Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

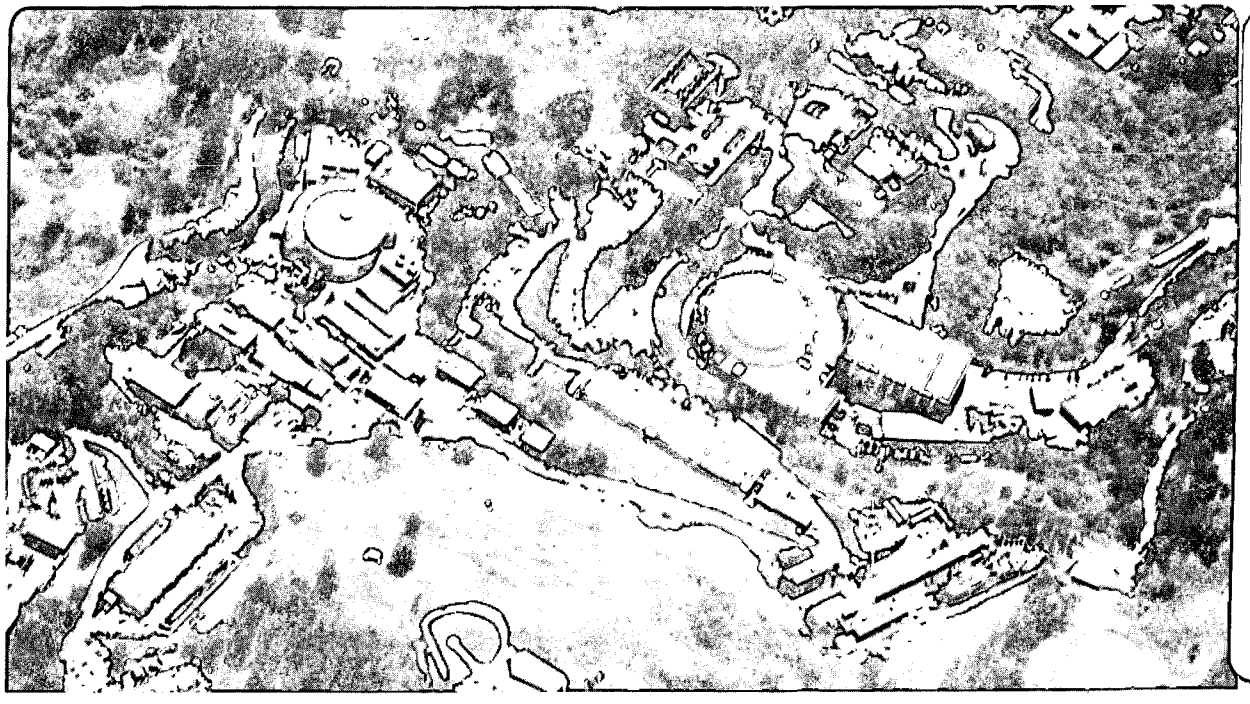
Information and Computing Sciences Division

To be presented at the 10th International Conference on the Entity Relationship Approach, San Mateo, CA, October 23–25, 1991, and to be published in the Proceedings

TOODM — A Temporal Object-Oriented Data Model with Temporal Constraints

E. Rose and A. Segev

April 1991



Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098

DISCLAIMER

LBL-30678

TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints

Ellen Rose & Arie Segev
Information & Computing Sciences Division
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

April 1991

This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints

Ellen Rose and Arie Segev

Walter A. Haas School of Business
The University of California and
Information and Computing Sciences Division
Lawrence Berkeley Laboratory
Berkeley, California 94720

ABSTRACT

A static Entity-Relationship (ER) or static Extended ER (EER) data model is not sufficient for representing the underlying time component of the data, more complex data types as found in planning, design and office automation applications or the operations required for this complex data. The decreasing cost of mass storage devices accompanied by an increased need for real-time systems and easier access to historical and planning data has made the study of the temporal aspects of data models more interesting both theoretically and practically. Furthermore, the ER-based data models can capture relationships between classes but they do not understand the object-oriented paradigm since they treat application-specific relationships and paradigm-specific relationships such as inheritance in the same manner. This shortcoming accompanied by a lack of support for the time dimension results in the specification of temporal relationships and constraints at the application level and often leads to inconsistencies in the data. In this paper, we extend the object-based ER model into a temporal, object-oriented model, incorporate temporal structures and constraints in the data model and propose a temporal, object-oriented query language for the model.

This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

1. Introduction

Most of the work in data modeling has either ignored the need for representing the time dimension or assumed that its relevance is self-evident. With databases becoming more complex in terms of the types of data which need to be represented, the types of manipulation operations needed and the information that needs to be extracted from the data, the requirements for richer data models have greatly increased. It is important that these modeling efforts address business and scientific application needs and requirements including the need for representing the time dimension.

A conceptual data model should serve as a common frame of reference for analysts, users and implementors throughout the database life cycle. It should provide a long-run, time-unrestricted view of those aspects of the real world scenario which are relevant to the application problem of concern [Bubenko 77]. The conceptual model should support and integrate structural, behavioral and temporal abstractions of object properties and provide a language for accessing information from the model.

A conceptual data model which incorporates past, present and future temporal semantics can be used as a building block to create a system which enables the user to schedule business events, formulate alternative plans, or request information about past events and planned products. Planning is based on future states of the real world and therefore can't be supported by existing static models which exclude temporal semantics. In this paper, we extend the object-based ER model into an object-oriented model and incorporate temporal structures and constraints in the data model. Our data model, TOODM, provides abstractions to capture the evolution of the schema and its instances and a means of expressing constraints on both of them.

The paper is organized as follows: related work is discussed in section 2, motivation and objectives are delineated in section 3, section 4 presents a preliminary model, section 5 discusses a temporal, object-oriented version of SQL (TOSQL), section 6 outlines implementation alternatives and section 7 concludes the paper with a discussion of future work.

2. Related Work

Only one study, [Ariav 87], of business and management application requirements for temporal data was found. This study suggests a need to support effective graphical representation, representation of states and multiple temporal orderings and support for differing user needs. Temporal properties including the merits of relative versus absolute time, the ability to identify an object as it evolves and the differences between events and states are discussed in [Bolour & Dekeyser 83]. Other properties such as precedence/succession relationships, intrinsic versus extrinsic time, granularity and lifespans [Segev & Shoshani 87] and types of time [Snodgrass & Ahn 85] have also been covered.

The bulk of the literature proposes adding temporal semantics at the logical level by extending the record-based relational model through tuple or attribute versioning. Research in this direction includes: [Clifford & Warren 83], [Navathe & Ahmed 86], [Abbod, etal. 87] and [Segev & Shoshani 88]. Generalized logical temporal models have recently been proposed by [Ariav 86] and [Segev & Shoshani 87]. A comprehensive bibliography on temporal databases and references to three previous bibliographies can be found in [Soo 91]. Several versions of temporal SQL's including [Fishman etal 87] can also be found in the literature.

