Temporal Conditions with Retroactive and Proactive Updates

February 2, 1995

Minglin Deng **A. Prasad Sistla** **Ouri Wolfson**
Department of Electrical Engineering and Computer Science,
University of Illinois at Chicago,
Chicago, Illinois 60680
deng@dbis.eecs.uic.edu, sistla@surya.eecs.uic.edu, wolfson@ouri.eecs.uic.edu

Abstract

In this paper, we extend the temporal rule language introduced in [1] to support pro- and retroactive updates. Such updates may cause the time when a condition is satisfied to differ from the time when it is detected. We enrich the rule language by providing a set of alternative actions in each rule; this allows execution of one of the alternative actions depending on the difference between condition-satisfaction time and detection time.

We identify a subset of the temporal condition language PTL [1], called *decomposable* PTL, that is amenable to more efficient evaluation. We present an evaluation algorithm for decomposable PTL. We also describe the implementation of the system on top of the Sybase DBMS.

1 Introduction

The most popular model of rules in active database systems is the ECA model [42, 7, 29, 6, 15, 19]. 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. One drawback of ECA is that the condition is static and cannot specify the way that the database evolves over time. In [1] we proposed 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, based on Past Temporal Logic (PTL), enables the specification of both, events and conditions on the evolving database, and thus is more powerful than existing rule languages.

In this paper we extend our work as follows. First, we enhance our active rule system by facilitating retroactive and proactive database updates. These are updates that apply to past and future database states respectively (see [13]). Retroactive and proactive updates occur naturally in many applications. For example, a postdated check is a proactive update, and a salary increase may be retroactive to some past date. We assume that the database system supports retroactive and proactive updates, and we enhance the rule system to enable retroactive and proactive rule firing.

This involves two extensions. First, the rule language is enhanced to enable different actions for a satisfied condition, depending on the difference between the detection time, and the time at which the rule-condition is satisfied. To see why this is necessary, consider a check that overdraws an account. Suppose that it is presented on 15th Dec., and it is postdated as 20th Dec. Now consider a rule that stipulates that when an account is overdrawn, an action has to be taken. The overdrawal may be detected on both dates, 15th and on 20th. However, it is reasonable to assume that the action to be taken depends on the date. On the 15th the action may be to send the customer a note saying that unless a deposit for a certain amount is made, the postdated check will be bounced on the 20th. On the other hand, the reasonable action on the 20th is to bounce the check. Therefore, we extend the definition of an active rule to enable different actions depending on the difference between the detection time and the condition-satisfaction time.

The second enhancement of this paper deals with pro- and retro- active updates, and it involves the evaluation algorithm. In [1] we proposed an algorithm to detect the satisfaction of a temporal condition. In this paper, we extend the algorithm to handle pro- and retro- active updates. The extended algorithm is clearly more complex in terms of time and space. Thus, as another contribution of this paper, we identify a subset of past temporal logic (PTL), called *Decomposable PTL* (DPTL). For rule-conditions defined in DPTL we provide a more efficient evaluation algorithm. An additional performance enhancement is the introduction of bounded temporal operators using the *Within* clause. Use of the bounded operators fecilitates easier specification, and more efficient evaluation, of real-time propeties.

In this paper we also describe our implementation of the proposed system on top of Sybase. This system contains a PTL interpretor, the temporal-condition evaluation algorithm and the rule processor, a database update tracking subsystem, and a graphical user interface. The system is currently being ported to support a medical application at the Rush Presbyterian Medical Center.

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

- In order to support pro- and retro- active updates, we extend the rule language to provide alternative actions in a rule, depending on the difference between condition-satisfaction time and detection time.
- We propose a more efficient evaluation algorithm for a subset of PTL, called Decomposable PTL, and for formulas employing bounded temporal operators.
- We describe the implementation of the system on top of the Sybase DBMS.

This paper is organized as follows. Section 2 describes the model that we use; section 3 introduces the temporal rule system with alternative actions. Section 4 defines the basic Past Temporal Logic (PTL) language, and Decomposable PTL. Section 5 describes the decomposable PTL evaluation algorithm, and the handling of retroactive and proactive updates. Section 6 describes the implementation on top of Sybase. Section 7 provides a comparison to relevant work.

2 System Model

A *database* consists of a set of database items. These are names of relations or object classes. A special data item, called *time*, gives the current time at every instant. Additionally, there is a set U of events that can occur at any time. For example, Transaction-begin, Transaction-commit, Rule-execute, Insert, Delete, 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. We impose the following constraint on system histories. The database states of two consecutive system states are identical, unless the event set E of the latter contains a database modification event (i.e. an Insert, Delete, or Update). If so, then the new database state reflects all and only the database changes made by the database modification events 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 modification events.

