Temporal Triggers in Active Databases

A. Prasad Sistla[1] Ouri Wolfson[1]

Department of Electrical Engineering and Computer Science, University of Illinois at Chicago Chicago, Illinois

Abstract

In this paper we propose two languages, called *Future Temporal Logic* (FTL) and *Past Temporal Logic* (PTL), for specifying temporal triggers. Some examples of trigger conditions that can be specified in our language are the following: "The value of a certain attribute increases by more than 10% in ten minutes", "A tuple that satisfies a certain predicate is added to the database at least 10 minutes before another tuple, satisfying a different condition, is added to the database". Such triggers are important for monitor and control applications.

In addition to the languages, we present algorithms for processing the trigger conditions specified in these languages, namely, procedures for determining when the trigger conditions are satisfied. These methods can be added as a "temporal" component to an existing database management systems. A preliminary prototype of the temporal component that uses the FTL language has been built on top of Sybase running on SUN workstations.

Index Terms: Active Databases, Triggers, Temporal Logics, Algorithms

1 Introduction

Modern systems, such as traffic control, securities trading and communication networks, are increasingly dependent on real-time software applications for monitor and control. At the center of such applications usually lies an active database, that represents the status of the system. This database is continuously updated by (often remote) sensors, and the software is expected to respond to predefined conditions. Often these conditions refer to the evolution of the database state over time (i.e. the database history). For example, in securities trading, the system may be requested to alert a trader when the value of a particular stock (given in database attribute A)

[1] This research is supported by AFOSR grant no. F49620-93-1-0059. Ouri Wolfson's research was also supported by NSF grant IRI9224605.

increases by more than 10% in 15 minutes. We call such conditions temporal triggers, i.e. triggers on the evolution of the database state over time. Furthermore, one may want to specify temporal triggers that also involve external events (such as transaction-begin, transaction-commit, invocation of a method, etc.) in addition to the database history. The following temporal trigger is one such example— the value of attribute A increases by more than 10% from the time when transaction X commits to the time when transaction Y starts. Existing database management systems do not provide the capability for specifying temporal triggers. In most of them, the condition part of a rule (in Hipac [8] terminology a rule is an event-condition-action triple) refers to either the current database state, or to the transition from one database state to the next, but not to a sequence of database states.

In this paper we propose two languages based on temporal logic, called *Future Temporal Logic* (FTL) and *Past Temporal Logic* (PTL)[1] for specifying temporal triggers. As we point out in section 5, it is easier to specify certain triggers in FTL, whereas it is easier to specify certain others in PTL. *Temporal Logic* [24] is a formalism for specifying and reasoning about time-varying properties of systems, and therefore it is an appropriate language for specifying temporal triggers. The main feature of FTL and PTL is the use of special operators that apply exclusively to the time dimension. The temporal operators used in FTL are Until , Nexttime , and Eventually , while the ones used in PTL are Since , Lasttime and Previously . Some other examples of trigger conditions that can be specified in our languages are the following: "A tuple that satisfies a certain predicate is added to the database at least 10 minutes before another tuple, that satisfies another condition, is added to the database". Or, "the value of an attribute stays above a certain threshold for at least 10 minutes". Or, "for a period of 10 minutes, two different values in the database increase simultaneously".

In addition to the languages, we present methods for processing the trigger conditions specified in both the languages, namely, procedures for determining when the trigger conditions are satisfied. These methods can be added as a "temporal" component to an existing database management systems, as follows. Based on a temporal trigger specification, we identify a set of materialized views that should be presented to the DBMS as traditional triggers. Namely, the DBMS is requested to invoke the temporal component whenever any of the materialized views changes (see [2, 3, 25] for maintenance of materialized views). When invoked, the temporal component determines whether or not the trigger is satisfied. If so, then the user is notified, or the associated action is taken. Otherwise, the temporal component saves a minimum amount of information from the current database state, in order to be able to determine in its future invocations whether or not the trigger condition is satisfied. We have implemented the temporal component for a subset of FTL on top of the Sybase DBMS running on SUN work-stations.

[1]Future Temporal Logic uses temporal operators that refer to the future database states while Past Temporal Logic uses temporal operators that refer to past database states.

The proposal of FTL and PTL as trigger specification languages is based on our experience with Net-mate [12, 13, 41]. Net-mate aims to develop a software environment for management of very large (hundreds of thousands of interconnected computers) communication networks. A fault in such a network is a failure or an overload condition, and an important goal in network management is to automatically detect and recover from this exceptional condition. The current network-state in Net-mate is represented by a management information base (MIB). In the process of developing Net-mate we were faced with the need for a flexible specification language, that enables frequent changing of the trigger conditions; these triggers should have the temporal capability offered by our proposed languages. In [40] we developed another language for temporal triggers, however, that language does not have the expressive power of our languages.

The rest of the paper is organized as follows. In section 2 we give an informal as well as a formal presentation of our language FTL, and we illustrate its application through examples. In section 3 we present the algorithm for detecting the trigger conditions specified inthe language FTL and demonstrate it through an example. This section also contains the proof of correctness of the algorithm, its optimization and implementation issues. In section 4, we present the language PTL and give an algorithm for detecting triggers specified in this language. In section 5 we discuss systems issues and extensions of our work. In section 6 we compare our work with the existing relevant work.

2 The Temporal Logic FTL

2.1 The Model

We assume that the database is composed of variables of various types, such as relations, objects, etc. We refer to these variables as *database variables.* A *database state* is a mapping that associates a value from the appropriate domain to each variable. The formulas of the logic FTL are interpreted over database histories. A *database history* is a finite sequence of database states, each with an associated *time stamp.* A new database state is added to the history whenever the database changes. The time stamp associated with the new database state denotes the time at which the change occurs, according to a fixed global clock[2]. We assume that the time stamps along the history are strictly increasing.

This model only refers to the database variables, and it does not explicitly consider external events such as—transaction-begin, transaction-commit, etc. We can easily incorporate the external events into our model by assuming that the database has a special relation which, at any instance of time, contains all the external events that occur at that time. This is explained in more detail in section 5.

[2] Our notion of time is discrete, but with slight modifications we can extend the model to the case of continuous time.

2.2 Syntax

The formulas of FTL use two basic future temporal operators Until and Nexttime . Other temporal operators, such as Eventually , can be expressed in terms of the basic operators. The symbols of the logic include various *type* names, such as relations, integers, etc. These denote the different types of the variables and constants in the database. We assume that, for each $n \geq 0$, we have a set of n-ary *function* symbols and a set of n-ary *relation* symbols. 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 retrieval queries on databases. For example, consider the following relational query, BUSY-LINKS, that retrieves all busy links from a relation called TRAFFIC-ON-LINK (the relation contains tuples consisting of a link name and the traffic on that link):

RETRIEVE (TRAFFIC-ON-LINK.name) WHERE TRAFFIC-ON-LINK.traffic$\geq$100

The above query is represented by the function symbol BUSY-LINKS, that takes a single argument which is a binary relation, and returns a unary relation. Functions of arity zero denote constants and relations of arity zero denote propositions.

In addition to database variables, we assume that we have another set of variable names, called *global variables*[3]. Each one of these variables is associated with a type. The global variables are used with the assignment quantifier, in order to capture values derived from the database at different instants of time. Specifically, the global variables retain the results of queries executed on different states of the history. The use of these variables will be clear when once we define the syntax and semantics of the logic. We assume that there is a special database variable called *time*, and its value denotes the time stamp associated with the database state.

The formulas of the logic are formed using the function and relation symbols, database variables and global variables, the logical symbols $\neg, \wedge$, the assignment quantifier $\leftarrow$, square brackets [,] and the temporal modal operators Until and Nexttime. In our logic, the assignment is the only quantifier. It binds a global variable to the result of a query in one of the database states of the history. A *term* is a variable or the application of a function to other terms. For example, $time + 10$ is a term; if x, y are variables and f is a binary function, then $f(x, y)$ is a term; the query BUSY-LINKS specified above is also a term. Well formed formulas of the logic are defined as follows. If $t_1, ..., t_n$ are terms of appropriate type, and R is an n-ary relational symbol, then $R(t_1, ..., t_n)$ is a well formed formula. If f and g are well formed formulas, then $\neg f$, $f \wedge g$, f Until g, Nexttime f and $([x \leftarrow t]f)$ are also well formed formulas, where x is a global variable and t is a term of the same type as x that does not contain any global variables; such a term t may represent a query on the database. A global variable x appearing in a

[3]The global variables are similar to the rigid variables in [24]

formula is *free* if it is not in the scope of an assignment quantifier of the form $[x \leftarrow t]$. A formula without any free global variables will be called a *trigger condition*, or, in short, a *trigger*.

We present below some example formulas in the logic. If the query called $traffic(link1)$ retrieves the traffic on link1 in a communication network, then the following formula asserts that the traffic on link1 continues to be zero until it jumps to 100:

$\mathit{traffic}(\mathit{link1}) = 0 \;\; \mathsf{Until} \;\; \mathit{traffic}(\mathit{link1}) = 100.$

The formula given below, which uses the assignment and the nexttime operators, indicates an update of traffic on link1:

$[x \leftarrow traffic(link1)] \;\; \mathsf{Nexttime} \; traffic(link1) \neq x.$

This formula should be read as follows: if the global variable x denotes the value of traffic(link1) in the present state, then the value of traffic(link1) in the next state is not equal to x. Notice how we used the global variable x to capture the value of traffic(link1) in the first state and compared it with the value of traffic(link1) in the next state. Traditional database triggers fire when a certain update occurs (e.g. the traffic on link1 increases). The above example shows that these triggers can be elegantly specified in our logic using the Nexttime and the assignment quantifier.

2.3 Semantics

Intuitively, the semantics are specified in the following context. Let s_0 be the state of the database when a trigger f is activated. At each future database state s_i, the formula f is evaluated on the history starting with s_0 and ending with s_i. If f is satisfied then the trigger is *fired*, namely the user is notified or the associated action is executed; in addition, f is reactivated starting at the next database state s_{i+1}.