Over the past decade, several researchers including [Klopprogge & Lockemann 83], [Ferg 85], and [Elmasri & Wuu 90] have proposed incorporating time into the object-based ER model as an entity or attribute. [Olen 85] and [Dogac, etal. 85] discuss alternative ways of classifying constraints for the ER model but make no mention of handling constraints on historical data. [Navathe & Pillalamarri 89] propose structural objectification of the ER model into their OOER model by adding the abstractions of generalization and classification. The OOER model is operationally object-oriented in the sense that generic operations in the data model can deal with complex objects. The OOER, however, is not behaviorally object-oriented in that it does not deal with the concepts of abstract data types, encapsulation, operator overloading and message passing. The concepts of schema and instance evolution are also not considered in the OOER model.

The literature search indicates that there is no single theory on how to treat time in a data model. Furthermore, no uniform means for expressing and enforcing temporal constraints was found. Issues such as how to provide for the database's evolution in terms of constraints, evolution of metadata, how to manage and group a large number of constraints, how to handle inconsistencies and how to handle exceptions to constraints remain open.

3. Motivation and Objectives

The object based entity-relationship model is well-known for its simplicity and clean, graphical notation [Mylopolous & Brodie 89]. As such, it has become the de facto standard for conceptual data base design in practice. Furthermore, the trend towards end-user development and the proliferation of data models brings the issue of usability into focus. A recent study by [Batra, etal 90] compares the usability of the record-based relational and object-based EER models. The study found that the EER model outperformed the relational model in all cases except the representation of unary relationships. The EER model was found to lead to better performance, but was not perceived by the users as being significantly easier to use than the relational model.

An increase in end-user development combined with the need for handling more complex objects and rapidly changing environments suggests a need for a more generalized data model such as an OODM which encapsulates both structural representation and behavioral abstractions. These trends lead to the main

motivation behind the design of a temporal object-oriented data model which provides a means of expressing and enforcing constraints in the data definition process and a means for allowing the data model to evolve to meet changes in organizational data and application needs. Furthermore, in current data models, changes in the problem domain are not easily transcribed into changes in the database solution domain due to the use of multiple paradigms in the development of the database. For example, in translating a graphical ER model representation into a tabular relational model, information is sometimes lost if it can't be represented with the structures provided in both paradigms. In the case of an OODM, the same paradigm is used in both the conceptual modeling and logical modeling phases thereby avoiding this problem.

As a first step in satisfying our objectives, a temporal object-oriented data model is developed to support the following functionality:

1) specification and enforcement of temporal constraints in the data model
2) support for past, present and future time points
3) support of type and instance evolution through the time sequence type, TS[T]
4) support for different user views of the same object using metadata
5) support for retro/proactive updates and queries through the design of a temporal, objected-oriented SQL data manipulation language
6) support for multiple time lines and corrections

To satisfy these objectives, abstractions to represent the structural and temporal properties of the data will be developed in the following section.

4. Model

In this section, a temporal object-oriented data model (TOODM) is developed. TOODM incorporates some of the functionality of the TEER model of [Elmasri & Wuu 90] and the generalized model of time sequences (TS's) of [Segev & Shoshani 87]. The basic assumptions, structures, representation of constraints and temporal operators of the model follow.

4.1. Assumptions

Since there is no single defining standard for an object-oriented data model (OODM), we specify the properties included in TOODM to be as follows:

1) multiple inheritance
2) encapsulation

3) object identity

Time in TOODM, is viewed as continuous and independent of events which are defined as durationless happenings in the real world that cause an object in the model to change state. We adopt the taxonomy of time terminology given in [Snodgrass & Ahn 85] for valid time (vt) and record time (rt). We also add event time (et) to represent the time when an event occurs. The state of an object persists over some duration of time and is represented by the values held by the instance variables of the object for this duration of time. The time interval over which a state holds is referred to as the valid time. The record time refers to when an object was recorded in the data model.

4.2. Structures

The concept of abstraction is used to stress the common properties and suppress unnecessary details. The main abstraction processes found in existing data models are generalization/specialization, association, aggregation, classification and identification. Figure 1 compares the EER and Object-Oriented data models in terms of the abstractions each can handle. For example, the association abstraction is represented as a relationship in the EER model and as an instance variable of type Class in each of the participating classes in the object-oriented model. The remainder of the notations are self-explanatory except for Ptype which is explained later on and can be viewed as a set of primitive values such as Integers. The OO-Model extends the EER model in that it also allows users to model behavior in terms of allowable operations on data associated with a particular type and the specification of explicit constraints in the data definition process.

Figure 1 - A Comparison of Abstractions in the EER and OO-Models

Abstraction	EER Model	OO-Model
Classification	Entity Type	Object Class
Aggregation	Aggregation Construct in EER only; Entity in ER	Object Instance
Association	Relationship	Instance Var: Class
Atomic Aggregate	Attribute	Instance Var: Ptype
Generalization	Generalization in EER Only	Super/Sub-Class Hierarchy
Identity	Primary Key Values	Object Identity

Our basic object-oriented data model includes the following primary constructors: objects, classes, types, methods, messages and collections. Every object has a type which is a template describing the structural and behavioral aspects of it's instances.

A basic OODM type-lattice of system-defined types is extended with new features in Figure 2. The type-lattice is a directed, acyclic graph (DAG) in which the nodes are type definitions and the edges are is_a links directed from the supertype towards its subtypes. OBJECT is the root node of the type-lattice. It is a lattice since subtypes may have more than one supertype. We discuss the new features in the remainder of this section.

Two new classes: V-CLASS and NV-CLASS represent versionable and non-versionable classes. A user-defined type can be a subclass of either. If the components (set of instance variables, messages/methods and constraints) of the type definition are allowed to change, the type should be defined as a subtype of V-CLASS as in Figure 3. The motivation behind the differentiation of versionable and non-versionable classes is to provide a mechanism for supporting both types of data since the non-versionable type components do not have a history as do those of the versionable type. This may have implications for storage management of the two types of data classes.

Figure 2 - New Types in the System Type-Lattice

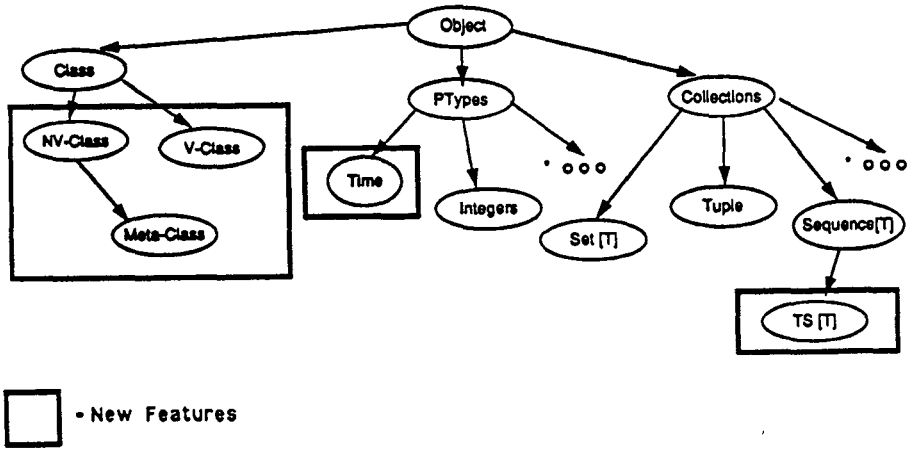


Figure 4 illustrates the definition of instance variables and messages for user-defined types. The type's instance variables are specified by (variable_name:domain) pairs. The domain can be a primitive type such as integer, a class or a collection. Each message has a target object from a particular domain to which the message is to be sent and an optional list of parameters. Parameters may represent inputs the user must supply to the message or variables to hold a return value for the message. Messages have the following format:

Message_name(Target; p-list) where p-list is a list of parameters

Figure 3 - Sales Office Example of User-Defined Classes

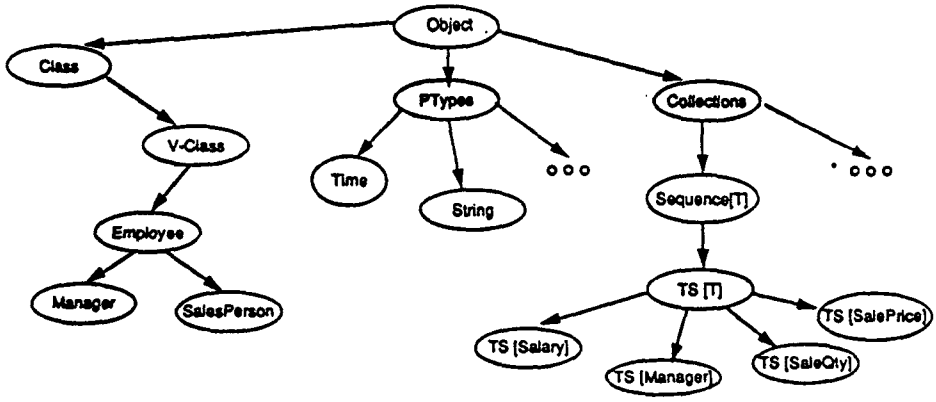
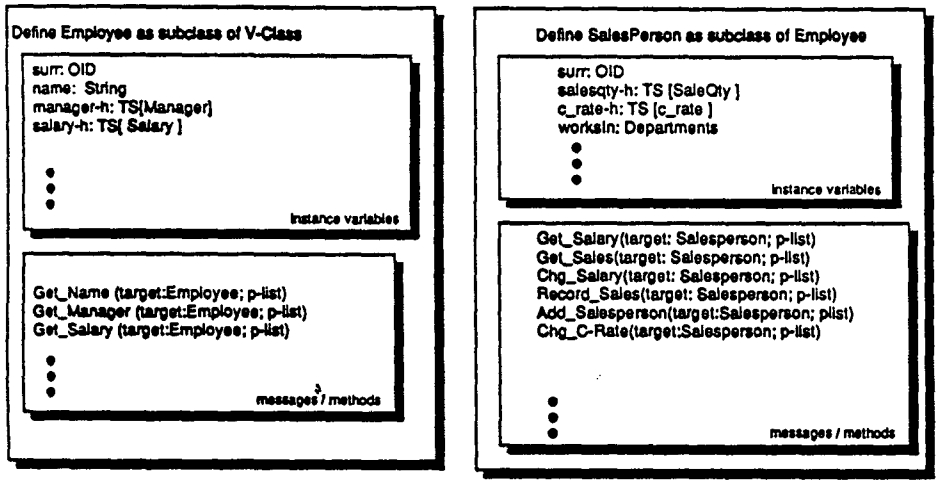


Figure 4 - Possible Type Definitions for Employee and SalesPerson



The parameters are given as parameter=value when they act as inputs to the message and as parameter when they hold outputs or serve as a variable input parameter in a query. Messages are sent by the objects who own them and must be recognizable by the recipient, target object. This visibility is determined by instance variables which represent relationships with other objects. The value of such an instance variable is referred to as a shared value and is really a reference to another object. This reference provides a logical access path to the related object.

These type definitions reference the TS[T] type which is explained later in this section. Messages can only be used to refer to the states of object instances in their defining type. Temporal predicates can be attached to messages to specify which version of the type definition is needed in a request so the user can ask for the information as it existed in the database as of some point in time from any other point in time. This will be

illustrated in section 5 in the discussion of TOSQL.

Figure 5 - Example Instances of the SalesPerson Type

SalesPerson

OID	name	salary-h	manager-h	salesqty-h
E1	Mary	SH1	MH1	SQH1
E2	Jones	SH2	MH2	SQH2

TS[Salary] where history := (<oid,salary,vt, rt,et>)

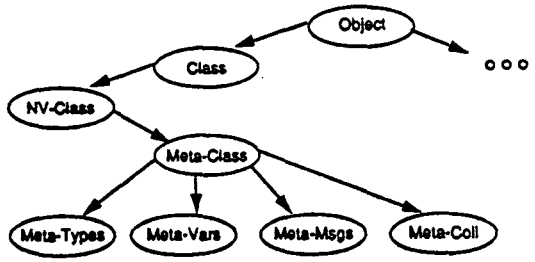
OID	history	corrections
SH1	((s1,20K,[1,4],1,1), (s2,25K,[5,now*],4,3))	((s1,22K,[1,4],4,1))
SH2	((s10,18K,[1,6],1,1), (s11,20K,[7,10],8,5))	

TS[SalesQty] where history := (<oid,qty,vt, rt,et>)

OID	history	corrections
SQH1	((q1,100K,1,2,1), (q2,50K,6,6,5), (q3,70K,8,4,8))	
SQH2	((q11,90K,1,4,1), (q12,100K,2,4,2), (q13,30K,10,7,9))	

Figure 5 shows some instances of the SalesPerson and two time sequence types. The values E1, SH1, MH1, q1, s1, etc. represent object id's (oid's) which serve as pointers to other objects defined in the system. All TS[T] types have a history instance variable which is a time ordered sequence of tuple values where the meaning of each position in the tuple is defined in the TS[T] type and its subtypes. Each member of the history set has an OID so it can be referenced.

Figure 6 - Meta-Objects and the Meta-Type Definition



Define Meta-Types as subclass of Meta-Class
typ-oid: OID
typ-name: TS [String]
superclasses: TS [Set [Meta-Types]]
var-list: TS [Set [Meta-Vars]]
msg-list: TS [Set [Meta-Msgs]]
creator: TS[UserID]
constraints: TS[Set [BOOL]]

Some of the instance variables (ex. salary) are of type TS[T] so each instance contains a value and associated time intervals or time points which are explained later in this section. Meta-class definitions can't be changed so they are subtypes of NV_CLASS as in Figure 6. The meta-data are timestamped with a record-

timestamp (rt) and include the user-id or group-id of the users who created the definition. Treating meta-data values as time sequences of values with user-group ids allows us to assign different names to the same object for different groups at different times.

