Temporal Conditions and Integrity Constraints in Active Database Systems *

A. Prasad Sistla, Ouri Wolfson
Electrical Engineering and Computer Science Department
University of Illinois
Chicago, Illinois 60680

Abstract

In this paper, we present a unified formalism, based on Past Temporal Logic, for specifying conditions and events in the rules for active database system. This language permits specification of many time varying properties of database systems. It also permits specification of temporal aggregates. We present an efficient incremental algorithm for detecting conditions specified in this language. The given algorithm, for a subclass of the logic, was implemented on top of Sybase. [0]

1 Introduction

The most popular model of rules in active database systems is the ECA model [41, 6, 28, 5, 14, 18]. It defines a rule to consist of three parts, event, condition, and action. The semantics is that whenever the event happens, the condition (which is usually a database query) is evaluated, and if satisfied then the action is taken. The event may be composite and temporal, such as, transaction A starts after transaction B ended. However, the condition is static in the sense that it refers to the current database state, but not to the way the database evolves over time. For example, the condition cannot specify that the value of a certain object in the database increases by 2% in 2 minutes. Additionally, this separation of event and condition parts introduces a contrived dichotomy between events and database states, since there are natural conditions which involve both, events and predicates on the database state [1]. For example, the temporal condition— "the value of attribute A remains positive while user X is logged in" means that the value of A remains positive between the X-login event, and the X-logout event.

In this paper we propose to eliminate the distinction between the event and the condition parts in rules, and replace both of them by the notion of a temporal-condition. The temporal-condition enables the specification of both, temporal events and conditions on the evolving database, and thus is more powerful than existing rule languages. We use Past Temporal Logic (PTL) for specifying temporal conditions. It allows us to specify conditions that involve both, events and database states, and their evolution over time. Basically, PTL consists of a regular query language augmented with a set of temporal operators, such as Previously and Since . Consequently, PTL is independent of the data model, or, in other words, it can be combined with any query language.

The logic proposed in this paper also allows temporal aggregates, i.e. aggregate functions over time. Therefore, we can specify a condition such as: "The moving average of a stock price in the last 20 minutes exceeds 50". Or, "the Dow Jones Industrial Average fell more than 250 points in the last 2 hours."

In addition to the language, we also present an efficient algorithm for incremental evaluation of the temporal conditions. The evaluation is incremental

*This research was supported in part by Grants NSF IRI-9224605, NSF CCR-9212183, AFSOR F49620-93-1-0059

[0]

[1] One may argue that in addition to semantics, the separation between event and condition parts was advocated from efficiency point of view, i.e. the condition part is going to be evaluated only when the event occurs. However, even if we combine these two, we will explain that we can still maintain the efficiency by first analyzing the events and going to the database only when necessary.

in the sense that when a new database state is created as a result of an update, the algorithm only considers the changes in the new database state in order to determine if the condition is satisfied, instead of considering the whole database history. The evaluation algorithm is also an add-on component, executed on top of, and using the existing query processing system.

Temporal integrity constraints are temporal conditions that must be satisfied at every commit point of a transaction. We propose a new framework for processing temporal integrity constraints. Existing methods for processing such constraints ([17, 38]) translate them into a set of rules with nontemporal conditions. In contrast, we process integrity constraints "from first principles" so to speak. Our rule conditions are temporal, thus we use the algorithm that processes temporal conditions for both, rule processing and integrity constraints processing.

We show that our proposed language and processing algorithm can work with *valid time* as well as *transaction time.* The important distinction between the two notions of time was introduced in the context of temporal databases [31], and is critical in rule and constraint processing, in the following sense. The database is usually a model of the real-world and the time at which changes happen in the real-world, namely the valid time, may precede the time at which these changes are posted in the database, namely the transaction time. Furthermore, the temporal conditions may refer to the valid time rather than the transaction time. For example, consider a transaction that updates the price of a stock, i.e., posts to the database the price of a stock sale. The sale may have occurred five minutes before the change in the stock price is committed to the database. It is easy to see that there are triggers that will fire with respect to the valid time but will not fire with respect to the transaction time, or vice versa. For example, consider the temporal condition: "the stock price remains constant for seven minutes". This condition may be satisfied with respect to the transaction time, but not with respect to the valid time, and vice versa.

We show that in the valid time model temporal conditions in rules and integrity constraints can have different semantics. For example, we show that an integrity constraint with respect to a database history can be *online* satisfied but not *offline* satisfied, and vice versa. Similarly, we show that execution of an action in a rule can be based on either *tentative* or *definite* history, and our language and processing algorithm can accommodate both notions.

We also demonstrate that our rule formalism also enables the specification of temporal actions, i.e., composite actions built from atomic ones using temporal operators. For example, the user can specify that when a certain condition is satisfied, the system should execute the BUY-STOCK transaction every 10 minutes (in order to prevent driving up the stock-price), as long as the stock-price remains below 50. Previously, such temporal actions required extended transactions models ([9, 10, 13]) that must be controlled by an additional software system running on top of the database management system. The temporal condition formalism also requires an additional software system, the condition evaluator, but this is the same system used for rule processing, integrity constraint enforcement, and temporal actions. Furthermore, the specification of temporal actions involving iteration and absolute and relative timing properties (every 10 minutes) is not straightforward in existing models of extended transactions.

In summary, the contributions of this paper are as follows.

- We introduce a temporal-logic based language and incremental processing algorithm. The language can be used for temporal integrity constraints, and for rules that are simpler (only involve conditions and actions) and more powerful than existing rule languages. It also supports temporal aggregates, and it is independent of the underlying data model.
- We define various semantics for rules and integrity constraints with respect to two notions of time, valid time and transaction time. We show how to use the processing algorithm for the various semantics.
- We show that the temporal conditions can be used for the specification of temporal actions, which are a form of extended transactions.

This paper is organized as follows. Section 2 describes the model that we use; section 3 introduces temporal rule systems. Section 4 defines the basic Past Temporal Logic (PTL) without aggregates. Section 5 presents the processing algorithm for PTL. Section 6 incorporates temporal aggregates in to PTL and shows how they can be processed. Section 7 shows how temporal and composite actions can be specified using our formalism. Section 8 describes the execution model. Section 9 shows how we can handle valid time model. Sections 10 and 11 contain comparison to other work and conclusions, respectively.