We define the formal semantics of our logic as follows. We assume that each type used in the logic is associated with a domain, and all the variables of that type take values from that domain. We assume a standard interpretation for all the function and relation symbols used in the logic. For example, $\leq$ denotes the standard less-than-or-equal-to relation, and $+$ denotes the standard addition on integers. Global variables capture information from various database states in the history. For this reason, we will define the satisfaction of a formula at a state on a history with respect to an evaluation, where an *evaluation* is a mapping that associates a value with each global variable. For example, consider the formula mentioned above, $[x \leftarrow traffic(link1)]$ $\mathsf{Nexttime}$ $traffic(link1) \neq x$, that is satisfied when the traffic on link1 is updated. The satisfaction of the subformula $traffic(link1) \neq x$ depends on the result of the query that retrieves the traffic on link1 from the current database, as well as on the value of the global variable x. The value assigned to x is the value of the traffic on link1 in the previous database state.

The definition of the semantics proceeds inductively on the structure of the formula. If the formula contains no temporal operators and no assignment (to the global variables) quantifiers, then its satisfaction at a state of the history depends exclusively on the values of the database variables in that state and on the evaluation. A formula of the form f Until g is satisfied at a state with respect to an evaluation ρ, iff one of the following two cases holds: either g is satisfied at that state, or there exists a future state in the history where g is satisfied and until then f continues to be satisfied. A formula of the form Nexttime f is satisfied at a state with respect to an evaluation, iff there is a next state and the formula f is satisfied at the next state of the history with respect to the same evaluation. Note that, for all f, the formula Nexttime f is not satisfied in the last state of a history. A formula of the form $[x \leftarrow t]f$ is satisfied at a state with respect to an evaluation, iff the formula f is satisfied at the same state with respect to a new evaluation that assigns the value of the term t to x and keeps the values of the other global variables unchanged. A formula of the form $f \wedge g$ is satisfied iff both f and g are satisfied at the same state; a formula of the form $\neg f$ is satisfied at a state iff f is not satisfied at that state.

In our formulas we use the additional propositional connectives $\vee$ (*disjunction*), $\Rightarrow$ (*logical implication*) all of which can be defined using $\neg$ and $\wedge$. We will also use the additional temporal operators Eventually and Always which are defined as follows. The temporal operator Eventually f asserts that f is satisfied at some future state, and it can be defined as *true* Until f. The temporal operator Always f asserts that f is satisfied at all future states, including the present state, and it can be defined as $\neg$ Eventually $\neg f$.

We would like to emphasize that, although the above context implies that f is evaluated at each database state, our processing algorithm avoids this overhead by specifying a set of materialized views to the database management system; the evaluation takes place only when one of the materialized views changes, i.e., only when an update that is relevant to f occurs. The algorithm performing the evaluation is given as a parameter the set of changes to the materialized views.

2.4 Examples

The following formula, called OVERLOAD, uses the assignment quantifier and the Eventually operator. It indicates that the traffic on link1 doubles within ten minutes. The formula states that there exists a future state such that: if x, t denote the values of traffic(link1) and *time* in that state, then there exists another state after it in which the value of traffic(link1) is at least $2x$ and the time is less than or equal to $t + 10$. Here, x and t are global variables. We split the formula over two lines.

(A) Eventually $[t \leftarrow time][x \leftarrow traffic(link1)]$

$$(\text{ Eventually } (traffic(link1) \geq 2x \wedge time \leq t+10)).$$

Now consider a trigger that is to fire when the traffic on any link doubles, not just when the traffic on link1 doubles. Let *Traffic-on-Link* be a relation that contains link names and the traffic on each link. The new trigger can be specified in our logic by a modest modification to formula A above. Specifically, the following formula asserts that the traffic on some link doubles within ten minutes.

(B) $\quad \textsf{Eventually} \ \ ([t \leftarrow time] \ [x \leftarrow \text{Traffic-on-Link}]$

$$\textsf{Eventually} \ \ (\textbf{query1} \wedge time \leq t+10)$$

In the above formula **query1** is a traditional database predicate. It refers to two relations x and $Traffic-on-Link$ each of which has two attributes $name$ and $traffic$. This predicate evaluates to **true** if there is a tuple t in x and a tuple u in $Traffic-on-Link$ such that $u.name = t.name$ and $u.traffic \geq 2(x.traffic)$. This predicate can easily be specified in any relational query language, e.g. SQL.

In the above logic, we are explicitly using the time variable to compare the time at different points on the history. We can make the formulas more readable by omitting the time variable from the formulas, and using time constants as subscripts of temporal operators. To do so, we augment the language with bounded modal operators of the form $\textsf{Until}_{\leq c}$, $\textsf{Until}_{\geq c}$ and $\textsf{Until}_{= c}$, where c is some positive constant. Intuitively, $f \ \textsf{Until}_{\leq 10} \ g$ asserts that within ten minutes the formula g holds, and until then f continues to hold. We also use the bounded operators $\textsf{Eventually}-\textsf{within}-\textsf{c} \ (g)$ as an abbreviation for the formula $True \ \textsf{Until}_{\leq c} g$.
Formal definitions of the above operators are given below. For any formulas f and g,

$$f \ \textsf{Until}_{\leq c} \ g \equiv [t \leftarrow time](f \ \textsf{Until} \ (g \wedge time \leq t+c)),$$

The formal definitions of the other bounded operators $\textsf{Until}_{= c}$ and $\textsf{Until}_{\geq c}$ are similar (using $=$ and $\geq$ in place of $\leq$). Using the bounded operators, the formula (A) can be expressed as:

(C) $\quad \textsf{Eventually} \ \ ([x \leftarrow traffic(link1)]$

$$\textsf{Eventually}-\textsf{within}-10 \ (traffic(link1) = 2x)).$$

Our logic is more expressive than some of the existing temporal database query languages, such as the Temporal Calculus of [37]. For example, consider the trigger SIMULTANEOUS-INCREASE that states the following condition: In a period of at least 50 minutes, whenever the traffic on link1 doubles in an update, then the traffic on link2 also doubles in the same update. This trigger can not be expressed in the Temporal Calculus of [37]. However, this trigger can be specified in our logic by the formula $g \ \textsf{Until}_{\geq 50} True$ where g is the following formula.

$[x \leftarrow traffic(link1)][y \leftarrow traffic(link2)]$

$$\textsf{Nexttime} \ (traffic(link1) \geq 2x \Rightarrow traffic(link2) \geq 2y).$$

The formula g $\mathsf{Until}_{\geq 50}$ $True$ states that the property g is satisfied continuously for at least the next 50 minutes, where g states the following property: if x,y denote the values of traffic(link1) and traffic(link2) in the present state, then in the next state $traffic(link2) \geq 2y$ if $traffic(link1) \geq 2x$.

3 Detection Of Trigger Conditions in FTL

In this section we present a method for detecting if a given trigger condition, specified in FTL, is satisfied by a database history. We let f denote the given trigger condition. We describe a component that should be added to a database management system in order to process the condition f. This component works as follows. It presents the database queries that appear in f as materialized views to the database management system. The component is activated by the database management system whenever a database update occurs. The component uses the following algorithm to determine if the trigger condition f is satisfied by the database history up to the current state. We make the assumption that the database is referred to using the materialized views; that is, all predicates referred to in the formulas only involve the views, the global variables, and the time variable.

3.1 Informal Description of the Algorithm

3.1.1 Requirements Graph

The algorithm maintains a *directed, rooted, acyclic, and-or* graph G, called the *requirements graph.* The nodes of G are classified as and-nodes or as or-nodes with the root being an or-node. All the terminal nodes (i.e. nodes having no outgoing edges) of the graph are also or-nodes. Each or-node of the graph is marked with a *requirement.* A requirement is a pair of the form (g, ρ), where g is a component formula of the original formula f (or the negation of such a formula), and ρ is an evaluation on those global variables that appear free in g. Recall that an evaluation is a mapping that assigns values to the global variables; the assigned values are instances of the materialized views in some past database states. We say that a requirement, (g, ρ) is satisfied by a history if the history satisfies the formula g with respect to the evaluation ρ.

The root node of G is labeled with the requirement (f, ξ) where f is the trigger formula. The graph G can be considered as the syntax diagram of a boolean formula over boolean variables representing the terminal requirements (i.e. requirements marking the terminal nodes). This boolean formula denotes which subsets of the terminal requirements need to be satisfied by the future history, in order for the trigger condition f to be satisfied by the entire history.

The following example illustrates the intuition behind the construction of G. Consider the formula f given by $\mathsf{Eventually}$ $(P \wedge (Q$ Until $R))$ where P, Q and R are predicates on the database state. Intuitively, f is satisfied in the following cases. Either $P \wedge (Q\ \mathsf{Until}\ R)$ is currently satisfied (i.e. in the first state), which means P is satisfied in the current state and $Q\ \mathsf{Until}\ R$ is currently satisfied, or $\mathsf{Eventually}\ (P \wedge (Q\ \mathsf{Until}\ R))$ is satisfied by the future history starting from the next state. The first of the above cases is further decomposed into the following subcases: P and R are satisfied in the current state, or P and Q are satisfied in the current state and $Q\ \mathsf{Until}\ R$ is satisfied by the future history starting from the next state. Now, assume that P and Q are satisfied by the current database state, while R is not satisfied by it. From the above analysis, it should be easy to see that the only way $\mathsf{Eventually}\ (P \wedge (Q\ \mathsf{Until}\ R))$ can currently be satisfied is if the future history satisfies either of the formulas $Q\ \mathsf{Until}\ R$ or $\mathsf{Eventually}\ (P \wedge (Q\ \mathsf{Until}\ R))$. This is indicated by the graph G which has the following nodes and edges after the first state (i.e. first update). There is an edge from the root to two and-nodes, denoted by x and y. There is an edge from node x to a terminal or-node labelled with the rquirement $(Q\ \mathsf{Until}\ R, \xi)$, and another edge from y to another terminal or-node labelled with the requirement $(\ \mathsf{Eventually}\ (P \wedge (Q\ \mathsf{Until}\ R)), \xi)$.

