AN OBJECT-ORIENTED TEMPORAL MODEL

Nina Edelweiss*
José Palazzo M. de Oliveira*
Barbara Pernici+,

* Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 - Bloco IV - Agronomia - Caixa Postal 15064
CEP 91501-970 - Porto Alegre - RS - Brazil
Fax +55(51)336.5576
e-mail: [nina,palazzo]@inf.ufrgs.br

+ Università di Udine and Politecnico di Milano
Italy
e-mail: pernici@uduniv.cineca.it

Abstract. The representation of complex objects and behaviors (state transitions) in information systems is a central issue in software engineering. In an information system, supported by a conventional database, the only available state is the set of present values. The object's behavior is controlled by integrity constrains defining only the valid states. Almost all the representation of the dynamic evolution is lost in the traditional modeling process. In this paper the main concern is to present the extensions made on an Object-Oriented Model, the F-ORM model [11] to expand the representation of the object's evolution and to support the temporal aspects involved. Temporal object-oriented models can be used to specify behavioral requirements of information systems. Four different modeling concepts are defined to represent temporal information: (i) a set of temporal data types and their associated functions, to be used in properties' definitions; (ii) time stamps associated to instances and to dynamic properties; (iii) a special *null* value for attribute values outside the validity period; and (iv) temporal conditions added to rules, written in a temporal logic language. An example showing the use of the Temporal F-ORM is developed.

1. INTRODUCTION

Information systems specification can be done through the use of data models. In the framework of object-orientation much work has been done on the modeling of static and behavioral properties, but temporal aspects have not been explored in profundity. One isolated example is the language RML [14]. Some studies about these aspects are available [4, 8].

Object-oriented models should provide definition of time properties when intended to be used in time critical systems, e.g., plant control systems, office information systems. Our basic concern is the modeling of *information systems*. Time aspects are necessary to represent objects dynamic evolution within these information system environments.

Temporal properties are used to define properties (attributes) with time values, to time stamp properties in temporal databases, order activities' execution through rules, etc.

Recent works present different forms of time modeling. Time definitions may be done explicitly, usually through timestamping (a time value associated with some object, e.g., an attribute value or a tuple) or implicitly, using some sort of temporal logic language.

The explicit time representation requires the choice of a primitive temporal element, like events (isolated instants of time) or intervals (the time between two events). When using *events* as the primitive temporal notion, there is a special time point corresponding to the *current time,* constantly moving along the temporal axis. The time concept may be represented as a continuous or a discrete variable. Events belong to the continuous time representation. An event is an isolated instant of time. It is said to occur at time *t* if it occurs at any time during the chronon represented by *t* [15]. A chronon is the shortest duration of time supported by a temporal DBMS; it belongs to the discreet time representation. Many applications show the need of defining different granularity for information: hours, days, years. This makes the retrieving of the temporal information a complex affair but produces a much better representation of reality. Other time domains, such as time intervals, may be defined as pairs of events, representing the lower and the upper end of the time interval.

In systems where reasoning on time duration is central, such as scheduling systems, the notion of *time interval* is a primitive and events are represented by very little intervals. One important approach for time modeling is Allen's interval algebra [3], were time intervals are related to each other by temporal relations, represented by predicates expressed in a temporal logic language. The languages TELOS [21] and RML [14] are based on this theory.

The use of temporal logic is also found in some systems and languages [5, 10, 14, 18, 19, 23]. In the Event Calculus [16], reasoning about events and time is performed within a logical programming framework. The most important contribution of this approach is the possibility of dealing with uncertain and imprecise information like *before* and *after*.

This papers main concern is to extend an object-oriented model, the F-ORM model [11], so that it supports temporal aspects. The chosen primitive temporal element is the event. We add four different modeling concepts for temporal definitions: (i) a set of temporal data types and their associated functions and operations, to be used in properties' definitions; (ii) time stamps associated to instances of objects in the database and to dynamic properties; (iii) a special *null* value for attribute values outside the validity period; and (iv) temporal conditions added to rules, written in a temporal logic language.

Snodgrass and Ahn proposed a taxonomy of time in databases [25], consisting of three distinct time concepts: (i) transaction time, the update time; (ii) valid time, the period of validity of the stored information and (iii) user-defined time, temporal properties defined explicitly on a time domain and manipulated by the user program. With the definition of the set of temporal data types, the need of user-defined temporal properties decreases. Dynamic properties timestamping and the *null* value represents the transaction and valid times. Temporal conditions added to static and dynamic integrity rules constrain the set of possible state transitions of the application.

The paper is organized as follows. A small application case is presented in Section 2, to be used in examples in the other sections. Section 3 describes briefly the main aspects of the F-ORM model [11,22]. The temporal requirements needed to specify information systems are listed in Section 4. In Section 5, special temporal data types and the corresponding functions and operations are presented. Section 6 describes the representation of transaction times and valid times and Section 7 introduces briefly the adopted temporal logic language.