2 Transaction Time System Model

A *database* consists of a set of database items. These are names of relations or object classes. Additionally, there is a set U of events that can occur at any time. For example, Transaction-begin, Transaction-commit, Rule-execute, Insert-tuple etc., are some of the events. Many of these events may be parameterized. For example, the Transaction-begin event may have a parameter denoting the id of the particular transaction. We assume that the events are instantaneous. Temporal conditions may involve both, database items and events.

A *database state* is a mapping that associates a value from the appropriate domain with each database item. A *system state* is a pair (S, E) where S is the database state and E is the set of events; intuitively, a system state is a snapshot of the system giving the database state and the set of events that occur at a particular instant. The formulas of the temporal logic, which we define later, are interpreted over system histories. A *system history* is a finite sequence $(S_0, E_0), ..., (S_i, E_i)$ of system states. A new system state is added to the history whenever an event occurs. Two or more events may occur simultaneously, but if so, then a single new database state is added to the history. We impose some constraints on system histories. First, we require that no two transactions commit simultaneously, i.e. the event set in each system state may contain at most one Transaction-commit event. Secondly, the database states of two consecutive system states are identical, unless the event set E contains the commit of a transaction. If so, then the new database state reflects all and only the database changes made by the transaction. In other words, the new database state is identical to the previous one, except for the changes to the database made by the transaction.

A *time stamp* is associated with each system state. It denotes the time at which the change to the previous state occurred, i.e. the time at which the event that produced the new system state occurred, according to a fixed global clock. We assume that the value of this time stamp is given by a data-item called *time*. Since events that occur simultaneously produce a single new database state, the time stamps along the history are strictly increasing.

3 Temporal Logic Rule System

In this paper we propose the Condition-Action (C-A) model of a rule. The condition part is specified by a Past Temporal Logic (PTL) formula. The PTL formula can use various past temporal operators and propositional connectives, events, database predicates involving database items, variables and temporal aggregate functions. The PTL formulas can also refer to a special predicate on rule executions, called *executed*, whose importance will be explained later. The variables can be instantiated inside the formula by binding them to a database query. They are, essentially, used to compare query values at different instances of time, and can be used to specify various patterns of change in the database state over time. The temporal aggregates allow us to use aggregates of various database query values over time. The condition part can have some free variables (i.e. variables which are not instantiated inside the condition). In this case, any assignment of appropriate values to the free variables can cause the condition to be satisfied and the rule to be fired.

The action part of our C-A rules may be a database operation, a program, or it may simply be an abort operation on the current transaction. Furthermore, the action part can refer to some of the free variables referred to in the condition part. This facilitates parameter passing from the condition part to the action part which is often very useful. This aspect of our rule system is not different than existing nontemporal active databases. Also, the options on the coupling between rule executions and transactions (see [24]) are the same for nontemporal and temporal active databases, except that in our model the event component of the rule is eliminated.

A rule is either a trigger or an integrity constraint. An *integrity constraint* is a rule in which the action is $abort(X)$, and the condition consists of the event $attempts\_to\_commit(X)$, and the negation of the integrity constraint. The rule is executed as part of each transaction when the transaction attempts to commit. A *trigger* is any other type of rule.

4 Past Temporal Logic (PTL)

4.1 Syntax of PTL Formulas

The formulas of PTL use two basic past temporal operators Since and Lasttime . Other temporal operators, such as Previously and Throughout_the_Past , can be expressed in terms of the basic operators. The symbols used in the logic are the following: temporal operators Since ; Lasttime ; Previously ; Throughout_the_Past ; boolean connectives $\neg, \wedge, \vee$; propositional constants $true$, $false$; assignment operator $\leftarrow$; variables denoted by X,Y,...; function symbols denoting database queries, relation symbols, event symbols denoting various events; integers and standard operations on integers etc. We assume that each function symbol and relation symbol has an associated arity denoting the number of parameters to it. Similarly, each event symbol has an associated arity with it. Each n-ary event symbol when used with actual parameter values specifies a single event. For example, Transaction-begin(30) denotes the beginning of transaction 30. Each n-ary function symbol denotes a function that takes n-arguments of particular types, and returns a value. For example, + and * are function symbols denoting addition and multiplication on the integer type. Similarly, $\leq, \geq$ are binary relation symbols denoting arithmetic comparison operators. The functions symbols are also used to denote queries on databases. For example, consider the following relational query, OVERPRICED, that retrieves all stocks with a price above a certain threshold from a relation called STOCK-FOR-SALE (the relation contains tuples consisting of a stock name, the price, the company's name and the category of products the company produces):
RETRIEVE (STOCK-FOR-SALE.name) WHERE STOCK-FOR-SALE.price$\geq$300
The above query is represented by the function symbol OVERPRICED, that takes a single argument which is a 4-ary relation, and returns a unary relation.

We first define the syntax of *terms* in the logic. Every variable and constant is a term. If f is a n-ary function then $f(t_1, ..., t_n)$ is a term where $t_1, ..., t_n$ are terms of appropriate type.

Now, we define a PTL formula recursively. If t_1, t_2 are terms and ρ is a comparison operator, $\rho \in \{>, \geq, <, \leq, =\}$, then $t_1 \ \rho \ t_2$ is a PTL formula. If R is a n-ary relation symbol or an n-ary event symbol and $t_1, \ldots, t_n$ are terms, then $R(t_1, \ldots, t_n)$ is a PTL formula. We call the above type of formulas as *atomic formulas.* If g is a PTL formula, X is variable and t is a function, then $[X \leftarrow t]$g is a PTL formula. This denotes assignment of the query t to the variable X. All occurrence of X inside g are said to be bound. If g_1 and g_2 are PTL formulas, then $\neg g_1$, $g_1 \wedge g_2$, $g_1 \vee g_2$, Lasttime g_1, Previously g_1, g_1 Since g_2 and Throughout_the_Past g_1 are also PTL formulas. true and false are PTL formulas.