In the above example, both the and-nodes have only one successor. In general, the and-nodes may have more than one successor. For example, consider the formula $f = f_1 \wedge f_2$ where f_1 is given by the previous formula and f_2 is $Q\ \mathsf{Until}\ R'$. Assume that in the first state P and Q are satisfied, while R and R' are not satisfied. In this case, using the previous arguments, it should not be difficult to see that G will have two and-nodes x and y as above, each having an additional terminal successor node marked with the requirement $(Q'\ \mathsf{Until}\ R)$. The requirements graph for this case is given in figure 1.

In the above example, all the evaluations were empty due to the fact the trigger did not have any global variables. In general, this may not be the case. The evaluations in the terminal requirements of G capture the information (from the past history) that is necessary in order to determine whether or not the trigger condition holds.

We say that a requirement (g, ρ) is a *temporal requirement* if g contains at least one temporal operator, otherwise we call it a *nontemporal requirement*. Note that the satisfaction of a nontemporal requirement only depends on the current state, not on the future history.

In summary, the graph G captures the information about the past history that is needed to monitor the trigger condition. It also specifies which combination of the different terminal requirements need to be satisfied by the future history in order for the trigger to be satisfied.

3.1.2 Overview

We assume that the database history starts when the trigger is entered. Every new update extends the history by appending the new database state to the current history. Whenever at least one of the materialized views is updated, the following algorithm will be executed. The algorithm is informed which views have changed, and what the changes are. The algorithm accesses the views during its execution. For ease of exposition, we assume that during the execution of the algorithm no further database updates will be allowed. In the Discussion section we will explain that this is not necessarily so.

As explained before, the algorithm maintains the requirements graph G. Each or-node of the graph is marked with a requirement. The graph is divided into levels numbered starting from 0 onwards. Level i corresponds to the i^{th} database update. Each level is divided into a sublevel of and-nodes, followed by a sublevel of or-nodes. The edges in the graph only connect the or-nodes of a level to the and-nodes of the next level, or the and-nodes of a level to the or-nodes of the same level. For example, in figure 1, the nodes x,y form the and-sublevel and the terminal nodes form the or-sublevel of a single level.

The root node is an or-node and is the only node at level 0. The marking (g, ρ) of an or-node at level $i-1$ denotes a requirement on the future history beyond the $i-1^{st}$ update.

When the trigger is first entered into the system, the graph is initialized to contain the root node, and it is marked with the requirement (f, ξ). After $i-1$ updates the graph would have been generated up to level $i-1$. In this graph, each or-node at level $i-1$ is a terminal node. When the temporal component is invoked after the i^{th} update, the graph G is extended. Additionally, a labeling procedure is executed. In this procedure each node v of G is labelled with two boolean values, $label_1(v)$ and $label_2(v)$. These labels are computed from scratch during each invocation. $label_1(v)$ pertains to the history up to the i^{th} update, while $label_2(v)$ pertains to the future history. More specifically, $label_1(v)$ will be set true if the history up to the i^{th} update satisfies the requirement labelling v; otherwise it will be set to false. $label_2(v)$ will be set true if some extension of the history beyond the i^{th} update can satisfy the requirement labelling v; otherwise it will be set to false.

Now we discuss the steps of the algorithm executed as a result of the i'th update in further detail. Consider a terminal node u of the requirements graph G. Let $r = (g, \rho)$ be a requirement with which u is marked. The procedure given in the next subsection is applied to the requirement r to compute a collection $\mathcal{C}^u$ of sets of requirements. For each C_i in $\mathcal{C}^u$, all the non-temporal requirements in it are evaluated in the current database state. Any C_i containing at least one of the non-temporal requirements that is not satisfied in the current database state, is immediately deleted from $\mathcal{C}^u$. This means that for each C_i remaining in $\mathcal{C}^u$, all non-temporal requirements are satisfied. If $\mathcal{C}^u$ contains at least one set all of whose requirements are non-temporal, then $label_1(u)$ is set to true; otherwise, it is set to false. If $\mathcal{C}^u$ is non-empty then $label_2(u)$ is set to true; otherwise, it is set to false. Note that if $label_2(u)$ is set to false then $label_1(u)$ would also have been set to false. After this, all the sets that only contain non-temporal requirements sets are deleted from $\mathcal{C}^u$. Therefore, each C_i's remaining in $\mathcal{C}^u$ satisfies the following condition. C_i contains a temporal requirement, and all its nontemporal requirements are satisfied in the current database state.

After the above procedure, the values of $label_1(u)$ and $label_2(u)$ are propagated upwards, i.e. to lower levels, in a straight-forward manner. See step 2 in the formal description.

$label_1(\epsilon)$ will be set to true by the above procedure iff the history up to the i^{th} update satisfies the trigger. In this case, the trigger will be fired. The condition $label_2(\epsilon) = \mathsf{false}$ implies that the the history up to the i^{th} update and no future extension of it can satisfy the trigger. If $label_2(\epsilon) = \mathsf{false}$ then the user is notified that the trigger can never be satisfied.

If neither of the above conditions is satisfied then the graph G is extended to the next level as given in step 4 of the formal description. After this the temporal component is exited.

3.1.3 Generating Sets of Requirements

The input to this step of the algorithm is a single requirement r of the form (g, ρ) marking a terminal node in the graph G. It returns a collection $\mathcal{C}$ of sets of requirements. Each requirement in each set of the returned collection , i.e. $\mathcal{C}$, is either a non-temporal requirement or is a temporal requirement of the form ($\mathsf{Nexttime}\ h, \delta$), i.e. the temporal formula in the requirement starts with the Nexttime operator.

The algorithm uses a special state predicate called *last-state*. This predicate is introduced only for convenience in the description of the algorithm, and is not interpreted in the database history. When the temporal component is exited, none of requirements in the sets belonging to $\mathcal{C}$ refer to this predicate. However, during the execution of the algorithm some of the requirement sets may contain a requirement of the form $(\textit{last-state}, \rho)$. The presence of this in a requirements set indicates that all the requirements in that set should be satisfied by the history up to the present state, i.e. the present state is the last state. For example, during the execution of the algorithm if a requirements set contains the requirements $(\textit{last-state}, \rho)$ and $(P\ \mathsf{Until}\ Q, \rho')$, then this requirements set indicates that Q should be satisfied in the present state, and its satisfaction cannot be postponed to the next state, since it is assumed that there is no next state.

The algorithm maintains a collection $\mathcal{C}$ of sets of requirements, which is first initialized to contain the singleton set $\{(g, \rho)\}$. During the execution of this step, the algorithm updates the collection as follows.

In this procedure the algorithm chooses one of the sets C in $\mathcal{C}$, and a temporal requirement (e, ρ) in C such that the outermost connective of the formula e is not the Nexttime operator. It takes one of the following actions, depending on the outermost connective of the formula e. The following cases are considered.

- The outermost connective of e is the temporal operator Until , and $e = g\ \mathsf{Until}\ h$. The formula e is satisfied in the current state if either h is satisfied in the current state, or g is satisfied in the current state and $g\ \mathsf{Until}\ h$ is satisfied in the next state. Using this observation, we replace C by the following two sets. The first set is obtained from C by replacing the pair (e, ρ) by the pair (h, ρ); this corresponds to the first possible way of satisfying g Until h. The second set is obtained from C by replacing the requirement (e, ρ) by the two requirements (g, ρ) and (Nexttime $(g$ Until $h), \rho)$; this corresponds to the second possible way of satisfying g Until h.

- The outermost connective of e is the assignment quantifier, and $e = [x \leftarrow t]g$. The formula e is satisfied with respect to the evaluation ρ, iff the formula g is satisfied with respect to the evaluation ρ' where the value of $\rho'(x)$ is the value of the term t in the current database state and for every other variable y, the value of y in ρ' is same as its value in ρ. The value of the term t in the current database state is retrieved from the materialized views. The requirement (e, ρ) in C is replaced by the requirement (e, ρ').

- The outermost connective of e is $\wedge$, and $e = g \wedge h$. Clearly, e is satisfied iff both g and h are satisfied. The requirement (e, ρ) in C is replaced by the two requirements (g, ρ) and (h, ρ).

- The outermost connective of e is $\neg$, and $e = \neg e'$. In order to process this case, we need to "push the negation down" in the formula e; we achieve this effect by considering the outermost connective of the formula e'. In other words, we consider the next outermost connective. We divide this case into the following subcases depending on the outermost connective of the formula e'.

  - If the outermost connective of e' is also $\neg$, i.e. $e' = \neg g$, then the requirement (e, ρ) is satisfied iff the requirement (g, ρ) is satisfied. The requirement (e, ρ) in C is replaced by (g, ρ).
  - The outermost connective of e' is Until , and $e' = g$ Until h. In this case $e = \neg(g$ Until $h)$, and it is satisfied at the current state iff g and h are both not satisfied at the current state, or h is not satisfied at the current state and $\neg$ Nexttime $(g$ Until $h)$ is satisfied in the current state. This observation can be seen from the definition of the semantics of the Until operator. In this case, the set C in $\mathcal{C}$ is replaced by the following two sets: the first set is obtained from C by replacing the requirement (e, ρ) by the to requirements $(\neg h, \rho)$ and $(\neg g, \rho)$; the second set is obtained from C by replacing the requirement (e, ρ) by the two requirements $(\neg h, \rho)$ and $(\neg$ Nexttime $(g$ Until $h), \rho)$.
  - The outermost connective of e' is $\wedge$, and $e = \neg e' = \neg(g \wedge h)$. Clearly, e is satisfied iff one of $\neg g$ and $\neg h$ is satisfied. The set C is replaced by the following two sets: the first set is obtained from C by replacing the requirement (e, ρ) by $(\neg g, \rho)$, and the second set is obtained from C by replacing the requirement (e, ρ) by $(\neg h, \rho)$.
  - The outermost connective of e' is the assignment quantifier, and $e = \neg e' = \neg[x \leftarrow t]g$. From the semantics of the assignment quantifier, it is easy to see that $\neg[x \leftarrow t]g$ is satisfied iff $[x \leftarrow t]\neg g$ is satisfied. The requirement (e, ρ) in C is replaced by the requirement $([x \leftarrow t]\neg g, \rho)$.

