BEHAVIOUR MANAGEMENT IN DATABASE APPLICATIONS

J.Y LINGAT * , P. NOBECOURT **, C. ROLLAND**

* *THOM'6 33, rue de Vouillé 75015 PARIS FRANCE*
** *Université PARIS 1 12, Place du Panthéon 75005 PARIS FRANCE*

ABSTRACT

Behavioural aspects of Information Systems are now taken into account in a lot of Conceptual Models. However, the behavioural concepts of these Models have rarely been fully implemented in DBMS.
RUBIS is an extended Relational DBMS which supports an extended relational schema (including event and operation concepts) and automatic control of the dynamic aspects of Applications, i.e event recognition, operation triggering and time handling.

After a short presentation of the basic concepts and the specification language used for the extended Schema, we focus on two internal mechanisms :
- the Temporal Processor, which manages the temporal aspects of specifications and recognizes temporal events,
- the Event Processor, which manages events treatment and synchronization.

These two mechanisms permit an automatic execution of the extended schema and so provide rapid prototyping capabilities.

INTRODUCTION

The dynamic aspect of data is increasingly taken into account by Conceptual Models and by Relational DBMS.
Numerous Semantic Data Models (SDM [HAMM78], TAXIS [MYLO80], [SMIT77],..) are only concerned with data structure.
More recent Models also permit the modelling of data behaviour (ACM/PCM [BROD82], CIAM [BUBE82], REMORA [ROLL82], [CRIS1], [BORG85]).
Finally, Object Oriented Models are now frequently encountered in Data Base works. The spirit of such models is also a mixed representation of the stuctural (static) ~~and behavioural~~ (dynamic) aspects of knowledge (SEMBASE [KING86], GODEL [KERS86]). But there are few realizations of DBMS wich fully support the dynamic concepts of these Models.

On the other hand, there are regular trials for integrating dynamic capabilities into existing DBMS.
There was first the notion of trigger in System R [ESWA76] and alerter in Daisy [BUNE79]; then, other trials were made ([LIN 84], [MELK83], [CHAN81],..) but no real complete integration of these mechanisms in a global model has been accomplished.

The aim of the RUBIS System is to provide a complete dynamic Model, fully supported by a Relational DBMS.
Our Model is based on REMORA [ROLL82]; the static objects are modelled by <u>relations</u>, while <u>operations</u> (elementary actions on an object) and <u>events</u> (elementary state changes triggering one or several operations) permit the modelling of the dynamic aspects of the objects. The Conceptual Schema is called the R-Schema (RUBIS-Schema). In this schema, the temporal aspects of the Application are also taken into account; they are modelled using the time types provided by the RUBIS Model.

In this paper, we are only concerned with :
- the R-Schema, which is specified using our Specification Language called PROQUEL (PROgramming QUEry Language).
  The possibilities of this language will be demonstrated by the examples given in the first section.
- physical handling of the dynamic concepts. This is achieved by the Temporal Processor, which manages temporal aspects of the specification; and by the Event Processor, which manages event recognition and synchronization.
  These two mechanisms will be described in section II.

I THE R-SCHEMA

I.1 UNDERLYING CONCEPTS

The R-Schema is based upon three kinds of elements which allow a complete description of a Database Application :
a) Relations represent entity types or relationship types from the real world (e.g CUSTOMERS, BANKS, LOANS,...).
b) Events represent special situations in the Database life cycle, in which one or several operations acting on the

Database must be triggered.
There are three kinds of events :

* an internal event describes a "noticeable state change" of one and only one relation (e.g an account becomes a debit account; an employee salary becomes greater than his manager's,...). The "noticeable state change" is specified in the event predicate and generally concerns two ~~successive states : s and s' (also called OLD and NEW)~~ of a relation tuple.
For instance : "the balance of an account was positive or nil (s.BALANCE >= 0) and is now negative (s'.BALANCE < 0)".
An internal event is thus said to "ascertain" its associated relation, because it ascertains the relation state changes.

* an external event describes the arrival of a message from the real world (e.g "loan requirement arrival", "cheque arrival",..). The external event predicate describes the acceptance condition of the message (e.g "the date of the cheque is valid").

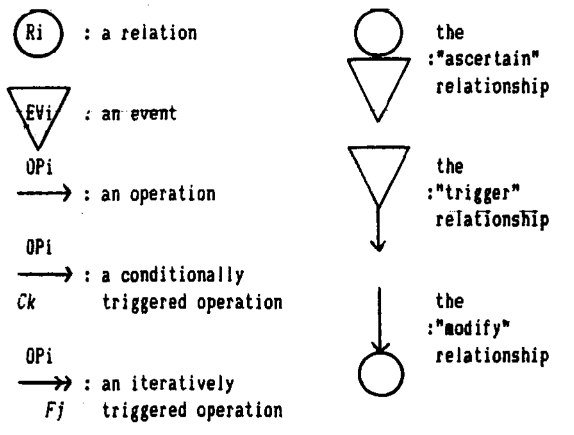
* a temporal event describes a situation with reference to time. This situation can be either an absolute reference (e.g 25/10/87), or a periodic reference (e.g the thirteenth day of each month), or a reference to another event (e.g 3 days after the occurrence of the "cheque arrival" event).

Successful testing of the event predicate means recognizing the event; it is at this moment that the event occurs; there is event occurrence.

c) **Operations** represent the elementary actions triggered by the events when they occur. An operation stands for an action type (e.g send a warning letter, modify an account balance,...) and can modify at most one relation. An operation instance (i.e operation executed in fact) can modify at most one relation tuple (e.g modify the account n° 44532), with respect to the elementarity principle.
The triggering of an operation can be :
- conditional, in this case, the operation is executed only if the **triggering condition** is true (e.g put the order note in the "wait" mode only if the stock is not sufficient).
- iterative, then the **triggering factor** computes all the tuples that will be used as effective parameters for the execution of the operation (e.g the sending of a Christmas letter to all "good" customers).

Notes:

1) Operations, conditions and factors can appear several times in the R-Schema : an operation can be triggered by several events and two different operations can have the same triggering condition or factor. In the same way, a relation can be modified and/or ascertained by several different operations and/or events. For this reason, events, relations, operations, conditions and factors can be specified independently.
2) Splitting update operations into : "elementary action + condition + factor" may seem quite restrictive but permits us to exercise entire control over system behaviour, as will be seen below. Morover, such splitting helps avoid redundancy and helps obtain a modular description of the processing.
3) The following notation is used to construct a graphic representation of the R-Schema :



Ri
: a relation
EVi
: an event
OPi
: an operation
OPi
: a conditionally triggered operation
Ck
OPi
: an iterative triggered operation
Fj
the :"ascertain" relationship
the :"trigger" relationship
the :"modify" relationship

4) According to the definitions of operations and events, the key concept of behaviour modeling is Dynamic Transition.
It is composed of :
- the event,
- all the operations it triggers,
- all the relations modified by these operations.
The following figure represents a dynamic transition :

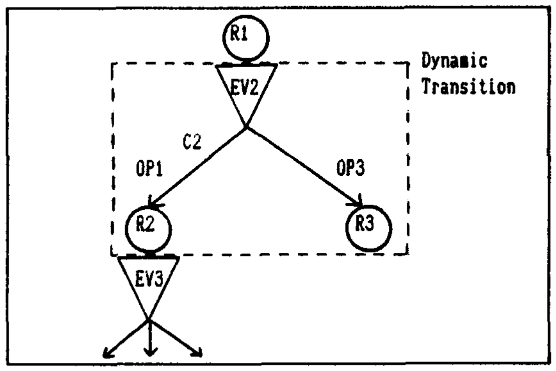


Figure 1 : A Dynamic Transition

This figure highlights an important aspect of behaviour modeling : the succedence of Dynamic Transitions. For example, in fig. 1, the transition of EV3 follows the transition of EV2. Most of RUBIS' work lies in the handling of these transitions and of their ordering, as we will see further on.