4.2 Semantics of PTL Formulas

The semantics of a PTL formula is defined with respect to a time-stamped system history. For a PTL formula f , let $free\_var(f)$ be the set of all variables that appear *free* in f, i.e. variables that are not bound in f. An evaluation ρ for f is a mapping which assigns appropriate values to the variables in $free\_var(f)$. The satisfaction of a formula f over a time-stamped system history $h = ((S_0, E_0), \ldots, (S_i, E_i))$ with respect to an evaluation ρ is defined inductively according to the following cases:

- If f is an *atomic formula* then f is satisfied by h with respect to ρ iff the system state (S_i, E_i) satisfies f with respect to ρ. More specifically, if f is an event then it is satisfied by h if the event f is in E_i. Otherwise, f is satisfied by h if the database state S_i satisfies f.
- If f is of the form $f = f_1 \wedge f_2$, or $f = f_1 \vee f_2$, or $f = \neg f_1$, then the satisfaction of f by h depends on, respectively, the fact that database state S_i satisfies both f_1 and f_2, or S_i does not satisfy f_1, or S_i satisfies either f_1 or f_2, with respect to ρ.
- If f is of the form $[X_1 \leftarrow t_1]f_1$, then h satisfies f with respect to ρ iff h satisfies f_1 with respect to ρ_1 where ρ_1 is an evaluation for f_1 in which $\rho_1(X_1)$ = value of function t_1 at database state S_i, and for all other variables $X \neq X_1$, $\rho_1(X) = \rho(X)$.
- If f = Lasttime f_1, then f is satisfied by h with respect to ρ iff history $((S_0, E_0), \ldots, (S_{i-1}, E_{i-1}))$ satisfies f_1 with respect to ρ. If $i = 0$, then f = false.
- If $f = f_1$ Since f_2, then f is satisfied by h with respect to ρ iff there exists a $j \leq i$, such that history $((S_0, E_0), \ldots, (S_j, E_j))$ satisfies f_2 with respect to ρ, and for all $k, j < k \leq i$, history $((S_0, E_0), \ldots, (S_k, E_k))$ satisfies f_1 with respect to ρ.
- If f = Previously f_1, then $f \equiv$ true Since f_1.

- If $f =$ Throughout_the_Past f_1, then $f \equiv \neg$ Previously $\neg f_1$.

4.3 Examples

The following formula, called SHARP-INCREASE, uses the assignment quantifier and the Previously operator. In this formula, $price(Motorola)$ is a database query that retrieves the price of the Motorola stock. The formula indicates that the price for Motorola stock increases by 10% within 30 minutes. The formula states that there exists a past state such that: if X, t denote the values of price(Motorola) and $time$ in current state respectively, then there exists a past state before it in which the value of price(Motorola) is at most 90% of X and the time is more than or equal to $t - 10$. Here, X and t are variables. We split the formula over two lines.

(A) $[t \leftarrow time]$ $[X \leftarrow price(Motorola)]$
Previously $(price(Motorola) \leq 0.9X \wedge time \geq t - 30)$.

We can specify multiple conditions in one formula by allowing free variables[2]. For example, the following formula uses the free variable Y to specify a sharp increase in price of the company denoted by Y.

(B) $[t \leftarrow time][X \leftarrow price(Y)]$
(Previously $(price(Y) \leq .9x \wedge time \geq t - 30)$.

In the above formula, the variable Y can be constrained to Computer companies by binding Y to a query that retrieves all names of all computer companies.

The following PTL formula is an example of a condition that uses external events. This condition specifies that the value of attribute A has gone above 50 while user X is logged in. Note that the second conjunct in the formula denotes that user X is still logged in.

$$A > 50 \wedge (\neg user\_X\_logs\_out \text{ Since } user\_X\_logs\_in)$$

5 Algorithm for Evaluation of PTL Conditions

In this section, we outline an approach for detecting triggers specified in PTL. We also discuss some implementation and optimization techniques that can be applied.

[2] These free variables can also be used in the specification of the action part of a rule.

The Basic Algorithm

Let f be the given PTL formula. The basic idea of the algorithm is to eliminate the temporal operators in each subformula g of the trigger f, and express each such subformula as a boolean expression involving results of database queries on past database states and variables that appear free in g.

Let $s_1, s_2, ...s_i, ...$ be a database history where s_i is the system state after the i^{th} update since the trigger is entered. For each subformula g of f and for each $i \geq 1$, we inductively define a boolean formula $F_{g,i}$ as follows.

- If $g = R(t_1, ..., t_k)$ where R is an arithmetic relation symbol such as $\leq$, $\geq$ etc., then $F_{g,i} = R(val(t_1), ..., val(t_k))$ where $val(t_j)$, for $j = 1, ..., k$, is the result of evaluating the term t_j in the database state s_i. Any variables appearing in t_j are left as they are. $F_{g,i}$ may evaluate to the constants true or false, or it may be boolean valued expression involving variables.
- If $g = g_1 \wedge g_2$ then $F_{g,i} = F_{g_1,i} \wedge F_{g_2,i}$.
- If $g = \neg g_1$ then $F_{g,i} = \neg F_{g_1,i}$.
- If $g =$ Lasttime g_1 then $F_{g,i}$ is defined as follows. If $i = 1$, i.e. this is the first update after activation of the trigger, then $F_{g,i} =$ false. Otherwise, $F_{g,i} = F_{g_1,i-1}$.
- If $g = g_1$ Since g_2 then $F_{g,i}$ is defined as follows. If $i = 1$ then $F_{g,i} = F_{g_2,i}$. Otherwise, $F_{g,i} = F_{g_2,i} \vee (F_{g_1,i} \wedge F_{g,i-1})$.
- If $g = [x \leftarrow q]\ g_1$ then $F_{g,i} = h$ where h is the expression obtained after substituting the result of executing the query q in the database state s_i into the expression $F_{g_1,i}$.

Note that of $F_{g,i}$ can be computed incrementally and inductively using the values of $F_{g,i-1}$ and $F_{h,i}$ where h is a proper subformula of g. After computing $F_{g,i}$ for all subformulas g of f, the previous values $F_{g,i-1}$ for different g can all be deleted.

The trigger detection algorithm operates as follows. After the i^{th} update, it simply computes $F_{g,i}$ for each subformula g and fires the trigger iff the formula $F_{f,i}$ evaluates to true. Also, it discards the previous values $F_{g,i-1}$ for each subformula g.

The following theorem can be proved by induction on i.

THEOREM 1: The above algorithm fires the trigger after the i^{th} update iff the formula f is satisfied at state s_i.

We illustrate the operation of the above algorithm on the trigger condition given by the formula f given below. This formula is satisfied if any time the price of IBM stock doubled in 10 units of time.

$$[t \leftarrow time]\ [x \leftarrow price(IBM)]$$
$$\textsf{Previously}\ (price(IBM) \leq .5x \wedge time \geq t - 10).$$

Now, we define two subformulas h and g of f.

$$h = price(IBM) \leq .5x \wedge time \geq t - 10;$$
$$g = \textsf{Previously}\ h.$$

In the operation of the algorithm we use the equivalences $\textsf{Previously}\ h = true\ \textsf{Since}\ h$. We consider the following history where each system state is represented as a pair containing the value of $price(IBM)$ and $time$ in that order. Here we do not show the update events.
(10,1) (15,2) (18,5) (25,8)...