– The outermost connective of e' is Nexttime , and $e = \neg e' = \neg$ Nexttime g. In this case, it should be easy to see that the formula $\neg$ Nexttime g is satisfied iff either there is no next state, i.e. the current state is the last state of the history and hence the state predicate *last-state* is satisfied in the current state, or Nexttime $\neg g$ is satisfied, i.e. there is a next state and $\neg g$ is satisfied in that state. The set C is replaced by the following two sets. The first set is obtained from C by replacing the requirement (e, ρ) by (*last-state*, ρ). The second set is obtained from C by replacing (e, ρ) by (Nexttime $\neg g, \rho$).

The above transformations to the collection $\mathcal{C}$ are carried out until each of the requirements in all the sets is either nontemporal, or is of the form (Nexttime g, ρ), i.e. the formula starts with the Nexttime operator. Notice that only temporal requirements whose formulas do not start with the Nexttime operator are replaced by the above algorithm. Whenever such a replacement takes place, the formula in each of the newly added requirement either starts with the Nexttime operator, or its length is less than the length of the formula in the requirement that is replaced. For this reason, it should be easy to see that the above procedure terminates with the collection of sets satisfying the desired property.

After the termination of the above procedure, any set C that contains a requirement of the form (*last-state*, ρ) and a temporal requirement will be removed. Also any set that contains conflicting requirements, i.e. two requirements of the form (g, ρ) and $(\neg g, \rho)$, will also be removed. In the resulting collection, all requirements of the form (*last-state*, ρ) will be deleted from all the sets. The resulting collection is returned.

3.2 Formal Presentation

A more formal description of the algorithm, overviewed in subsection 3.1.2, is given below. The algorithm maintains a directed, acyclic, and-or graph G. In this graph, with each node u we maintain three fields $type(u)$, $mark(u)$ and $level(u)$. $type(u)$ denotes if u is an and-node or it is an or-node. $mark(u)$ gives the requirement associated with u. This field is defined only for the or-nodes. $level(u)$ gives the level number of u. If $level(u) = i$ then u was created when the temporal component was invoked after the i^{th} update to the database.

When the trigger is entered, G is initialized to contain a single node u with $type(u) = or$,$mark(u) = (f, \xi)$ and $level(u) = 0$. After each update the following algorithm is executed.

This algorithm uses a procedure *comp-sets*() which takes as argument a single requirement and returns a collection of requirements sets. This function essentially implements the algorithm given in subsection 3.1.3.

1. For each terminal node u in G, perform the following steps.

   (a) $\mathcal{C}^u = comp\text{-}sets(mark(u))$;

(b) The collection of requirements sets $\mathcal{C}^u$ has the following property: each temporal requirement in all the sets is of the form ($\mathsf{Nexttime}\ g, \rho$). Delete from $\mathcal{C}^u$ those sets that have at least one nontemporal requirement that is not satisfied in the current database state.

(c) Save $\mathcal{C}^u$ for later use. Set $label_1(u)$ and $label_2(u)$ as follows. If $\mathcal{C}^u$ contains at least one set all of whose requirements are non-temporal, then set $label_1(u) = \mathsf{true}$; otherwise, set $label_1(u) = \mathsf{false}$. After this, if $\mathcal{C}^u$ is non-empty then set $label_2(u) = \mathsf{true}$; otherwise, set $label_2(u) = \mathsf{false}$. From $\mathcal{C}^u$, delete all sets that only contain non-temporal requirements.

2. For each non-terminal node u, compute $label_1(u)$ and $label_2(u)$ from the terminal nodes to other nodes in decreasing level numbers as follows. For each and-node (resp., or-node) u, $label_1(u)$ and $label_2(u)$ are set to the conjunction (resp., disjunction) of the corresponding values of its successors.

3. If the above procedure sets $label_1(\epsilon) = \mathsf{true}$ (recall that ϵ denotes the root node), then notify the user that the trigger is satisfied, and reactivate the trigger by initializing the graph G to consist of a single or-node marked with the requirement (f, ξ) and exit the temporal component. If $label_2(\epsilon) = \mathsf{false}$, then notify the user that the trigger will never be satisfied and purge the graph G. If neither of the above cases hold then go to the next step. It can be shown that if $label_2(\epsilon) = \mathsf{false}$ then $label_1(\epsilon) = \mathsf{false}$.

4. Delete from G all nodes v such that $label_2(v) = \mathsf{false}$. For each of the remaining terminal node u, do the following.

5. For each $C_i \in C^u$, perform the following actions. Create a new and-node v in the graph and set $level(v) = level(u) + 1$. For each temporal requirement of the form ($\mathsf{Nexttime}\ g, \rho$) in C_i do the following. If there already exists an or-node w such that $level(w) = level(u) + 1$ and $mark(w) = (g, \rho)$ (such a node could have been introduced when processing some other terminal node u') then introduce an edge from v to w. Otherwise, create such a node w with $level(w) = level(u) + 1$ and $mark(w) = (g, \rho)$, and introduce an edge from v to w.

The procedure *comp-sets* is given below. It takes as argument a requirement (g, ρ). It computes a collection $\mathcal{C}$ of sets of requirements. It first initializes $\mathcal{C}$ to contain the single set $\{(g, \rho)\}$. After this it updates $\mathcal{C}$ using the following iterative procedure.

While there exists a temporal requirement (e, ρ) in some requirements set C of $\mathcal{C}$

such that e does not start with $\mathsf{Nexttime}$ operator **do**

if $e = g \wedge h$ **then** replace (e, ρ) in C by the requirements (g, ρ) and (h, ρ);

if $e = [x \leftarrow t]g$ **then** replace (e, ρ) in C by (g, ρ') where $\rho'(x)$ is the value of the term t in the current database state, and for each other global variable y that is free in e, $\rho'(y) = \rho(y)$;

if $e = g$ Until h **then** replace the set C by the two sets $C_1 = C - \{(e, \rho)\} \cup \{(h, \rho)\}$ and $C_2 = C - \{(e, \rho)\} \cup \{(g, \rho), (\text{ Nexttime } e, \rho)\}$;

if $e = \neg(g \wedge h)$ **then** replace C by the two sets $C_1 = C - \{(e, \rho)\} \cup \{(\neg g, \rho)\}$ and $C_2 = C - \{(e, \rho)\} \cup \{(\neg h, \rho)\}$;

if $e = \neg$ Nexttime g **then** replace C by the two sets $C_1 = C - \{(e, \rho)\} \cup \{(\textit{last-state}, \rho)\}$ and $C_2 = C - \{(e, \rho)\} \cup \{(\text{ Nexttime } \neg g, \rho)\}$;

if $e = \neg(g$ Until $h)$ then replace C by the two sets $C_1 = C - \{(e, \rho)\} \cup \{(\neg g, \rho), (\neg h, \rho)\}$ and $C_2 = C - \{(e, \rho)\} \cup \{(\neg h, \rho), (\neg \text{ Nexttime } (g \text{ Until } h), \rho)\}$;

if $e = \neg(\neg g)$ **then** replace the requirement (e, ρ) in C by the pair (g, ρ);

if $e = \neg([x \leftarrow t]g)$ **then** replace the requirement (e, ρ) in C by $([x \leftarrow t]\neg g, \rho)$;

End of do

From $\mathcal{C}$, delete any set that contains conflicting requirements, i.e. requirements of the form (g, ρ) and $(\neg g, \rho)$. Also delete any set that contains a requirement of the form $(\textit{last-state}, \rho)$ and a temporal requirement. Delete any requirement of the form $(\textit{last-state}, \rho)$ from all the sets in $\mathcal{C}$. Return the final value of $\mathcal{C}$.

3.3 Example

We illustrate the operation of the algorithm using the OVERLOAD trigger given by the formula (A) in section 2. For the purpose of exposition we denote this formula by f, and rewrite it as $f =$ Eventually g where $g = [x \leftarrow traffic(link1)][t \leftarrow time]h$, $h =$ Eventually l, and $l = (traffic(link1) \geq 2x \wedge time \leq t + 10))$. In the operation of the algorithm, we use the equivalences, Eventually $g \equiv (true$ Until $g)$ and Eventually $l \equiv (true$ Until $l)$. Note that g, h and l are all subformulas of f. When the trigger is activated, the graph G has only one node marked with the requirement (f, ξ). Now consider the database history given below. In each database state, we only give the value of $traffic(link1)$ since that is the only database variable that appears in our history. Successive elements of the history are given by a pair where the first component of the pair is the value of $traffic(link1)$, and the second component is the value of $time$. The history has the following elements:

$(10, 1)(15, 2)(18, 5)(25, 8)...$

Note that in the above history, $traffic(link1)$ has the values $10, 15, 18$ and 25 at the times $1, 2, 5$ and 8 respectively.

For this example, whenever the temporal component is entered the graph G satisfies the following properties. All the and-nodes in G have only one successor since the formula does not have any conjunction of subformulas containing temporal operators. Thus, the trigger f is satisfied by a future history iff any of the requirements marking the terminal nodes is satisfied by the future history. Since the markings of all the terminal nodes are distinct, we can identify a terminal node u with the requirement that marks it. For example, $\mathcal{C}^{(f,\xi)}$ denotes $\mathcal{C}^u$ for the terminal node u with $mark(u) = (f, \xi)$. At each invocation of the temporal component, we give the requirements that mark the terminal nodes, instead of showing the entire graph G.