2. AN APPLICATION EXAMPLE

This paper uses a part of a Video Rental Store specification in the examples: the information on clients, employees, tapes and rentals. A *client* is identified by a unique code, a name and an address. Additional information may be necessary, like his or her inscription date in the Video Rental Store, all the tapes he or she rented and the corresponding periods, and if he or she is allowed to rent tapes. An *employee* is identified by the name and has the properties: address, the hiring date and salary. The *tapes* are identified by a unique tape code. Each tape has the following information: the movie name, the category and the date of acquisition. A *rental* is made associating the client's and the tape's codes, and a starting date. A rental is only possible if some conditions are satisfied: the tape must be in the shop, the client must be allowed to rent tapes, can have a maximum of 5 tapes and is not allowed to keep up a tape for more than 30 days.

3. THE F-ORM MODEL

The F-ORM model (Functionality in Object with Roles Model) [11] is an object-oriented design framework for information systems requirements' specification[1]. Objects' behavior is described using the concept of roles. Two distinct types of classes are identified: *resource classes* and *process classes*. A resource class defines the structure of the resources (agents, data and documents) in terms of roles that the resource can have in its life-cycle. Process classes integrate the resource classes allowing to describe how the work is actually performed in its organization and in the cooperation among agents. The concept of role in process classes models the different tasks executed in the process and their relationships in terms of communication and cooperation rules together with the involved resources.

A class is defined by a name c_n and a set of roles R_i, each one representing a different behavior of this object:

$$\text{class} = (c_n, R_0, R_1, \ldots, R_n)$$

Each role R_i consists of a *role name* Rn_i, a set of *properties* P_i of that role (abstract descriptions of data types implemented as instance variables), a set of *abstract states* S_i that the object can be at while playing this role, a set of *messages* M_i that the object can

[1] Project INFOKIT, Italian National Research Council and Project ITHACA, ESPRIT II.

receive and send in this role, and a set of *rules* Ru_i (the state transition rules and integrity rules):

$$R_i = < Rn_i, P_i, S_i, M_i, Ru_i >$$

All instances of roles evolve independently, the interactions being allowed through message passing. An object can play different roles at different times, can play more than one role at the same time, and can have more than one instance of the same role at the same time.

Every object has a base role R_0 that describes the initial characteristics of an instance and the global properties concerning its evolution. The properties of the base role are inherited by the remaining roles; the messages are used to add, delete suspend and resume instances of other roles; the possible states are pre-defined, *active* and *suspended*; and the rules define transitions between roles and global constraints for the class. Properties' definitions describe the domain each property should have.

A class can be defined as a subclass of one or of several classes (multiple inheritance). A subclass inherits all components specified in the parent class or classes. New components can be added to a subclass definition in two ways: (i) adding specifications of new roles and (ii) modifying the specification of inherited roles.

Considering the class *tape* of the proposed application case, the properties of the base-role are the tape's code, the name of the film and the type of the movie (e.g., drama, comedy). The following roles can be identified: (i) *Life-time*, modeling the actions to be executed to buy the tape, let the tape available for rental during a period of time and sell it afterwards; (ii) *Rentals*, modeling the possible rentals of a tape; and (iii) *Tape_loss*, modeling what shall be done when a tape is lost. Considering the role *Rentals*, some required properties are the client's code and the rental starting and ending dates. Possible states in this role are *available* and *rented*. This role can receive and send the following messages: *rental* from Rental_control, *tape_devolution* from Rental_control and *rented_time* to Rental_control. State transition rules control the behavior. Representing incoming messages by the prefix "←" and outgoing messages by "→ ", we have the following class definition:

```
resource class (
  TAPE,
  < base_role,
   properties =        {       (tape_number, INTEGER),
                               (tape_film, STRING),
                               (film_type, STRING)      },
   messages =          { ... },
   states =     { ... },
   rules =      { ... },
  >,

< Life_time,
    ...
  >,
< Rentals,
properties = {      (client_code, INTEGER),
                    (beginning_date, DATE),
                    (end_date, DATE)          },
messages = {        rental (Tape:INTEGER, Client:INTEGER) from
                            Rental_control,
                    tape_devolution (Tape:INTEGER) from Rental_control,
                    rented_time (Time:INTEGER,Client:INTEGER)) to
                            Rental_control },
states = { available, rented },
rules = {  rule1 : msg(←add_role) ⇒ state(available),
           rule2 : state(available), msg(←rental(T,C)) ⇒ state(rented),
           rule3 : state(rented), msg(←tape_devolution(T)) ⇒
                    msg(→ rented_time(T,C)), state(available) }
>,

< Tape_loss,
  ...
> )
```

4. TEMPORAL REQUIREMENTS FOR INFORMATION SYSTEMS

Temporal aspects are important in information systems not only to represent temporal information to be introduced in the corresponding database but also to model the interaction of the possible processes to be executed. Temporal F-ORM, an extension of the F-ORM model, was created to permit temporal modeling. Analyzing the domain of information systems, two different types of temporal requirements can be identified: unconditional and conditional temporal information about events.