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. 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.

Next we discuss the occurrence time of events (remember, the occurrence time determines the timestamp of a system state), and retroactive and proactive database modification events. The *occurrence time* of events other than database modification events is the current time (value of the time data item) when they occur. For database modification events the occurrence time is either overwritten by the user, or it is the commit-time of the transaction that issued the modification. If that transaction aborted, then the modification is disregarded. More precisely, every database modification event may be issued by the user with an occurrence time. The occurrence time is the time for which the database modification becomes valid, and this time may precede or succeed the time at which the modification event is presented to the database management system.

If the occurrence time precedes the current time, then the modification is *retroactive*, and if it succeeds the current time then it is *proactive*. For example, the update u = (IBM:=72) (indicating that the price of the IBM stock is 72) may be presented at 1pm and have an occurrence time of 12:50pm. This update is denoted (u, 12:50pm), and the database management system makes the change retroactively. If the user does not specify an occurrence time for u, then the occurrence time of u is the time when the transaction that issued u commits.

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 some actions (such as firing a missile) can only be based on definite values in the database. To accommodate these situations we introduce the concept of a *maximum retroactivity delay*, denoted τ. It means that for every update, the maximum difference between the time at which it is presented to the DBMS and the occurrence 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 retroactivity delay assumption is that a committed value in the database can be in one of two states, definite or tentative. If the value was put in the database by a modification event that occurred less than τ time units ago, then the value is *tentative*. Otherwise the value is *definite*. Observe that definite values can be updated later, but a value that is definite at 1pm cannot be retroactively updated as of 1pm. In other words, the value at 1pm becomes definite at time $T = 1pm + \tau$, which means that after time T the value cannot be updated as of 1pm.

3 Temporal Logic Rule System

In this paper we assume 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 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 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 fire.

In principle, the condition part of a rule is evaluated for every system state in the history, when the state is added to the history. Furthermore, if a proactive or retroactive update is applied to a system state with a particular occurrence time, the condition may be reevaluated for that system state. We say that a condition C *is satisfied at time* t if it is satisfied by the history up to the state s with occurrence time t. Observe that because of retroactive and proactive updates, s may not be the last state in the history. In other words, the satisfaction of C at time 1pm may be detected at time 2pm, and at 2pm the state with time-stamp 1pm may not be the last in the history. This late detection may be due to a retroactive update that pertains to 1pm, but was presented at 2pm; possibly this was the update that made the condition C **true**.

Suppose that a condition C is satisfied at time t, and this satisfaction is detected at time T. Then we say that C is *satisfied as of* T. If $T \geq t+\tau$, then no subsequent update may change the fact that C is satisfied at time t, and we say that C is *definitely satisfied as of* T. If $T < t+\tau$, then we say that C is *tentatively satisfied as of* T. (Observe the T may actually be smaller than t because of proactive updates).

An *action* is a database operation, a program to be executed, or it may simply be an abort operation on the current transaction. Furthermore, the action may refer to some of the free variables in the condition part. This facilitates parameter passing from the condition part to the action part which is often very useful.

Now consider a C-A rule, and suppose that C is satisfied at time t as of T, i.e. the satisfaction is detected at time T. The action to be taken (at time T) may depend on the difference between t and T. If the satisfaction is tentative, one action may be taken, and if the satisfaction is definite another action may be taken. For example, suppose that a check that overdraws an account is presented. If the check is postdated (proactive withdrawal) the customer may be issued a warning-letter saying that unless a deposit is made, the check will be bounced at the postdate. If the check is predated, then the check is bounced.

Furthermore, the action to be taken for a tentative satisfaction may depend on the difference between t and T. Thus, the *action part* of a C-A rule is a sequence of pairs, $(A_0, I_0), ..., (A_n, I_n)$, where each A_j is an action, and each I_j represents a time interval. The semantics is the following. If C is satisfied at time t, where t is in the interval I_0 relative to the current time T (the interval +[5,8] denotes a time between 5 and 8 time units after T) then action A_0 is executed; if t is in the interval I_1 relative to the current time T then action A_1 is executed; etc.

4 Past Temporal Logic (PTL)

In this section, we define the syntax and semantics of the basic PTL. We also give the syntax of decomposable PTL formulas and a set of examples of temporal conditions expressed in the decomposable PTL.