In addition, we also show the collection $\mathcal{C}^u$ after step 1a and after step 1b. At each of these steps, each set in $\mathcal{C}^u$ is a singleton set, and we simply show the corresponding requirement contained in it. Each requirement is indicated by a triple, or by a pair if the evaluation is empty; in case of a triple, the first component gives the formula, and the next two give the values of the variables x and t, respectively.

At the first state:	
Terminal nodes at the beginning of step 1:	(f, ξ)
After 1 iteration in *comp-sets*, $\mathcal{C}$ contains	(g, ξ); $(\textsf{ Nexttime } f, \xi)$
After 2 iterations	$(h, 10, 1)$; $(\textsf{ Nexttime } f, \xi)$
	**requirement (g, ξ) processed
After 3 iterations	$(l, 10, 1)$; $(\textsf{ Nexttime } h, 10, 1)$; $(\textsf{ Nexttime } f, \xi)$
	**$(h, 10, 1)$ processed
$\mathcal{C}^{(f,\xi)}$ after step 1a	$(l, 10, 1)$; $(\textsf{ Nexttime } h, 10, 1)$; $(\textsf{ Nexttime } f, \xi)$
$\mathcal{C}^{(f,\xi)}$ after step 1b	$(\textsf{ Nexttime } h, 10, 1)$; $(\textsf{ Nexttime } f, \xi)$
	**the requirement $(l, 10, 1)$ is not satisfied at state 1
Terminal nodes after step 4:	$(h, 10, 1)$; (f, ξ)
At the second state:	
Terminal nodes at the beginning of step 1:	$(h, 10, 1)$; (f, ξ)
$\mathcal{C}^{(h,10,1)}$ after step 1a	$(l, 10, 1)$; $(\textsf{ Nexttime } h, 10, 1)$;
$\mathcal{C}^{(h,10,1)}$ after step 1b	$(\textsf{ Nexttime } h, 10, 1)$;
	** $(l, 10, 1)$ is not satisfied and deleted;
$\mathcal{C}^{(f,\xi)}$ after step 1a	$(l, 15, 2)$; $(\textsf{ Nexttime } h, 15, 2)$;$(\textsf{ Nexttime } f, \xi)$
$\mathcal{C}^{(f,\xi)}$ after step 1b	$(\textsf{ Nexttime } h, 15, 2)$;$(\textsf{ Nexttime } f, \xi)$
	**$(l, 15, 2)$ deleted;
Terminal nodes after step 4:	$(h, 10, 1)$; $(h, 15, 2)$; (f, ξ)
At the third state:	
At the beginning of step 1:	$(h, 10, 1)$; $(h, 15, 2)$; (f, ξ)
$\mathcal{C}^{(h,10,1)}$ after step 1a	$(l, 10, 1)$; $(\textsf{ Nexttime } h, 10, 1)$;
$\mathcal{C}^{(h,10,1)}$ after step 1b	$(\textsf{ Nexttime } h, 10, 1)$;

	**$(l, 10, 1)$ deleted
$\mathcal{C}^{(h,15,2)}$ after step 1a	$(l, 15, 2)$; ($\mathsf{Nexttime}\ h, 15, 2)$;
$\mathcal{C}^{(h,15,2)}$ after step 1b	($\mathsf{Nexttime}\ h, 15, 2)$; ** $(l, 15, 2)$ deleted
$\mathcal{C}^{(f,\xi)}$ after step 1a	$(l, 18, 3)$; ($\mathsf{Nexttime}\ h, 18, 3)$;($\mathsf{Nexttime}\ f, \xi)$
$\mathcal{C}^{(f,\xi)}$ after step 1b	($\mathsf{Nexttime}\ h, 18, 3)$;($\mathsf{Nexttime}\ f, \xi)$
	**$(l, 18, 3)$ deleted
Terminal nodes after step 4:	$(h, 10, 1)$; $(h, 15, 2)$; $(h, 18, 3)$; (f, ξ)

At the fourth state:

$\mathcal{C}^{(h,10,1)}$ after step 1b	$(l, 10, 1)$; ($\mathsf{Nexttime}\ h, 10, 1)$;

Requirement $(l, 10, 1)$ is satisfied. In step 1c, the $label_1$ for terminal node $(h, 10, 1)$ is set to true. In step 3 $label_1$ of the root node ϵ is set true and the trigger is fired.

The requirements graph after the third update is shown in figure 2. In this figure all $\bullet$ nodes are and-nodes and all $\circ$ nodes are or-nodes.

3.4 Correctness and Optimization

Correctness

We prove the correctness of the previous algorithm and discuss some important optimization techniques.

The following theorem asserts the correctness of the algorithm. Because of space limitations the proof is omitted. It can be found in [36].

THEOREM 3.1: Consider an invocation of the temporal component. Let h be the history since the time when the trigger f is entered or reactivated. During the execution of step 3, the algorithm indicates the satisfaction of the trigger condition f iff h satisfies f. Also during this step, if the algorithm indicates that the trigger will never be satisfied then, for every possible future history h', hh' does not satisfy f.

Optimization

The algorithm can be optimized to further eliminate certain requirements sets that will never be satisfied. For example, currently substitution of values for global variables occurs only in nontemporal formulas (in step 1b of the algorithm). If the resulting formulas are not satisfied in the current database state, then the corresponding sets of requirements are eliminated. We can modify the algorithm to substitute for certain global variables (such as variables containing past time values) in temporal formulas as well. If such formulas are found to be unsatisfiable, the corresponding requirements sets will be eliminated in step 1b. For example, after substitution of the values in a temporal formula assume that the resulting formula is $\mathsf{Eventually}\ (g \wedge time \leq c)$, where c is a constant.

Recall that *time* is the time-stamp variable. Now, if the value of the current time-stamp is greater than c, then it is obvious that the current database state and none of the future database states will satisfy the above formula, because the time-stamp value is always increasing. In this case, the corresponding requirement and the associated requirements set can be eliminated. This type of optimization is particularly useful for formulas with bounded temporal operators. The effect of this optimization in our OVERLOAD example is that the algorithm will only maintain the traffic on link1 for the last ten minutes.

3.5 Implementation

We briefly discuss here some of the implementation issues of the above procedure, particularly the maintenance of the evaluations in the requirements sets. We will describe the data structures needed to maintain these evaluations using the facilities of the DBMS. We will discuss this issue focusing on the relational data model, but the ideas can be easily extended to the object-oriented model. Our implementation involves maintaining some auxiliary relations R_{x_i}. For the object-oriented model, each one of these relations can be made into an object class.

We assume that in the original trigger f, for each global variable x, there is exactly one assignment of the form $[x \leftarrow q]$ where q is a database query. (If this condition is not satisfied, then we can rename the global variables to satisfy it). Clearly, all occurrences of x are within the scope of such an assignment quantifier because f has no free variables. q is a materialized view defined on the underlying database. We call q the query that x represents.

Now consider any requirement of the form (g, ρ). Clearly, the satisfaction of the requirement depends only on the values assigned to the global variables that are free in g. Thus, in each requirement it is enough to maintain the values of $\rho(x)$ for each x free in g. Let the free variables in g be $x_1, x_2, ..., x_m$, and let $q_1, q_2, ..., q_m$ be the queries denoted by these variables, respectively. Clearly, $\rho(x_i)$ is the value of the query q_i in a past database state. We store the above requirement as an $m+1$ tuple whose first component is the subformula g. Its $i+1$st component, for $i > 0$, is given as follows. If the value retrieved by q_i is always a single value, then we store $\rho(x_i)$ as the $i+1$st component of the tuple. For example, consider the OVERLOAD trigger in subsection 3.3. In this trigger there are two global variables, x and t. The queries assigned to these variables only retrieve single values, and each requirement set has only one requirement. An instance of a requirement set is $\{(h, 10, 1)\}$ where h is the subformula $\mathsf{Eventually}\ (traffic(link1) \geq 2x \wedge time \leq t + 10))$. Then 10,1 are the values of x and t respectively.

If q_i retrieves a relation, then the $i+1$st component is simply the value of some past time-stamp t, and in this case $\rho(x_i)$ is retrieved as a selection followed by a projection operation on an auxiliary relation R_{x_i} using the value of t. This will be made precise after the next paragraph.

Now we define the *auxiliary* relation R_{x_i}. Intuitively, R_{x_i} captures the tuples of the materialized view defined by q_i on some past database states as follows. If q_i retrieves a k-ary relation, then R_{x_i} has $k+2$ attributes. The first k attributes denote a tuple in a materialized view defined by q_i, and the last two attributes are the time attributes *beg-time* and *end-time*; they specify the beginning and ending of the time interval during which this tuple is in the materialized view, namely, is *valid*. If this tuple is currently valid, then the value of the attribute *end-time* has some large value, say MAX. Initially, R_{x_i} is set so that the projection on its first k attributes is the view defined by q_i on the initial database state, and in all the tuples the value of the *beg-time* attribute is set to the initial time stamp, and the value of the *end-time* attribute is set to MAX.

Subsequently, if the materialized view defined by q_i changes, then R_{x_i} is updated as follows. Assume that A and D, respectively, are the sets of tuples added to and deleted from the materialized view due to the change in the database state. Upon invocation, these are passed as parameters to the temporal component. Before deriving new requirements sets, the first major step updates the auxiliary relations as follows. For each tuple $a = (a_1, ..., a_k) \in A$, the tuple $(a_1, a_2, ..., a_k, T, MAX)$ is added to R_{x_i}, where T is the current time. For each $d \in D$, there is a tuple in R_{x_i} whose first k components constitute d, and whose *end-time* attribute has the value MAX; for this tuple, the value of the *end-time* attribute is changed to be $T - 1$, where T is the current time. Now, it is easy to see that the query q_i at some time t is a selection on R_{x_i}, followed by a projection. The selection is of the set of tuples for which the condition *beg-time*$\leq t \leq$*end-time* is satisfied. Then the attributes *beg-time* and *end-time* are projected out of the result.