4.1. UNCONDITIONAL TEMPORAL REQUIREMENTS

Unconditional temporal requirements are explicitly defined, representing a specific moment of time associated to an information. These requirements can be well-defined and incompletely defined. *Well-defined static requirements* are of three different types:

- registration of a temporal element associated to an event, like the birthday of a person or the hour of a meeting;
- the duration of an event, like the duration of a meeting;
- the period of time during which a value is valid, as the case of an exchange rate that is valid during a certain period.

These requirements are represented in Temporal F-ORM through appropriate temporal data types, presented in Section 5. The valid times of an information, analyzed in Section 6, also represent well-defined static temporal requirements.

In many applications there is also the need to represent *incomplete information*:

- the occurrence of an event *before* or *after* a date, as the case of the Individual Income Tax Return form that must be posted before the end of May;

- the occurrence of an event *within* an interval, like the inscription of a student in a course that must be done during the first week of September.

Incomplete information is also defined through some specific data types, as will be explained in Section 5.4.

4.2. CONDITIONAL TEMPORAL REQUIREMENTS

Two types of conditional temporal requirements can be identified:

- conditional temporal requirements that represent casual constraints on the possible execution of processes, like the case of a process that is executed when an event occurs, or before (or after) the occurrence of another event; these requirements control the beginning and coordinate the execution of concurrent processes through constraints on the temporal order in which messages can be sent and received by instances of roles;

- another form of conditional temporal restrictions is used to represent implicitly or explicitly defined temporal information, relative to other information; as an example, there can be a restriction that controls the fact that a new salary is never less than a previous one.

Conditional temporal information is represented through the use of logic conditions associated to the transition and integrity rules, as explained in Section 7. These conditions modeling temporal requirements represent constraints to the information systems evolution.

5. TEMPORAL DATA TYPES

A set of temporal data types is used on properties' definition to represent well-defined static temporal requirements. These data types present different time granularity like hour, year, interval. The different granularities are necessary to make it possible to model reality in a natural way with concepts we are used to.

The F-ORM method has the following pre-defined domains: BOOLEAN, DATE, IMAGE, INTEGER, PLACE, STRING, TEXT, TIME, TITLE. Two temporal data types, DATE and TIME, are already defined in F-ORM. Depending on the application to be modeled other temporal data types are necessary. Four different temporal types can be identified [1, 87]: time points, intervals, span (duration) and periodic time. In our approach we introduce the first three, considering that periods could be defined using rules defining temporal constraints on intervals.

5.1 TIME POINTS

The selected primitive temporal element for explicit time representation is the time point. Analyzing the applications in the information systems domain, the finest time granularity necessary to define human activity is the *minute*, chosen to be the chronon [15] of the data model. To define completely a time point it is necessary to set a *date* (year, month and day) and a *time* (hour and minute). This is done defining the basic temporal type INSTANT. Using a simplified BNF notation, the basic temporal data type INSTANT is defined as:

```
<instant> ::= <year> "/" <month> "/" <day> <hour> ":" <minute>
```

Other temporal types with different time granularity, needed to model information systems applications, can be derived from this basic one. The derivation mechanism is based on a restriction applied to the basic type. The derived temporal types defined in this model are: DATE, YEAR, MONTH, DAY, TIME, HOUR and MINUTE. Some other types are needed to model reality in a natural way, representing special intervals that are considered as time points: WEEK and SEMESTER. The set of temporal types representing time points is completed with the type WEEKDAY representing the correspondent information for a date in the enumeration Sunday to Saturday.

5.2. INTERVALS

Intervals of time can be used to define a set of time points between two limit events. The two limits of an interval must have the same granularity. The chronon within the interval is implicitly defined by the limits' type. Four different types of intervals can be defined, depending if the limit events belong to the interval or not: *closed interval*, when the interval contains both limits; *semi-open interval*, when one of the limits belongs to the interval; *open interval* when both limits are not in the interval; and *floating interval*, in case one of the limits is the actual time. For instance, a closed interval is defined as:

```
<closed interval> ::= <limit> ":" <limit>

<limit> := INSTANT | DATE | TIME | YEAR | MONTH | DAY | HOUR | MINUTE
```

As an example of time points and interval's definition consider the resource class PERSON properties (Appendix 1):

```
resource class (
 PERSON,
 < base-role,
        static properties = { (name, STRING), (birthday, DATE) },
        dynamic properties = {  (object_instance, INSTANT), (end_object, INSTANT),
                (address,STRING),(vacations, INTERVAL(DAYS, CLOSED)) }
```

5.3. SPAN

Another important type is the span (directed duration) of an activity. This information is represented by an integer number followed by the appropriate time unit - e.g., days, hours, weeks. A possible span type is:

```
<month_span> := <integer> MONTH
```