We give the values of $F_{h,i}, F_{g,i}$ and $F_{f,i}$ after simplification and rearrangement, for each $i = 1, 2, 3$ and 4.
$F_{h,1} = F_{g,1} = (10 \leq .5x \wedge 1 \geq t - 10);\ \ F_{f,1} = false;$
$F_{h,2} = (15 \leq .5x \wedge 2 \geq t - 10);\ \ F_{g,2} = F_{h,2} \vee F_{h,1};\ \ F_{f,2} = false;$
$F_{h,3} = (18 \leq .5x \wedge 5 \geq t - 10);\ \ F_{g,3} = F_{h,3} \vee F_{h,2} \vee F_{h,1};\ \ F_{f,3} = false;$
$F_{h,4} = (25 \leq .5x \wedge 8 \geq t - 10);\ \ F_{g,4} = F_{h,4} \vee F_{h,3} \vee F_{h,2} \vee F_{h,1};\ \ F_{f,4} = true;$

¿From the analysis, we see that $F_{f,4}$ evaluates to $true$ and the trigger gets fired.

Implementation Using Auxiliary Relations

It is to be noted that each of the formulas $F_{g,i}$ contains the results of database queries on previous database states. The values of these queries can be maintained in separate auxiliary relations and the query values can be retrieved from the auxiliary relations by a selection operation that uses time stamps. It is enough if these time stamps are maintained as part of the formulas $F_{g,i}$. Also, the formulas $F_{g,i}$ can be maintained as an *and-or* graph. The use of auxiliary relations is elaborated below.

We assume that, in the given PTL formula f, each bound variable x is assigned a query value at most once in the formula; if this condition is not satisfied, we can simply rename some of the ocuurrances of x to satisfy the condition. Furthermore, we assume that each reference to a database query in f is through a variable. Now, consider a variable x which is bound to the the query q. Note that the value retrived by q can be a scalar or it can be a relation. Assume that the query q retrieves a k-ary relation. Now, corresponding to x, we use an auxiliary relation R_x with $k+2$ attributes. This relation captures the values of the query q at different instances of time. The first k attributes of R_x correspond to the attributes of the relation retrived by q. The last two attributes, denoted by T_start and T_end, denote an interval of time during which the particular tuple in the relation is valid. Initially, i.e. when the trigger condition f is first entered at time T, R_x is set to the relation retrieved by q on the database at that time, and by having $T\_start = T$ and $T\_end = MAX$ in each tuple. . Later, as the value of query q changes (due to updates on the database), T_start and T_end are appropriately modified. It should be easy to see that the value of the query q at any previous time can be retrived by performing a selection, followed by a projection on R_x.

Optimization

Clearly, the size of the formulas $F_{g,i}$ keeps growing with i, i.e. with the number of updates. We can use optimization techniques to reduce the size of the formulas $F_{g,i}$. Suppose g has a clause of the form $t \leq c$ where t is free variable in g, c is a constant, and t is assigned the value of $time$, then we can use the following optimization technique. If the value of $time$ in s_i is greater than c, then it clearly is the case that the clause $t \leq c$ will never get satisfied in the future. In this case, we can replace the clause $t \leq c$ by the constant $false$ and simplify the formula. We illustrate this by considering the previous example for the following history.
(10,1) (15,2) (18,5) (11,20)...
The first three states in the history are same as before. Hence, $F_{h,i}, F_{g,i}$ and $F_{f,i}$ are same as before for $i = 1, 2, 3$. However,
$F_{h,4} = (t \leq 30 \wedge x \geq 22);\ \ F_{g,i} = F_{h,4} \vee F_{h,3} \vee F_{h,2} \vee F_{h,1};\ \ F_{f,4} = false;$

Since the value of $time$ in the last state is 20, it should be obvious that $F_{h,i}$ for $i = 1, 2, 3$ will never be satisfied when t is substituted by any future values of $time$, and hence we can replace all these clauses by $false$ and simplify $F_{g,4}$ to get $F_{g,4} = (t \leq 30 \wedge x \geq 22)$.

The above method applied to triggers formed using only bounded temporal operators, allows us to keep only bounded information from the past history.

6 Temporal Aggregates

6.1 Definition

We extend the basic PTL, defined earlier, by using temporal aggregate functions. These functions allow us to refer to different aggregates of database values over time. A temporal aggregate function has the form $f(q, \phi, \psi)$, where f is the name of the aggregate function, q is a database query, ϕ and ψ are PTL formulae. The above function denotes an aggregate of the database query q starting from the latest point when ϕ is satisfied. The formula ψ denotes the points (i.e. sampling points) that should be considered for computing the aggregate. We call ϕ and ψ as the *starting* formula and *sampling* formula, respectively.

The formal semantics for the above aggregate function can be defined as follows. Let $h = (S_0, E_0), (S_1, E_1) ..., (S_i, E_i)$ be a history, and j be the highest integer less than or equal to i such that the prefix of the history up to (S_j, E_j) satisfies ϕ. Also, let $k_1, k_2, ..., k_p$ be all the integers between j and i, such that the history up to the system state (S_{k_r}, E_{k_r}) satisfies ψ. Then, the value of $f(q, \phi, \psi)$ (at the end of above history) is the aggregate (as specified by f) of the values of q at the system states given by $k_1, k_2, ..., k_p$.

Now consider the stock market example. The function $sum(price(IBM), time = 540, time \textbf{ mod } 60 = 0)$ denotes the sum of the the prices of the IBM stock taken every hour since 9AM (i.e. $time = 540$); here, we are assuming that value of $time$ is given in minutes. The following expression denotes the hourly average of the IBM stock price since 9AM— $sum(price(IBM), time = 540, time \textbf{ mod } 60 = 0)/sum(1, time = 540, time \textbf{ mod } 60 = 0)$. Note that the denominator, in the above expression, denotes the number of hour boundaries since 9AM.

In the above example, by appropriately choosing the starting and sampling formulae, we can specify hourly or daily averages, or averages using sampling points where some condition is satisfied. Suppose, $update\_stocks$ denotes the $Transaction\_Commit$ event for a periodically run transaction that updates all stock prices. Then, the following expression denotes the average of IBM stock price since 9AM, where the sampling points are instances when stock prices are updated— $sum(price(IBM), time = 540, update\_stocks)/sum(1, time = 540, update\_stocks)$.