It is implicit in the above discussion that tuples are never deleted from an auxiliary relation. When tuples are deleted from the database, only their *end-time* is updated in the auxiliary relation. However, for bounded temporal operators tuples can be deleted from the auxiliary relations. Intuitively, if a trigger condition refers only to the history in a time-window of 10 minutes, then tuples that were deleted from the database more than 10 minutes ago can also be deleted from the auxiliary relations. Our algorithm can be modified to take advantage of this optimization, but the details are beyond the scope of this paper.

3.6 Complexity issues

Here we will comment on the space and time complexities of our algorithm. We consider three different parameters for analyzing the complexity. These are the length of the trigger specification, called m, the length of the history, called n, and the size of view-history, called p. The length of the history is simply the total number of states in the history. The size of the view-history is defined as the size of the materialized views when the trigger is entered, plus the total size of the updates to the views from the time the trigger is entered until it ends (i.e. until it fires or until it is deleted as a result of the determination that it will never fire). Note that both n and p increase with the length of the history.

The total space used by the algorithm consists of the auxiliary relations and the requirements graph. The size of the auxiliary relations is bounded by the view history, p. The requirements graph has n levels where each level corresponds to an update point,i.e. state. The number of or-nodes at any level can be bounded as follows. Corresponding to each requirement, there is at most one or-node at any level. Since each requirement consists of a subformula of the trigger specification f and an evaluation on the global variables appearing in it, the number of distinct requirments is bounded by the number of distinct subformulas of f multiplied by the number of distinct evaluations on the global variables. Since the number of global variables is bounded by m, the number of distinct evaluations is bounded by n^m. From this, it follows that the number of or-nodes at any level is $O(mn^m)$. Therefore the total number of or-nodes in the requirements graph is $O(mn^{m+1})$. Also, each or-node has at most 4^m successor and-nodes. (This follows from the fact that the procedure *comp-sets* given in subsection 3.2 generates at most 4^m sets of requirements for each invocation. This is due to the fact that the evaluation in any requirement (g', ρ') belonging to any set is uniquely determined by g', the input requirement to the procedure, and the current state of the database. As a consequence, the total number of sets in the returned collection is bounded by 2^m, which is the total number of distinct subsets of subformulas of f.) From this, we see that the total number of nodes in the requirements graph is $O(mn^{m+1}4^m)$. The size of each node in the graph is $O(m)$. Therefore, the size of the requirements graph is $O(n^{cm})$ for some small constant c. Hence the total space requirment is $O(n^{cm}) + O(p)$.

The time complexity of the algorithm is $O(n^{cm}p^d)$ for some constants c and d. This is obtained as follows. The total time taken for each invocation of the temporal component consists of two parts. The first part is the evaluation of nontemporal requirements on the auxiliary relations, in step 1b of the algorithm. Each such evaluation can be done in time polynomial in p, the sizes of the auxiliary relations. The total number of such evaluations can be shown to be $O(n^{cm})$ which together take time $O(n^{cm}p^d)$. The second part of the time complexity involves extending the requirements graph, and computing the values of $label_1$ and $label_2$. All this can be done in time linear in the size of the requirements graph.

For a trigger that uses only bounded temporal operators, the space and time complexities of the current algorithm (with the optimization given in subsection 3.4) depends only on the view-history and the number of updates in the the bounded history.

The performance can be further enhanced by combining multiple updates into one, as explained in this section under the “transactions” heading.

4 Past Temporal Logic PTL

In the previous sections we considered FTL as a language for specifying triggers. In this section we investigate the use of Past Temporal Logic (PTL) for specification of triggers. We also present an algorithm for monitoring triggers specified in this logic.

4.1 Model, Syntax and Semantics

The model that we use is same as that used in the case of FTL. The syntax of PTL is same as FTL except that the future temporal operator Until is replaced by its past counter part Since , and the future operator $\mathsf{Nexttime}$ is replaced by the past counter part $\mathsf{Lasttime}$. Thus, if f and g are PTL formulas then $\mathsf{Lasttime}\ f$ as well as $f\ \mathsf{Since}\ g$ are also PTL formulas.

As in the case of FTL, the PTL formulas are interpreted over time stamped database histories. The semantics of a formula is defined inductively on the structure of the formula. The formula $f\ \mathsf{Since}\ g$ is satisfied at a state s in a history h with respect to an evaluation ρ iff one of the following two cases holds: either g is satisfied at s in h with respect to ρ , or there exists a past state s' where g is satisfied with respect to ρ and since then f continues to be satisfied with respect to ρ, i.e. f is satisfied in all states occurring between s' and s (including s but excluding s'). The formula $\mathsf{Lasttime}\ f$ is satisfied at a state s in h with respect to an evaluation ρ iff s is not the first state in h and f is satisfied at the state that immediately precedes s in h. We use additional past operators $\mathsf{Previously}$ and $\mathsf{Throughout-the-Past}$ defined as follows. The formula $\mathsf{Previously}\ f$ asserts that sometime in the past f is satisfied, and it can be defined as $true\ \mathsf{Since}\ f$. The formula $\mathsf{Throughout-the-Past}\ f$ asserts that f is satisfied in all past states including the present state, and it can be defined as $\neg\ \mathsf{Previously}\ \neg f$.

A trigger in PTL is simply a PTL formula without any free global variables. We say that a history satisfies a trigger f if the last state of the history satisfies f with respect to the empty evaluation.

4.2 Examples

Below, we present some example triggers specified in PTL. The following trigger fires if the current time is past 5 O'clock and the traffic on link1 remained above 100 since 2 O'clock. In this formula, $time$ denotes the value of the time-stamp associated with a database state.

$(time \geq 5 \wedge (traffic(link1) \geq 100\ \mathsf{Since}\ time = 2))$.

The following trigger is fired if an update of traffic on link1 occurs.

$[x \leftarrow traffic(link1)]\ \mathsf{Lasttime}\ traffic(link1) \neq x$.

The above formula should be read as follows. If the value of the global variable x denotes the value of $traffic(link1)$ in the current state of the histroy, then the value of $traffic(link1)$ in the previous state is not equal to x. The following formula, called PAST-OVERLOAD, specifies the same property as the OVERLOAD trigger of section 2.4. It indicates that the traffic on link1 has doubled within ten minutes. The formula states that if x, t denote the values of traffic(link1) and $time$ in latest state, then there exists a past state in which the value of traffic(link1) is at most $.5x$ and the time is greater than or equal to $t - 10$. Here, x and t are global variables.

(A') $[t \leftarrow time][x \leftarrow traffic(link1)]$

$$(\ \mathsf{Previously}\ (traffic(link1) \leq .5x \wedge time \geq t - 10)).$$

As in the case of FTL, we can augment PTL with bounded operator $\mathsf{Since}_{\leq c}$, $\mathsf{Since}_{\geq c}$ and $\mathsf{Since}_{= c}$ where c is a positive constant. Intuitively, f $\mathsf{Since}_{\leq 10}$ g asserts that some time with in that past 10 units of time g holds and since then f continues to hold.

4.3 Detection of PTL triggers

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

Let f be the given trigger. 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 global variables that appear free in g.

Let $s_1, s_2, ...s_i, ...$ be a database history where s_i is the database 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 global 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 global 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 $g =$ $\mathsf{Lasttime}$ 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 $F_{g,i} = \mathsf{false}$. 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 4.1: 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 PAST-OVERLOAD trigger given by A'. For the purpose of exposition, we denote the formula A' by f and rewrite it as $f = [t \leftarrow time]\,[x \leftarrow traffic(link1)]\; g$ where $g =$ Previously h and $h = traffic(link1) \leq .5x \wedge time \geq t - 10$. In the operation of the algorithm we use the equivalences Previously $h = true$ Since h. We consider the following history (same as the one used in subsection 3.3) where each database state is represented as a pair containing the value of $traffic(link1)$ and $time$ in that order.

(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.

$F_{h,1} = F_{g,1} = (t \leq 11 \wedge x \geq 20)$; $F_{f,1} = false$;

$F_{h,2} = (t \leq 12 \wedge x \geq 30)$; $F_{g,2} = F_{h,2} \vee F_{h,1}$; $F_{f,2} = false$;

$F_{h,3} = (t \leq 15 \wedge x \geq 36)$; $F_{g,3} = F_{h,3} \vee F_{h,2} \vee F_{h,1}$; $F_{f,3} = false$;

$F_{h,4} = (t \leq 18 \wedge x \geq 50)$; $F_{g,4} = F_{h,4} \vee F_{h,3} \vee F_{h,2} \vee F_{h,1}$; $F_{f,4} = true$;

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

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 (as in the implementation of FTL triggers) and the query values can be retrived 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 as in the case of FTL.

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, similar to those used in case of FTL triggers, 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 global 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,

$F_{h,4} = \ (t \leq 30 \wedge x \geq 22); \quad F_{g,i} == \ F_{h,4} \vee F_{h,3} \vee F_{h,2} \vee F_{h,1}; \quad F_{f,4} = \ false;$

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.

5 Discussion

In this section, we will discuss various issues concerning our language and the processing algorithm.

External Events: In section 2 we indicated that external events, such as user-login, user-logout, transaction-begin and transaction-commit, can be easily handled in our model. This is achieved by having a special relation (or an object class), called EXT-EVENTS, which contains at any instance of time all the external events that occur at that time. We assume that the external events are instantaneous. This means that if the relation EXT-EVENTS is non-empty at any instance of time, then at the next-instance (i.e. the next clock tick) this relation will change. For example, if at time t, the relation EXT-EVENTS contains a tuple indicating the login of user X, then at time $t + 1$ this tuple will not be present in EXT-EVENTS.

Now, consider the trigger— "the value of attribute A remains positive while user X is logged in". This is expressed by the following FTL formula:

$$\mathsf{Eventually}\ \ (\textit{user-X-logs-in} \wedge (A > 0\ \ \mathsf{Until}\ \ \textit{user-X-logs-out}))$$

In the above trigger *user-X-logs-in* and *user-X-logs-out* are queries on the relation EXT-EVENTS. They check for the existence of tuples that indicate the occurrence of the events login and logout of user X.