4.1 Syntax of PTL Formulas

Past temporal logic has 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. For example, $f =$ Previously f_1 is equivalent to $f = true$ Since f_1.

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 denoting database relations, 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 STOCKS (the relation contains tuples consisting of a stock name, the price, the company's name and the category of products the company produces):

RETRIEVE (STOCKS.name) FROM STOCKS WHERE STOCKS.price≥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 also expanded the logic by introducing a time constraint operator, $Within\ \tau$, where τ is a constant denoting time limit. The operator will bound a temporal formula to a time period. For example, **Previously** f_1 $Within$ 10 specifies that the formula f_1 was satisfied within 10 time units in the past.

We first give the definition of a term 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.

Definition A: *PTL Formulas*

1. 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.
2. 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.
3. If g is a PTL formula, X is a 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.
4. 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, **Throughout_the_Past** g_1, **Previously** g_1 $Within\ \tau$,
   g_1 **Since** g_2 $Within\ \tau$ and **Throughout_the_Past** g_1 $Within\ \tau$ are PTL formulas.
5. **true** and **false** are PTL formulas.

We call the first two types of PTL formulas as *atomic formulas*. We will demonstrate in section 5 how the *atomic formulas* can be directly evaluated by posting queries over the database history.

If all the variables in a PTL formula are bound, namely, there is an assignment inside the formula for each variable, then the formula is called *closed formula*, otherwise, it is called *open formula*. If all the variables in the formula are unbound, then the formula is called *fully open formula*. The example (A) in section 4.4 is a closed formula, and example (B) in section 4.4 is a open formula.

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 $f = \mathsf{Lasttime}\ f_1$, 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 $f = \mathsf{false}$.
- 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 $f = f_1$ Since f_2 $Within\ \tau$, 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 there is a $j < l \leq i$, such that the time difference between (S_l, E_l) and (S_j, E_j) is $\leq \tau$, and for all k, $j < k \leq l$, 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 Decomposable PTL Formulas

Now, we define the syntax of decomposable PTL formulas. A PTL formula f is said to be a *decomposable formula* if it satisfies the following condition: for every subformula g of f, all the free variables in g are instantiated (i.e. are assigned query values) at a single place in f.

The most common case of a decomposable formula is when f is of the form $[X_1 \leftarrow t_1] \ldots [X_k \leftarrow t_k] g$, where $X_1, \ldots, X_k$ are all the global variables in g. We call such decomposable formulas as *basic* decomposable formulas.

An example of a non-decomposable formula is the formula
$[X \leftarrow price(IBM)]$ Previously $[Y \leftarrow price(IBM)]$ Previously $(price(IBM) = X + Y)$
Note that in this formula, the free variables X, Y of the subformula $(price(IBM) = X + Y)$ are instantiated at two different place in the main formula.

The class of decomposable PTL formulas is expressive enough to be able to capture many of the trigger conditions occurring in practice. The examples of temporal conditions in this paper are all given by decomposable formulas. Since the all global variables appearing free in every subformula of a decomposable formula are instantiated in a single place, the evaluation procedure for the decomposable formulas is much simpler.

4.4 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 denotes the values of price(Motorola) in current system state, then there exists a past state before it in which the value of price(Motorola) is at most 90% of X and the time difference is less than or equal to 30. Here, X is a variable and it is assigned the query $price(Motorola)$. We split the formula over two lines.

(A) $[X \leftarrow price(Motorola)]$
Previously $(price(Motorola) \leq 0.9X)$ $Within$ 30).

We can specify multiple conditions in one formula by allowing free variables[1]. For example, the following formula uses the free variable Y to specify a sharp increase in price of the company denoted by Y.

(B) $[X \leftarrow price(Y)]$
Previously $(price(Y) \leq .9X)$ $Within$ 30).

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

In the above formula, the variable Y can be constrained to Computer companies by binding Y to a query that retrieves all names of 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 Evaluation of Decomposable PTL formulas

In this section, we present an efficient algorithm for evaluation of trigger conditions specified by a decomposable PTL formula. A general evaluation algorithm for PTL formulas (including the non-decomposable ones) was presented in [1]. This general algorithm requires the maintenance of a directed acyclic and-or graph, whose size can grow with the database history. In this section, we present a simpler and a more efficient algorithm for decomposable PTL formulas, which avoids the maintenance of the and-or graph.

