Computación y Sistemas Vol. 4 No.4 pp. 327 - 339
© 2001, CIC - IPN. ISSN 1405-5546 Impreso en México

# Temporality Specification in Automatic Production Software Environments Based on Object-Oriented Conceptual Models

**Carlos Meneses**[1], **Oscar Pastor**[2], **Juan Carlos Molina**[2] and **Emilio Insfrán**[2]
[1]Programa de Ingeniería de Sistemas y Computación (PISC)
Universidad Tecnológica de Pereira, La Julita Pereira, Colombia
[2]Departament de Sistemes Informàtics i Computació (DSIC)
Universitat Politècnica de València, Camí de Vera s/n 46022
València, España. Teléfono:+34-96 387 7734
e-mails: {einsfran, opastor, cmeneses, juamoud}@dsic.upv.es

Article received on December 2, 1999 ; accepted on February 21, 2001

## Abstract

*In order to represent the static and dynamic aspects of a system, according to the real world modeling necessities in a suitable way, it is important to include a temporal expressivity whose specification allows to obtain an application that is functionally equivalent to the conceptual model.*

*In the context of the IDEAS (Ingeniería De Ambientes Software - Software Environments Engineering) project*[1]*, and in particular in its task titled "Specification methods for transactions in phase of conceptual modeling in OO environments", the research group DSIC-UPV (Valencia, Spain) works in the definition of a automatic production software environment based on temporal conceptual models. The work is oriented to the specification and design of transactions from its methodological OO-Method approach. The objective of this article is to present how the properties that allow to capture the temporal expressivity of object oriented conceptual models are adequately included in the OO-Method preserving the automatic generation of the corresponding final software product.*

**Keywords:** Object oriented, temporal expressivity, requirements analysis, automatic code generation, conceptual modeling.

## 1 Introduction

The software development classic life cycle offers a sequential approach starting from the most abstract aspects to the most concrete (system engineering, analysis, design, codification, test and maintenance). For the last stages (design, codification…) developers could use automatic tools, which help assure the quality of the resulting product. The same does not occur with the previous stages, which usually are error and inefficiency source, affecting the quality and the costs and for these reducing the productivity of the developed product.

Nowadays, efforts to introduce tools based on formal methods are more notable. These efforts heighten a new discipline (Requirement Engineering) that bases the requirements specification on conceptual models from the information that is relevant for the application that wants to be represented.

In the Object Oriented modeling and design a form of thinking is underlayed in which problems are solved from models based on real world concepts. This paradigm is considered, by its expressivity, the most adequate approach for the process of conceptual modeling. In this process, it is possible to represent all the aspects, not only static and dynamic, but also its temporal evolution in the application domain, without considering implementation aspects.

The potentiality of the temporary information is perceived when the possibility of consulting historic data is verified (past and present states of the system that allow to analyze and to plan future situations for the decision making). The

specification of temporal aspects in information systems makes possible the handling of temporized data and tasks that can have temporal preconditions associated to their execution or can present temporal interactions with others for the control of their execution. In addition, it is not possible to stop considering the importance of the correct representation of the temporal integrity conditions in the future database.

In order to treat the problem of how to correctly specify that temporary information, diverse models of temporal data have appeared (Edelweiss *et al.*, 1994; Özsoyoglu and Snodgrass, 1995; Tansel *et al.*, 1993). Most of them constituting temporal extensions of already existing models. Within these we can emphasize the ones based on relational models (Clifford and Crocker 1987; Gadia, 1988; Lorentzos and Johnson, 1988; Navathe and Ahmed, 1989; Sarda, 1990; Snodgrass, 1987; Tansel, 1986), E-R (Elmasri and Kouramajian, 1992; Elmasri *et al.*, 1993; Loucopoulos *et al.*, 1991; Tauzovich, 1991; Lee and Elmasri, 1998) and the ones based on object-oriented models (Edelweiss *et al.*, 1993; Kafer and Schoning, 1992; Rose and Segev, 1991; Su and Chen, 1991; Wuu, 1993). Recently, important efforts in the area of temporal modeling have been made on the definition of the temporal consulting language TSQL2 (Snodgrass, 1995), which tries to be standard for the manipulation of temporal relational databases. In any case, works dealing with the specification of temporal properties (problem space) along with implementation aspects (solution space) in an unified framework are not frequent.

In that context, the objective of this article is clear: to enrich an automatic code generation environment, which is based on the object-oriented model, with primitives that allow to specify adequately the temporal information commented previously. To do it, the software production method chosen is the OO-Method, considering that the presented ideas are applicable to any object-oriented software production method based on, for example, a standard like UML. In addition, this enrichment includes the conversion mechanisms of the temporal specification primitives, introduced in the model, in its corresponding software representations. In this way, we preserve the model-based code generation facilities that are offered by proposals like the OO-Method.

According to the previous, the article structure is as follows: section 2 presents a brief introduction of the OO-Method proposal and the formal specification language OASIS that is the basis of the OO-Method. Section 3 shows the representation mechanisms of the temporal properties in conceptual modeling (problem space). Section 4 indicates the implementation of the temporal expressivity in the development environments used by the code generators implemented in the OO-Method (solution space). The final section presents some conclusions and the further work.

# 2 OO-Method

The OO-Method (Pastor, 1992; Pastor *et al.*, 1996; Pastor *et al.*, 1997) is an OO methodology for automatic software production based on formal specification techniques and the use of graphical models similar to those employed by conventional methodologies (OMT, Booch...) or UML. The OO-Method offers a rigorous framework for the specification of information systems that include a powerful and simple graphical notation to deal with the analysis phase, ***OASIS*** (Pastor and Ramos, 1995) as an OO formal specification language that constitutes the high-level repository of the system, and in addition, the definition of a precise *execution model* that guides the implementation phase.