An example of this data type is the span of time a tape is held in the store for locations before it is sold:

```
resource class (
  TAPE,
      ...
  < Life_time,
      dynamic properties = { role_instance, INSTANT), (end_role, INSTANT),
            (time,SPAN(YEARS)) },
      ...
```

5.4. DATA TYPES FOR INCOMPLETE TEMPORAL INFORMATION

Some applications need the representation of incomplete temporal information. For these, Temporal F-ORM presents some specific temporal data types. When the requirement only states that an event shall occur after or before a specific instant one of the following types can be used considering a date as limit type:

```
<limit_date> ::= AFTER <date> | BEFORE <date>
```

The meaning of this data type is the same as an interval with one limit equal to infinity, and considering only one point within the interval. Another special data type for incomplete information is used to represent only one point within an interval, not defining which point:

```
<within_interval> ::= WITHIN <interval>
```

5.5. FUNCTIONS AND OPERATIONS ON DATA TYPES

The uses of data types of different granularity offer some difficulties in the manipulation and operation of the different times [9, 26]. A set of functions (predicates) that convert the different types is defined to accomplish this manipulation. The specialization of classes with inheritance of properties and messages allows the definition of functions for INSTANT (basic temporal type) that can be specialized for DATE or TIME, as needed. The functions return a temporal information, e.g., the weekday corresponding to a date, a different temporal granularity of a given time point, the span of an interval. Some examples are:

year(<instant>)	- extracts the year of an instant
weekday(<date>)	- returns the weekday of the given date
begin(<interval>)	- returns the lower bound of an interval

span(<interval>) - computes the span of an interval

Operations using different time granularity are also defined:

- arithmetic operations like sum and subtraction can be applied in particular cases, like: (i) the two operands are of the type SPAN, resulting in a value of the same type; (ii) the first operand is of type INSTANT, DATE or TIME and the second of type SPAN, resulting the type of the first operand; (iii) the two operands are INTERVALS having the same granularity, resulting in an INTERVAL or in an undefined result;

- the relations "<" (less than), ">" (greater than), "=" (equal to), "≤" (less than or equal to) and "≥" (greater than or equal to) can be used to compare two time points or two spans of different granularity converting internally the values to the finest granularity and resulting in a logical value; the allowed types for time points are INSTANT, DATE, TIME, YEAR, MONTH, DAY, HOUR, MINUTE;

- operations defined for sets that can be applied to INTERVALS, like *union*, *intersection* and *ownership*, resulting intervals or undefined results, or logical values.

6. TRANSACTION AND VALID TIME REPRESENTATION

Two different time concepts must be represented in an application - transaction time and valid times [15, 25]. The transaction time represents the time when an information is stored in a database; valid time corresponds to the time when the information model reality.

6.1. TRANSACTION TIME

The transaction time is implicitly defined by the DBMS. The definition of transaction times in Temporal F-ORM is done timestamping dynamic properties and timestamping objects' instances.

PROPERTIES TIMESTAMPING

An analysis of the possible properties a role can present shows that there are some properties that never change, like the "social security number" for a person object. These properties are called *static properties*; they are supposed to be defined once and valid all over the instance's life.

The properties that may change with time are defined as *dynamic properties*. Transaction times are associated to dynamic properties by timestamping. A dynamic property consists of a set of pairs mapping the definition time to the property's value domain, as proposed in [8]. The temporal domain of these time stampings is the concatenation of the date and the time - the INSTANT data type. Appropriate operations can be used in the query language and in the rules to compare these values and to extract specific information (e.g., year, month, hour, minute, weekday). The instances' whole history may be retrieved through the dynamic properties' values.

As an example, static and dynamic properties are defined in the role employee of the PERSON class:

Employee,
    static properties = { (name, STRING) },
    dynamic properties = { (salary, REAL), (hire_date, DATE), (out_date, DATE) }

Considering the dynamic property *salary*, it is represented by the pairs:

    salary: INSTANT X REAL

A special *null* value can be used to represent the periods of time during which dynamic properties have undefined values. Immediately after the creation of the instance, all the properties receive a default *null* value. Static properties hold this value until the first non null value is assigned to them, and then retain the new value during all their lifetime. For the dynamic properties the *null* value holds until a new value is defined. All the changes of dynamic properties' values are time stamped. During an application, there can be periods of time when a property has an undefined value. This can be represented associating again the *null* value to this dynamic property. This special value can be used in all the property domains.

INSTANCES TIMESTAMPING

Instances of objects are managed through special messages. A special dynamic property of the basic role, *object_instance*, keeps the time correspondent to the creation of an instance of that object, done through the message *create_object*. This property may have one of the two special values defined to represent the instances' life span - *null* and *nonull*. An instances' existence starts at the creation moment, with the value *nonull* associated to *object_instance*, and may have some valid disjoint periods, depending on the income of the messages *suspend_object* and *resume_object*. The beginnings of the periods when the instance is suspended are represented in *object_instance* through the value *null* associated to the corresponding temporal information. The moment the instance is resumed is again associated to *nonull*. The message *kill* terminates the object instances' life. As the implemented database corresponding to this model will be a temporal database, keeping all the values of the past, the active life of the instance is discontinued but the instance is not removed. The instant when an instance is killed is kept in another special dynamic property, called *end_object*, referencing the end of the instances' life.