Figure 7 - Instances of Meta-Types and Meta-Vars

Meta-Type Instances

OID	name(string,creator,rt,vt)
T1	(Employees, all, 1, [1, now+])
T2	(Salespersons, all, 2, [2,now+])(Salespeople, jones+++s, 4, [4,now])

Meta- Variable Instances

OID	Var_Name(string,creator,rt,vt)	Var_Type(Meta-Type,creator,rt,vt)
V1	(name, all, 1, [1,now+])	(string, all,1, [1,now+])
V2	(address, all, 1, [1,now+])	(string,all,1, [1,now+])
V3	(phone, all, 1, [1,now+])	(Integer,all,1, [1,now+]), (phone,jones,9, [9,now+])
V4	(salary-h, all, 1, [1,now+])	(TS[Salary], all, 1, [1,now+])

Figure 8 - TS[T] Type and the Time Primitive Type

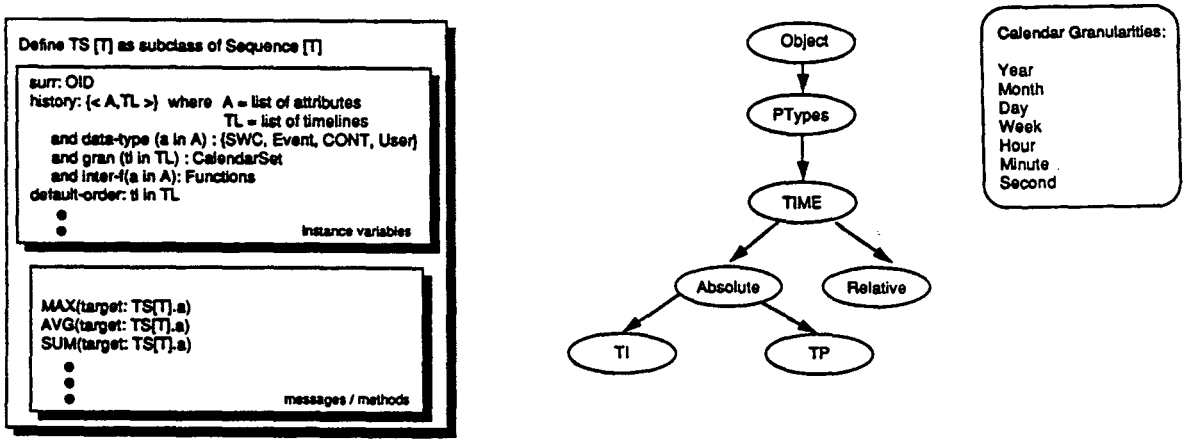
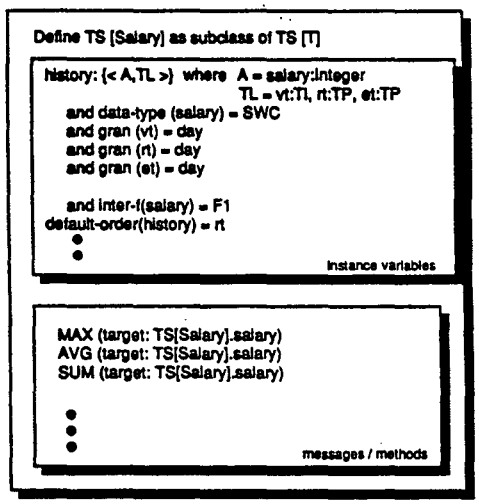


Figure 7 gives examples of instances of the Meta-Type and Meta-Vars definitions. Two names for the SalesPerson type: SalesPerson and SalesPeople exist. Each name refers to the same object type and each was created by a different user at a different time. The user-id called "all" is the default. If user "jones" sends the Add_Salesperson() message to the object Salesperson using either name, the correct object can be found using the meta-data in the meta-types object which allows different users to view the same object in different ways. Primitive types, PTypes, represent objects which do not have instance variables. Ptypes usually serve as a

simple domain for the instance variables of other objects. The TIME ptype can be specialized into subtypes as shown in Figure 8 where the time points can have different calendar granularities. Translation between calendar granularities in operations involving multiple time lines will be provided automatically with the default being to translate to the greater granularity. TP represents time point and TI represents a time interval in Figure 8.

The COLLECTION type has three pre-defined subtypes: SET[T], TUPLE, and SEQUENCE[T]. A set is a collection of unordered, homogeneous objects that doesn't contain any duplicates. A tuple is a collection of heterogeneous objects ordered by object name as found in the relational model. A sequence is an ordered collection of objects with a pre-defined subclass called time sequence (TS[T]) which is ordered on time.

Figure 9 - TS[Salary] Type Definition



TS[T] is a parameterized type where the parameter T refers to the user-defined or system-defined type upon which we are collecting a time sequence of information. The TS[T] subclass can be used to represent the concept of a time sequence as defined in [Segev & Shoshani 87]. Figure 9, the TS[Salary] type definition, defines the data-type of the salary attribute of the history and the granularity of the timelines. The predicates data-type(x) and gran(x) are used to define constraints on the attributes and timelines in the history. These predicates are inherited from the generic TS[T] type.

The evolution of an object is represented by time sequences of the values the object's instance variables have held. For example, in Figure 9 each salary value that an employee has held in the past, present or future is associated with three timelines: a record time (rt), a valid time (vt), and an event-time (et). This ability to represent multiple timelines allows us to get information from the data model as it existed at some point in the

past or as it will exist in the future based on the information available at some particular time point in the model. Missing data points can be interpolated using the data-type information and an interpolation function. For example, a particular employee may have been given a raise on March 1, 1989 (et) which becomes effective on April 1, 1989 (vt) or is even valid retroactively on January 1, 1989 (vt). This fact may not be recorded in the database until March 5, 1989 (rt). Other timelines could be formed by providing other interpretations to time values such as correction times. (Note: In the figures, ordinal numbers versus actual dates are used for simplicity.)

The following section discusses the constraints which control how the model evolves. A brief general discussion of the types of constraints to be included in the model precedes the discussion of temporal constraints.

4.3. Constraints

Constraints are logical restrictions on data and operations on that data. A constraint may be inherent in the structure, explicitly specified or implied. Inherent constraints are application-independent constraints and explicit constraints are application-dependent constraints specified by the user. In TOODM, application-dependent constraints will be defined in the type descriptions of user-defined types and inherent constraints in the system-defined types of Figure 2. Constraints are timestamped and restricted in scope since it would not be valid to use a current set of constraints to control a correction on old data and user-defined rules may change over time as the business changes. The property of encapsulation provides a natural means of restricting scope and of separating user-defined and system-defined constraints.

The constraints in TOODM can be grouped into static, dynamic and temporal restrictions based on the states that need to be considered when determining whether or not the constraints hold. In this paper, we focus on the temporal constraints but we briefly distinguish between the three groupings as follows. A constraint is static if it imposes restrictions on moving to the next state without concern for any other states. Therefore, in the case of a temporal database, a static constraint refers to conditions that must be true on any snapshot of the database. Dynamic constraints are specifications of allowable changes that move a database to the next state taking both the previous and next states into consideration. These constraints have a set of pre-conditions that indicate certain propositions which must hold true in order to initiate a state change and a set of post-conditions which must hold true for the new state to be created. Dynamic constraints only refer to changes from the current state to a new state and do not reference historical or future states. Temporal constraints including time-related static constraints will be discussed in the following section.