In the software development process two main phases are defined:

1. *Conceptual Modeling.* The graphical analysis models (objects, dynamic and functional) collect the relevant properties that define the system to be developed without considering the implementation. With this description, a formal specification of the OO system in OASIS is generated automatically, that constitutes a complete and formal high-level repository of the system.
2. The application of an *Execution Model.* The formal specification is the source of the *execution model* that is defined in a precise way. This model determines in an automatic way the system-dependent characteristics of the implementation (user interface, access control, service activation, information consulting and manipulation, etc).

## 2.1 OASIS: An Object Oriented Formal Model

The OO-Method has been created on the formal basis of OASIS, a formal OO specification language for information systems. In an OASIS specification, a class is a pattern, or set of properties, that all of its instances share. This pattern has a name and allows the declaration of an identification mechanism. The signature of the class includes attributes, events and a set of formulas of dynamic logic that allow to describe the rest of their properties: *static and dynamic integrity restrictions*, *evaluations* that specify how the attributes change due to the occurrence of events, *derivations* that specify the value of a derived attribute based on others, preconditions that establish conditions that must be satisfied to activate a service and *triggers* that cause the spontaneous activation of a service as a result of the satisfaction of a condition.

In addition, an object can be defined as an *observable process*, reason why the class specification is enriched with a

basic algebra processes, that allows to declare the possible lives of an object as terms of this algebra, and whose elements are events and transactions combined with alternative and sequence operators.

## 2.2 Conceptual Model

The analysis phase begins with the construction of three models (Objects, Dynamic and Functional) that gather the relevant properties (determined by the different specification sections in OASIS) of an information system and describe the society of objects from three complementary points of view and within a rigorous object oriented framework.

The ***Object Model*** is described graphically by a Classes Diagram (DC) that shows the structure of all the classes identified in the problem domain as well as its relationships. A class is represented graphically as a box divided in sections where information on its attributes and services is gathered. The services are declared specifying their name and arguments, distinguishing between the transactions and the creation events, deletion and the shared events (Pastor *et al.*, 1996). For managing complexity, structural relationships can be defined in terms of aggregation (part-of) and inheritance (it-is).

The aggregation relationship between classes includes information on cardinalities (minimum and maximum), which determine how many component objects can be related to a compound object and inversely. In addition, an aggregation must be indicated as inclusive/referential or static/dynamic. A detailed explanation of these properties can be found in Pastor *et al.* (1996).

The ***Dynamic Model*** specifies the aspects related to the control, possible lives, sequence of services and interaction between objects. It is represented using two types of diagrams: State Transition Diagrams (STD) and Object Interaction Diagrams (OID).

The *State Transition Diagrams* (STD) describe the object behavior establishing *possible lives*. A *valid life* is a correct sequence of states that characterize a valid behavior for all the objects of a class. The states denote situations in which the objects can be found because of the occurrence of relevant services. The transitions represent the changes of valid states. The use of conditions allows to eliminate the indeterminism and creates restrictions in the state transitions.

The *Object Interaction Diagrams (OID)* graphically model the interaction between objects and specify two types of interactions. *Triggers*, which are services of a class that are activated in an automatic way when a condition in an object is satisfied, and the *global interactions*, that are *transactions* composed by services of different objects.

The ***Functional Model*** captures the semantic associated with the changes of state in an object as a result of the occurrence of an event. The value of each attribute can be modified depending on the action that took place, the arguments of the event and/or the present state of the object.

The specification of the change of state is determined by the *categorization of attributes* (Pastor *et al.*, 1997) between a predefined set of categories, which indicate what information is needed to determine how the attribute value changes before the occurrence of certain events.

## 2.3 The Execution Model

The *execution model* is an abstract behavior pattern applicable to any conceptual model constructed following the OO-Method proposed strategy. The idea consists of giving a vision of the model that will directly determine the programming pattern to follow when a developer (or a CASE tool) arrives to the implementation phase and it defines the final software product characteristics in terms of the user access control, service activation, user interface, etc. This model has three essential phases:

- *Access control:* in the first place, the object that wishes to connect to the system should identify itself as a member of the society of objects.
- *System vision:* once a user has connected (one object of the system), it will have a clear vision of the society of objects in terms of what classes of objects can be seen, which services it can activate and which attributes it can consult.
- *Service activation:* finally, the object should be able to activate any available service and make the pertinent observations.

Any ***service request*** is characterized by the following sequence of actions: identification of the server object, introduction of arguments of the service to be activated, validation of the transition between states, satisfaction of preconditions, accomplishment of the evaluations (modification of the object state), verification of the integrity restrictions in the new state and verification of the trigger relationships. These steps, clearly defined in Pastor *et al.* (1997), guide the implementation process assuring the functional equivalence between the system description gathered in the conceptual model and its representation in a programming environment.

# 3 Problem Space. Temporality in the OO-Method Conceptual Models

**Temporal marks.** There can be four different forms of marks (*temporal primitives*) for the temporal specification (Jensen, 1994):

- *Temporal moment.* It represents a point in time. For example, the moment in which a soccer game begins. In this case, only a temporal data is registered.
- *Temporal interval.* It is defined by two temporal moments, indicating the beginning and the end of the interval. For example, the intervals of the last contract of an employee.
- *Temporal element.* It is constituted by the finite union of the temporal intervals. For example, the different intervals in time in which an employee has been contracted for the company.
- *Temporal duration.* It corresponds to a time period without considering the moments in which it begins and ends. It can be of two types, depending on the context in which it is defined: fixed and variable. A fixed duration is independent of the context of its definition (for example, one hour always corresponds to 60 minutes) and a variable duration depends on the context (for example, a month can have from 28 to 31 days) (Edelweiss, 1988).

In the model presented in this document, because of the easier representation and to facilitate queries, a *temporal interval* specification is used in the historic objects (that allows to use relational databases that fulfill the 3NF– third normal form -; and that in addition, provides explicit information supporting the notion of *temporal element* using a finite set of moments). The agent relationship is the exception where to model a change of state that occurs in a moment of time, the *temporal moment* is used (it provides implicit information).