In Temporal F-ORM there are not only object instances, but also role instances, managed through the messages *add_role*, *resume_role*, *suspend_role* and *terminate_role*. The validity periods of a role instance are stored the same way as the object instances, using a special role dynamic property, called *role_instance*. The end of an instance of a role is stored in another dynamic property: *end_role*.

The validity of a role instance depends on the validity of the corresponding objects' instance. Therefore, the values stored in *object_instance* and *end_instance* are temporal restrictions imposed on those kept in *role_instance* and *end_role*. The termination of an objects' instance kills all the role instances of that object.

The dynamic properties *object_instance* and *end_instance* are implicitly defined for each base role, and *role_instance* and *end_role* for each role.

6.2. VALID TIME

The valid time corresponding to an information can be different from the transaction time. Both these values shall be stored in the database that models the application. Two ways can be used to represent the valid time: (i) augment the definition of dynamic properties to three dimensions, representing respectively the transaction time, the valid time and the information domain; and (ii) to define special dynamic properties that keep the valid time, present only when defined. As the existence of valid time definition is not so frequent, the first option would lead to an unnecessary augment in storage. In Temporal F-ORM the second alternative is used. The name of the special properties used to define valid time is formed by the concatenation of the prefix *valid_* and the name of the property. For instance, considering the property *salary* of the role employee of the PERSON class, when a valid time is defined it is stored in a property called *valid_salary*, associated with the value of the information. For each defined dynamic property there is the implicit creation of a corresponding valid dynamic property. In the Appendix 1 example, the dynamic properties of *employee*, a role of the object *person*, are internally defined as:

```
< Employee,
        dynamic properties = {   (role_instance, INSTANT), (end_role, INSTANT),
                                 (salary, REAL), ( valid_salary, REAL),
                                 (hire_date, DATE), (valid_hire_date, DATE),
                                 (out_date, DATE), (valid_out_date, DATE),
                                 (function, INTEGER), (valid_fuction) ,INTEGER},
                ...
```

The values stored in the database are defined as arguments by messages sent and received by the roles. A special argument is used to define valid times: *Valid_Time*. This argument is optional, used only when the valid time can be different from the transaction time. The corresponding query language must take care of the possibility of existence of a valid time definition, when storing and retrieving an information.

As an example considers the change of an employee's salary. Let's suppose that on the 92/May/10 (transaction time) there is the definition of a new salary for the employee, and that this new salary counts from the first day of that month (valid time). Two possible messages are:

```
messages = { modify_salary(Value:REAL, Valid_Time:DATE)
                        from employee_control,
             end_employment (Valid_Time:DATE)
                        from employee_control    }
```

7. TEMPORAL LOGIC

Two types of rules are used in Temporal F-ORM: state transition rules and integrity rules. A *state transition rule* characterizes valid transitions between states, eventually depending on the arrival of a message. The transition may cause the sending of another message. In some applications there is the need to define dynamic integrity conditions - conditions that compare two different states of the application's information. Therefore a condition was added to the Temporal F-ORM transition rules. This condition is evaluated just before the state transition is activated. The transition between the two states will only happen if the condition is satisfied. The extended state transition rule has one of the following forms:

```
<state transition rule> ::= <rule identifier>":"
        <state1>[","<message1>] "⇒"[<message2>]","<state2>[";"<temporal condition>]

<state transition rule> ::= <rule identifier>":"
        <message1> "⇒"[<message2>]","<state2>[";"<temporal condition>]
```

The second form, when there is no <state1> defined, represents a special state transition rule; the arrival of the message <message1> will cause the transition to the new state, independently of the actual state. This makes it possible to represent *active objects*, that has a defined behavior when receiving a message with a pre-defined temporal argument. In these cases an object *clock* sends messages (eventually virtual messages) at a defined interval to all the active objects. A condition added to these rules represents a limitation of that behavior - the rule will be executed at the moment this condition is satisfied, without considering the actual state.

Integrity rules represent static integrity conditions - conditions that must always hold; they must be satisfied by all instances of a role at all times. An integrity rule has the following form:

```
<integrity rule> ::= "constraint"  "(" <condition1> "⇒" <condition2> ")"
```

The rule represents a constraint: if the first condition is satisfied, than the second condition must also hold. A constraint is evaluated the first time at the moment when the roles' instance is created. It must be satisfied, otherwise the instance will be discontinued. Afterwards, the constraint will be evaluated each time a state transition involving the first conditions' parameters is executed, just before the sending of the outgoing message.

If it happens that the constraint is not satisfied, the transition will be undone; if there is an outgoing message, it will not be sent; and a *NAck* message is sent to the role that sent the incoming message. If the constraint is satisfied, then the outgoing message is sent.