4.3.1. Temporal Constraints

Temporal constraints can be thought of as rules for providing a temporal ordering and as rules for establishing precedence/succession relationships on sequences of constraint and object states or on non-adjacent states. Specifying when constraints and objects are valid allows us to create a "time filter" to prevent the user from viewing inconsistent data. The time filter is needed because we allow the addition of inconsistent facts to the model as long as the facts do not exist at the same time. For example, an employee may be a manager at some point in time recorded in the database and he may not be one at some other point in time recorded in the database. We classify temporal constraints as follows:

1.0 Time-Related Static Constraints

2.0 Constraints on Sequences of States and Non-Adjacent States

3.0 Constraints on Relationships between Time Sequences

4.0 Constraints between States in a Time Sequence

Category 1.0 constraints help to avoid confusion as to which properties and operations can be associated with an object instance at any point in time.

EX: Every object has a non-null system assigned timestamp of type record time which indicates when the information was recorded in the database.

EX: Every component of a type which is a subtype of V-Class has an associated time-interval of type VALID TIME which the system will prompt the user for when the type information is updated or recorded.

These constraints imply that variables which are time invariant may not require a valid-timestamp but they do require a record-timestamp since they are subject to corrections. The record-timestamp can be used to eliminate corrections that did not exist when viewing the state of the data base at some point in time prior to when the correction was made. Inherent constraints on lifespans would also fall under category 1.0.

Since object types can be versioned, all past and planned definitions for an object type can be stored. In TOODM, this is done by timestamping the properties of the type definition with valid and record times. The valid timestamp may be a specific time interval or one of the temporal modifiers below. Each method can also be labeled with a temporal modifier which indicates the time period over which it is assumed to be valid. The first eight of the following modifiers were specified in a first order logic model presented in [Kung 84]. These modifiers can be used in specifying allowable sequences in category 2.0 constraints.

TM1: always in the past, [p-now]

TM2: always in the past & present, [p+now]

TM3: sometime in the past, [sp-now]

TM4: sometime in the past or present, [sp+now]

TM5: always in the future, [f-now]

TM6: always in the future & present, [f+now]

TM7: sometime in the future, [sf-now]

TM8: sometime in the future or present, [sf+now]

TM9: in window = [now - k, now]: where k is some number of time units

TM10: in window = [now, now + k]

TM11: in window = [t1,t1 + k] where t1 is some reference point other than now

TM12: always in the past, present & future, [pf+now]

TM13: sometime in the past, present or future, [spf+now]

An explicit example of a category 2.0 constraint is "An employee can't be rehired." This constraint refers to non-adjacent states and means attaching TM's to the add_employee message's definition in type Employee. This constrains the value of Employee to be someone who was not an employee sometime in the past including the present as shown below:

Assert constraint_id:

ON: add_employee(Employee)

Condition: Employee.oid not in Employee [sp+now]

An example of a category 3.0 temporal constraint between TS's is: "A salesperson's commission rate history must start 1 year after their base pay history." This might be expressed as follows:

Assert constraint_id:

ON: Employee.salary-h, Salesperson.c_rate-h

Condition: lifespan.start(salary-h.history) 1 year before lifespan.start(c_rate-h.history)

An example of a category 4.0 temporal constraint between instances of a TS is: "John's salary history is contained in the history of his work relationship with the company." This might be expressed as follows:

Assert constraint_id:

ON: Employee Instance John

Condition: lifespan(John.salary-h.history) always during lifespan(John.worksIn)

where salary-h and worksIn are instance variables of instance John of type Employee and lifespan is a predicate defined in the TS[T] type which refers to the union of the valid time intervals for the instance variable under

consideration.

4.4. Operators

Support for schema evolution, a primitive time type and time variant attributes of objects requires the development of new operators and changes the semantics of some existing operators such as deletion. Since a query can include any collection of user-defined operations (messages) from user-defined types, it may be more difficult to find equivalence preserving transforms in an OODM. The addition of each new class or type introduces new operators which leads to a new algebra whose operators are not known to the optimizer due to the encapsulation property of OODM's.

Messages (operators) can perform update, correction or retrieval operations on the values of the instance variables of objects. Update messages perform the insertion of new values as well as deletion and change operations to existing values, object classes and their instances. The semantics of the delete and change operations differ from those of a static data model. A deletion implies the end of an object's lifespan without the removal of the information from the database and that the object can't be referenced by any other objects from the time of deletion forward until the time if any that the object re-enters the system.[1] A change to existing values or type definitions implies ending its valid time interval at the time of the change say t-1 time units and starting the valid time interval of the new value or definition at time t. The endpoint of the new value's valid time interval could be a specific time point or now+ if it is assumed to hold in the future and present. These messages should be defined in the type OBJECT of Figure 2. Messages can be refined and constrained to express the semantics of the application in the subtype definitions. Since all user-defined classes are instances of CLASS it defines operations (messages + methods) to:

-add, drop and rename classes, messages and instance variables

-add or drop a superclass

-expand the domain of variables or change their parent or default

-change a message's origin or the method it attaches to

Methods need to be defined for each of the system-defined classes in the type-lattice of Figure 2. Each level down in the lattice will add new methods specific to objects of its type. The system will automatically create a class object which is a SET[Type] object when a new type is added to the lattice in order to facilitate query processing.

[1] The historical information of the "deleted" object can still be queried since the history of the object's past states prior to the deletion still exists in the database.

Temporal operators are messages and special predicates that reference the temporal ordering of time sequences or the time values held by objects in those sequences. These operators are defined in the TS[T] class and refined in its subclasses to accommodate user-defined temporal operators or different semantics for inherited operators. Several possible temporal operators are illustrated in the following section in sample queries.

5. A Temporal Object-Oriented Query Language - TOSQL

Since messages are used to address one instance at a time of a particular type in which the message is defined we need a means of getting information about groups of objects that meet specified conditions. This can be done using one of the higher-level navigational languages associated with ER and object-oriented models or by using a modified declarative high level language such as SQL. SQL is chosen here since it is endorsed by most major developers of the relational model which is the current standard. Furthermore, SQL is the only relational language for which a standard has been developed and it is also being promoted as the interface language of choice for databases.

SQL can be made object-oriented by allowing messages to appear in the WHERE and SELECT clauses and by using direct references to objects as opposed to primary key values. Several Object-Oriented SQLs have been proposed in the literature including OSQL used in IRIS where the SELECT clause contains nested functions [Fishman, etal 87]. Our TOSQL differs from these SQLs in its inclusion of temporal operators and clauses and the use of nested messages versus functions where the result of the innermost message is an input to the next innermost message etc.

Temporal extensions to SQL include the addition of a WHEN clause, a TIME-SLICE clause and a MOVING-WINDOW clause. Timestamps or intervals may also be associated with instance-variables in the SELECT clause and with variables or messages in the WHERE clause. Temporal predicates such as DURING, BEFORE, AFTER, etc. as defined in several previous works including [Navathe & Ahmed 86] [Snodgrass 87] can also be used in comparing time intervals within a query. References to the inherent temporal ordering in a time-sequence through predicates such as FIRST, LAST, T-LAST, T-NEXT, V-LAST and V-NEXT[2] as found in [Segev & Shoshani 87] can also be used to answer queries which are not possible in static data models.