The kind of *temporal mark* used for the specification of a *temporal moment* depends on the system environment, the implementation decision and the *granularity* (sequential internal number, date and hour with its possible combinations of year, month, day, hours, minutes, seconds, hundredth of a second, etc.) offered in the data model used.

In reference to the *meaning of the temporal frame*, the following can be used:

- *Transaction time.* The time in which the information was introduced in the database that implements the application.
- *Validation time.* The time in which the information is valid in the reality modeled (corresponds to the existing time or life of an object in the system).
- *Both.*

The *validation time* is used in the OO-Method, because it is relevant in the phase of the conceptual model, in which relative aspects of the physical implementation do not have to be considered (except again in the temporized agents relationships that use the transaction time to precise the moment at which the event was activated).

**Specification Levels.** The specification of the temporal information in an object-oriented model can be done under five different levels: attributes, objects, aggregation relationships, inheritance relationships and agents. In an application, the values taken by the data under these levels of specification can change with time. When it is necessary to represent these changes and to conserve the historical information, a representation that indicates a temporal behavior is introduced.

It is important to remark that in an object, not all the attributes are temporized, for this reason it must be specified in the conceptual modeling which ones they are. In this way would be possible to know when and how one temporized attribute has changed. For objects, temporal information about when it was created can be required, during what periods of time it existed (it was active) for the system, when it stopped existing, etc. For the aggregation relationships there can be variations of their cardinality in time or variations in their value; the management of these temporized relationships well be considered as object-valued attributes. For the inheritance relationships, some objects can inherit properties during different intervals in time. For the agents, who act as service actuators, it must be possible to know at what instant of time an agent has activated a service, as well as when and which services an agent executed or when and which agents executed a service.

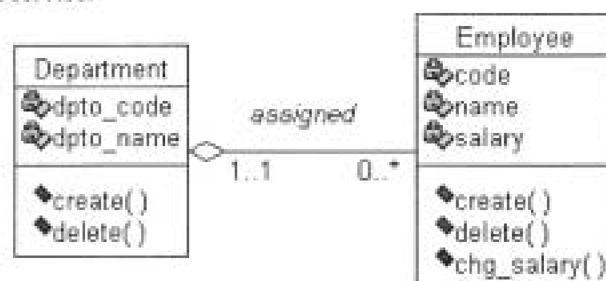



Figure 3.1 Object Model Example

The importance of including the temporality specification at conceptual level is because a great part of the systems requirements involve the handling of temporal and historical information with which it allows the analyst to specify, for example, that some attributes of a class, a class, an aggregation, inheritance or agent relation, have associated a *historical register* of all of their variations in time. Next, we will show how to capture in the OO-Method the temporal expressivity in each one of the five determined levels of specification.

## 3.1 Classes

A *temporized class* is that class in which it is interesting to conserve the historical information of the different temporal

intervals in which each object of the class has existed. The associated information to *classes of temporized objects* indicates a temporal evolution of the object as a whole – when it was created, when it stops existing, periods of activity suspension, etc.

When an object of a *temporized class* is eliminated, it will not disappear from the database used, but it will close adequately its existence interval. For the example, in Figure 3.1, if a class *employee* is declared as temporized, we will be specifying that one person can appear as an instance of the class *employee* associated to different existence intervals.

| *Existence* | *Code* | *Name* | *Salary* | ... |
|---|---|---|---|---|
| [3,10] ∪ [20,null] | e1 | José Perez | 180.000 [3,6] ∪ 225.000 [7,10] ∪ 270.000 [20,null] | |
| [8,null] | e2 | Manuel Gómez | 175.000 [8,20] ∪ 185.000 [21,null] | |
| [7,35] | e3 | María López | 160.000 [7,20] ∪ 195.000 [21,35] | |
| ... | ... | ... | ... | |

Table 3.1 Temporized class and attribute *salary* of the class *employee*

## 3.2 Attributes

The *temporized attributes* are those attributes whose values can be changed throughout the life of the object, but these sequences of values are necessary to record. Each of the different values that the *temporized attributes* can have throughout its evolution has its corresponding temporal mark. Not all the attributes of an object need to be temporized. It is important that a temporal model allows the *coexistence* of temporized attributes and others that do not have associated temporal information.

Each interval of existence associated to an object of a temporized class can, additionally, have related one or more existence intervals associated to the attribute values that have been declared as temporized. In the previous example, if the attribute *salary* is linked to a temporal dimension, an employee in a given existence interval could have had different salaries (assuming that, as we said before, salary is a temporized attribute) in its different existence period as is shown in Table 3.1.

## 3.3 Aggregation Relationships

In the *temporized aggregation relationship*, a history of the aggregations in time is maintained. In this case, the cardinalities are considered as restrictions referent to a moment in time. The distinction of a *temporized aggregation relationship* allows keeping information on the time intervals in which an object is composed (or forms part) of others. For example, considering the aggregation relationship between the temporized classes *department* and *employee*, the objects of the class *department* are composed of 0 or N objects of the class *employee*, and an object *employee* forms part of an object *department* at a determined moment. An aggregation in the OO-Method can be seen as an object-valued attribute and it is conceptually considered as another temporized attribute for the implementation.

## 3.4 Inheritance Relationships

By means of the *temporized inheritance relationships*, it is possible to conserve a historical register of the intervals existence associated to the roles that have the objects that form part of the specialized class. The temporal model must allow the *coexistence* of some temporized specializations and others that are not. For example, the objects of the class *employee* according to the task carried out they are classified as *operative* or *administrative* (being able to have emergent attributes and services for each specialty); we could be interested in maintaining the historical information of who and when they have occupied administrative positions, but without requiring the historical information of those that have occupied operative positions.

## 3.5 Agent Relationships

Of the *temporal relationships for agents*, who act as service actuators, the specification of the object model must allows to conserve a history of the instants in time in which an agent activates a service or transaction. This historical information corresponds to a "log" that allows the users to control the execution of the system specified. For example, if the agent relationship between the class *employee* and a service *change_salary* is specified as temporal, each time this service is executed by an *employee* a historical register is created indicating the agent and service involved, the instant in time and if the execution was successful. This facilitates to know when an agent executed (or tried to execute) a service, which services it has executed, as well as when and which agents executed a service.

