Triggers on Database Histories

A. Prasad Sistla   Ouri Wolfson

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

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 as a database attribute A) 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 an object 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, prototypes, and proposed languages, do not provide the capability for specifying temporal triggers. In most of them, the condition part of a rule (in Hipac [1] 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 the complete database history.

2. Temporal Logic Based Languages for Triggers

Temporal Logic (TL) [3] 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 TL is that it has special operators that apply exclusively to the time dimension. *Until*, *Since*, *Next-time*, *Last-time*, *Eventually* and *Previously* are some of the widely used temporal operators. In our case, a trigger is a formula of TL, and it is interpreted over a history of system states and a *reference* time. A *system state* is a triple *(database-instance, set-of-external-events, time-stamp)*. Intuitively, a system state defines the database-state and the external events that occur at a particular time. A *history* is a set of system states with distinct time-stamps.

For triggers specified in TL, the history starts at the time when the trigger is entered into the system. The different temporal operators are classified as *past* operators and *future* operators. *Until*, *Next-time* and *Eventually* are future operators, while *Since*, *Last-time* and *Previously* are past operators. The fragment of TL that uses only the past operators is called *Past TL* (PTL) and the fragment the only uses the future operators is called *Future TL* (FTL).

The following are examples of PTL and FTL triggers. Suppose that predicate P on the database state is "A=50"; predicates Q and R (which refer to external events) are "transaction T begins" and "transaction T commits", respectively. Then, "R and (P *Since* Q)" is a PTL trigger. Intuitively, this trigger will fire at the time when T commits, provided that the value of A is 50 continuously from the time when T begins. The same trigger is specified in FTL as "Q and (P *Until* R)"[1].

[1] Strictly speaking, we need to have an additional *Eventually* operator in front to allow the beginning of transaction T any time after entering the trigger

It is to be noted that when interpreting PTL formulas as triggers, the reference time is taken as the latest time-stamp in the history. In other words, PTL triggers refer to the system states that have a time-stamp smaller than the latest time in the history. In contrast, when interpreting FTL formulas as triggers, the reference time is taken as the time when the formula is entered as a trigger.

PTL and FTL have the same expressive power, but in some cases one is easier and more natural to use than the other. For example, consider the following trigger specification in FTL where P1, P2, Q1 and Q2 are predicates on the database state:

(P1 *Until* Q1) and (P2 *Until* Q2).

For instance, the above trigger becomes reasonable for a stock trader under the following semantics. P1 states that "the IBM stock is less than 90", and P2 states that "the GM stock is less than 38", and Q1 states that "the DJ industrial average is more than 3000", and Q2 states that "the S&P average is more than 400".

The above trigger fires when the latter of the Q1 and Q2 is satisfied (provided that P1 and P2 hold respectively).

Assuming that the trigger is entered at 2pm, in past logic this trigger is expressed as follows.

[Q2 and (P2 *Since* (P1 and P2 *Since* 2pm))] or [Q1 and (P1 *Since* (P1 and P2 *Since* 2pm))]

Clearly the past logic formula is more complex. Actually, if there are k conjuncts (rather than 2) in the future logic formula, then the size of the past logic formula will be exponential in k. Intuitively, the reason for this is that one does not know the order in which the Q's occur, therefore the past logic formula has to account for all possible orders.

One can also use temporal formulas that combine both past and future operators to specify triggers.

In addition to the temporal operators, TL allows the use of global variables that enable comparison of database values derived at different system states. For example, consider the trigger mentioned in the introduction, the value of attribute A increases by more than 10% in 15 minutes. In FTL it can be expressed as follows.

$$\begin{array}{l} \textit{if } x \leftarrow A \textit{ then} \\ \textit{eventually within 15} \\ \quad A \geq 1.1x \end{array}$$

In the above formula x is a global variable, and eventually-within-15 is a composite FTL operator that is defined in terms of *Until* and a global variable for time. The above formula should be read as follows: if x is the value of attribute A at the reference time (i.e. when the trigger is entered), then within 15 minutes A has a value grater than or equal to $1.1x$. By adding the *Eventually* operator in front of the above formula we obtain a trigger that fires whenever the value of A increases by 10% in any period of 15 minutes, not just in the first 15 minutes after entering the trigger.