1.2 THE TIME MODEL

1.2.1 TIME IN DATABASE APPLICATIONS

The time concept occurs at different levels during the specification of static and dynamic aspects of data [BOLO82].
* On one hand, time enables us to express some static properties of entities or relationships (for instance, the "OBTAINING_DATE" of the "LOAN" relation).

* ~~On the other hand, as in Historical Databases, time is used to~~ manage successive versions of data (the "timestamp notion" [DADA84], [ADIB86]) and to access these different versions by asking questions like: "what was the address of the subscriber "Jones" at 08/09/86 ?".
* Finally, the concept of time allows automatic triggering of some actions according to temporal assertions (e.g "send an acknowledgement no more than three days after the order-note arrival").

In the RUBIS time model, imprecisely specified times and different abstraction levels of temporal specifications are supported. For instance, it is possible to express "predefined temporal types" (points, intervals, durations, periodic times) ~~at all abstraction levels~~.

Some functions are provided for handling relationships between times. This is necessary for specific applications like planning systems, where causal relationships between times are more important than precise times.
Two kinds of specifications are handled : absolute times (e.g dates) and relative times, such as times defined relative to an event occurrence (e.g three days after the order-note arrival) [BARB85]. In the following, we briefly define time types and the primitives used in RUBIS.

1.2.2 TIME TYPES

Time assertions are described using a calendar. The predefined calendar is the common gregorian calendar augmented with hours, minutes and seconds.
Time may be specified at six levels of abstraction : year (1986), month (1986/12),...second (1986/12/04:23h54m03s).
Year is considered to be a higher level than month, which is a higher level than day, and so on. Elements within a given level are specified using only upper levels.

For each level of abstraction, the following types are defined :
- **Time Point** type : The time point type is based on the primitive concept of the temporal axis origin. A time point is defined using the calendar schema. For instance, "1986/05/11" is a valid specification at the day abstraction level.
- **Time Interval** type : A time interval is defined by its bounds. which are of point type. For instance, [1986/05/11-1986/05/14] is a valid interval at the day abstraction level.
- **Duration** type: duration type allows reference to the distance between two points. A value from this type is defined in terms of elementary durations (according to the calendar schema).
  For instance, "1 year, 3 months, 20 days", and "15 days" are valid durations at the day abstraction level.
- **Periodic Time** type : A periodic time is defined by its base (point or interval type) and its period (duration type). For instance, "the 25th day of each month" is a valid periodic time at the day abstraction level.
  A periodic time may be limited by an interval, so : "every fortnight from order-note arrival and until delivery" is a valid periodic time too.

Time functions and operations, such as before, after, equal..., are provided. Conversion functions are also provided (when moving from a given level of abstraction to an other). For instance, the ~~following specification : "at month(1986/05/11)"~~ is equivalent to "at 1986/05", and "after minutes(15 days)" is equivalent to "after 21600 minutes"

The way in which temporal assertions (expressed via the above types, functions and operations) are organized to provide a structure for automatic triggering of operations will be discussed in subsequent sections.

1.3 DESCRIBING THE R-SCHEMA

The description of the R-Schema can be made incrementally :
- first, the static sub-schema can be described with relation specifications ~~(introduced by **DEFINE RELATION**)~~.
- Second, a first version of the dynamic sub-schema can be obtained by specifying dynamic transitions (these specifications are introduced by **DEFINE EVENT**).
- Third, the dynamic sub-schema can be completed by operation, condition and factor specifications (respectively introduced by **DEFINE OPERATION**, **DEFINE CONDITION**, and **DEFINE FACTOR**).

The static sub-schema used in the examples (drawn from a Bank Application) is shown below. Figure 2 presents the LOAN relation specification.

```
CUSTOMER (CUST#, CUSTNAME, CUSTADR, TYPE)
LOAN (LOAN#, CUST#, OBTAINING_DATE, AMOUNT, REF_NB, FREQ)
ACCOUNT (ACC#, CUST#, BALANCE)
SAVINGS_ACCOUNT (SAV#, CUST#, BALANCE, OPENING_DATE,RATE)
CEILINGS_HISTORIC (HDATE, CEILING)
```

```
DEFINE RELATION LOAN
    ( LOAN# : INTEGER KEY;
      CUST# : INTEGER;
      OBTAINING_DATE : DATE;
      AMOUNT : DOLLARS;
      REF_NB : INTEGER;  /* total number of refunds */
      FREQ : DURATION );   /* refunds frequency */
```

Fig. 2 : Specification of the LOAN relation

EXAMPLE 1: INTERNAL EVENT SPECIFICATION

Figure 3 associates the textual, graphic and formal specifications of a savings-account management rule.

* The ascertained relation name and the type of the state-change are introduced by **ON**.
* **PRED** contains the "noticeable state change" statement.
  Here, the operator **LAST** helps to retrieve the CEILING that was in effect just before the SAVINGS_ACCOUNT opening date.
  If the predicate is complex (such as here) the final computation of the "return value" of the event predicate is made using the **RETURN** operator. The predicate can be empty if the state-change is a simple insertion, deletion or update.

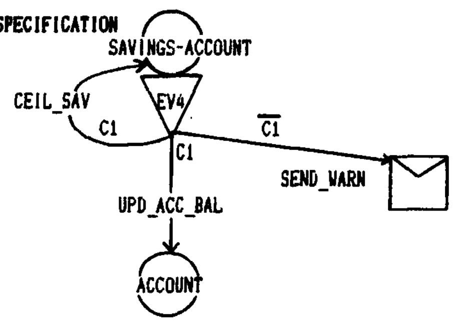
In each internal event specification, the ascertained relation (here: SAVINGS_ACCOUNT) is the implicit parameter of the

assertion introduced by **PRED**. The formal parameter is the relation name, while the effective parameter, also called "context", is composed of the tuple pair which defines two successive states of the modified entity (here: the savings-account state before and after its BALANCE modification).
The **OLD** prefix and the **NEW** prefix allow us to reference (and to differentiate) the two tuples or, to be more precise, the old and new values of their attributes (e.g NEW.BALANCE).
The **CONTEXT** prefix is used (e.g CONTEXT.OPENING_DATE) when no differentiation has to be made (the attribute value hasn't changed).