# 4 Solution Space. Processing of the Temporal Properties in the Execution Model

The **specification of the temporal expressivity in the Object Model** of the OO-Method (see Figure 3.2) is represented by

means of a mark (hourglass) located next to the temporal element. In the case of classes, attributes and aggregation relationships the mark is next to the name. For agent relationships, the mark is on the dotted line that specifies this relation. In the case of inheritance, the mark is on the continuous arrow proper of the specialization.

To represent the temporal evolution of the system, the data model used must allow the specification for each one of the previous levels during the phase of conceptual modeling.

The main characteristic of the OO-Method is its capacity to automatically generate the implementation of the system modeled in industrial environments of software production using the abstract execution model presented previously. These applications use a Relational Database Management System (RDBMS) as a data repository, whereas the services declared in the classes are represented as stored procedures, class methods or using programming structures (depending on the development environment used to implement them).

The fact that the OO-Method uses in its production environments the RDBMS as data containers orients us towards how to appropriately and of an automatic way represent that temporal expressivity now introduced in the conceptual models created with the OO-Method. Next, we will show how the processing of the temporal characteristics will be introduced in the execution model and how it is going to affect the automatic code generation in the OO-Method.

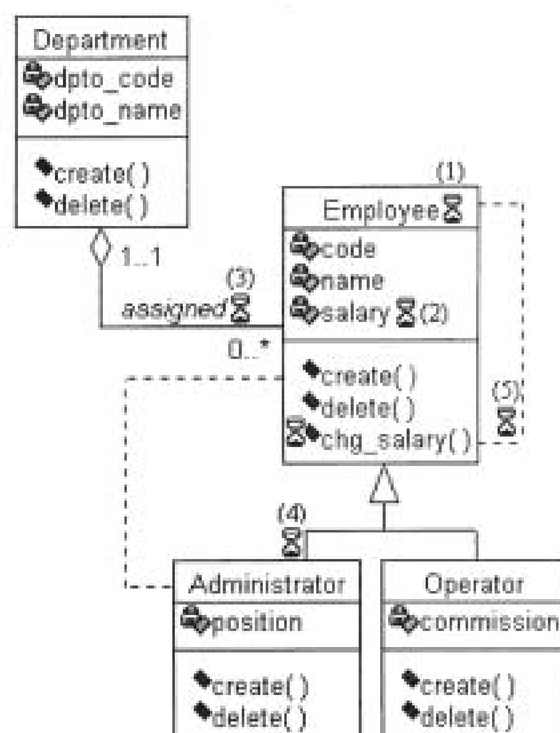



Figure 3.2 Temporality specification in the OO Method: (1) Classes. (2) Attributes. (3) Aggregation Relations. (4) Inheritance. (5) Activator agents of services.

The first decision that needs to be taken is to represent the data of a temporized class in a unique table with the present data or to separate them in a different table. The fact of using a unique table obliges to work on the present data through a *view* in which the attribute that represents the instant of modification is *null* and the handling of a set of historical values must be handled for an instance of a temporal element. In order to avoid this situation, we are going to choose a representation based on different tables for the present and historical information. In addition, we will also avoid problems associated to the normalization for the handling of tables and the processing of missing information that are present in relational implementations.

The following aspects determine how the software representation should be associated in the OO-Method to the temporal characteristics introduced in the conceptual models of the method:

## 4.1 Classes

Each class is represented as a table that is called *base table*. If the class is specified as *temporized*, it will have associated to its base table a new *historical table* that includes the eliminated instances of the base table with the information of the existent temporal interval closed for these eliminated objects, and opened for the present instances in the base.

The **base table** has the following structure:

(oid, {attribute list})

where, *oid* is the object identifier and *attribute list* has the rest of the attributes associated to the base table.

The **historical table** has the following structure:

(oid, begin_time, end_time, {attribute list})

where, *oid* is the object identifier in the base table, *begin_time* references the moment in which the existing temporal interval considered was opened, instantiated when creating the object in the base table. The attribute *end_time* represents the closing moment of this interval, and *attribute list* includes the set of attributes associated to the moment at which this tupla is introduced in the historical table and which are then modified when the existent object interval is closed. This moment will precisely be the one of the *destroy event* occurrence declared in the class.

| *Code* | *Name* | *Salary* |
| --- | --- | --- |
| e1 | José Fernández | 270.000 |
| e2 | Manuel Gómez | 185.000 |
| ... | ... | ... |

Table 4.1 Base table of the class *Employee*

| *Begin_time* | *End_time* | *Code* | *Name* | *Salary* |
|---|---|---|---|---|
| 3 | 10 | E1 | José Fernández | 225.000 |
| 7 | 35 | E3 | María López | 195.000 |
| 8 | Null | E2 | Manuel Gómez | 175.000 |
| 20 | Null | E1 | José Fernandez | 270.000 |
| ... | ... | ... | ... | ... |

Table 4.2 Historical table of the class *Employee*

The primary key in the historical table corresponds to *oid+begin_time*. For each foreign key that references to a historical table of another temporized class, an attribute is added that contains thé "begin time" in existence of this foreign key to relate with the corresponding historical table.

The generation of an instance in the historical table, allows two implementations:

- That the implementation of the *new* service of the implicated class (in base table) be the one in charge of executing the corresponding *insert* in the historical table, and that the implementation of the *destroy* service of the class be the one in charge of executing the corresponding *update* in the same historical table.
- That a *trigger* associated to the insertion and another one to the deletion of tuplas of that base table be defined. This option will be the one chosen if the RDBMS allows the trigger definition of the database; otherwise, the first one should be used.

The attributes derived from a temporized class do not appear in the base table structure, but do in the historical table structure, with the purpose of facilitating the consultations.