Our new algorithm, for decomposable formulas, is based on determining, for each subformula g of the given formula f, a set of state intervals (of the history) in which g is satisfied. Consider the atomic subformula $price(Motorola) \leq 0.9X$ in the example (A) of section 4.4. Now, if we assign a value to the variable X, then the set of system states in the history where this subformula is satisfied can be represented by a set of disjoint intervals on this history. Now consider an arbitrary closed formula g. Assume that g is satisfied in the states $S_j, S_{j+1}, \ldots, S_k$ of the history. Then, we call $[S_j, S_k]$ as a *state interval* that satisfies g. The collection of all state intervals in the history that satisfy g is called the *state interval set* of g. To compute the state interval sets, the algorithm maintains some auxiliary relations that capture past database values. A detailed discussion on the the auxiliary relations is given in a later subsection.

The algorithm we present here computes the state interval set of each subformula g of the given formula inductively by using the state interval sets of the the smaller subformulas. This inductive computation uses the following operations on the state interval sets— union, intersection etc.

5.1 Computation of the State Interval Sets

Let f be a decomposable PTL formula. We assume that f is a *basic* decomposable subformula, i.e., f is of the form $[X_1 \leftarrow q_1]...[X_k \leftarrow q_k]f'$ where all the variables appearing in f' are those from the set $\{X_1, ..., X_k\}$ and none of these variables are instantiated in f'. We will present an algorithm that computes the state interval set of the formula f by using the state interval sets of the subformulas of f. We later argue that this algorithm for the basic decomposable formulas can be extended to handle arbitrary decomposable subformulas.

The evaluation algorithm for f, given above, over a history $h = ([S_0, E_0], \ldots, [S_i, E_i])$ is given below.

First, compute the values of the queries $q_1, ...q_k$ on the latest system state $[S_i, E_i]$ and substitute the resulting values in the formula f'. Let the resulting formula be f''. Now, compute the state interval set associated with f'' inductively as follows.

1. For each *atomic formula* g of f'', determine the state intervals in which g is satisfied, by querying the auxiliary relations that capture the relevant parts of the system history.

2. For eacn non-atomic subformula g of f'' (including f'' itself) the state interval set of g is computed based on the following cases.

   (a) $g = g_1 \wedge g_2$: The state interval set of g is computed as the set of intervals obtained by taking the intersection of a state interval of g_1 with a state interval of g_2.

   (b) $g = \neg g_1$: The set of states where g is satisfied is the complement of the set of states where g_1 is satisfied. The actual state intervals of g are computed as follows. If $[u_1, v_1], [u_2, v_2], ..., [u_k, v_k]$ are indices associated with the state intervals of g_1 when arranged in increasing values of the indices, then the indices for the state intervals of g is given by $[v_1 + 1, u_2 - 1], [v_2 + 1, u_3 - 1], ..., [v_k + 1, i]$.

   (c) If $g = g_1 \vee g_2 \equiv \neg(\neg g_1 \wedge \neg g_2)$: The set of state intervals of g is a union of the sets of state intervals of g_1 and g_2 .

   (d) $g =$ Lasttime g_1: The state interval set for g is computed using the following observation. If (S_u, S_v) is a state interval of g_1 then (S_{u+1}, S_{v+1}) is a state interval of g.

   (e) $g = g_1$ Since g_2 *within* τ: The state interval set for g is computed using the following observation. Assume that (S_j, S_k) is a state interval of g_2. For each such state interval do as follows. Determine if there exists a state interval (S_u, S_v) of g_1 such that $u = k + 1$ or $u \leq k \leq v$, i.e. (S_u, S_v) overlaps with (S_j, S_k) on the right. If such a state interval exists determine the latest state S_w in the interval (S_u, S_v) such that the difference between the time of S_w and S_k is less than or equal to τ; then add the interval (S_j, S_w) to the set of state intervals of g. If no such overlapping interval exists then simply add (S_j, S_k) to the set of state intervals of g. Now, it should be easy to see that the state intervals of g can be computed efficiently by taking the sorted lists of the state intervals of g_1 and g_2, and by making one pass over them simultaneously.

   It is to be noted that the case when $g = g_1$ Since g_2 can be handled using the bounded Since and taking a value of ∞ for τ.

   (f) $g =$ Previously $g_1 \equiv$ true Since g_1. If S_t is the earliest state in the history at which g_1 is true, then the state interval set of g consists of the set consisting of the single interval (S_t, S_i).

3. It is not very difficult to see that f is satisfied at the latest system state, i.e. at $[S_i, E_i]$, iff f'' is satisfied at this system state. The later can be determined by checking if there is an interval ending with S_i in the set associated with f''.

