THE DESIGN AND IMPLEMENTATION OF EPL: AN EVENT PATTERN LANGUAGE FOR ACTIVE DATABASES*

G. Giuffrida and *C. Zaniolo*

Computer Science Department
The University of California
Los Angeles, California 90024
giovanni@cs.ucla.edu

Abstract

The growing demand for intelligent information systems requires closer coupling of rule-based reasoning engines, such as CLIPS, with advanced Data Base Management Systems (DBMS). For instance, several commercial DBMS now support the notion of triggers that monitor events and transactions occurring in the database and fire induced actions, which perform a variety of critical functions, including safeguarding the integrity of data, monitoring access, and recording volatile information needed by administrators, analysts and expert systems to perform assorted tasks; examples of these tasks include security enforcement, market studies, knowledge discovery and link analysis. At UCLA, we designed and implemented the Event Pattern Language (EPL) which is capable of detecting and acting upon complex patterns of events which are temporally related to each other. For instance, a plant manager should be notified when a certain pattern of overheating repeats itself over time in a chemical process; likewise, proper notification is required when a suspicious sequence of bank transactions is executed within a certain time limit.
The EPL prototype is built in CLIPS to operate on top of Sybase, a commercial relational DBMS, where actions can be triggered by events such as simple database updates, insertions and deletions. The rule-based syntax of EPL allows the sequences of goals in rules to be interpreted as sequences of temporal events; each goal can correspond to either (i) a simple event, or (ii) a (possibly negated) event/condition predicate, or (iii) a complex event defined as the disjunction and repetition of other events. Various extensions have been added to CLIPS in order to tailor the interface with Sybase and its open client/server architecture.

INTRODUCTION

A growing demand for information systems that support enterprise integration, scientific and multimedia applications has produced a need for more advanced database systems and environments. In particular, active rule-based environments are needed to support operations such as data acquisition, validation, ingestion, distribution, auditing and management —both for raw data and derivative products. The commercial DBMS world has sensed this trend and the more aggressive vendors are moving to provide useful extensions such as *triggers* and *open servers*. These extensions, however, remain limited with respect to functionality, usability and portability; thus, there remains a need for an enterprise to procure a database environment that is (1) more complete and powerful, and

* This work was done under contract with Hughes Aircraft Corporation, Los Angeles, California.

thus supports those facilities not provided by vendors, and (2) is more independent from specific vendor products and their releases and helps the enterprise to manage their IMS and cope with multiple vendors, data heterogeneity and distribution.

A particularly severe drawback of current DBMS is their inability of detecting patterns of events, where an event is any of the possible database operation allowed by the system; typically they are: *insertion*, *deletion* and *updating*. Depending on the application, sequences of events temporally related to each other, might be of interest for the user. In addition to basic database events, management of long transactions and deferred actions may be involved in such patterns. Practical examples of such meaningful patterns of events are:

- Temperature which goes down for three consecutive days;
- 2 day delayed deposit for out-of-state checks;
- 30 days of inactivity on a bank account;
- IBM shares increased value consecutively within the last week;
- Big withdrawal from a certain account followed by a deposit for the same amount on another account.

These and similar situations may require either a certain action to take place (e.g.: buy IBM shares) or a warning message to be issued (e.g.: huge transfer of money is taking place.) EPL gives the user the ability to handle such situations.

The purpose of this paper is to propose a rule-based language and system architecture for data ingestion. The first part of this paper describes the language, then the system architecture is discussed.

EPL DESCRIPTION

An EPL program consists of several named modules; modules can be compiled, and enabled independently. The head of each such module defines an **universe** of basic events of interest, which will be monitored by the EPL module. The basic events being monitored can be of the following three types:

```
insert(Rname), delete(Rname), update(Rname)
```

where *Rname* is the name of a database relation.

A module body consists of one or more rules having the basic form:

```
event-condition-list
→
action-list
```

The left hand side of the rule describes a certain pattern of events. When such a pattern successfully matches with events taking place in the database the set of actions listed in the right hand side are executed.