It is not difficult to see that one may need aggregate functions in which the starting and sampling formulas are temporal. In addition, in our formalism, aggregate functions can be nested, i.e. the starting and sampling formulas in an aggregate function may themself contain aggregate functions.

The following PTL formula asserts that the hourly average of the IBM price has remained above 70 since 9AM (i.e. time = 540 minutes). It is to be noted that, in this formula, the left side of the Since operator denotes the moving hourly average of the IBM stock price.

$$[u \leftarrow time]$$
$$(Avg(price(IBM), time \leq u - 60, update\_stocks) > 70)$$
$$\textsf{Since } time = 540$$

6.1.1 Extensions to the basic algorithm

Here we show how the previous algorithm can be extended to handle the temporal aggregates when the aggregates do not contain any free variables. Consider a rule r that uses the temporal aggregate $f(q, \phi, \psi)$ in its condition; here q is a database query, ϕ and ψ are PTL formulas. Assume that $f(q, \phi, \psi)$ does not contain any free variables. At any time the above aggregate denotes the aggregate of the value of the query q calculated since the latest time when ϕ is satisfied, and the aggregation is taken only at those times when ψ is satisfied. The computation of this aggregate can be programmed into our rule system. We first introduce a new database item F and replace every occurrence of the function $f(q, \phi, \psi)$ in the condition of r by F. Next, we introduce two new rules that update F. F is initialized by a rule with condition ϕ, and is updated by a rule with condition ψ. We illustrate this construction using the following example. Consider the following rule r.

$$(Avg(price(IBM), time = 9AM, update\_stocks) > 70) \rightarrow A$$

The above trigger fires if the the average price of the IBM stock since 9AM is higher than 70. We replace the average by $sum(price(IBM), time = 9AM, update\_stocks)/sum(1, time = 9AM, update\_stocks)$.
Since we have two different aggregates, we introduce two new database items –CUM_PRICE and $TOTAL\_UPDATES$. The single rule r is replaced by the following three rules.

$$r: \quad (CUM\_PRICE/TOTAL\_UPDATES > 70) \rightarrow A;$$

$r_1: \ time = 9AM \ \rightarrow$
$CUM\_PRICE := 0 \ ; TOTAL\_UPDATES := 0;$
$r_2: \ update\_stocks \ \rightarrow$
$CUM\_PRICE = CUM\_PRICE + price(IBM)$
$TOTAL\_UPDATES++;$

It is easy to see that the above approach can be used, for any temporal condition, if the aggregates do not have any free variables. Note that all of the above can be done automatically.

In a more general case, i.e. when $f(q, \phi, \psi)$ has free variables, we need to have multiple database items, indexed with different values for the free variables. For example, the following rule $r(x)$, which is similar to r except that the stock name IBM is replaced by a free variable x.

$$(Avg(price(x), time = 9AM, update\_stocks) > 70) \rightarrow A$$

To handle the above rule, we introduce two sets of database items, $CUM\_PRICE(x)$ and $TOTAL\_UPDATES(x)$, corresponding to the different values of x. As before, the rule $r(x)$ will then be replaced by three rules $r(x), r_1(x)$ and $r_2(x)$.

7 Temporal and Composite Actions

In this section, we describe how certain types of composite and temporal actions can be implemented elegantly by using a special predicate *executed* as part of the temporal conditions in rules. Essentially, the *executed* predicate denotes which rules were executed at what time. The *executed* predicate takes three arguments r, x, t where r is a rule name, x is a list of parameters and t is time. The predicate $executed(r, x, t)$ is satisfied at time T if $t < T$ and the rule r was executed with parameter list given by x (i.e. the condition part of r was satisfied, the rule was fired and the action part of the rule was committed by the time t);

Now, we show how the composite action can be expressed using the *executed* predicate. A composite action is specified by a set of atomic actions together with a partial order on them and a set of timing constraints on their execution. The partial denotes the order in which the individual atomic actions should be executed. Consider the a composite action $A(x)$ which consists of two atomic actions $A_1(x)$ and $A_2(x)$. Here x denotes the parameters that are passed to the action. The timing constraints require that A_2 be executed ten minutes after the execution of A_1. Now consider the following rule r.
$r: C(x) \quad \rightarrow \quad A(x)$
The rule r can be implemented by the following two rules r_1 and r_2.

$$r_1: C(x) \quad \rightarrow \quad A_1(x)$$
$$r_2: executed(r_1, x, t) \wedge time = t + 10 \ \rightarrow \ A_2(x)$$

Many types of dependencies between commitment of different transactions/actions of a nested transactions have been studied in literature. These dependencies can be programmed into our formalism by using the different types of predicates/events. Timing dependencies among them can also be specified elegantly in our formalism.

We can also use the *executed* predicate for programming certain types of temporal actions. For example, a rule r may say that whenever condition C is satisfied execute an atomic action A every ten minutes for the next one hour. For example, the condition C might say that $price(IBM) < 60$ and action A may be buying of 50 IBM stocks[3]. The following two rules together implement the above rule.

$$r_1: C \quad \rightarrow \quad A$$
$$r_2: executed(r_1, t) \wedge (time - t \leq 60) \wedge (time - t) \ mod \ 10 \ = 0 \ \rightarrow \ A$$

The *executed* predicate can be handled as follows. The temporal component needs to maintain an additional auxiliary relation denoting containing the information about the execution of each rule, including the parameter values and time. When this is done, we simply consider *executed* as a predicate on this auxiliary relation. In this auxiliary relation, only information necessary for future evaluation of conditions will be maintained. All other information will be removed as and when it is not needed.

All the optimizations discussed in the previous sections can also be applied to the extensions discussed here.

8 Execution Model

In this section we will address some issues about the relationship between the user transactions and

[3] Actually, this temporal action can be specified in future temporal logic cite SW93.

the execution of the rules. As mentioned, the user can specify for each rule any possible coupling between user transactions and rules. For example, TC-A denotes the coupling in which the condition of the rule is executed as part of the user transaction, right before the latter's commitment, i.e. when an event *attempts_to_commit* occurs. Integrity constraints are TCA rules, i.e. their condition and action are executed as part of the user's transaction.

Now consider T-CA or T-C-A rules, i.e. rules that are executed independently of user transactions. These rules seem to be more natural in temporal rule systems, since the condition satisfaction is usually not attributable to a single transaction, but is the cumulative effect of the execution of many transactions over time. In principle, each one of these independent rules are evaluated whenever a new system state is added to the history, i.e., whenever an event occurs. Thus, whenever an event occurs the database management system invokes the *temporal component*, i.e. a system that executes the temporal condition evaluation algorithm for each trigger. Consistency constraints are evaluated only when the $attempt_t o_c ommit$ event occurs.