For the example of Figure 3.1, the resulting base tables have the following structure:

EMPLOYEE:
(<u>code</u>, name, salary, contribution_3x100, address, fk_code_department

DEPARTMENT:
(<u>code_department</u>, name_department)

The *fk_code_department* attribute of *Employee* is a foreign key and references to *Department*.

With the non-temporal class *Department*, the historical table for *Employee* (temporal) is:

H_EMPLOYEE:
(code, <u>begin_time</u>, end_time, name, salary, Contribution_3x1000, address, fk_code_department).

The *fk_code_department* attribute of the H_Employee table is not a foreign key. It only indicates which was the department code for each employee without guaranteeing that it exists in the Department base table.

The historical database tables possess referential integrity between each other by means of their foreign keys, but they are not related by referencing to the associated base tables because integrity is not guaranteed. The reason is that the objects always exist in the historical tables but not in the base tables, where they are eliminated when its existent interval closes.

If the class *Department* is specified as temporal, then the corresponding historical table *(H_Department)* is created and the attribute *begin_time_department* in the table *H_Employee* is added. The attributes *fk_code_department* and *begin_time_department* are foreign keys referenced to the *H_Department* table.

## 4.2 Attributes

Each attribute that is specified as temporized (problem space) generates in the database (solution space) the creation of a *historical attribute table* associated to the base table of the class in which the attribute is defined and in addition, originates the creation of a *historical class table* (if it does not yet exist) associated to the same base table of the class.

The *historical attribute table* includes instances with the identification of the object, the historical values that the temporal attribute has had for this object in the base table with the existent temporal interval of these values, and also the existent present values for each object in the base table with the existent temporal interval opened.

The structure of the base table of the class is not affected by the inclusion of temporized attributes.

The structure of the **historical attribute table** is:
(oid, begin_oid, begin_attribute, end_attribute, attribute)

where *oid* is the object identifier, *begin_oid* is the instant from which the object is created with that oid, *begin_attribute* references the instant in which the considered existent temporal interval was opened (in which the historical tupla for the attribute was generated), instantiated when the object is created in the base table or when modifying the value of the temporal attribute in the base table. *End_attribute* represents the closing instant of this interval, instantiated each time the attribute value changes or when the object on the base table is eliminated and *attribute* corresponds to the value the attribute takes, associated to the considered interval.

The value the attribute *begin_attribute* is taken when

creating a historical attribute instance, the first time it coincides with the instant in which the object is created in the base table. When modifying the attribute value for the object the existent interval of the previous value closes and a new historical instance is created with the new value. When deleting the object instance in the base table the existent interval of the last value closes for each one of the temporal attributes.

| *Begin_employee* | *Code* | *Begin_salary* | *End_salary* | *Salary* |
|---|---|---|---|---|
| 3 | e1 | 3 | 6 | 180.000 |
| 3 | e1 | 7 | 10 | 225.000 |
| 7 | e3 | 7 | 20 | 160.000 |
| 8 | e2 | 8 | 20 | 175.000 |
| 20 | e1 | 20 | null | 270.000 |
| 7 | e3 | 21 | 35 | 195.000 |
| 8 | e2 | 21 | null | 185.000 |
| .... | ... | ... | ... | ... |

Table 4.3 Historical table for the temporal attribute salary

The primary key in the historical attribute table corresponds to *oid+begin_oid+begin_attribute*. The reference to the base table of the class to which the temporal attribute belongs is made with *oid+begin_oid*.

The generation of an instance in the historical attribute table, allows two possible implementations:

- That the implementation of the *new* service of the class in which the attribute is defined, and also each service that affects the temporal attribute be in charge of executing the corresponding *insert* in the table "historical attribute", and that the implementation of the service *destroy* of the class and each service that affects the temporal attribute will be the one in charge of executing the corresponding *update* (corresponding modification at the interval closing for the attribute value) in the same "historical attribute" table.
- That a *trigger* be defined associated to the object creation and another one to the attribute modification to open the temporal interval of the attribute value. In addition, a trigger must be defined (with which the interval of the value of the attribute in the historical table is closed) for the modification of the attribute and the deletion of the object. This option will be the chosen one if the RDBMS allows the definition of triggers of the database; otherwise, the first one must be used.

The variable attributes are the only ones specified as temporal. A temporal attribute can take a null value, which is stored historically as any other data.

In order to guarantee the referential integrity, when an attribute is specified as temporal, the class in which is defined is automatically treated as temporal (although it has not specified as so). In the example, if the attribute *Salaray* is specified as temporal, the *historical table* is created:

H-EMPLOYEE_SALARY (code, begin, employee, begin_salary), end_salary, salary)and in addition, the class employee is treated as temporal (although it war not been market as so).

## 4.3 Aggregation Relationships

The aggregation is implemented as a foreign key restriction (that can be formed by more than one attribute) between the corresponding *base tables* to the related classes. For M:M relationships, the foreign keys are located in the generated MM table. For relationships 1:1 or 1:M, the cardinality of the relation will determine where to locate this foreign key (in the compound or in the component).

When specifying a *relation of temporized aggregation*, a *historical table of aggregation* is created, that will gather the information on the time intervals in which the relation is maintained. The *historical table of aggregation* includes instances with the identification of the two objects of the related classes and the existent temporal interval of this relation.

The structure of the base tables of the related classes is not affected by the inclusion of the temporized aggregation relations. The structure of the **historical aggregation table** is:

(oid1,begin_oid1,oid2,begin_oid2,
begin_*compound_component*,
end_*compound_component)*

where, *oid1* is the object identifier in the compound class, *begin_oid1* is the instant in which the oid1 takes the assigned value, *oid2* is the object identifier in the component class, *begin_oid2* is the instant in which the oid2 takes the assigned value, *begin_compound_component* references the instant in which the aggregation relation was created (it opens the considered existent temporal interval and generates the historical tupla for the aggregation relation), *end_compound_component* represents the instant in which the relation stops existing (the closing of this interval).