Reactivation of Triggers: In our algorithms for detecting the triggers, after a trigger is fired the trigger is reactivated starting from the next database state as the first state of the new history. The reactivation of the trigger from "scratch" means that detection of the trigger condition is more selective. For example, consider the trigger condition OVERLOAD mentioned in section 2 (this trigger is satisfied if the traffic on link1 doubles within ten minutes). Suppose that at time T the traffic is 100, and at time $T + 2$ it is 200. Then at time $T + 2$ the trigger will fire and the user will be notified. Now, suppose that at the time instances $T + 3$ through $T + 20$ the traffic is 201. Then, the trigger will not fire again during this period, although technically speaking, at time $T + 3$ the OVERLOAD condition is satisfied (because the traffic at $T + 3$ is at least double the value at T).

An alternative strategy, that detects all the "firings" of the trigger, is to continue the previous activation instead of reactivating the trigger from the beginning. However, this approach may cause the firing of the trigger continuously in the above example, which may be undesirable. This can be avoided by attaching additional conditions to the the trigger specification. For example, we might add an auxiliary condition to the trigger, stating that there should be some minimum time difference between successive firings of the trigger. This necessitates keeping an additional variable that records when a trigger was last fired. This new variable needs to be updated appropriately by the temporal component.

The user may be allowed to select one of the two reactivation strategies discussed above.

Future Versus Past: In this paper, we have considered two different temporal logics— FTL which uses future operators and PTL which uses past temporal operators. It is to be noted that in both these cases we only consider the history from the time the trigger is entered or last re-activated, until the latest update.

From the point of view of expressiveness, it is well known that, when global variables are not used, both logics have the same expressive power. However, certain types of triggers can be specified concisely in FTL, while others can be concisely specified in PTL. For example, consider the FTL trigger given by the following formula, in which $I, P1, P2, Q1$ and $Q2$ are database predicates.

$\mathsf{Eventually}\ \ (I \wedge g)$ where g is given by the following formula

$(P1\ \mathsf{Until}\ Q1)\ \mathsf{and}\ (P2\ \mathsf{Until}\ Q2)$.

Intuitively, the above trigger states that there is a future instance when I is satisfied, and from that point onward $P1$ Until $Q1$ holds and $P2$ Until $Q2$ holds. The trigger is to be fired when the later of $Q1$ and $Q2$ is satisfied. Specification of the above trigger in PTL is more complex. It has to consider the different orders in which $Q1$ and $Q2$ are satisfied. If there are k conjuncts in g (rather than two), then the length of the equivalent PTL formula is exponential in k.

Similarly, there are PTL triggers that do not have a concise specification in FTL.

For future research, it will be interesting to investigate logics that permit the use of both, future and past operators in the same trigger.

Transactions: Next we comment on the coupling between the database update and the resulting temporal-component execution. We have assumed that the temporal component is invoked after each update. The first question arising is "what is considered an update"? We propose to define an update to be the set of changes made by a single committed transaction, i.e., updates of aborted transactions are ignored. There are other possible options for this definition of an update, but we will omit them here. For example, updates of aborted transactions may be important for tracking security violations.

The next question arising is the following. Should the invocation of the temporal component on behalf of a transaction T, be part of T? In active database literature, there have been studies on the coupling between the application transaction and the execution of the condition-action part of a rule [9][4]. We assume that the execution of the condition-action part is decoupled from the application transaction (it can be argued that this assumption is natural for conditions which are temporal triggers). Therefore, we assume that each invocation of the temporal component is executed as a separate transaction. The execution of the action part of the rule can be part of the transaction corresponding to the invocation of the temporal component, or it can be a separate transaction. Note that for future trigger monitoring, the action part is regarded as an application transaction.

Our method is as follows. At any time, for each trigger, there is at most one temporal-component invocation executing in the system. Furthermore, there is one temporal-component invocation per transaction. The invocation of the temporal-component on behalf of transaction T, gets the commit-time of T as a parameter. The invocation of the temporal component can be modified to process all the triggers affected by T at the same time.

In case of heavy application-transactions activity, the number of completed invocations of the temporal-component may increasingly fall behind the number of committed application-transactions. In other words, the number of pending invocations of the temporal component may grow unboundedly. We propose to solve this performance problem by redefining an update to be the set of changes of multiple committed transactions. More specifically, whenever the number of pending temporal-component invocations increases beyond some threshold, then we replace all the pending invocations by a single invocation, which considers the changes of all the corresponding application-transactions to be a single update. According to the model, this update must have a unique occurrence time. This can be, for example, the commit time of the last application transaction. Therefore, this scheme trades off precision for performance (precision is compromised since the occurrence time of some changes is modified).

There are other systems issues to be investigated here, which will be the subjects of future research.

Generic Triggers: This comment concerns the collective specification of multiple triggers in one formula. Such a collective specification is called a generic trigger. For example, suppose that the user needs to specify one OVERLOAD trigger for every link located in Chicago; the trigger for that link should fire when the traffic on the link doubles within 10 minutes. Our language can support generic triggers by allowing free database variables[5] in the definition of a trigger (recall that, according to our definition, a trigger is a formula without any free variables). Then, different no-free-variable-triggers can be created, one for each assignment of values to the free variables.

Therefore, a generic trigger is specified by a query on the database state, and a trigger with free variables. For example, the above generic trigger is specified by the free-variable-trigger:

[4] As mentioned in the introduction, a temporal trigger forms the condition part of a rule.

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

$$\textsf{Eventually}\ \ ([x \leftarrow traffic(y)]$$

$$\textsf{Eventually} - \textsf{within} - 10\ (traffic(y) \geq 2x))$$

together with the query q defined as: "RETRIEVE y where $location(y) = Chicago$".

Initially, a no-free-variable-trigger will be generated for each value of y retrieved by the query q on the initial database state. Subsequently, if the view defined by q changes, e.g. some new links located in Chicago are added, and some existing links located in Chicago are deleted. Then new triggers are activated, corresponding to the new values added to the view; and old triggers are deactivated, corresponding to the values deleted from the view.

Since multiple triggers corresponding to one generic trigger share the global variable, they can use the same auxiliary relations (see subsection 3.5). Also, the processing algorithm can be modified to handle all the triggers corresponding to one generic-trigger more efficiently than handling each one independently.

Aggregates: Another issue is how to use aggregates over time in trigger conditions. For example, we may want to define a trigger that fires if the average value of traffic(link1) in one hour time exceeds a certain threshold. Clearly, we cannot specify this in FTL or in PTL right now. However, we can easily modify the logics and the detection algorithms to allow terms that return some simple aggregate values. For example, we can use an aggregate function defined as SUM OF X FROM f where X is a database variable and f is nontemporal database predicate; this function, at any state in the history, returns the sum of the values of X from the last time when the database predicate f was satisfied. Further research is required on specifying and handling more complex aggregate functions that use temporal predicates as part of the aggregates, i.e. when f is a temporal formula.

Current Project Status: We have implemented the temporal component for a subset of the language FTL. The implementation is on top of the Sybase DBMS. The implementation consists of two subsystems. The first subsystem takes as an input an FTL trigger and defines a set of Sybase update triggers. When any of these triggers is fired by Sybase, then the second subsystem is invoked. This subsystem maintains the requirements sets and detects the satisfaction of the FTL trigger.

Observe that for the implementation of our trigger processing mechanism, it is not necessary that the underlying database system supports materialized views. We have made the assumption in the paper for the purpose of ease of exposition. Also observe that this implementation does not necessitate any modification to the Sybase code.

6 Comparison to Relevant Literature

First Order Logic: Query languages based on First Order Logic (FOL), such as SQL, can also be used to specify temporal triggers. In this case each relation, or type, must be augmented with the time attribute, and there must be separate relations containing a representation of the current external events. FTL and PTL are more intuitive since time is an attribute with special properties, e.g., it is monotonically increasing, and these logics have special operators for dealing with time naturally (i.e. the way it is used in natural language). Additionally, the triggers specified in FTL or PTL allow us to identify the least amount of information that needs to be saved over time in order to monitor the trigger, and to perform the monitoring incrementally. On the other hand, the FOL approach does not enable the easy identification of such minimal amount of information from the specification, and is not naturally amenable to incremental processing.

The following example illustrates the above point. Consider the simple FTL formula (P Until Q). In *First Order Logic* (FOL) this is expressed as: $\exists t[t \geq starttime \wedge Q(t) \wedge \forall t'\{(starttime \leq t' < t) \Rightarrow P(t')\}]$ where $starttime$ is the time when the trigger is entered. It is easy to see that the FOL formula is more complicated. It is also difficult to determine the past-database information that has to be saved in order to determine satisfaction of the trigger, and how to minimize this information. A straightforward way to evaluate the above trigger is to save the value of P and Q each time they change, and to evaluate the formula from scratch over this history of changes; the evaluation is performed after each change. On the other hand, our algorithm evaluates the FTL formula as follows. When the trigger is entered the algorithm checks whether the current database state satisfies Q. If so, the trigger is satisfied. Otherwise, the algorithm checks whether P is satisfied. If not, the formula is false (will never be satisfied). If P is satisfied, the above procedure is repeated after the next database update which changes either P or Q. Therefore, clearly, no information has to be saved from one database state to the next, and the evaluation is incremental. []

Other Temporal Logics: The work that is closest to ours is presented in [4, 5, 22, 23]. In [4, 5], 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 is similar to PTL, except that FPTL uses first order quantifiers whereas PTL uses the assignment operator. Additionally, the processing algorithms given in Chomicki's works can only handle the subclass of FPTL formulas f with the property that all subformulas of f are *domain independent* (see [38]). This restriction limits the applicability of his work for active databases. For example, consider the trigger condition PAST-OVERLOAD given in section 4. This can be expressed in FPTL as follows.