Note that the above algorithm only checks if f is satisfied at the latest system state. In order to compute the set of state intervals where f is satisfied, we need to repeat the above procedure for each value of i starting with the latest value upto 1.

The state interval set for an arbitrary decomposable PTL formula f can be computed by identifying the basic decomposable subformulas of f, and by applying the above algorithm inductively using the state interval sets of these subformulas.

5.2 Auxiliary Relations and System history

Auxiliary relations are employed in our system to keep track of the relevant database history needed for the evaluation of the PTL formula f. We assume that each reference to a database query in f is through a variable. Intuitively, if a variable X in f is bound by a query q, then we have an auxiliary relation R_q that captures the tuples of the materialized view defined by q on some past database states as follows. If q retrieves a k-ary relation, then R_q has k+2 attributes. The first k attributes denote a tuple in a materialized view defined by q and the last two attributes T_start and T_end are state counters that identify the interval of the history during which this tuple is in the materialized view. Each such auxiliary relation R_q is created when the formula is installed into the database system; at this time the relation R_q is initialized so that its projection on its first k attributes is the view defined by q on the database state at that time, and the value of T_start in each tuple is set to 1 and the value of T_end is set to MAX.

Every database update that changes the view defined by q will appropriately update the auxiliary relation R_q as follows. Assume that A and D, respectively, are the sets of tuples added to and deleted from the materialized view. Then for each tuple $a = (a_1, \ldots, a_n) \in A$, the tuple $(a_1, a_2, \ldots, a_k, c, MAX)$ is added to R_q , where c is the current database state counter. For each tuple $d \in D$, the T_{end} attribute value of the corresponding tuple in R_q is set to $c - 1$ where c is as given above. Now, it's quite obvious that the value of query q at any database state S_i can be obtained by a selection on R_q, followed by a projection. The selection is the set of tuples for which the condition $T\_start \leq i \leq T\_end$ is satisfied.

In our algorithm, we need to evaluate the state intervals associated with atomic formulae. It is not difficult to see that such state intervals can be obtained by posing a query on the auxiliary relations and then projecting the resulting tuples on to the attributes T_{start} and T_{end}. The details will be given in the full paper.

In order to associate time with each database state, we introduce a special auxiliary relation called *update_times*. This relation has two attributes, *State_number* (database state counter) and *StartT* (starting time). Each tuple (i, T) in this relation indicates that update i on the database occurred at time T. By using this, we can easily find the time of occurrence of each update. These times are used in evaluation of bounded temporal operators.

Basically, the evaluation algorithm and the implementation of auxiliary relations, take advantage of the retrieval capability of the underlying database system.

5.3 Retroactive and Proactive Update Handling

In this section we will elaborate how the system deals with retroactive and proactive updates. Assume that when the user issues a transaction, he basically specifies the update event with two parts, (u, t), u is the update event and t is the *occurrence time* of the event. If no *occurrence time* specified, then the default value is the transaction commit time. We also assume that there is a *maximum retroactive delay*, denoted as τ in the system. So any retroactive update which goes back more than τ time units will be simply rejected.

Each legal retroactive/proactive update event will trigger the update of the auxiliary relations and a re-evaluation of the PTL condition. Assume there is such an update (u, t) which changes the material view defined by a query q. We assume that the auxiliary relation R_q and the relation *update_times* is modified by a separate component as follows.

1. Using the *update_times* relation, find the update number i such that the value of t falls between the times of the i^{th} and $(i+1)st$ updates. Modify the *updates_times* relation to reflect that the current update is the $(i+1)st$ update.
2. A new tuple is added to the auxiliary relation R_q denoting the change of the view defined by q at time t.
3. The update may have caused a change in the values of views at times later than t. We assume that the auxiliary relation is updated to reflect these changes in the views at these later times.

The re-evaluation of the PTL condition will have to start at the occurrence time t of the current update and end at the maximum occurrence time present in the history. The re-evaluation has to be done for each state inside this period. To do this, we execute our previous evaluation algorithm at each of these intermediate system states. If the PTL formula is satisfied at any of these system states, then the time stamp of the system state is passed as a parameter to the component that deals with the action part. The action part will execute an appropriate action based on the difference between the current time and the time value passed to it.

6 System Implementation in Sybase

We have designed and implemented a temporal-condition evaluation system on top of Sybase; this system is based on the evaluation algorithm for decomposable PTL presented in this paper. This evaluation system takes a decomposable PTL temporal condition (trigger) as an input, and then automatically creates a temporal component that communicates with the DBMS and detects the satisfaction of the rule condition. This temporal component maintains the auxiliary relations by capturing the relevant database updates, and evaluates the rule condition at appropriate times.