The conditions used in state transition and integrity rules are expressed in temporal logic. Temporal logic is a specialization of modal logic - while the interpretation domain of the last is a generic set of states and the relations between these states, temporal logic requires that these states constitute a linear discrete sequence [20]. A two-state discrete sequence can be used to model dynamic changes at discrete instants.

Situations that change due to the passage of time can be represented using the use of temporal logic. We assume that the time variation is discrete, presenting a linear past and allowing branching in the future. Logic formalisms have been widely used to express requirements involving time and to model dynamic applications [7, 6, 12, 13, 17, 24]. One of the advantages of this formalism is that the use of temporal operators, as *since* or *until* make it possible to represent incomplete information.

The symbols that can be used in the condition formula are: (i) atomic propositions, referencing values of static and dynamic attributes and names of states; (ii) values transmitted as arguments by the incoming message; (iii) relational operators; (iv) Boolean connectives *and*, *or* and *not*; (v) existential and universal quantifiers *formal* and *exist*; and (vi) a set of temporal operators, listed in Table 1.

Operator	Semantics
sometime past A	*A* held at sometime in the past
immediately past A	*A* held at the previous moment
always past A	*A* held at all times in the past
sometime future A	*A* will hold sometime in the future
immediately future A	*A* will hold in the next moment
always future A	*A* will hold at all times of the future
A since B	*A* held at all times since *B* held
A until B	*A* holds at all times until *B* holds
A before B	*A* held sometime before *B* hold
A after B	*A* held sometime after *B* hold

Table 1: Temporal Operators

As we are in an object-oriented framework, operators referencing the past consider only those times beginning with the creation of the instance of the role; as for the future, they will be limited by the life span of the instance.

As an example of a state transition rule with an associate condition, we can use the representation of an employee's salary update. Suppose there is a law stating that an employee's salary can never decrease. The corresponding state transition rule is:

r_i: state(employed), MSG(modify_salary(V)) $\Rightarrow$ state(employed) ;

immediately past exists V1 (salary(V1) and V>V1)

8. CONCLUSION

Socio-technical systems compose a category of applications to which temporal modeling is an essential requirement. In this class of systems the close interaction associating human and automated activities requires modeling and control of temporal characteristics. Object-oriented models are a good option to represent this interaction. They should provide definition of time properties to be used in time critical systems. Time aspects are necessary to represent objects dynamic evolution within an information system environment. The F-ORM model is an object oriented model, and has the pre-defined domains DATE and TIME to support explicit time manipulation.

This work describes Temporal F-ORM, an extension of this model for temporal modeling purposes. A set of different data types was defined to represent unconditional temporal requirements. Not only transaction times are considered but also the valid time corresponding to the moment when information model reality.

State transition and integrity rules are used in Temporal F-ORM to represent complex objects and behaviors (state transitions) in information systems. A transition rule characterizes valid transitions between states. A condition added to the Temporal F-ORM transition rules enables the representation of dynamic integrity rules -- conditions that compare two different states of the application's information. This condition is evaluated just before the state transition is activated, and the transition will only happen if the condition is satisfied. The extended transition rule allows the representation of complex temporal behavior and enables the modeling of conditional temporal requirements.

A support environment is being implemented, with tools that support the requirements' specification, using a class library for reusing former defined requirements. The implementation is based on a deductive temporal database and a Prolog-like query language. The history of the objects' instances is held in the database, which enables the retrieving of several different versions of the object, relative to logical and physical time.

9. REFERENCES

[1] M. Adiba; N.B. Quang; J.Palazzo M. de Oliveira. Time concept in generalized data bases. In: ACM Annual Conference, Denver, Oct. 14-16, 1985. Proceedings. New York, ACM, 1985. p.214-23.

[2] M. Adiba; N.B. Quang; C. Collet. Aspect temporels, historiques et dynamiques des bases de données, TSI - Technique et Science Informatiques, AFCET-Bordas, v.6, n.5, p.457-478, 1987.

[3] J.F. Allen. Maintaining knowledge about temporal intervals. Communications of the ACM, New York, v.26, n.11, p.832-43, Nov. 1983.

[4] C. Arapis. Specifying object interactions. D. Tsichritzis (ed.) Objects Composition. Genebra, Université de Genève, 1991. p.303-22.

[5] A. Bolour; L.J. Dekeyser. Abstractions in temporal information. Information Systems, Great Britain, v.8, n.1, p.41-9, 1983.

[6] J. Carmo; A. Sernadas. A Temporal logic framework for a layered approach to systems specification and verification. In: C. Rolland; F. Bodart; M. Leonard (eds.) Temporal Aspects in Information Systems. Amsterdam, North-Holland, 1988. p.31-46.

[7] J.M.V. Castilho; M.A. Casanova; A.L. Furtado. A Temporal framework for database specifications. In: International Conference On Very Large Data Bases, 8., Mexico City, Sept. 1982. Proceedings. Mexico City, 1982. p.280-91.