The general form of a TOSQL query follows where [ ] indicates optional clauses, | indicates "or" and { } indicates that the group can repeat in a list. A detailed BNF Grammar of the syntax appears in the appendix.

[2] These predicates may appear in a GROUP TO clause as in GROUP TO V-LAST 3, where each value in the original sequence is replaced by the sum of itself and the two numbers before it. In the case of V-NEXT each value is replaced by itself and the specified n-1 values which follow it in the original sequence. T-LAST/NEXT are similar predicates except we refer to time points versus value points. Value points have actual recorded values associated with them whereas time points are all potential points (based on the specified granularity) that could have values.

SELECT [tem-seq] msg-exp-list

[FOR EACH class-name variable-name]

[WHERE clause]

[WHEN clause]

[GROUP-TO temporal-pred ON timeline]

[MOVING WINDOW clause]

[TIME-SLICE clause]

The FOR EACH clause is similar to the FROM clause in standard SQL. Since some queries may refer to a particular object and each object has a unique oid we don't need to specify which class (relation) the object is from unless we want to iterate over all members of the class in the query.

The following two queries show the use of the WHEN clause and direct access through the object-id (oid) which is represented by the Employees Name for simplicity.

Q1: What was Mary's salary on January 10,1990?

SELECT Get_salary(Mary.salary-h.history; salary)

WHEN '01/10/1990' DURING historyt.vt

Q2: Who was Mary's manager after Rachel was her manager and when was the change made?

SELECT Get_manager(Mary.mgr-h.history; mgr, et)

WHERE PREV history.mgr = Rachel

WHEN history.vt FOLLOWS Rachel.vt

In Q2, the predicate PREV refers to the object in the history set of Mary's manager history object which contains the value of Rachel in the mgr attribute. Since Rachel may have acted as Mary's manager more than once this query would only find the first occurrence.

The following queries make use of the inherent temporal ordering of a time-sequence. Since we allow for multiple timelines we will need to create indexes for each in order to effect different temporal orderings for these queries. The default ordering is by record time since we have an append only data model.

Q3: Find the third change of manager for Mary and the duration over which he/she was Mary's manager.

SELECT 4th Get_manager(Mary.mgr-h.history; mgr, DURATION vt)

Q4: List the names and salaries of all employees who started with a first salary of at least 30K.

SELECT Get_name(e: name), Get_salary(e.sal-h.history; salary)

FOR EACH Employee e

WHERE FIRST e.sal-h.history.salary > 30K

The following query illustrates the use of the TIME-SLICE clause which selects only those objects which were valid during the given time period specified in the clause.

Q5: List the manager history of all employees who were employed sometime during the last 4 years.

SELECT Get_manager(e.mgr-h.history; mgr, vt, et, rt)

FOR EACH Employee e

TIME-SLICE year e.mgr-h.history.vt:[now - 4,now]

The next query makes use of aggregation over time intervals using the predicate DURATION to get the length of each time interval that meets the specified conditions. Aggregate operations include MAX, MIN, SUM and COUNT. These aggregation operators map a set of points that fall into a given time interval into one point in the result set.

Q6: How long did Mary work for Rachel?

SELECT SUM(Get_manager(e.mgr-h.history; DURATION vt))

FOR EACH Employee e

WHERE e.mgr-h.history.mgr = 'Rachel'

The following query involves accumulation of time points to produce another set of time points as in [Segev & Shoshani 87]. The special predicate T-LAST 7 appears in the GROUP TO clause and vt (valid time) is the timeline of interest as indicated in the ON part of this clause. The granularity of valid time is specified in the type definition of TS[Salary] and will be used to determine the time points. A new time sequence of average salary values where each value in the sequence is in 1 to 1 correspondence with each time point in the lifespan of the original sequence is produced in the result. The values of the time points in the new sequence are equal to the average of the original value and the 6 actual or interpolated values that precede it.

Q7: Produce a 7-day moving average of sales.

SELECT AVG(Get_qty(e.saleqty-h.history; qty))

FOR EACH Employee e

GROUP TO T-LAST 7 ON e.saleqty-h.history.vt

A MOVING WINDOW clause is used in the following query where only the length of a time interval is known. Moving windows allow us to obtain aggregate information about the moving time interval over the lifespan of the object specified in the query. The granularity of the timeline in the ON part of the MOVING WINDOW clause specifies the granularity of the timeline selected, the "5" indicates the length of the window and "years" the granularity of the window. We begin with the first data point in the sequence of prices ordered on valid time. The start time point of the valid time interval of the first object in the TS[Price] object extension

is used to calculate the first window which is the time start plus 5 years. Next the aggregate operator, MAX is applied to the group of price values associated with the data points in s that fall in this window. The time start of the vt interval of the next object in the sequence s is then used to determine the next window and so forth until the last point in the lifespan is reached.

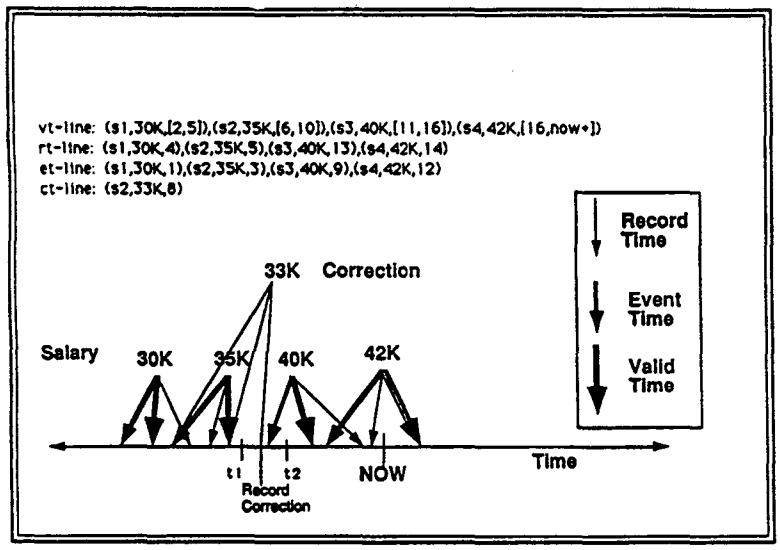
Q8: Find the 5 year period where price increased the most.

SELECT MAX(Get_price(s.price; price))

FOR EACH TS[SalePrice] s

MOVING WINDOW 5 years ON s.price.vt

Figure 10 - Multiple Timelines and Corrections



Update and correction messages can be defined in a similar manner such as in the following correction. This results in the addition of a pointer to a correction time-sequence for each object in the time sequence. A built-in constraint would be that the correction sequence's record time must be greater than the record time of the base sequence.

C1: Correct the salary amounts of all employees to be 10% more than previously recorded.