The implemented system uses Sybase stored procedures, Sybase triggers, embedded C, Lex and Yacc in Unix Client-Server environment. The implemented system has the following subcomponents:

PTL interpreter, condition evaluation sub-system, a database update tracking sub-system and the graphical user interface. We give a description of each of these sub-systems below:

1. **Past Temporal Logic Interpretor**
   This interpretor is used to check the syntax of the given PTL formula and then to translate the formula into predefined data structures. The data structures are specifically designed to represent temporal operators, logic connectives, and to reflect recursive structure of the formula. This subsystem contains an SQL parser to interpret SQL queries appearing in PTL conditions (note that such queries are used in assignment of a query to a variable in a PTL formula). The interpreter is implemented using Lex and Yacc.

2. **Database Update Tracking Sub-System**
   This sub-system takes the data structures from PTL interpretor as an input, and automatically creates auxiliary relation tables, Sybase triggers and stored procedures. The Sybase triggers and stored procedures are used to capture related database updates, and to modify the auxiliary relation tables correspondingly. The procedures are also used to query the auxiliary relation tables, and to obtain the state intervals for all the atomic formulas in the PTL formula and pass the result to the PTL evaluation subsystem. The Sybase triggers are also used to invoke the PTL formula evaluation sub-system after each related database update.

3. **PTL Formula Evaluation Sub-system**
   This sub-system implements the DPTL evaluation algorithm given in the paper. It takes the data structures and state intervals of atomic formulas from the previous subsystems and computes state intervals for each subformula recursively.

4. **Graphic User Interface**
   A Motif graphic user interface is built for the whole evaluation system. First, this interface allows the user to edit the input temporal condition (specified by a PTL formula). Secondly, this interface allows the user to open windows for reviewing the evaluation process and the result.

The implemented system is currently being ported to the Informix DBMS to support a medical application.

7 Comparison to Relevant Literature

The work described in this paper extends our earlier work in [1]. This paper, for the first time, identifies the dichotomy between the condition satisfaction time and the detection time caused by the retroactive/proactive updates. This gave rise to the introduction of the new rule formalism, where alternative actions are supported. In the remainder of this section, we compare our formalism for specifying temporal rule conditions with other approaches.

Other Temporal Logics: The work that is closest to ours is presented in [2, 3, 23, 24]. In [2, 3], 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 [40] of the formula. For example, the trigger condition SHARP-INCREASE given in subsection 4.4 is natural, but it is considered unsafe and cannot be handled by the methods in [2, 3]. Furthermore, certain types relative time conditions cannot be expressed concisely using the formalism of [3]. 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 [23, 24] 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 [37] 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 [16, 17]. 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 [36]) 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 [17]) are not used. In this case, the space complexity of our algorithm does not suffer from this super exponential blow up.

The work in [16, 17] 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 [16, 17] 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. [33].

Other work on Triggers in Active Databases: The other relevant work is the work on active databases (e.g. [6, 8, 5, 20]), on triggers [4, 12, 26], and on rule-languages [22, 27, 29, 42, 21]. 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 [30] 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 [13] [2] 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 [34, 35, 31, 28, 41]. 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.

8 Conclusions

In the [1] paper, we have presented a unified formalism, based on Past Temporal Logic (PTL), for specifying events and conditions in active database systems. This formalism offers elegant specification of temporal conditions as well as composite actions. In this paper, we extended our system by considering and handling retroactive and proactive update events. In addition to this extension, we have presented an efficient method for evaluation of trigger conditions for a subclass of PTL formulas called decomposable formulas; many of the naturally occurring temporal conditions fall into this class. We have also discussed how this algorithm can be extended to handle retroactive/proactive updates. We have also described a system that was implemented on top of Sybase.

References

[1] P. Sistla, O. Wolfson, *Temporal Conditions and Integrity Constraints in Active Database Systems*, to appear at Sigmod,1995.

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

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

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

[2] Actually [13] also considers decision time, and it makes interesting observations concerning various notions of time in active databases.

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

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

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

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

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

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

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

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

[13] 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.

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

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

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

[17] 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 .

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

[19] 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.

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

[21] 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.

[22] 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.

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

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

[25] 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.

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

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

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

[29] 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.

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

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

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

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

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

[35] 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.

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

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

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

[39] 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.

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

[41] 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.

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