[8] J. Clifford; A. Croker. Objects in time. Data Engineering, Washington, v.11, n.4, p.11-18, Dec. 1988.

[9] J. Clifford; A. Rao. A Simple, general structure for temporal domains. In: C. Rolland; F. Bodart; M. Leonard (eds.) Temporal Aspects in Information Systems. Amsterdam, North-Holland, 1988. p.17-28.

[10] E. Corsetti; E. Crivelli; A. Mandrioli; A. Montanari; A.C. Morzenti; P. San Pietro; E. Ratto. Dealing with different time scales in formal specifications. International Workshop On Software Specification And Design, 6., Como, Italy, Oct. 25-6, 1991. Proceedings. IEEE Computer Society Press, 1991. p.92-101.

[11] V. Deantonellis; B. Pernici; P. Samarati. F-ORM Method: a F-ORM Methodology for reusing specifications. In: F.V. Assche; B. Moulin; C. Rolland (eds.) Object Oriented Approach in Information Systems. Amsterdam, North-Holland, 1991. p.117-35.

[12] M. Finger; P. Mcbrien; R. Owens. Databases and executable temporal logic. In: Esprit '91 Annual Esprit Conference, Brussels, Nov. 25-29, 1991. Proceedings. Brussels, ECSC, 1991. p.289-302.

[13] D. Gabbay; P. Mcbrian. Temporal logic & historical databases. In: International Conference On Very Large Databases, 17., Barcelona, Sept. 3-6, 1991. Proceedings. Barcelona, Industria Grafica, 1991. p.423-30.

[14] S.J. Greenspan; A. Borgida; J. Mylopoulos. A Requirements modeling language and its logic. In: M.L. Brodie; J. Mylopoulos (eds.) On Knowledge Base Systems. Springer-Verlag, New York, 1986. p.471-502.

[15] C.S. Jensen et al. A Glossary of temporal database concepts. SIGMOD Record, v.21, n.3, p.35-43, Sept. 1992.

[16] R. Kowalski; M. Sergot. A Logic based calculus of events. New Generation Computing, 4, 1986. p.67-95.

[17] U.W. Lipeck, G. Saake. Monitoring dynamic integrity constraints based on temporal logic. Information Systems, GB, v.12, n.3, p.255-69, 1987.

[18] P. Loucopoulos; P. Mcbrien; U. Persson; F. Schumacker; P. Vasey. TEMPORA - Integrating database technology, rule-based systems and temporal reasoning for information systems development. (to be included in the IEEE Knowledge Engineering Newsletters, Feb. 1991.

[19] R. Maiocchi; B. Pernici; F. Barbic. Automatic deduction of temporal information. University of Udine, Dipartimento de Matematica e Informatica, 1991. 58p. (Research Report). (to be published in ACM Transactions on Database Systems)

[20] Z. Manna; A. Pnueli. Verification of concurrent programs: the temporal framework. In: B. Moore (ed.) The Correctness Problem of Computer Science. Academic Press, 1981. p.215-73.

[21] J. Mylopoulos; A. Borgida; M. Jarke; M. Koubarakis. Telos: representing knowledge about information systems. ACM Transactions on Information Systems, New York, v.8, n.4, p.325-62, Oct. 1990.

[22] B. Pernici. Objects with Roles. In: Conference on Information Systems, Cambridge, Massachussetts, April 25-27, 1990. Proceedings. SIGOIS Bulletin, v.11, n.2-3, p.205-15, 1990.

[23] U. Schiel. An Abstract introduction to the Temporal-Hierarchic Data Model (THM). International Conference On Very Large Data Bases, 9., Florence (Italy), Oct. 31 - Nov. 2, 1983. Proceedings. Italy, VLDB, 1983. p.322-30.

[24] Segev,A. & Shoshani,A. Modeling temporal semantics. In: Rolland,C.; Bodart,F.; Leonard,M. (eds.) Temporal Aspects in Information Systems. Amsterdam, North-Holland, 1988. p.47-57.

[25] R. Snodgrass; I. Ahn. A Taxonomy of time in databases. In: ACM SIGMOD International Conference On Management Of Data, Texas, May 28-31, 1985. Proceedings. New York, ACM, 1985. p.236-46.

[26] G. Wiederhold; S. Jajodia; W. Litwin. Dealing with granularity of time in temporal databases. In: International Conference CAISE'91, 3., Trondheim, Norway, May 13-15, 1991. Proceedings. Berlin, Springer-Verlag, 1991. p.124-40.