Obviously, this is only a schematic way of viewing the evaluation process, and many optimizations are possible. For example, rules may be associated with relations or object classes, and evaluated only when an event relating to the object class (e.g. an update of an object in the class) occurs.

In the ECA rule model the Event part played a performance optimization role, by forcing condition evaluation to be executed only when the respective event occurs. We can obtain the same effect in our model as follows. Rules that refer in the condition part to events are considered only when the respective events occur, and disregarded otherwise. Rules that do not refer to events, only to database evolution over time, are considered only at commit points, [4] since this is the only time when the database changes. Rules that refer only to database histories, but their condition can be satisfied only as a result of the execution of some transaction T, can add to the condition part the event $commit(T)$, and thus constrain the evaluation to occur only when T commits.

Thus, whenever an event occurs, the temporal component considers only the relevant triggers. Even so, the overhead may become excessive, and trigger evaluation may fall behind event occurrence. To overcome this problem the temporal component invocation can be executed for multiple events at the same time. The only implication of such an action is that trigger firing may be delayed, but not go unrecognized.

[4] A *commit point* in a history h is a state that contains the commit transaction event.

9 Valid-time system model

9.1 The model

In the model we discussed so far we considered the transaction time rather than the valid time. Specifically, our model and the temporal conditions referred to the time at which the values assigned to database items are committed.

In order to accommodate triggers that refer to the valid time we will assume that that every update presented to the database management system is associated with a valid time, and this valid time may precede the current time. For example, the update (IBM, 72) indicating that the price of the IBM stock is 72 may occur at 1pm and have a valid time of 12:50pm. We assume that when this happens the database management system makes the change retroactively.

In order to distinguish in the system history between the valid time and the transaction time, we modify our model. The valid time model is identical to the transaction time model, except for the following revisions. An update of the database is an event in both models. However, in contrast to the transaction time model, the event is assumed to have occurred at the valid time, say v. If there is a system state with time-stamp v then the update is added to the set of events E of that system state. Otherwise a new system state is added to the history with time-stamp v.

Furthermore, in the valid time model we do not assume that the changes occur at commit time, but at the valid time. Specifically, the database state of two consecutive system states is identical, unless the event set E contains an update of the database. If so, then the new database state reflects all and only the database changes made by the updates (possibly from different transactions) that appear in E. In other words, the new database state is identical to the previous one, except for the changes to the database made by the updates that occurred at that instance in time.

In the valid-time model we are still interested only in committed updates, and we ignore updates of aborted transactions. Indeed, it does not make sense to fire a trigger based on updates that will be aborted. However, the time of the committed updates is not the time of the transaction commit as in the model discussed in previous sections, but the time at which the update occurs. This is a subtle but important distinction, which, as explained in the introduction, may make a trigger fire in one model but not in the other.

In order to restrict attention to committed updates we first need to define the concept of a committed system history. Intuitively, given a system history h, the committed system history at a time t is a subhistory consisting of the changes made by transactions that committed before t. Formally, given a system history h, an update made by a transaction T is *committed in h* if h contains a system state in which the commit of T appears. Given a system history h, a *committed history* at time t is the prefix p of h consisting of states having a time-stamp not higher than t, in which each system state that contains an update that is uncommitted in p is modified as follows. The database state in the system state is modified to eliminate the effects of the each uncommitted update.

9.2 Triggers

Now consider rules in the valid time model. Can a trigger fire, i.e. the action part of the rule be executed, when the temporal condition is satisfied by a committed history? If the history can be modified retroactively at any point in time in the future, this means that any history is tentative in the sense that each database value is uncertain, and furthermore, it remains tentative forever.

Suppose that triggers are *tentative* in the sense that their actions can be taken based on tentative values. In order to evaluate tentative triggers, the temporal component algorithm is executed when a system state is added to the history, as before. However, the temporal component does not consider only the latest system state. It incrementally performs the evaluation algorithm for each state starting with the oldest system state that was updated by the transaction, until the last system state in the history. If the condition is satisfied for one of these states, then the trigger is satisfied and the action is executed.

Now assume that tentative values can be queried and viewed, but actions can only be based on definite values. When do values become definite? To answer this question we introduce the concept of a maximum delay, denoted τ. It means that for every update, the maximum difference between the current time and the valid time is at most τ. In other words, an update cannot make a retroactive change which goes back more than τ time units. The implication of the maximum delay assumption is that a committed value in the database can be in one of two states, definite or tentative. If the commitment occurred less than τ time units ago, the value is tentative, otherwise it is definite.

Suppose that triggers are *definite* in the sense that their actions can be taken based only on definite values. In this case a trigger fires if its temporal condition is satisfied by a history that is at least τ time units old. Observe that in this case the firing is delayed, and it occurs at least τ after the condition is satisfied. In other words, definite triggers inherently imply a delayed firing.

In order to evaluate definite triggers, the temporal component algorithm is still executed whenever a system state is added to the history. However, it only considers the system states that have a time-stamp that is at least τ time units smaller than the current time. Since the algorithm is incremental, it actually considers only the system states that have not been considered in the prior invocation of the algorithm.

9.3 Integrity constraints

When does a committed history satisfy a temporal integrity constraint? It turns out that there are two possible ways to define the notion of integrity constraint satisfaction, one is offline and the other is online. Suppose that h is a *complete* history, i.e. an history in which every started transaction is either committed or aborted. A temporal integrity constraint c is *online-satisfied* in h if the temporal formula c is satisfied by the committed history at time t, for all times t which denote commit points of transactions. A temporal integrity constraint c is *offline-satisfied* in h if the following condition is satisfied for the committed history at time infinity, h'. For all times t which denote commit points of transactions the temporal formula c is satisfied by the committed history at time t.

Intuitively, the difference between the two notions of satisfaction is the following. Offline satisfaction means that the condition is satisfied at every commit point p; the history up to p contains all updates, including the updates of transactions that commit after p. Note that these updates are not "visible" at time p. In contrast, online satisfaction means that the condition is satisfied at every commit point

p; the history up to p contains only the updates of transactions that commit at p or before p.

Now let us demonstrate that the two satisfaction notions are different. Consider for example the following temporal condition. Whenever update u2 occurs in the history, it is preceded by update u1. Now suppose that update u1 is issued by transaction T1, and update u2 by is issued by transaction T2, and consider the history in which the events occur in the following order:
u1, u2, commit-T2, commit-T1.
It can be verified that this history offline-satisfies the temporal condition, but it does not online-satisfy it. Similarly, it can be shown that a history can online-satisfy a temporal condition, although it does not offline-satisfy it.