TEXTUAL SPECIFICATION

* Each savings-account possesses a ceiling that should not be overstepped (this ceiling depends upon the opening date).
* If the customer also possesses a current-account in the bank, the surplus is transferred into it.
* If the customer doesn't possesses another account, a warning letter is send to him.

GRAPHIC SPECIFICATION



PROQUEL SPECIFICATION

```
DEFINE EVENT EV4  IS ACC_OVER
   ON UPDATE  OF  SAVINGS_ACCOUNT
   COMMENT "The savings-account exceeds its ceiling"
   PRED
     ( VAR $CEIL : DOLLARS;
       $CEIL:=SELECT CEILING FROM LAST CEILINGS_HISTORIC
              WHERE HDATE <= CONTEXT.OPENING_DATE;
       RETURN NEW.BALANCE > $CEIL )
   TRIGGER
       IF C1 THEN ( UPD_ACC_BAL(NEW.BALANCE-$CEIL)
                        ON ACCOUNT;
                    CEIL_SAV($CEIL) ON SAVINGS-ACCOUNT )
          ELSE SEND_WARN;
```

Fig. 3 : Specification of the internal event EV4

* The **TRIGGER** part introduces the operations, their respective triggering conditions and factors, and the relations they modify (if any). In the example, UPD_ACC_BAL (OP3: transfer the surplus of the new BALANCE to the customer's current-account) and CEIL_SAV (level the savings-account balance to the ceiling) are executed if the condition C1 (the customer possesses a current-account) is true. If not, SEND_WARN (send a warning letter) is executed.

The specification of Dynamic Transitions corresponds to the first version of the schema definition.

After checking for consistency, second level R-Schema specification can start. This includes the definition of conditions, factors and operations texts.
An example of such definitions is given in figures 4 and 5, which respectively introduce the C1 and OP3 specifications.

```
DEFINE CONDITION C1 IS CURR_ACC
   COMMENT "The customer possesses a current-account"
   TEXT  EXISTS ACCOUNT WHERE CUST# = CONTEXT.CUST#;
```

Fig. 4 : Specification of the triggering condition C1

```
DEFINE OPERATION OP3 IS UPD_ACC_BAL
 MODIF ACCOUNT
 TYPE UPDATE
 COMMENT "Modify the customer's current-account balance"
 INPUT ($credit : DOLLARS)
 TEXT UPDATE ACCOUNT
      SET BALANCE = BALANCE + $credit
      WHERE CUST# = CONTEXT.CUST#;
```

Fig. 5 : Specification of the operation OP3

Conditions, factors and operations have two kinds of parameters:

- their **explicit parameters** (ex: $credit), introduced by **INPUT** and receiving a value during the call (i.e in the **TRIGGER** part of the triggering event(s)),
- an **implicit parameter** : that of the triggering event (here: the two tuples representing the modified savings-account). The attributes of this implicit parameter are retrieved using the OLD, NEW or CONTEXT prefix (e.g CONTEXT.CUST#).

In an operation specification, the modified relation name is introduced by **MODIF**. The **TYPE** part introduces the modification type (INSERT, DELETE or UPDATE). The **TEXT** part contains the operation's algorithmic specification.
Each operation possesses an implicit output parameter: the two versions of the tuple it modifies.

NOTE: These two specifications could have been incorporated in the EV4 specification, using a special notation :
IF C1 [DEFINED AS ..] THEN ...

EXAMPLE 2: EXTERNAL EVENT SPECIFICATION

Figure 6 represents the specification of the external event : "Arrival of a list of movements for a savings-account" (EV3).

An external event ascertains the arrival of an appropriate message. The structure of this message (which is the external event "context") can be quite complex and is described in a special part.

* Since an external event ascertains no relation, the **MESSAGE** keyword is used in the **ON** part.
* **PROP** describes the structure of the message the event is waiting for. This structure isn't in 1st Normal Form, so it may contain embedded structures, optional fields and lists (c.f LIST_OF if fig. 6).

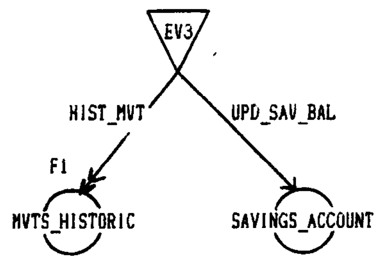
* The message validity condition is described in the **PRED** part (here: "the SAVINGS_ACCOUNT referenced by its number in the message, must already exist in the Database").
* The **TRIGGER** part of an external event is identical to that of an internal event. In this example, note that UPD_SAV_BAL (update the SAVINGS_ACCOUNT balance) is unconditional, and HIST_MVT (save a movement into the historic) is iterative (cf. factor F1, described further on).

* The message is the "context" of the external event. The values of this message are still accessible from the **PRED** and **TRIGGER** parts of the event, and also from all the specifications of conditions, factors and operations which appear in the **TRIGGER** part of the event. Here again, the prefix to use is **CONTEXT** (e.g CONTEXT.svnum).

TEXTUAL SPECIFICATION

* On each arrival of a list of movements for a savings -account, update the savings-account.
* The savings-account referenced by the message must already exist in the Data Base.
* Each savings-account movement must be historicized.

GRAPHIC SPECIFICATION



PROQUEL SPECIFICATION

```
DEFINE EVENT EV3 IS MVT_ARR
  ON MESSAGE
  COMMENT "Arrival of a list of movements
          for a savings-account"
   PROP
    ( svnum : INTEGER; (* savings-account number*)
      l_mvt : LIST_OF ( date_mvt : DATE;
                        mvt : DOLLARS ) )
  PRED EXISTS SAVINGS_ACCOUNT WHERE SAV#=CONTEXT.svnum
  TRIGGER
  (UPD_SAV_BAL ON SAVINGS_ACCOUNT;
   HIST_MVT(CONTEXT.svnum,FACT.dat,FACT.move,FACT.type)
            ON MVTS_HISTORIC  FOR F1 );
```

Fig. 6 : Specification of the external event EV3

Fig. 7 presents the F1 triggering factor specification. The **OUTPUT** part describes the structure of the FACT relation, which is the implicit output parameter of every factor. How the FACT tuples are generated is described in the **TEXT** part. These tuples will be used as effective parameters during the call of the operation to trigger iteratively (cf. "FACT.xx" in the EV3 **TRIGGER** part).

```
DEFINE FACTOR F1  IS ALL_MVTS
 COMMENT "For all the movements included in the message"
 OUTPUT (dat: DATE, move: DOLLARS, type: MVT_TYPE)
 TEXT FOR EACH m IN CONTEXT.l_mvt
      DO IF m.mvt >= 0
         THEN INSERT INTO FACT (m.dat_mvt,m.mvt,'CREDIT')
         ELSE INSERT INTO FACT (m.dat_mvt,-m.mvt,'DEBIT');
```

Fig. 7 : Specification of the triggering factor F1

EXAMPLE 3: TEMPORAL EVENT SPECIFICATION

Each temporal event is associated with the predefined relation named **CALENDAR**. Its predicate is a temporal assertion which is defined using ~~all temporal model capabilities.~~ In this section, time assertions have been specified using the following subset of operations and language clauses :

OPERATORS

=	:equality redefined on time types.
*	:multiplication of an integer and a duration. The result is a duration.
+	:addition of a point and a duration. The result is a point.