INSERT INTO TS[Sal-Corr]

VALUES (salary=1.10*salary, vt=vt, rt='time corrected', et=et)

FOR EACH TS[Salary].history.LAST

where the 'time corrected' refers to the time when the correction is made and LAST refers to the last element of each object's history sequence. Figure 10 illustrates making a correction to one value in the time sequence of a particular employee's salary history. If we ask what the salary was as of time t1 we would get 35K whereas if we asked what the salary was as of t2 we would get 33K since the correction didn't exist at time point t1. Also

note that 42K is a future salary value since its valid time exceeds the current time NOW.

6. Implementation

Viable approaches for implementing this system include the following:

1) build from scratch using an OOPL
2) build from scratch using an extended language such as C++
3) build on top of an existing DBMS

A major disadvantage of the build from scratch approaches is the need to build all the database capabilities into the system. A disadvantage of using conventional languages and their extensions is the possibility that knowledgeable programmers will use their programming skills to ignore the encapsulation built-in by the system-designers. An advantage of using an existing DBMS is that it provides support for database capabilities such as persistence, transactions, concurrency control, querying and recovery. We are using the third option, building on top of the extended relational database management system, Postgres. This option was advantageous to us for the aforementioned reasons and the source code for the system is available. Postgres also includes some object-oriented features such as support for oid's and temporal features such as using an append-only storage format where updates are actually insertions. When a change is made to an object, the current end point of the object's valid time is set to the change time and the new version of the object is inserted with a valid start time of the change time plus one time unit.

Our database system will be split into two subsystems: an interpreter and a storage manager. The interpreter will provide the operational semantics of TOODM. It will enforce encapsulation and execution of methods and will call the Postgres storage manager to perform physical data access and manipulation. The Postgres storage manager will provide secondary storage of objects and will be responsible for moving data back and forth between main memory and secondary storage. The storage manager is also responsible for creating new objects, concurrency control, recovery and indexing. New indexes for the management of temporal data will need to be added to the Postgres storage system. Which indexes are necessary will depend on the access patterns of the applications using the database.

The Postgres storage manager can be outlined as follows. Postgres stores the state of objects as an anchor tuple with a pointer to the delta tuple which represents the next change. This means the object is stored apart from its component objects which will mean better performance for queries that range over all versions of an object and worse performance for those that require all subcomponents of the object. The decision of how to

cluster objects on secondary storage pages is really dependent on access patterns for the applications using the system since objects can only be stored in one way unless redundancy is introduced but this will have update penalties. Trying to cluster an object with all the objects it references is also difficult since the object may be referenced by many other objects. Postgres uses a standard two-phase locking policy for concurrency control and can recover instantly from crashes since there is no recovery code to run. Instead, each record has an additional eight fields which consist of the oid, the transaction identifier of the interaction that inserts the record (Xmin), the commit time of Xmin (Tmin), the command identifier of the interaction inserting the record (Cmin), the transaction identifier of the interaction deleting the record (Xmax), the commit time of Xmax when the record becomes invalid (Tmax), the command identifier of the interaction which deletes the record (Cmax) and the pointer (PTR) to the changed data (delta). If the storage manager knew very little about the data model it could support several front ends but it must know something about interobject references in order to support index maintenance and constraint enforcement on objects.

The TOODM itself will be implemented in the interpreter subsystem as previously mentioned. The interpreter is essentially a frontend system for the storage manager that implements our model. Our interpreter will contain a parser to generate a parse-tree from user supplied queries in TOSQL. The query-rewrite rule system of the parser will be used to enforce constraints on objects of the target class(s) of the query by adding the constraint(s) to the query. The architecture of Postgres is modularized to allow the addition of other parsers which facilitates this aspect of the design. In addition an optimizer module will convert the parse-tree into an execution plan. The execution module will be invoked by the optimizer and will initialize access methods and a constraint-enforcement system to run the execution plan.

7. Conclusions and Future Work

Some of the basic features of our TOODM have been outlined. These features include support for modeling changing type definitions, addition/deletion of types and recording changes in the state of instances. Support for these features requires the definition of static, dynamic and temporal constraints both by the system and user-defined types. We have focused on the temporal features in this paper leaving discussion of the other constraint categories for another paper. Addition of temporal and dynamic features results in an active data model which provides better support to the changing problem-domain that characterizes most CAD, OIS and Business applications. The inclusion of past, present and planned information and alternate ways of describing objects in different time periods provides support for planning and decision-support applications.

We have provided some of the capabilities needed to support objective six by adding valid time, record time and user-defined time as possible interpretations to the time component of objects of type TS[T].

Objectives one, two and five are partially satisfied by incorporating valid and record time-lines for instance variable values that vary with time. Future values are represented by valid times greater than now, the present with a valid time of now and the past by valid times prior to now. Objective three is partially satisfied in the development of temporal constraints on the TS[T] type. The incorporation of the meta-data into meta-class objects also aids in satisfying our fourth objective of allowing different user views of the same object. A set of meta-rules to check new constraints against existing constraints is also needed. Preliminary development of a temporal, object-oriented extension to SQL has also been discussed.

Some of the constraints and operators which must be supplied by this model have been outlined. Future work will include defining operators that allow for the merging of alternate versions into a final version, rules for conflict resolution in the cases of multiple inheritance of properties and ways to evolve messages without necessitating major rewrites to their methods.

We also need to explore the issue of defining relationships between objects that do not exist at the same time such as the relationship between sales forecasts over different time periods or between an employee and the disbursement of his/her death benefits. Efficiency issues involving the definition of indexes and storage methods and design of a query optimizer which utilizes semantic query optimization based on constraints for an implemented version of the model also remain open for future exploration.

Appendix

We include only the temporal and object-oriented extensions to SQL in this BNF based on the one found in [Navathe & Ahmed 86]. Standard SQL is assumed for other operations excluded from this grammar such as the GROUP BY clause. Expressions for CREATE, INSERT, DELETE, etc are not yet included.

```
query := query-exp [ORDER BY order-by-list]
query-exp := query-block
                | query-exp query-block
                | query-exp
query-block := select-clause
                [FOR EACH class-name variable-name]
                [WHERE clause]
                [WHEN clause]
                [GROUP TO t-seq-pred ON t-line]
                [MOVING WINDOW clause]
                [TIME SLICE clause]
order-by-list := message | order-by-list, message
                | nested-messages | order-by-list, nested-messages

select-clause := SELECT [tem-seq] msg-exp-list
msg-exp-list := msg-exp | msg-exp-list, msg-exp
msg-exp := message | nested-messages | t-agg-term
t-agg-term := agg-fn(msg-exp)
agg-fn := COUNT | MAX | MIN | AVG | SUM

WHERE clause := WHERE boolean1
boolean1 := bool-term1 | boolean1 OR bool-term1
bool-term1 := bool-term1 AND bool-fac1 | bool-fac1
bool-fac1 := [NOT] bool-prim1
bool-prim1 := pred1 | boolean1
pred1 := exp1 comparison exp1
          | q-spec comparison q-spec
q-spec := query-block | query-exp | [tem-seq]
          msg_result | constant
exp1 := arith-term | exp1 add-op arith-term
arith-term := arith-fac | arith-term mult-op arith-fac
arith-fac := [add-op] primary1
primary1 := agg-fn(exp1 | DURATION ) | COUNT (*)
                | constant | {exp1} | obj-des
obj-des := tem-seq msg_result [bf-af tem-seq BREAK]
                | [tem-seq] msg_result
tem-seq := PREV | NEXT | FIRST | SECOND | THIRD | Nth | LAST
comparison := comp-op | IN | NOT IN | CONTAINS
                | NOT CONTAINS
comp-op := > | < | >= | <= | = | !=
add-op := + | -
mult-op := * | /
constant := quoted-string | number
```