3. Comparison to other Specification Formalisms

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. TL is more intuitive since time is an attribute with special properties, e.g., it is monotonically increasing, and TL has special operators for dealing with time naturally (i.e. the way it is used in natural language). Additionally, the triggers specified in TL allow us to identify the least amount of information that needs to be saved over time in order to monitor the trigger. Whereas the FOL approach does not enable the easy identification of such information from the syntax of the specification. The following example illustrates the above point.

Example 1. Consider the simple TL 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. On the other hand, in [5] we proposed an incremental algorithm that evaluates the TL 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 from this description, no database information has to be saved from one database state to the next. In [5] this processing methodology is generalized to handle an arbitrary FTL formula. Of course, for the general case there may be information that needs to be saved from one state to the next, but the algorithm in [5] minimizes the amount of such information. []

Extensions of SQL to deal with temporal databases [4] were not designed for the specification of triggers. Therefore they also do not enable efficient processing in real time. Additionally, it is hard to reason about the semantics and the expressive power of such extensions, whereas the expressive power of TL has been studied extensively.

Event expressions (EE) is another formalism for specifying triggers [2]. 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. For example, the trigger "A,B,C" will fire if A occurs, followed by B, followed by C. The TL formalism allows one to specify time varying properties of external events and database predicates in a unified manner, whereas event expressions are mainly appropriate for specifying the order of external events. Furthermore, real-time properties, i.e. properties on the time of occurrence of different events, cannot be elegantly specified in event expressions. The following property is not easily expressible using event expressions: Events A, B, and C occur in this order, within 60 minutes. This property is easily expressed by the following FTL formula.

A and *if* $T \leftarrow$ *currenttime then*
Eventually
  (B and *Eventually*
    (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 $T + 60$.

An event expression is usually processed by constructing a finite-state automaton. Another drawback of the EE approach is the high complexity of the automaton-size. It can be superexponential in the size of the event-expression. When global variables are not used, the trigger processing algorithm given in [5] for FTL reduces to a finite state automaton run on the history. The size of the automaton in this algorithm does not suffer from the same high complexity as in the case of the EE approach.

At the level at which we discuss FOL, TL, and EE, in this paper, they are incomparable with respect to expressive power. Specifically, there are variants of TL that are more expressive than variants of EE and FOL, and vice versa. Particularly, it can be shown that a variant of TL without global variables is equivalent in expressive power to EE.

4. Conclusion and Systems Issues

In this short paper we considered the problem of specifying triggers on the evolution of database over time. We have proposed languages based on temporal logics for this purpose. We discussed two types of temporal logic, one based on future temporal operators and the other based on past temporal operators. Finally, we compared this approach with other formalisms, such as Event Expressions and First Order Logic based languages.

Regardless of the formalism in which triggers are expressed, there has to be an algorithm that is invoked at different transition points to detect the satisfaction of the trigger. A *transition point* is either a change to the system state (occurrence of an external event, or update to the database), or a commit of a transaction (when all its updates are considered to have occurred atomically), or a periodic time-instance (say every 10 seconds). How should this invocation be incorporated into the transaction processing mechanism, if at all? Should this invocation be a separate transaction that runs concurrently and serializably with other transactions, some of which may be invocations of the algorithm for other transition points? In case the invocation is due to the commit of a transaction, should it part of the transaction? We have investigated some of these issues, and the results will be presented in a forthcoming paper.

References

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

[2] N. H. Gehani et. al., *Composite Event Specification in ACtive Databases: Model& Implementation*, Proceedings of the 18th International Conference on Very Large Databases, Vancouver, Canada, August, 1992.

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

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

[5] A. P. Sistla and O. Wolfson, *Temporal Triggers in Active Databases*, Technical Report, Department of Electrical Engineering and Computer Science, Univ. of Illinois at Chicago, 1992.