CLAUSES

d <u>after</u> p	:d is a duration, p is a time point. The result is a time point z, such as z = p + d.
<u>from</u> p	:time interval starting at point p
<u>until</u> p'	:time interval ending at point p'
<u>every</u> d	:periodic time defined with a duration (this duration represents the distance between two realizations).
at <time-assertion>	:refers to the first point for which the assertion is true. For example, "at day=26" defines the first point from current time, for which day=26.
each <time-assertion>	:refers to the set of points from current time, for which the assertion is true. For example, "each day=15" generates one realization per month.

Figure 8 describes the temporal event EV1 which triggers the pay-roll publishing periodically. Here, the F4 factor means "for all the employees".

```
DEFINE EVENT EV1
ON CALENDAR
PRED  each day=25
TRIGGER  OP4 FOR F4 ;
```

Fig. 8 : Specification of the EV1 temporal event

If EV5 is the external event ascertaining an order-note arrival, then "EV5.time" represents the real occurrence time of an EV5 instance (the occurrence time is an implicit property of every event). Thus, the predicate of event EV6 : "at most 3 days after each order-note arrival" can be specified in the following way :

PRED <u>until</u> 3 days <u>after</u> EV5.time

Here, EV6 has been defined "by reference to EV5", there will be one occurrence of EV6 after each occurrence of EV5.

More complex temporal assertions can be expressed. For example, if we consider the operation of "automatic levy for loan refunds", pay-days are defined by the following expression :

"on the 26th day of the month, each FREQ, from the month following OBTAINING_DATE and during a period equal to (REF_NB * FREQ)"

FREQ, OBTAINING_DATE, and REF_NB are attributes of the LOAN relation which is described in Fig. 2.

```
PRED ( ALIAS OF LOAN IS L;
       from 1 month after Month(L.OBTAINING_DATE);
       at day=26;
       every L.FREQ;
       until (L.REF_NB * L.FREQ)
                after ( Month(L.OBTAINING_DATE) + 1) )
```

Figure 9 : A more complex predicate specification

Finally, an interesting application of temporal events is the automatic management of Database Snapshots, which can be easily modelled using such events.

II MANAGING DYNAMICS

II.1 GLOBAL ARCHITECTURE OF RUBIS

Automatic management of the database dynamics from the R-Schema specification involves :

- automatic recognition of events,
- automatic triggering of appropriate operations when an event occurs,
- operations execution control,
- event synchronization.

Attaining such automation requires :

a) a Relational DBMS to deal with :
- managing the relations and the Meta-Base corresponding to the R-Schema specifications,
- executing operations texts and evaluating factors, conditions and predicates; this requires an interpreter more powerful than a simple SQL interpreter;

b) a mechanism able to :
- recognize an event,
- determine which operations to execute,
- trigger and control operations execution,

and to synchronize event-chaining.

This mechanism is similar to the inference engine of a forward chaining expert system, whose cyclic function is to :
- test the rules premises,
- choose a candidate rule,
- execute the action part of the rule,

and which possesses a rule-chaining strategy.
Such a mechanism has to exist in the DBMS itself for an efficient management of the database dynamics.

The mechanism we propose is composed of three units managing all kinds of events :
- the Applications Monitor recognizes external events,
- the Temporal Processor recognizes temporal events,
- the Event Processor recognizes internal events and treats all events and their synchronization.

II.2 THE RUBIS' RUNNING

* The Meta-Base contains the relational description of the R-Schema. Texts, like any other component are stored in it; thus, they can be modified easily. For example, modifying a text (like an event predicate or an operation) doesn't imply recompiling the application; it doesn't even imply stopping the users' activities if the text isn't used at that moment.

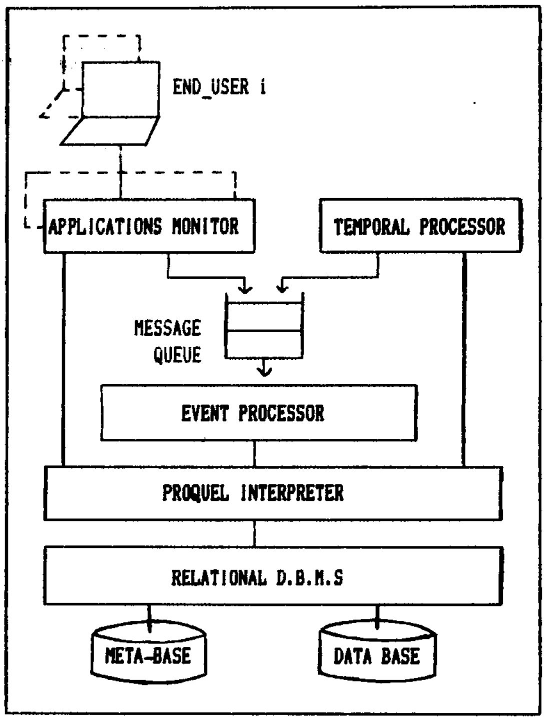


Fig. 10 : Global Architecture

* The Applications Monitor is the end-user interface. For each external event specification, a corresponding Application Program (A.P.) is generated. The A.P. construction is based upon the event structure (its PROP part) and predicate (which is the message validation condition).
The Applications Monitor executes Application Programs according to end-user requests. In fact, executing an A.P. corresponds to a message acquisition and validation. When the A.P. is correctly finished, the Applications Monitor sends the valid message into the Message Queue.
Since the external event predicate is verified by the corresponding A.P., one may consider the reception of a valid message in the Message Queue as an external event occurrence.
Each time a user is connected to RUBIS, a process containing an Applications Monitor is created.

* The Temporal Processor works independently. It sends a message into the Message Queue each time it recognizes a temporal event.

* The Event Processor recognizes internal events, takes into account, processes and synchronizes events.
  - taking into account an external or temporal event is accomplished by removing the corresponding message from the Queue.
  - processing an event includes :
    - evaluation of all conditions and factors appearing in the TRIGGER part of the event,
    - controlled execution of all operations having a true condition and a not empty factor.
      Event processing is the atomic execution unit, which means that it must be executed entirely and in one block or not at all. Furthermore, it is also the consistency unit, since it must leave the database in a valid state.
  - event synchronization is based upon the following strategy: the internal consequences of an external or temporal event (i.e the internal events it may generate), receives priority processing before any other external or temporal event can be taken into account .