$\forall x\, \forall t\; ((x = load(link1) \wedge t = time) \rightarrow\; \mathsf{Previously}\; (load(link1) \leq .5x \wedge time \geq t - 10))$

However, the subformula ($load(link1) \leq 0.5x$) in the above formula is not domain independent, since, for any given value of $load(link1)$, it is satisfied by an infinite number of values of x. Hence, this trigger canot be processed using the algorithm given in [4, 5].

Also, Chomicki's logic and processing method are strongly tied to the relational model. The reason is that in his logic temporal operators are embedded in relational-database queries. In contrast, in FTL and in PTL database queries are functions that do not have temporal operators. For example, query1 in the quantifier $[x \leftarrow$ **query1** $]$ is not allowed to have temporal operators, thus it is specified in the language of (and processed by) the underlying DBMS. Therefore, the expressive power of our language depends on the expressive power of the query language of the underlying DBMS. Additionally, although we have assumed the existence of materialized views that have been studied particularly for the relational model, the concept of a materialized view applies to other data models as well.

Another difference between our approach and Chomicki's is that we concentrate on issues of maintaining triggers in active databases, whereas he concentrates on maintaining integrity constraints. Thus, we address the issues of trigger reactivation, integration with the transaction system, etc., which are not addressed in his works.

Finally, the length of the specifications of triggers in Chomicki's language is often proportional to the value of time constants in the trigger. This is obviously undesirable. For example, consider the following trigger: Events A, B, and C occur in this order, within 60 minutes. The only way to express this trigger in the logic of [5] is as follows: (B occurs within 1 minute before C , and A occurs within 59 minutes before B), or (B occurs within 2 minutes before C, and A occurs within 58 minutes before B), or... ,or (B occurs within 59 minutes before C, and A occurs within 1 minute before B). This specification will have 60 clauses. The above trigger can be expressed more concisely in the logic of [4]; however, the processing method of [4] is not space efficient, since it uses unbounded past temporal operators (the purpose of [5] is to rectify this problem). In contrast, the following FTL formula concisely expresses the above trigger, and it is processed by our algorithm (with the optimization given in subsection 3.4) in a space efficient way.

$$\text{A and } [T \leftarrow currenttime]$$
$$\quad \text{Eventually} \quad (\text{B and} \quad \text{Eventually} \quad (\text{C and } currenttime \leq T + 60)\ )$$

In the above formula T is a global variable and *currenttime* is a database variable that gives the value of the time. The above formula states that if A is satisfied in the current state and T has the value of *currenttime*, then eventually B occurs, followed by C; additionally, at the occurrence of C the value of *currenttime* is less than or equal to $T + 60$.

The work presented in [22, 23] considers temporal logic with future operators for specifying dynamic integrity constraints. In this work a procedure for maintenance of the integrity for a subset of the logic is given. This logic, called Propositional Temporal Logic, does not allow any first order quantifiers. Our logic is more expressive because it allows quantifiers. In addition, the model used in [22, 23] does not have time stamps associated with the database states. Due to this, they can specify the relative order of occurrences of different events, but cannot easily specify absolute timing properties of events, such as— event B occurs within 10 minutes after the occurrence of event A.

Event Expressions: Event expressions (EE) is another interesting formalism for specifying temporal triggers [14, 15]. 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. The trigger specified by the regular expression "$A \cdot B \cdot C$" will fire if A occurs, followed by B, followed by C.

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 [15]) are not used. In this case, the space complexity of our algorithm does not suffer from this super exponential blow up.

In this paper, we concentrated on specifying and efficiently processing the absolute timing properties. Our main algorithm together with the proposed optimizations (given in section 3.4) maintain the least amount of information needed for monitoring the trigger. In contrast, [14, 15] do not explicitly address the specification of absolute timing properties. One may speculate that in order to specify absolute timing properties of events in EE, one needs to assume 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 EE expressions.

Another difference between our logics and EE is that [14, 15] do not discuss temporal properties of database predicates (e.g. "attribute A is 50"); they concentrate on external events. Although it is possible that such predicates can be incorporated in the language, it is not readily apparent how to do so.

Our work also addresses systems issues such as transactions, no-update triggers, etc.

Other work on Triggers in Active Databases: The other relevant work is the work on active databases (e.g. [8, 9, 7, 18]), on triggers [6, 11, 26], and on rule-languages [21, 28, 30, 39, 20]. These works lack the temporal component in the following sense. They concentrate on the specification of triggers that involve two database states: the current one, and the previous one. In contrast, in this paper we address the more general temporal aspects in triggers.

Temporal Databases: The other database literature area that is relevant to our paper is temporal databases [33, 34, 32, 29]. The main difference between the present work and the work on temporal databases is that FTL is designed for triggers given a priori, whereas the latter is designed for ad hoc queries. Moreover, the work on temporal databases usually assumes an explicit time-attribute in the relation schemas, and proposes query languages with temporal capabilities. In this work we do not assume any explicit time attribute associated with relations or objects of the database. Of course, such an attribute can be automatically added by the DBMS to each relation schema, or object type. Then the prior work can be used by having the DBMS create a time-stamped snapshot of the database (or of the relation, or of the tuple, or of the attribute-value) at each update. This is the approach taken in [19]. This approach is reasonable when the updates are infrequent, and the DBMS should support any query on the past history. However, when the database is updated frequently, as in the case of real-time monitor-and-control applications, maintaining the whole past history is not feasible. In our case, the triggers are given a priori. We identify and save only the information of the database history that is relevant to the determination of whether or not the trigger will be satisfied in the future. Furthermore, we discard obsolete information (for example, for the trigger "the value of attribute X increases by 50% in 10 minutes", information becomes obsolete after 10 minutes). Regular query languages, that are designed to cope with arbitrary queries on the database history, are not equipped to make use of information in predefined triggers.

Another distinction is that most of the existing works on temporal databases concentrate on the relational model (see for example, [31, 19]), and add qualifiers to handle the temporal aspects of the database. In contrast, as explained above, FTL is independent of the underlying data model.

In [37] another logic called Temporal Calculus was proposed as a query language for temporal databases. However, our logic is more expressive than Temporal Calculus due to the capability to nest the temporal operators and the assignment quantifiers. For example, consider the trigger that states the following condition: In a period of one hour, whenever the traffic on link1 doubles in an update, then the traffic on link2 also doubles in the same update. In section 2 we demonstrated that this trigger can be expressed in our language; however, it cannot be expressed in the Temporal Calculus of [37] since it lacks the assignment quantifier. Also, we have presented a method for evaluating trigger conditions, whereas no such method is given in [37].

Work on Temporal Logic in other areas: There has also been work on interval logic [27] and maintenance of temporal databases in Artificial Intelligence [10]. However, the emphasis of this research is on the *inference* of temporal knowledge, rather than the *specification* of temporal constraints.

The assignment quantifier that we use is similar to the freeze quantifier defined in [17].

References

[1] R. Alur and T. Henzinger, *Real-time Logics: Complexity and Expressiveness*, Proceedings of the Fifth IEEE Symposium on Logic in Computer Science, pp 390-401, 1990.

[2] J.A. Blakeley, P.-A. Larson, F.Wm. Tompa, *Efficiently Updating Materialized Views*, Proceedings of the ACM-SIGMOD 1986, International Conference on Management of Data, Washington, D.C., May 1986; pp. 61-71.

[3] J.A. Blakeley, N. Coburn, P.-A. Larson, *Updated Derived Relations: Detecting Irrelevant and Autonomously Computable Updates*, ACM Transactions on Database Systems, 14(3), 1989.

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

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

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

[7] S. Ceri and J. Widom, *Production Rules in Parallel and Distributed Database Environments*, Proceedings of VLDB92, 1992.

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

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

[10] T. Dean, *Using Temporal Hierarchies to Efficiently Maintain Large Temporal Databases*, JACM 36(4), Oct. 1989.

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

[12] A. Dupuy, S. Sengupta, O. Wolfson, and Y. Yemini, *Design of the Netmate Network Management System*, Integrated Network Management II, I. Krishnan and W. Zimmer (Ed.), Elsevier Science Publishers B. V. (North Holland).

[13] A. Dupuy, S. Sengupta, O. Wolfson, Y. Yemini; *NETMATE: A Network Management Environment*, to appear in the special issue on Network Operations and Management, IEEE Network (The Magazine of Computer Communications), 1991.

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

[15] N. H. Gehani et. al., *Composite Event Specification in Active Databases: Model& Implementation*, Proceedings of the 18th International Conference on Very Large Databases, Aug. 1992.

[16] E.N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang, *A predicate matching algorithm for database and rule systems*, Proceedings of the ACM-SIGMOD 1990, International Conference on Management of Data, Atlantic City, NJ, May 1990; pp. 271-280.

[17] T. Henzinger, *Half-order Modal Logic: How to prove Real-time Properties*, Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing, pp 281-296, 1990.

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

[19] C.S. Jensen, L. Mark and N. Roussopoulos, *Incremental Implementation Model for Relational Databases with Transaction Time*, IEEE Trans. on Knowledge and Data Engineering, Dec. 1991.

[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] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems—Specification*, Springer-Verlag 1992.

[25] X. Qian and G. Wiederhold, *Incremental Recomputation of Active Relational Expressions*, IEEE Transactions on Knowledge and Data Engineering, 3(3), 1991.

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

[27] R. R. Razouk and M. M. Gorlik, A Real Time Interval Logic for Reasoning About Executions of Real-Time Programs, Proc. ACM-SIGSOFT Conf., 1989.

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

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

[30] 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.

[31] R. Snodgrass, *The Temporal Query Language TQuel*, ACM Trans. on Database Systems, 12(2), 1987.

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

[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, 1994.

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

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

[40] O. Wolfson, S. Sengupta, and Y. Yemini, *Managing Communication Networks by Monitoring Databases*, IEEE Transactions on Software Engineering, Vol. 17(9), Sept. 1991, pp. 944-953.

[41] Y. Yemini, O. Wolfson, *Netmate: Management of Complex Distributed Networked Systems* (project synopsis), Proc. 1st Int. Conf. on Parallel and Distributed Information Systems (PDIS), Dec. 1991.