Ideally, one would like to enforce offline-satisfaction of temporal integrity constraints. However, practically only online satisfaction can be enforced, since at some point in time the system does not know which transactions will commit and which ones will abort; therefore it cannot consider current updates of transactions that will commit in the future.

The way to enforce the integrity constraint is to make the auxiliary relations changes and invoke the temporal component at every commit point of a transaction. The temporal component algorithm should evaluate the temporal condition at commit points in the history, starting with the one immediately following the earliest update of the current transaction, and ending with the committing transaction. If the condition is violated at any one of these points, then the transaction attempting to commit is aborted. This procedure enforces both online and offline satisfaction, i.e. the resulting history online- and offline- satisfies the integrity constraints. However, when considering offline satisfaction, it may have aborted transaction that shouldn't have been aborted.

A system history in the transaction-time model can be obtained from a committed system history in the valid-time model by changing the database states in the history to reflect all the database changes made by a transaction at commit time of the transaction, rather than the update time. A committed system history so modified is called a *collapsed committed history.*

The following theorem indicates that although the two satisfaction definitions differ in the valid-time model, they coincide in the transaction-time model.

THEOREM 2: Suppose that h is a complete history, and let h' be its collapsed committed history. A temporal integrity constraint c is online-satisfied on h' if and only if it is offline-satisfied on h'.

10 Comparison to Relevant Literature

Other Temporal Logics: The work that is closest to ours is presented in [1, 2, 22, 23]. In [1, 2], Chomicki considers a first order temporal logic with past temporal operators (FPTL) for specifying and maintaining Real-time Dynamic Integrity Constraints of relational databases. FPTL uses first order quantifiers, whereas PTL uses the assignment operator. This operator can be viewed as a form of quantification that naturally ensures safety [39] of the formula. For example, the trigger condition SHARP-INCREASE given in subsection 4.3 is natural, but it is considered unsafe and cannot be handled by the methods in [1, 2]. Furthermore, certain types relative time conditions cannot be expressed concisely using the formalism of [2]. The following condition is one such example: Three events A, B, C occur in that order within a span of 60 minutes.

Additionally, Chomicki's logic and processing method are strongly tied to the relational model.

The work presented in [22, 23] considers temporal logic with future (as opposed to past) operators for specifying temporal integrity constraints. This logic, called Propositional Temporal Logic, does not allow quantifiers and thus is less expressive. In [36] we also concentrate on future TL (that uses future temporal operators such as Until, Nexttime etc.) and do not discuss temporal actions, aggregates, or valid and transaction times.

Event Expressions: Event expressions (EE) is another elegant formalism for specifying temporal conditions [15, 16]. Event expressions are based on regular expressions. They consider the basic events to be the letters of the alphabet, and the expression defines the order in which these basic events occur.

Regular-expressions and temporal-logic are two different and widely-accepted formalisms for the specification and verification of concurrent programs. While the former is algebraic based, the latter is logic based.

An event expression is processed by constructing a finite-state automaton. Since event expressions use all the operators of regular expressions and also use negations, it can easily be shown (see [35]) that the size of the automaton can be superexponential in the length of the event-expression, even when variables (called attribute values in the terminology of [16]) are not used. In this case, the space complexity of our algorithm does not suffer from this super exponential blow up.

The work in [15, 16] does not explicitly address the specification of relative timing properties. We assume that relative timing properties of events can be handled in EE by using a special clock-tick event, i.e., a special event that occurs at every clock-tick. However, it is not clear how to efficiently process such event expressions.

It is to be noted that [15, 16] also propose to simplify the ECA rules by combining the event and condition parts of a rule. However, their processing algorithm seems to emphasize the event part.

All the languages mentioned so far in this section cannot handle temporal aggregate functions. The importance of such functions was observed by the temporal-database researchers, e.g. [32].

Other work on Triggers in Active Databases: The other relevant work is the work on active databases (e.g. [5, 7, 4, 19]), on triggers [3, 11, 25], and on rule-languages [21, 26, 28, 41, 20]. These works lack the temporal component in the following sense. They concentrate on the specification of triggers that involve at most two database states: the current one, and the previous one. In contrast, in this paper we address the more general temporal aspects in triggers. Also, the notion of time travel in POSTGRES [29] falls short of providing the ability to specify complex queries on the way the database evolves over time.

Valid and Transaction Times: These two notions of time were discussed in the context of active databases in [12] [5] but that discussion was restricted to append-only databases, and it did not provide a formalism for dealing with these various notions of time.

Temporal Databases: The other database literature area that is relevant to our paper is temporal databases [33, 34, 30, 27, 40]. Temporal databases are designed to save information over time in order to accommodate ad hoc queries. Therefore, for example, when a stock price changes the new value does not overwrite the previous one.

In contrast, in this work we assume that the database is designed to represent only the current information, and new values overwrite old ones. Our algorithm determines, based on analysis of the given temporal condition, which information to save, and for how long to do so.

[5] Actually [12] also considers decision time, and it makes interesting observations concerning various notions of time in active databases.

11 Conclusions

In this paper, we have presented a unified formalism, based on Past Temporal Logic (PTL), for specifying events and conditions in active database systems. Our formalism allows temporal aggregates. It also permits elegant specification of temporal as well as composite actions. In addition to the language, we have also presented an incremental algorithm for processing conditions specified in the language. We have also shown that that one can use either valid time or transaction time in our system.

Based on the work presented in this paper, a system for processing trigger conditions specified by a subclass of PTL formulas called *decomposable formulas* was implemented[8]. This system operates as follows. When a trigger condition is first entered, it automatically identifies and creates auxiliary relations. Later, whenever the database is updated, the temporal component part of the system is invoked. The temporal component updates the auxiliary relations and checks for the satisfaction of the condition. This whole system was implemented on top of Sybase using Sybase triggers.

As part of the future work, it will be interesting to see if we can extend the specification logic and the processing algorithm to include both the future and past temporal operators (in our earlier paper [36], we used only future temporal operators such as Until, Nexttime etc.). It would also be interesting to provide a graphical user interface for specifying the temporal conditions.

References

[1] J. Chomicki, *History-less Checking of Dynamic Integrity Constraints*, IEEE International Conference on Data Engineering, Phoenix, Arizona, Febraury 1992.