* The PROQUEL Interpreter evaluates predicates, conditions and factors, and supervises execution of operations. It executes all texts written in PROQUEL, by sending queries to the DBMS and managing : local variables, control structures and parameter passing.
  It is being developped using the LEX and YACC tools of the UNIX System. Queries (expressed in relational algebra) are sent to a small Relational DBMS called PEPIN [BOUC81].

We will now focus on the two Processors, which are the most interesting parts of the System.

II.2 THE TEMPORAL PROCESSOR

The role of the Temporal Processor is to recognize automatically each occurrence of a temporal event, and to inform the Event Processor of such occurrences (by sending a message into the Queue). Its strategy is based on a dynamic management of the agenda.

The agenda is a chronologically organized list describing a pertinent subset of future temporal event occurences.
This list is constructed :
  - either by directly using the R-Schema specifications (the case for temporal events in which the predicate defines an absolute time),
  - or a time propagation through the Temporal RElationships Graph (the case for temporal events in which the predicate defines a relative time).

II.2.1 THE AGENDA STRUCTURE

1) The occurrence domain notion

In theory, a temporal event occurs when its predicate becomes true for Current-Time ("Current-Time" is the value returned by the "now" function which reads the computer clock).
In fact, due to our model, such a predicate may be true during a time interval. Therefore, in order to deal with these "imprecisely defined times", the <u>occurrence-domain</u> notion must be distinguished from the <u>occurrence-time</u> notion.

The occurrence-domain of a given (temporal event) EVi instance is a time interval during which the EVi predicate is true; that is to say during wich an instance of EVi may occur. As shown in fig. 8, the occurrence-domain of the EV1 instance X could be:
[1986/12/23:00h00m00s - 1986/12/23:23h59m59s]
and the occurrence-domain of the instance X+1 :
[1986/12/24:00h00m00s - 1986/12/24:23h59m59s]

Occurrence-time is the precise time at which a given event is effectively ascertained. An event occurrence is then instantaneous and the assertion : "occurrence-time <u>is during</u> occurrence-domain" must always be true.

2) The agenda organization

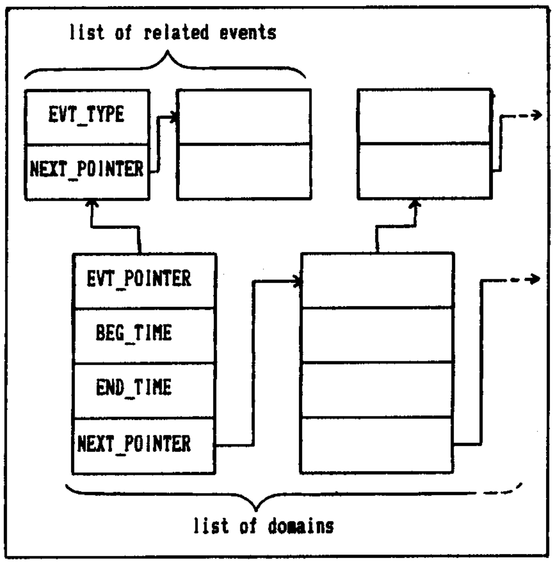


Fig. 11 : The AGENDA structure

Each element within the list of domains :
  - corresponds to a particular occurrence domain,
  - is associated with the events whose predicate is always true during this domain.

Occurrence domains are defined in terms of time intervals like [BEG_TIME - END_TIME]. Therefore, it is possible to represent :
  - occurrence-domains reduced to a single point (if BEG_TIME = END_TIME),
  - occurrence-domains infinite in the past (or the future) by assigning to BEG_TIME (or END_TIME) the "INFINITE" value.

The agenda is sorted in increasing order of "BEG_TIME"; a domain B appears "later" than a domain A :
  - when the assertion : "B.BEG_TIME <u>is after</u> A.BEG_TIME" is true; no assumption is made concerning A.END_TIME and B.END_TIME; so, domain B can be included in domain A when the assertion "A.END_TIME <u>is-after</u> B.END_TIME" is true.
  - or when the two assertions : "B.BEG_TIME <u>equals</u> A.BEG_TIME" and "B.END_TIME <u>is-after</u> A.END_TIME" are true.

II.2.2 ORDERING ABSOLUTELY DEFINED EVENTS

Formally, a temporal event is absolutely defined if its predicate directly or indirectly refers to the calendar origin.
For example, all the following predicates define absolute times:

PRED at 1986/05/11
PRED during [1986/05/11 - 1986/06/11]
PRED from 1986/05/11 and until 1986/06/11
PRED at 10 days after 1986/11/30

The absolute definition is transitive, so a time expressed relatively to an absolute time is interpreted as an absolute time.
Insertion of an absolutely defined event into the agenda can be executed immediatly after validation of its specification. The Temporal Processor then executes the following actions :

- it examines the event specification (located in the R-Schema) in order to determine the occurrence-domain of the event;
- it searches for this occurrence-domain in the agenda;
- if the occurrence-domain is already present, it simply adds the event into the domain's related events list;
  if not, it inserts a new occurrence-domain into the agenda, The related event list of this domain contains at this point only the considered event. Domain location within the agenda is determined by comparing the domain bounds with those of the other domains.

NOTE : if the event predicate defines a periodic time, only the first future occurrence is inserted into the agenda.

II.2.3 ORGANIZING RELATIVELY DEFINED EVENTS

A temporal event is relatively defined if its predicate doesn't refer to the calendar origin at specification time. This is the case when the event predicate references :

- the occurrence-time of another event,
- a temporal attribute of a database relation.

Therefore, it is impossible to determine the occurrence-domain of a relatively defined event just after its specification.
For example, if the EVk predicate is the following :

PRED until 3 days after EV3.time

the corresponding occurrence-domain can be determined only when EV3.time (i.e the occurrence-time of an instance of EV3) is known. At this moment, an absolute time can be derived from the relative specification. Consequently, a relatively defined event can become an absolutely defined event (applying transitiveness) during the system evolution.
In order to allow such transformations, the Temporal Processor manages a graph which organizes temporal events according to their relationships [KAHN77] [MITT82].

1) The Temporal REferences Graph (TREG)

This graph is a directed graph which consists of a finite set of vertices and edges.
A vertex V describes a non-temporal event (and then corresponds to a root) or a relatively defined event.
Two vertices V1 and V2 are connected by an edge (V1,V2) if V2 is defined by reference to V1.