WHEN clause := WHEN boolean2
boolean2 := bool-term2 | boolean2 OR bool-term2
bool-term2 := bool-fac2 | bool-term2 AND bool-fac2
bool-fac2 := [NOT] bool-prim2
bool-prim2 := pred2 | boolean2
pred2 := exp2 t-comp-op exp2
exp2 := tem-seq ts-var [bf-af tem-seq BREAK]
| [tem-seq] ts-var
| tem-constant
tem-constant := t-term | "["t-term,t-term"]" | path.vt
t-term := t-fac add-op number
t-fac := t-point | NOW | TM | number granularity
tem-comp-op := BEFORE | AFTER | DURING | OVERLAPS | MEETS
| EQUIVALENT | ADJACENT | FOLLOWS | PRECEDES

bf-af:= BEFORE | AFTER
t-seq-pred := LAST | FIRST | T-LAST n | V-LAST n | T-NEXT n | V-NEXT n

t-point := date | path.rt | path.et | path.user-def-t-line
t-line := path.vt | path.rt | path.et | path.user-def-t-line

MOVING WINDOW clause := MOVING WINDOW length granularity ON t-line

TIME SLICE clause := TIME SLICE granularity t-message: t-interval
t-message := (message that returns a t-line)
granularity := years | months | weeks | days | hours | minutes | seconds

Bibliography

[Abbod, etal. 87] Abbod, T., Brown, K. and Noble, H., Providing Time-Related Constraints for Conventional Database Systems, *Proceedings of the 13th International Conference on VLDB,* Brighton, 1987, pp.167-175.

[Ariav 86] Ariav, G., A Temporally Oriented Data Model, *ACM Transactions on Database Systems,* V. 11, N. 4, December 1986, pp. 499-527.

[Ariav 87] Ariav, G., Design Requirements for Temporally Oriented Information Systems, *TAIS Conference,* May 1987, pp. 3-16.

[Banerjee etal 87] Banerjee,J., Chou, H., Garza,G., Kim, W., Woelk,D. and Ballou, N., Data Model Issues for Object-Oriented Applications, *ACM Transactions on Office Information Systems,* V. 5, N. 1, 1987.

[Bolour & Dekeyser 83] Bolour, A. and Dekeyser, L.J., Abstractions in Temporal Information, *Information Systems,* V. 8, N. 1, 1983, pp. 41-49.

[Clifford & Warren 83] Clifford, J. and Warren, D.S., Formal Semantics for Time in Databases, *ACM Transactions on Database Systems,* V.8, N. 2, June 1983, pp. 214-254.

[Elmasri & Wuu 90] Elmasri, R. and Wuu, G.T.J., A Temporal Model and Query Language for ER Databases, *Proceedings of the International Conference on Data Engineering,* May 1990, pp. 76-83.

[Ferg 85] Ferg, S., Modelling the Time Dimension in an Entity-Relationship Diagram, *Proceedings of the 4th International Conference on the ER Approach, In Entity-Relationship Approach,* Ed. Chen, P.P.S., Elsevier Science Publishers B.V. North-Holland, 1985, pp. 280-286.)

[Fishman etal 87] Fishman, D.H., etal., IRIS: An Object-Oriented Database Management System, *ACM Transactions on Office Information Systems,* V. 5, N. 1, 1987.

[Kappel & Schrefl 89] Kappel, G. and Schrefl, M., A Behavior Integrated Entity-Relationship Approach for the Design of Object-Oriented Databases, *Proceedings of the 7th International Conference on the ER Approach, In Entity-Relationship Approach,* Ed. Batini, C., Elsevier Science Publishers B.V. North-Holland, 1989, pp. 311-328.)

[Klopproge & Lockemann 83] Klopprogge, M.R., Lockemann, P.C., Modelling Information Preserving Databases: Consequences of the Concept of Time, *Proceedings of the 9th International Conference on VLDB,* Florence, Italy, 1983, pp.399-416.

[Kung 84] Kung, C.H., A Temporal Framework for Database Specification and Verification, *Proceedings of the 10th International Conference on VLDB,* Singapore, pp. 91-99, 1984.

[Manola & Dayal 86] Manola, F. and Dayal, U., PDM: An Object-Oriented Data Model, *International Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, September, 1986.

[Mylopolous & Brodie 89] Mylopolous, J. and Brodie, M., Introduction in: *Readings in Artificial Intelligence and Databases* Morgan Kaufman Publishers, Inc., 1989.

[Navathe & Ahmed 86] Navathe, S.B. and Ahmed, R., A Temporal Relational Model and Query Language, *Information Sciences, 49* 1989, pp. 147-175.

[Navathe & Pillalamarri 89] Navathe, S.B., and Pillalamarri, M.K., OOER: Toward Making the E-R Approach Object-Oriented, *Proceedings of the 7th International Conference on the ER Approach, In Entity-Relationship Approach,* Ed. Batini, C., Elsevier Science Publishers B.V. North-Holland, 1989, pp. 185-206.)

[Olen 85] Olen, O., Integrity Constraints in the Conceptual Schema SYSDOC, *Proceedings of the 4th International Conference on the Entity-Relationship Approach,* Chicago, Il, Oct 28-30, 1985, pp.288-294.

[Segev & Shoshani 87] Segev, A. and Shoshani, A., Logical Modelling of Temporal Databases, *Proceedings of ACM SIGMOD International Conference on the Management of Data,* May 1987, 454-466.

[Segev & Shoshani 88] Segev, A. and Shoshani, A., The Representation of a Temporal Data Model in The Relational Environment, *An Invited Paper to the 4th International Conference on Statistical and Scientific Database Management,* LBL-25461, August 1988.

[Soo 91] Soo, M.D., Bibliography on Temporal Databases, *SIGMOD Record,* V. 20, N. 1, March 1991, pp. 14-23.

[Snodgrass & Ahn 85] Snodgrass, R. and Ahn, I., A Taxonomy of Time in Databases, *Proceedings of ACM SIGMOD International Conference on the Management of Data,* May 1985, pp. 236-246.

[Snodgrass 87] Snodgrass, R., The Temporal Query Language TQUEL, *ACM Transactions on Database Systems,* June 1987, pp. 247-298.

[Stonebraker & Rowe 87] Stonebraker, M. and Rowe, L., The POSTGRES Data Model, *Proceedings of the 13th VLDB Conference,* Brighton, pp. 83-94, 1987.

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
INFORMATION RESOURCES DEPARTMENT
BERKELEY, CALIFORNIA 94720