[2] J. Chomicki, *Real-Time Integrity Constraints*, ACM Symposium on Principles of Database Systems, June 1992.

[3] D. Cohen, *Compiling Complex Database Triggers*, Proceedings of ACM SIGMOD 1989.

[4] S. Ceri and J. Widom, *Production Rules in Parallel and Distributed Database Environments*, Proceedings of VLDB, 1992.

[5] S. Chakravarthy et. al., *HiPAC: A Research Project in Active, Time-Constrained Database Management*, TR XAIT-89-02, Xerox Advanced Information Technology.

[6] S. Chakravarthy et. al., *Composite Events for Active Databases: Semantics, Contexts and Detection*, Proc. VLDB 94.

[7] U. Dayal, *Active Database Management Systems*, Proceedings of the Third International Conference on Data and Knowledge Bases—Improving Usability and Responsiveness, Jerusalem, June 1988.

[8] M. Deng, *Past Temporal Logic Trigger Evaluation System*, Masters Project Report, University of Illinois at Chicago, Department of EE and CS, 1994.

[9] U. Dayal, M. Hsu, R. Ladin *Organizing Long-Running Activities with Triggers and Transactions*, Proc. ACM-SIGMOD Conf. 1990.

[10] U. Dayal, M. Hsu, R. Ladin *A Transactional Model for Long-Running Activities*, Proceedings of the 17th Conf. on VLDB, 1991.

[11] M. Darnovsky and J. Bowman, "TRANSACT-SQL USER'S GUIDE" Document 3231-2.1 Sybase Inc., 1987.

[12] O. Etzion, A. Gal and A. Segev, *Retroactive and Proactive Database Processing*, Proc. of the 4th Int. Workshop on Research Issues in Database Engineering, Feb. 94.

[13] A. Elmagarmid, editor, Data Eng. Bulletin, Special Issue on Unconventional Transaction Management, March 1991.

[14] S. Gatziu and K. Dittrich, *SAMOS: an Active Object-Oriented Database System*, Data Engineering Bulletin, Dec. 92.

[15] N. H. Gehani, H. V. Jagadish and O. Shmueli, *Event Specification in an Active Object-Oriented Database*, ACM-SIGMOD 92.

[16] N. H. Gehani, H. V. Jagadish and O. Shmueli, *Composite Event Specification in Active Databases: Model& Implementation*, Proceedings of the 18th International Conference on Very Large Databases, Aug. 1992.

[17] M. Gertz and U. Lipeck, *Deriving Integrity Maintaining Triggers from Transition Graphs*, Proc. Int. Conf. on Data Eng., 1993.

[18] E.N. Hanson, *Rule Condition Testing and Action Execution in Ariel*, Proceedings of the ACM-SIGMOD 1992, International Conference on Management of Data, June 1992.

[19] E. N. Hanson and J. Widom, *An Overview of Production Rules in Database Systems* Research Report RJ9023, IBM Research Division, 1992.

[20] A. Kotz, K. Dittrich and J. Mulle, *Supporting Semantic Rules by a Generalized Event/Trigger Mechanism*, Proc. of the EDBT'88, Springer Verlag LNCS 303.

[21] G. Kiernan, C. de Maindreville, E. Simon, *Making Deductive Database a Practical Technology: A Step Forward*, Proc. of the ACM-Sigmod International Conf. on Management of Data, 1990.

[22] U. W. Lipeck and Gunter Saake, *Monitoring Dynamic Integrity Constraints Based on Temporal Logic*, Information Systems, 12(3):255-269, 1987.

[23] U. W. Lipeck and Gunter Saake, *Using Finite-Linear Temporal Logic for Specifying Database Dynamics*, Lecture Notes in Computer Science, Springer-Verlag 1988.

[24] D.R. McCarthy and U. Dayal, *The Architecture of An Active Database Management System*, Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data, Portland, Oregon, May-June 1989, 215-224.

[25] T. Risch, *Monitoring Database Objects*, Proc. VLDB, Aug. 89.

[26] T. Sellis, Ed., Special Issue on Rule Management and Processing in Expert Database Systems, SIGMOD RECORD, 18(3), Sept. 1989.

[27] A. Segev and H. Gunadhi, *Event-Join Optimization in Temporal Relational Databases*, Proc. VLDB, Aug. 1989.

[28] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, *On Rules, Procedures, Caching and Views in Database Systems*, Proc. of the ACM-Sigmod International Conf. on Management of Data, 1990.

[29] M. Stonebraker and G. Kemnitz, *The Postgres Next-generation Database Management System*, CACM, Oct. 91.

[30] R. Snodgrass, ed., Data Engineering, Special Issue on Temporal Databases, Dec. 1988.

[31] R. Snodgrass and I. Ahn, *The Temporal Databases*, IEEE Computer, Sep. 1986.

[32] R. Snodgrass, S. Gomez, E. McKenzie, *Aggregates in the Temporal Query Language TQuel*, IEEE Trans. on Knowledge and data Eng., Oct. 1993.

[33] A. Segev and A. Shoshani, *Logical Modeling of Temporal Data*, Proc. of the ACM-Sigmod International Conf. on Management of Data, 1987.

[34] A. Segev and A. Shoshani, *The Representation of a Temporal Data Model in the Relational Environment*, 4th Int. Conf. on Statistical and Scientific Data Management, June 1988.

[35] L. J. Stockmeyer (1974), *The complexity of decision procedures in Automata theory and Logic*, Doctoral Dissertation, MIT, Cambridge, Project MAC Technical Report TR-133.

[36] A. P. Sistla and O. Wolfson, *Temporal Triggers in Active Databases*, Univ. of Illinois at Chicago, EECS Dept. Tech. Report, 1993.

[37] A. Tuzhilin and J. Clifford, *A Temporal Relational Algebra as a Basis for Temporal Relational Completeness*, Proc. of the 16th VLDB Conference, 1990.

[38] D. Toman and J. Chomicki, *Implementing Temporal Integrity Constraints Using an Active Database*, Proc. of the 4th Int. Workshop on Research Issues in Database Engineering, Feb. 94.

[39] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1988.

[40] X. Wang, S. Jajodia, V. Subrahmanian, *Temporal Modules: An Approach Toward Federated Temporal Databases*, Proc. of the ACM-Sigmod International Conf. on Management of Data, 1993.

[41] J. Widom, S. Finkelstein, *Set-Oriented Production Rules in Relational Database Systems*, Proc. of the ACM-Sigmod International Conf. on Management of Data, 1990.