In the graph presented in figure 12, the EV49 predicate is the most complex and has the form :

PRED from 7 days after EV47.time
and until 1 month after EV41.time;

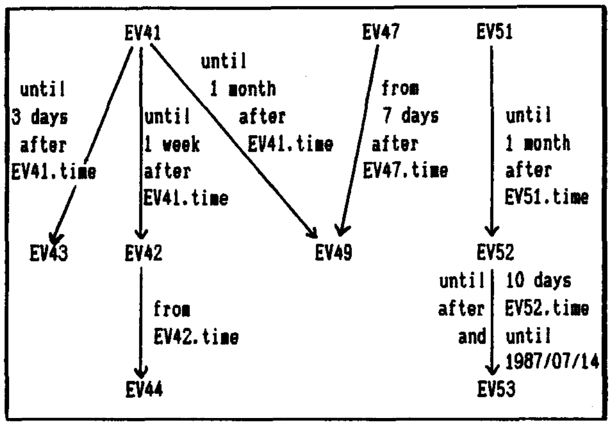
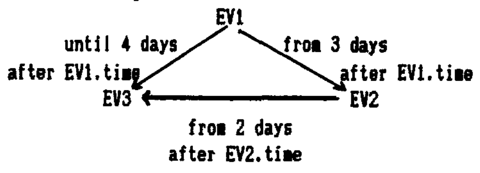

EV41
EV47
EV51
until 3 days after EV41.time
until 1 week after EV41.time
until 1 month after EV41.time
from 7 days after EV47.time
until 1 month after EV51.time
EV43
EV42
EV49
EV52
from EV42.time
until 10 days after EV52.time and until 1987/07/14
EV44
EV53

Fig. 12 : An example of Temporal REferences Graph (T.R.E.G)

* The TREG is built by the temporal language interpreter, which tests the consistency of each new temporal specification. For example, incoherent constructs such as the one below are detected using the transitive closure mechanism.


EV1
until 4 days after EV1.time
from 3 days after EV1.time
EV3
EV2
from 2 days after EV2.time

* The TREG will also be used by a "Time Expert" in order to answer questions like : "can EV43 occur before EV42 ?" or "what is the distance between EV42 and EV41 ?".

* Events from the TREG are grouped together in precedence classes. P_Class(EVi) is composed of all the events which are defined relatively to EVi. Five P_Classes can be derived from the TREG of Fig. 12 :
  P_Class(EV41) = { EV43, EV42, EV44, EV49 }
  P_Class(EV42) = { EV44 } P_Class(EV47) = { EV49 }
  P_Class(EV51) = { EV52,EV53 } P_Class(EV52) = { EV53 }

2) Propagation mechanisms

* When an event EVi occurs, the Temporal Processor verifies the existence of P_Class(EVi). If this class exists, it then propagates the EVi.time value through the TREG, in order to determine the occurrence-domains of the next instances of events belonging to the class. In fig. 12, the EV41.time propagation determines the occurrence-domains of new instances of events EV43 and EV42; and fixes the upper bound of the occurrence-domain of a new EV49 instance. This mechanism is called forward propagation.

* A second mechanism, called backward propagation, is used to

determine the occurrence-domains more precisely.

As shown in figure 12; if "day(EV51.time)" = 1987/07/01, then the two following occurrence-domains are deduced using forward propagation : occ_dom(EV52)=[1987/07/01 - 1987/08/01]

then, occ_dom(EV53)=[EV52.time - 1987/08/11]
<u>inter</u> [EV52.time - 1987/07/14]

But obviously, if day(EV52.time) is greater than 1987/07/14, it is impossible for EV53 to occur. Therefore, backward propagation is used to reduce the occurrence-domain of EV52 so that EV53 may occur : occ_dom(EV52)= [1987/07/01-1987/07/14]

Non-ascertainment of an event is still possible. For example, this situation occurs when EV51.time is greater than 1987/07/14. For this type of situation, the designer must define exception handling statements.

When a referenced event EVi occurs, forward and backward propagations are executed. Then, the "now completely defined occurrence-domains" and their related "future events occurrences" are inserted into the agenda.

II.3.3 THE TEMPORAL PROCESSOR ALGORITHM

```
WHILE there are some events related to the considered
      occurrence-domain
DO BEGIN
              /* . send to the Events Processor a message*/
              /*   notifying the occurrence of the first */
 PROCESS 1;   /*   event of the related list,            */
              /* . propagate the event occurrence-time in*/
              /*   the TREG,                             */
              /* . insert, if necessary, deduced domains */
              /*   and events into the agenda;           */

 IF the recognized event predicate corresponds to a
    periodic time
 THEN
                 /* .determine the next occurrence domain*/
    PROCESS 2;   /* .insert this domain and its related  */
                 /*  event into the agenda;              */

 PROCESS 3;   /* . delete the recognized event from the  */
              /*   agenda;                               */
END;
              /* . delete the occurrence-domain from the */
 PROCESS 4;   /*   agenda and sleep until the beginning  */
              /*   of the next domain.                   */
```

Fig. 13 : Global algorithm of the Temporal Processor

Only future domains and events are present in the agenda. The first element in the list of domains is always the next domain to occur. Therefore, it determines the next event occurrences to be ascertained. So, the Temporal Processor must wait for "Current-Time" to belong to this first domain. When true, the processor executes the set of processes shown above.

NOTE : this algorithm is a basic version of the Processor. It is, indeed, possible to take into account event priorities (depending on resources allocation, for instance). In this case, the Processor should dynamically reorder the different lists of the agenda after each temporal event occurrence.

II.4 THE EVENT PROCESSOR

The event processor fulfils three main functions :
- takes into account external and temporal events;
- processes events;
- orders them.

The first function is based on a FIFO management of the Message Queue. The second function consists of a meta-base search for appropriate conditions, factors and operations that will be evaluated or executed by the relational DBMS. These two functions do not present any major difficulties, as opposed to the third function, presented in the following section.

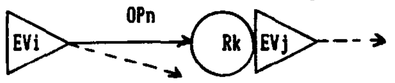
II.4.1 EVENT SYNCHRONIZATION

The chosen strategy is based on the induction notion, and on the use of the induction graph, which is derived from the R-Schema.

a) The induction notion

DEFINITION :
An event EVi inducts an event EVj if and only if :
- EVi triggers OPn which modifies the relation ascertained by EVj,
- an occurrence of EVi, followed by the execution of OPn <u>can</u> produce an occurrence of EVj.

Graphically, the situation is the following :



The notation used to represent an induction is : $EVi \xrightarrow{OPn} EVj$

But, if OPn has a triggering factor, the induction is called a

multiple induction and is written : $EVi \overset{OPn}{\twoheadrightarrow} EVj$
This means that an occurrence of EVi can induce several occurrences of EVj.

The EV3 and EV4 events (described in section I, Fig.6 and Fig.3) give us the following induction :

$$EV3 \xrightarrow{OP1} EV4$$

this is because OP1 modifies the SAVINGS_ACCOUNT relation and may produce a "ceiling exceeded" event (EV4).

B) The Induction Graph construction

The Induction Graph uses the above notation. It contains :
- vertices representing R-Schema events,
- directed edges representing inductions,
- weights on the edges, which represent operations and are used as "induction conditions".

The Induction Graph construction is accomplished in two steps :
- an automatic step, producing the Maximal Induction Graph,
- a manual step transforming the Maximal Induction Graph into the Induction Graph.

1st STEP

The Maximal Induction Graph can be automatically deduced from the R-Schema :
- "a priori possible chainings" are obtained by analysing the ON, MODIF and TRIGGER parts of event and operation specifications.
  For a given event EVi, the chain is composed of all those events ascertaining relations modified by the operations triggered by EVi,
- in order to keep only "structurally possible chainings", the occurrence of each operation's TYPE (INSERT, DELETE, UPDATE) is checked within the ON part of the internal event(s) it seems to induce. So, impossible chainings like "an account closure produces an overdraft" will be removed from the graph.