In the example of Figure 3.1, if the assigned aggregation relationship is specified as temporized, the historical table must include the intervals (determined by the attributes begin_dept_employee and end_dept_employee) in those that existed (and exist) the allocation relation of each employee in each one of the departments of the company.

| *Code* | *begin_employee* | *code_department* | *begin_depto* | *begin_dept_employee* | *end_dept_employee* |
|---|---|---|---|---|---|
| e1 | 3 | d1 | 2 | 3 | 5 |
| e1 | 3 | d2 | 1 | 6 | 10 |
| e3 | 7 | d2 | 1 | 7 | 35 |
| e2 | 8 | d1 | 2 | 8 | 15 |
| e2 | 8 | d3 | 1 | 16 | Null |
| ... | ... | ... | ... | ... | ... |

Table 4.4 Historical table for the aggregation relation *assigned*

The generation of an instance in the historical aggregationtable. allows two possible implementations:

- That the implementation of the *new* service of the class in which the foreign key is defined and each service that affects this foreign key (of the aggregation relation defined as temporal) is the one in charge of executing the corresponding *insert* in the "historical aggregation" table, and that the implementation of the *destroy* service of the class and each service that affects the foreign key at issue, will be the one in charge of executing the *update* (corresponding modification at the closing of the interval for the aggregation relation) in the same "historical aggregation" table.
- That a *trigger* be defined associated to the object creation and another one to the modification of the foreign key to open the temporal interval of the aggregation relation. In addition, a trigger must be defined (with which this interval is closed) for the modification of the foreign key at issue and the erasure of the object in the base table. This option will be the one chosen, if the RDBMS allows the definition of triggers of the database; on the contrary, the first one must be used.

When an aggregation relation is specified as temporal, to guarantee the referential integrity, the related classes are automatically treated as temporals (although they were not defined as so).

For the example, if the aggregation relation is specified as temporal, then the classes *Employee* and *Department* are automatically treated as temporals, generating the corresponding historical tables.

The primary key of the historical table for an aggregation relation corresponds to the identifiers of the classes related and the initial interval time in which the aggregation relation is valid was added. In the previous example, if the aggregation relation is specified as temporal, the *historical table* is created: H_DEPARTMENT_EMPLOYEE:

(code, begin_employee, code_department,begin_department, begin_department_employee, end_department_employee)

The foreign keys *code+begin_employee* and *code_department+begin_departamento* reference to the historical tables of *employee* and *department* respectively.

## 4.4 Inheritance Relationships

For the temporized inheritance relations, the specialized class is treated as temporal at a conceptual modeling level, and the corresponding historical table will gather the information on the time intervals in which the inheritance relation exists, in the same form we previously presented when studying the representation of temporized classes. This means, an inheritance relation that is specified as temporal, automatically treats the specialized class as temporal (although it has not been defined as so) and it is given the same temporized class processing.

When an inheritance relation is specified as temporal, besides treating the specialized class as temporal, all the classes that are ancestors in the ascending hierarchy line until the root of the inheritance tree are also treated as temporal (although they have not been defined as such). This is done with the purpose of conserving the temporal object information as a whole and to maintain the scheme of the normalized relational database.

For the previous example, if *Employee* is a specialization of the superclass *Person* and if the inheritance relation between the classes *Employee* and its specialization *Administrative* is defined as temporal, then the classes *Person, Employee* and *Administrative* are automatically treated (although they have not been defined as such), and the corresponding historical tables are generated.

## 4.5 Agent Relationships

The temporality specification (problem space) in each *agent* relationship originates in the database (solution space) the creation of a *historical agent table (or "log"),* which contains the historical information by means of which the service execution activities can possibly be known and controlled. As the service execution is considered an atomic process, the temporal mark used is the temporal instant.

In order to store the historical data generated by the temporal agents relations, a historical table ("log") by each agent class who has at least one *temporal agent relation* is generated. The maximum number of tables corresponds to the number of agent classes. The *historical table for an agent relation* includes instances, each one of which contains the identification of the agent object, as well as the name of the class and service executed, the temporal moment of execution, identification information of the affected data and an indication if it could or not execute (and why).

The structure of the base tables of the agent classes is not affected by the inclusion of temporized agent relations.
The structure of the **historical agent table** is:

(oid, begin_oid, instant, name_class, name_service, arguments, result, message)

where o*id* is the agent object identifier, *begin_oid* is the instant in which the oid takes the assigned value, *instant* references to the moment in which the service is executed, *name_class* corresponds to the name of the class that contains

the executed service, *name_service* is the name of the executed service, *arguments* correspond to a "string" with information that allows to identify the data that are parameters affected during the service execution, *result* indicates if it could execute or not the service and *message* corresponds to an error message in case of failure in the service execution.

| *Agent* | *Begin_agent* | *Instant* | *Class* | *Service* | Arguments | Res | Message |
|---|---|---|---|---|---|---|---|
| e1 | 3 | 20 | Employee | Change_salary | Emp=1234 | ok | |
| e2 | 8 | 26 | Department | Create | Dept_code=d5 | fail | error 132 |
| ... | | ... | ... | ... | ... | ... | ... |

Table 4.5 Historical table for agent log

The instant in which an agent activates the execution of a service (whose relation is specified as temporal), it implements itself making the executed service by the agent be in charge of executing the corresponding *insert* to the affected historical agent table(s).

To specify an agent relation who activates a service as temporal, implies that the class agent is treated as temporal (although it has not been specified as so), which allows to maintain the integrity of the foreign key (oid of the agent included in the log table) that references the historical table of the class. For the example of Figure 3.1, if the agent relation between the class *Employee* and the service *create* of the class *Department* is specified as temporal, the historical table of employee (H_EMPLOYEE) and the log table (H_LOG_EMPLOYEE) are created.

The primary key of the historical table for an agent relation corresponds to the identifier of the agent class (*oid+begin_oid*) and the time instant in which the service is executed is added. In addition, references of alien keys are created from each log table, towards their respective historical table of the class. The key *oid+begin_oid* in the log table is foreign and references to the historical table of the class that is a temporal agent.