APPENDIX 1

VIDEO RENTAL STORE APPLICATION EXAMPLE

Three classes of the Video Rental Store applications are illustrated. As the objective of this example is just to show the temporal extensions defined for the F-ORM model, we don't present the whole specification, just some important parts.

```
process class (
 Rental,
 < base-role,
          static properties = { (creation, INSTANT) },
          dynamic properties = { (client, CLIENT), (tape, TAPE) },
          rules = {rule1 : msg(←create_object) ⇒ msg(→ allow_role(rental_service)),
                   rule2 : msg(←create_object) ⇒
                           msg(→ allow_role(tape_devolution)),
                   rule3 : msg(←create_object) ⇒ msg(→ allow_role(rental_control)),
                   rule4 : msg(←create_object) ⇒ msg(→ allow_role(clients_rental))}
 >
```

/* The attendant receives the client's rental request and asks for information about the client; if the client is allowed to rent the tape, the attendant informs the rental control and gives the tape to the client; if not, he gives this information to the client. */

```
< Rental_service,
          static properties = { ... }
          states = { wait_request, wait_check }
          messages = {
                   tape_request(client:CLIENT, tape:TAPE) from clients_rental,
                   check_client (client:CLIENT) to rental_control,
                   allowed from rental_control,
                   rejected from rental_control,
                   begin_rental to rental_control,
                   request_denied to clients_rental }
          rules = {rule1 : msg(←add_role) ⇒ state(wait_request),
                   rule2 : state(wait_request), msg(←tape_request) ⇒
                           msg(→ check_client), state( wait_check),
                   rule3 : state(wait_check), msg(←rejected) ⇒
                           msg(→ request_denied), state(wait_request),
                   rule4 : state(wait_check), msg(←allowed) ⇒
                           msg(→ begin_rental), state(wait_request) }
 >,
```

/* The attendant receives the devolution of a tape and sends the information to the rental control. */

```
 < Tape_devolution,
          ...
 >,
```

/* Receives request of a clients checking, verifies if the client is allowed to rent a new tape and sends the answer to Rental_service; receives information of beginning and end of rentals and stores the correspondent information. */

```
 < Rental_control,
          ...
 >,
/* The client makes a rental request to Rental_service, receives a denial or the tape; if
he receives the tape, he later returns the tape to Rental_service. */
< Clients_rental,
        ...
>)

process class (
 ACCOUNTING,
 < base-role,
        ...
 >
 < employee_control,
        ...
 >,
 < rental_account,
        ...
 >)

resource class (
 PERSON,
 < base-role,
        static properties = { (name, STRING), (birthday, DATE) },
        dynamic properties = {  (object_instance,INSTANT), (end_object,INSTANT),
                (address, STRING), (vacations, INTERVAL(DAYS,CLOSED)) },
        messages = { ... },
        states = { ... },
        rules = { ... }
 >,
 < Client,
        ...
 >,
 < Employee,
        dynamic properties = {     (role_instance, INSTANT), (end_role, INSTANT),
                (salary, REAL), (hire_date, DATE), (out_date, DATE), (function,
INTEGER) },
        messages = {
                modify_salary(Value:REAL, Valid_Time:DATE) from
                        employee_control,
                end_employment (Valid_Time:DATE) from employee_control,
                employment_time(Time:SPAN(DAYS)) to employee_control,
                employment_ended from employee_control ,
                ... },
        states = { employed, waiting_end_emplyment, disconnected },
        rules = {rule1 : msg(←add_role) ⇒ state(employed),
                rule2 : state(employed), msg(←modify_salary(V)) ⇒ state(employed) ;
                        immediately past exists V1 (salary(V1) and V>V1),
                rule3 : state(employed), msg(←end_employment) ⇒
```

```
                          msg(→ employment_time(T)),
                          state(waiting_end_employment),
                    rule4: state(waiting_end_employment), msg(←employment_ended) ⇒
                          state(disconnected)
                    ... }
 >)

resource class (
 TAPE,
 < base_role,
      static properties = { (object_instance:INSTANT), (end_object, INSTANT),
            (tape_number, INTEGER) },
      dynamic properties = { (tape_film, STRING), (film_type, STRING) },
      messages = { ... },
      states = { ... },
      rules = {... },
 >,
 < Life_time,
      dynamic properties = {(role_instance, INSTANT), (end_role, INSTANT),
            (time,SPAN(YEARS)) },
      ...
 >,
 < Rentals,
      dynamic properties = {(role_instance, INSTANT), (end_role, INSTANT),
            (client_code, INTEGER), (beginning_date, DATE), (end_date, DATE)
},
      messages = {
            rental (Tape, INTEGER, Client:INTEGER) from Rental_control,
            tape_devolution (Tape:INTEGER, Client:INTEGER) from
                  rental_control,
            rented_time (Time:SPAN(DAYS), Client:INTEGER) to
                  rental_account},
      states = { available, rented },
      rules = {rule1 : msg(←add_role) ⇒ state(available),
            rule2 : state(available), msg(←rental(T,C)) ⇒ state(rented),
            rule3 : state(rented), msg(←tape_devolution(T,C)) ⇒
                  msg(→ rented_time(T,C)), state(available) }
 >,

 < Tape_loss,
      ...
 > )
```