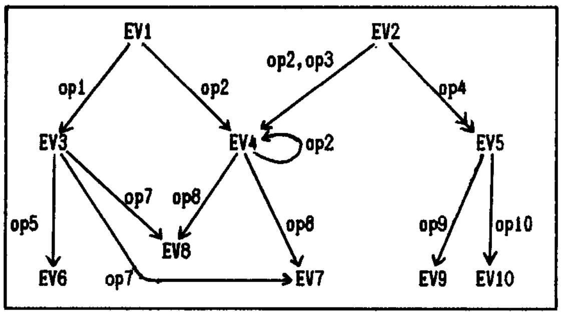


Fig. 14 : A Maximal Induction Graph

2nd STEP

The designer now manually modifies the Maximal Induction Graph, until he obtains the final Induction Graph.
During this step, the designer removes all the chainings that seem impossible to him from the graph.
For example, the EV4 event (cf. figure 3) seems to induce itself:

$$\begin{tikzcd} EV4 \arrow[loop right, "op2"] \end{tikzcd}$$

this is because EV4 ascertains a SAVINGS_ACCOUNT update, while OP2 updates SAVINGS_ACCOUNT. In reality, two EV4 occurrences will never be chained, because OP2 always produces a BALANCE decrease, while EV4 ("ceiling exceeded") can only occur when the BALANCE increases.
This kind of "false induction" cannot be detected automatically since it involves a semantical interpretation of predicates, conditions, factors and operations.

The final Induction Graph is an optimized and generally non-connected graph, which contains only "semantically possible chainings". Figure 15 presents the Induction Graph corresponding to the Maximal Induction Graph of figure 14.

If there are cycles in the Induction Graph, they are detected automatically, and the designer is asked to a confirm an "impossible infinite loop".

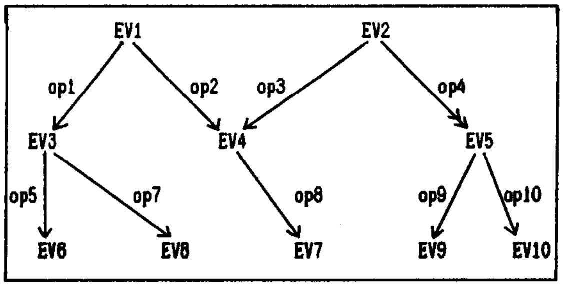


Fig. 15 : Induction Graph

C) Internal event chaining strategy

Given an external or temporal event to be processed, the chosen strategy is based on a "breadth-first" evaluation of the event Induction sub-graph.

** The induction sub-graph of an event is the maximal connected component, whose root is the event concerned.
By using this kind of sub-graph when an external or temporal event EVi occurs, the Event Processor can learn immediately what "the set of internal events it will probably have to process" is. This set of internal events is called the EVi Induction Class and is written C_{EVi}. For example, the EV1 Induction Class is :

$$C_{EV1} = \{ EV3, EV4, EV6, EV7, EV8 \}$$

** The internal event sequence construction is based on a breadth-first traversal of the Induction sub-graph.

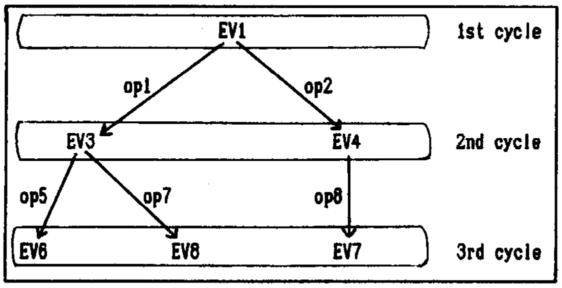


Fig. 16 : Internal event sequence

For example, if EV1 occurs, the complete processing cycle will include :

1st cycle: EV1
2nd cycle: EV3 + EV4
3rd cycle: EV6 + EV7 + EV8

It means that within each cycle, all events from the same level are processed.

The processing is the same if there is multiple induction. For example, if EV2 occurs, the sequence will be :

1st cycle: EV2
2nd cycle: EV4 + $EV5_1$ + $EV5_2$ + ... + $EV5_n$
3rd cycle: EV7 + $EV9_1$ +...+ $EV9_m$ + $EV10_1$ +...+ $EV10_p$

II.4.2 GLOBAL ALGORITHM OF THE EVENT PROCESSOR

```
WHILE TRUE
DO BEGIN
 WHILE there are some messages in the Queue
 DO BEGIN
            /* . take 1st message in the Queue;           */
  PROCESS 1; /* . generate appropriate external-event     */
            /*   and message references;                  */

  WHILE there are some event references
  DO BEGIN /* beginning of basic cycle */
             /* For all event references do :              */
             /* . identify operations to trigger,          */
             /* . evaluate all triggering conditions       */
   PROCESS 2; /*   and factors,                            */
             /* . generate "operations to execute"         */
             /*   references,                              */
             /* . generate "may-be-induced" event          */
             /*   references;                              */

   IF there are "operations to execute" references
   THEN BEGIN
                  /* . execute all operations,            */
                  /* . generate references for those      */
        PROCESS 3; /*   state changes which may           */
                  /*   correspond to induced events;      */

     IF there are "may-be-induced" event references
     THEN
                     /* . evaluate their predicate        */
                     /* . generate references for         */
           PROCESS 4; /*   recognized events and          */
                     /*   correponding state changes;     */
     END;
  END; /* of basic cycle */
 END;
END;
```

Fig. 17 : Global Algorithm of the Event Processor

The requirements for this algorithm (given in PROQUEL) are :

1) existence of a Meta-Base describing the R-Schema part containing all information on events, operations and the Induction Graph.
2) Management of References containing local information for each Event Processor cycle :
   - event occurrences (recognized event instances),
   - "operations to execute" instances,
   - "may-be-induced" events (not yet recognized event instances),
   - messages,
   - factor results,
   - state changes that may correspond to internal events,
   - state changes that correspond in fact to internal events.

II.4.3 SOME REMARKS ON EVENT PROCESSOR PERFORMANCE

1) Strategy

The "breadth-first" strategy (e.g EV1, EV3+EV4, EV6+EV7+EV8) has a real advantage over a "depth-first" (EV1, EV3, EV6, EV4, EV7) or a "random" strategy (EV1, EV3, EV6, EV4, EV7, EV8). Indeed, this strategy permits optimal management of the input/output implicit parameters. Idle time between :

- generation of an "operation output parameter",
- and its use as input parameter to process the event induced by this operation,

is minimal. Internal events are recognized as soon as "noticeable state changes" occur (in fact : just after all operations triggered at the same level have been executed); and these events are processed as soon as they are recognized (i.e during the next basic cycle of the event processor).