For a temporal agent relation with a service, besides treating the class agent as temporal, the agent relations between the subclasses (from all the inferior levels in an inheritance hierarchical relation) and the service at issue must automatically be treated as temporals, and therefore, the same subclasses to conserve the referential integrity of the base of the historical data are treated as temporals (although they have not been specified as such). That is to say, if a temporized agent relation exists that activates a service, any object agent that identifies itself with the primary key of one of its subclasses, can activate the same service and this event must be registered historically in a log.

For the same example, if the agent relation between the class *employee* and the service *create* of the class *department* is specified as temporal, besides creating the historical table H_EMPLOYEE and the log table H_LOG_EMPLOYEE, the historical tables of the subclasses *employee(administrative, operative...)*, and their log tables: (H_LOG_ADMINISTRATIVE and H_LOG_OPERATIVE) are created.

# 5 Conclusions and Future Work

It is not habitual to jointly approach the problem of adequately specifying temporal properties (introducing the necessary primitives) and the problem of adequately representing them in the resulting software product of a quality software production process. The contribution of this work is precisely the boarding of those two problems in a unified frame, and within a method of automatic production of software (OO-Method) based on an object oriented model with a solid formal base.

The object model version presented increases the expressivity of the OO-Method adding the specification of this type of temporal properties when defining the classes of the system. In particular, each class (simple or complex), relations (of aggregation, inheritance and agents) between classes, and the variable attributes could be declared as temporized in the line of the previously exposed.

The temporality specification of the *agent* that acts as an actuator of a *service*, enriches the semantics of the system since it allows to have a *historical agent table (or log)* that allows to know and to control the activities of execution of services.

In this work, a version of the OO-Method is presented that includes the specification of temporal properties in the phase of the conceptual modeling, with its corresponding software representation according to the automatic programming paradigm associated to the OO-Method. With this, works made in the context of specification of temporal properties to conceptual models are complemented, with works that based on the object-oriented model providing environments of automatic code generation.

Adding the temporal expressivity to the OO-Method CASE tool, has allowed obtaining as a result a historical relational database with all the information that the system analyst determines necessary to conserve from the application object model.

From the inclusion of the temporal expressivity, the OO-Method is appears as a methodology for the automatic software production that allows to specify a much more ample

range of information systems, including those in where the requirements imply a handling of temporal information, with historical data and execution control. In fact, the method extended with the temporal expressivity is being used successfully in the resolution with the OO-Method of real Information Systems, in the environment of diverse projects of I+D made with local companies dedicated to the software production.

This effort made has as a future projection of being able to use temporal logic in preconditions and restrictions, proposing the specification of temporal properties in the environment of the dynamic and functional models. In the case of the dynamic model, it would allow to specify temporized preconditions (that is to say, that are only valid in determined periods of the life of the object). Applied to the functional model, temporized évaluations could be specified analogically.

The temporal expressivity in the OO-Method presented in this work can be projected towards the development of a temporal query language oriented to the handling of the historical information registered in the temporal relational database. It is hoped from this temporal query language, not only to have access to the information for historical consultations, but also to look for a mechanism that allows to establish queries where actual data is mixed with historical information.

The use of a temporal query language allows to know all of the past and present states of an application, from which future states can be projected in order to facilitate managers to take decisions diminishing the projected level of uncertainty presented by the incapacity to know historical information.

# References


**Antunes, D.C.; Heuser C.A.; Edelweiss N.** *Modelagem temporal de transacoes em banco de datos.* Actas de las I Jornadas Iberoamericanas de Ingenieria de Requisitos y Ambientes de Software, IDEAS 98, Torres (Brasil), Abril 1998.

**Bergamaschi, S.; Sartori, C.** *Chrono: A conceptual Design Framework for Temporal Entities.* Italy, 1998.

**Clifford, J.; Crocker, A.** *The Historical relational data model (HRDM) and algebra based on lifespans.* Proceedings of the 3rd International Conference on Data Engineering, Feb. 1987, Los Angeles, California. p.528-537..

**Edelweiss, N.; Oliveira, J. Palazzo M.; Pernici, B.** *An Object-Oriented Temporal Model.* Proceedings of the 5th International Conference on Advanced Information System Engineering - CAISE'93, June 8-11, 1993, Paris. Heidelberg: Springer-Verlag, 1993. p.397-415. (Lecture Notes in Computer Science 685)..

**Edelweiss, N.; Oliveira, J. Palazzo M.** *Modelagem de Aspectos Temporais de Sistemas de Informação.* Recife: Universidade Federal de Pernambuco, 1994. Livro texto da 9a Escola de Computação, 1994, Recife..

**Edelweiss, N.** *Bancos de Dados Temporais: Teoria e Prática.* Report UFMG. P 225-282, Joao Paulo Kitajima 1998

**Elmasri, R.; Kouramajian, V.** *A temporal query language based on conceptual entities and roles.* Proceedings of the 11th International Conference on the Entity Relationship Approach, 1992, Karlsruhe, Germany. Berlin: Springer Verlag, 1992. p.375-388. (Lecture Notes in Computer Science, v.645).

**Elmasri, R.; Wuu, G. T. J.; Kouramajian, V.** *A temporal model and query language for EER Databases.* In: TANSEL, A. et al. (Eds.). Temporal databases: theory, design and implementation. Redwood City: The Benjamin/ Cummings Publishing, 1993. p. 212-229.

**Gadia, S.** *A homogeneous relational model and query language for temporal databases.* ACM Transactions on Database Systems, New York, v.13, n.4, p.418-448, Dec. 1988.

**Heuser, C.; Antunes, D.** *Conceptual Modeling of Transactions Combined with Temporal ER Diagrams.* Report Interno UFRGS – Instituto de Informática, Porto Alegre RS (Brasil), 1997.

**Jensen , C. S. (Ed).** *A consensus glossary of temporal database concepts.* ACM SIGMOD Record, New York, v.23, n.1, p. 52-64, Mar 94.