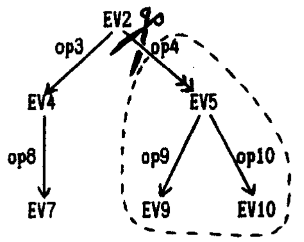
In this manner, there is no parameter waiting for use during a complete basic cycle. This is not true with other strategies; for instance, in the "depth-first" strategy, the EV4 input parameters must be kept in memory as long as EV3 and EV6 are still being processed.

2) The basic cycle

Recognizing induced events at the end of the basic cycle (cf. PROCESS 4) allows us to keep only these state changes corresponding to real internal events in memory from one cycle to another. The set of such state changes is called minimal context and contains only those input parameters essential to the next level of processing.

3) Using the Induction Graph

During the generation of "operations to execute" references (cf. PROCESS 2), a preliminary selection of "may-be-induced" events may be made. For example, as early as the first cycle of EV2 processing, if OP4 isn't in the list of operations to execute, a whole part of the EV2 Induction sub-graph can be pruned off :



So it permits :

- avoidance of useless predicate tests (EV5, EV9, EV10),
- avoidance of useless parameter recording,
- an earlier freeing of resources ("read-locked" relations for predicate, condition and factor evaluation; "write-locked" relations for operation execution).

It appears that an external or temporal Induction Class will become smaller and smaller after each basic cycle, and will finally reach the empty state. (another external or temporal event will then be processed.)

At any moment, the Event Processor knows what it is processing and what it must deal with next; so it controls the whole process fully.

In a future version, this will surely provide efficient management of parallelism and concurrency. Now, the parallelism criterion we are implementing is based on an "occurrence-time" reservation of all resources needed for each external event processing. Several external events can be processed together if they or their induced events can't produce any concurrency conflict. The dynamic release of the resources (cf. pruning) provides real-time management of concurrency.

CONCLUSIONS

This paper has presented RUBIS as an extended relational DBMS meant to :

* allow uniform and modular description of data (relations) and processing (events and operations),
* immediately take into account any schema change : all texts are interpreted and the schema is stored in meta-relations,
* automatically recognize predefined situations connected with :
  - external information arrival,
  - noticeable internal state changes,
  - time flow;
* automatically trigger appropriate operations and control their execution,
* synchronize event processing.

Such a system permits better management of applications consistency, and some transaction-writing facilities (which implies an error rate reduction).

Four kinds of Designer interfaces are actually being developped:

- a menu interface based on the PROQUEL language (introduced in section I),
- a graphic interface using an icon-based representation of the R-Schema concepts,
- a Natural Language interface based on a French Language subset [ROLL86],
- a Semantic Model interface managing high level concepts (such as generalization) which are then mapped onto the Relational Model [CAUV86].

A prototype version of RUBIS (based on the PEPIN DBMS [BOUC81]) is running on VAX/UNIX. The first applications of RUBIS are Information System Rapid Prototyping. In this kind of application, the RUBIS' schema modification capabilities are useful.

REFERENCES

[ADIB86] ADIBA M., BUI QUANK N. : "Aspects Historiques dans les Bases de Données Généralisées" Conf. BD3 1986, Giens FRANCE, INRIA 1986.

[BARB85] BARBIC F., PERNICI B. : "Time Modeling in Office Information Systems" ACM SIGMOD 85, Austin, Texas, May 1985.

[BOLO82] BOLOUR A. and al. : "The Role of Time in Information Processing : A Survey" SIGART Newsletters, Apr. 1982.

[BORG85] BORGIDA A. :"Features of Languages for the Development of Information Systems at the Conceptual Level" IEEE Software, Vol. 2, N° 1, Jan. 1985.

[BOUC81] BOUCHET and al. : "Databases for Microcomputers : the PEPIN Approach" ACM SIGMOD/SIGSMALLS, Orlando, Florida, Oct.1981.

[BUNE79] BUNEMAN O.P, CLEMONS E.K : "Efficiently Monitoring Relational Databases" ACM TODS Vol.4, N°3, Sept. 1979.

[BROD82] BRODIE M.L, SILVA E. : "Active and Passive Component Modeling" in [CRIS1].

[CAUV86] CAUVET C., LINGAT J.Y., NOBECOURT P.: "RUBIS: a DBMS for the description and management of Data Dynamics" Proc. Information Communication 86, PARIS, June 1986.

[CHAN82] CHANG J.M, CHANG S.K : "Database Alerting Techniques for Office Activities" IEEE COMM Vol.30, N°1, Jan. 1982.

[CRIS 1] "Information Systems Design Methodologies : a Software, Vol. 2, N° 1, Jan. 1985.

[DADA84] DADAM P., LUM V., WERNER H.D. : "Integration of Time Versions into a Relational Database System" VLDB 1984.

[ESWA76] ESWARAN K.P : "Specifications, Implementations and Interactions of a Trigger Subsystem in an integrated Data Base System" IBM Research Report RJ1820, Aug 1976.

[GUST82] GUSTAFSON M., KARLSON T., BUBENKO Jr J. : "A Declarative Approach to Conceptual Information Modeling" in [CRIS1].

[HAMM78] HAMMER M., Mc LEOD D. : "The Semantic Data Model: a Modelling mechanism for Database Applications" Proc. 1978 ACM SIGMOD Conf. on Management of Data.

[KAHN77] KAHN K., GORRY G. A. : "Mechanizing Temporal Knowledge" Artificial Intelligence, Vol.9, N°1, Aug. 1977.

[KERS86] KERSTEN M.L, SCHIPPERS F.H : "Using the Guardian Paradigm to support Database evolution" IFIP TC2 working conf. on Knowledge & Data

[KING86] KING R. : "A Database Management System Based on an Object-Oriented Model" Proc. Expert Database System Conf, 1986.

[LIN 84] LIN W.K, RIES D.R, BLAUSTEIN B.T, CHILENSKAS R.M : "Office Procedures as a distributed Database Application" Data Base, Vol 15, n°2, 1984.

[MELK83] MELKANOFF M., CHEN Q. : "Integrating Action Capabilities into Information Databases" 2nd Int. Conf. on Data Bases, Cambridge, Sept 83.

[MITT82] MITTAL S. : "Event-based Organization of temporal Databases" Proc. CSCSI/SCEIO Conf. 82, Saskatoon, Saskatchewan, 17-19 May 1982.

[MYLO80] MYLOPOULOS J., WONG H. : "Some features of the TAXIS Data Model" VLDB 1980.

[OVER82] OVERMYER R., STONEBRAKER M. : "Implementation of a Time-Expert in a Database System" ACM SIGMOD Vol.12, N°3, Apr. 1982.

[ROLL82] ROLLAND C., RICHARD C. : "The REMORA Methodology for Information Systems Design and Management" in [CRIS1].

[ROLL86] ROLLAND C., PROIX C. : "An Expert System Approach to Information System Design" IFIP Conf. 1986, Dublin, Oct. 1986.

[SMIT77] SMITH J.M, SMITH D. : "Database Abstractions : Aggregation and Generalization" ACM TODS 1977.