**Kafer, W.; Schoning, H.** *Realizing a temporal complex-object data model.* Proceedings of the ACM SIGMOD International Conference on Management of Data, June 2-5, 1992, San Diego. p.266-275.

**Lee, J. Y.; Elmasri, R. A.** *An EER-Based Conceptual Model and Query Language for Time-Series Data.* U.S.A. 1998

**Lorentzos, N.A.; Johnson, R.G.** *Extending relational algebra to manipulate temporal data. Information Systems.* v.13, n.3, p.289-296, 1988.

**Loucopoulos, P.; Theodoulidis, C.; Wangler, B.** *The entity*

*relationship time model and conceptual rule language.* Proceedings of the 10th International Conference on the Entity Relationship Approach, 1991, San Mateo, California.

**Navathe, B.; Ahmed, R.** *A Temporal relational model and a query language.* Information Sciences, v.49, p. 147-175, 1889.

**Özsoyoglu, G.; Snodgrass, R. T.** *Temporal and real-time databases: a survey.* IEEE Transactions on Knowledge and Data Engineering, New York, v.7, n.4, p.513-532, Aug.1995.

**Pastor, O.** *OO-Method: An Object Oriented Methodology for Software Production.* Actas de DEXA 92, Springer-Verlag, pp 121-127. ISBN: 3-211-82400-6. 1992

**Pastor, O.; Ramos, I.** *OASIS 2.1.1.: A class-Definition Language to Model Information Systems Using an Object Oriented Approach.* February 94 (1 ed.), Mars 95 (2 ed.), Oct 95 (3 ed.). 1995.

**Pastor, O.; Pelechano, V; Bonet, B.; Ramos, I.** *An OO Methodological Approach for Making Automated Prototyping Feasible.* Proc. DEXA 96. Springer-Verlag. September 1996.

**Pastor, O.; Insfrán, E.; Pelechano, V.; Romero, J.; Merseguer, J.** *OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods.* CAiSE97. June 1997

**Pelechano, V.; Insfrán, E.; Pastor, O.** *El Metamodelo OO-Method y su Repositorio Relacional.* UPV, 1998.

**Rose, E.; Segev, A.** *TOODM: A Temporal object-oriented data model with temporal contraints.* Proceedings of the 10th International Conference on the Entity Relationship Approach, Oct. 1991, San Mateo, California.

**Sarda,N.L.** *Extensions to SQL for historical databases.* IEEE Transaction on Knowledge and Data Engineering, v.2, n.2, p. 220-230, June 1990.

**Snodgrass, R.** *The Temporal query language TQuel.* ACM Transactions on Database Systems, New York, v.12, n.2, p.247-298, June 1987.

**Snodgrass, R. T. (Ed.)** *The TSQL2 Temporal Query Language.* Norwell: Kluwer Academic Publishers, 1995. 674p.

**Su, S.Y.W.; Chen, H.-H. M.** *A Temporal knowledge representation model OSAM*/T and its query language OQL/T.* Proceedings of the 17th International Conference on Very Large Data Bases, Sept. 1991, Barcelona. p.431-442.

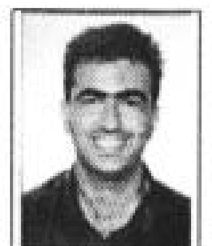
**Tansel, A.U.** *Adding time dimension to relational model and extending relational algebra.* Information Systems, v.11, n.4, p.343-355, 1986.

**Tansel, A. *et al.* (Eds.).** *Temporal Databases: Theory, Design and Implementation.* Redwood City: The Benjamin/Cummings Publishing, 1993.

**Tauzovich, B.** *Towards temporal extensions to the entity-relationship model.* Proceedings fo the 10th International Conference on the Entity Relationship Approach, 1991, San Mateo, California.

**Wuu, G. T. J.; Dayal, U.** *A uniform model for temporal and versioned object-oriented databases.* In: TANSEL, A. *et al.* (Eds.). Temporal databases: theory, design and implementation. Redwood City: The Benjamin/Cummings Publishing, 1993. p.230-247.

***Carlos Meneses*** *is an invited researcher in the Department of Information Systems and Computation (DISC) at the Valencia University of Technology, Spain. His research interests are object orientation, temporal conceptual modeling, requirements engineering and specification languages. He received a degree in Computer Science from the Pereira University of Technology in Colombia.*



***Oscar Pastor*** *is currently the Head of the Dept of Computation and Information Systems Department, Valencia University of Technology (Spain), and the leader of the Research Group on Object-Oriented Methods of Software Production in the same Dept. He received his phD degree from the Valencia University of Technology in 1992, after a research stay in HP Labs, Bristol, UK. He is currently Professor of Software Engineering at the Valencia University of Technology. His research activities has been involved with object-oriented conceptual modelling, requirements engineering, information systems, web-oriented software technology and model-based code generation. Author of over 100 research papers in conference proceedings, journals and books, he has received numerous research grants from public institutions and private industry, and devoted considerable effort to issues of technology transfer from academia to industry.*

***Juan Carlos Molina*** *is working at CARE Technologies S.A. in Denia, Spain. He is the head of the Automatic Code Generation Technologies Department. He is also a Ph.D student in the Information Systems and Computation Department (DISC) at the Valencia University of Technology. His research interests are object orientation, automatic code generation, conceptual modeling, requirements engineering, component-based software systems and specification languages. He received a degree in Computer Science from the Valencia University of Technology, and is a Member of the Logic Programming & Software Engineering Research Group at the DISC.*

***Emilio Insfrán*** *is an Assistant Professor in the Department of Information Systems and Computation (DISC) at the Valencia University of Technology, Spain. His research interests are OO methodologies, conceptual modeling, requirements engineering, specification languages and databases. He received a degree in Computer Science from the National University of Asunción, Paraguay, a MS degree in Computer Science from the Cantabria University, Spain. He is a Member of the Logic Programming & Software Engineering Research Group at the DISC and of IEEE Computer Society.*