For instance, assume that we have a database relation describing bank accounts whose schema is: **ACC(Accno,Balance).** We want to monitor the deposits of funds in excess of more than $100,000 into account 00201. In EPL, this can be done as follows:

```
begin AccountMonitor
    monitor update(ACC), delete(ACC), insert(ACC);

    update(ACC(X)),
    X.old_Accno = 00201,
    X.new_Accno = 00201,
    X.old_Balance - X.new_Balance > 100000
    ->
    write( "suspect withdraw at %s", X.evtime)
end AccontMonitor.
```

The lines "begin AccountMonitor" and "end AccountMonitor" delimit the module. Several rules may be defined within a module (also refereed as "monitor".) The second line in the example define the *universe* for the module, in this case any update, delete and insert on the **ACC** table will be monitored by this module. Then the rule definition comes. Basically this rule will be fired by an update on the **ACC** table, The variable *X* denotes the tuple being updated. The EPL system makes available to the programmer both the new and the old attribute values of X, these are respectively refereed by means of prefixes "new_" and "old_". An additional attribute, namely "evtime", is available. This contains the time when the specific event occurred.

In the previous rule, the event-condition list consists of one event and three conditions. The action list contains a single write statements. In general one or more actions are allowed, these actions are printing statements, execution of SQL statements, and operating system calls.

The previous rule can also be written in a form that combines an event and its qualifying conditions, that is:

```
update(ACC(X),
       X.old_Accno = 00201,
       X.new_Accno = 00201,
       X.old_Balance - X.new_balance > 100000)
->
write("suspect withdraw at %s", X.evtime)
```

In this second version, the extent of parentheses stresses the fact that conditions can be viewed as part of the qualification of an event. A basic event followed by some conditions will be called a **qualified event.**

The primitives to monitor sequences of events provided by EPL are significantly more powerful than those provided by current database systems. Thus, monitoring transfers from account 00201 to another account, say 00222, can be expressed in a EPL module as follows:

```
update(ACC(X), X.old_Accno = 00201,
       X.old_Balance - X.new_Balance > 100000)
update(ACC (Y), Y.old_Accno = 00222,
       X.old_Balance - X.new_Balance = Y.new_Balance - Y.old_Balance)
 ->
write("suspect transfer")
```

Thus, the succession of two events, one taking a sum of money larger than $100.000 out of account 00201 and the other depositing the same sum into 00222, triggers the printing of a warning message.

For this two-event rule to fire, the deposit event must *follow immediately* the withdraw event. (Using the concept of *composite events* described later, it will also be possible to specify events that need not immediately follow each other.)

The notion of *immediate following* is defined with respect to the universe of events being monitored in the module. Monitored events are arranged in a temporal sequence (a history). The notion of universe is also needed to define the negation of *b*, written *!b*, where *b* stands for a basic event pattern. An event *e1* satisfies *!b* if *e1* satisfies some basic event that is not *b*.

We now turn to the second kind of event patterns supported by EPL: **clock events**. Each clock event is viewed as occurring immediately following the event with the time-stamp closest to it. But a clock event occurring at the same time as a basic event is considered to follow that basic event. For example, say that our bank make funds available only after two days from the deposit of a check. This might be accomplished as follows:

```
insert( deposit( Y), Y.type = "check"),
clock( Y.evtime + 2D)
 ->
... action to update balance ...
write( "credit %d to account # %d", Y.amount, Y.account).
```

In this rule the "clock" event makes the rule waiting for two days after the check deposit took place.

The EPL system assumes that there is some internal representation of time, and makes available to the user a way to represent constant values expressing time. In particular, any constant number can be followed by one of the following characters: 'S', 'M', 'H', 'D', which stand, respectively, for seconds, minutes, hours and days. In the previous example the constant *2D* stands for two days. A value for time is built as the sum of functions that map days, hours, minutes and seconds to an internal representation. Thus *2D+24H+61M* will map to the same value of time as *3D+1H+1M*. Thus, EPL rules are not dependent on the internal clock representation chosen by the system. Observe that a clock can only take a constant argument —i.e., a constant, or an expression which evaluates to a constant of type time.

Patterns specifying (i) basic events, (ii) qualified events, (iii) the negations of these, and (iv) clock events are called *simple event patterns*. Simple patterns can always be decided being true or false on the basis of a single event. *Composite event patterns* are instead those that can be only satisfied by two or more events. Composite event patterns are

inductively defined as follows. Let $\{p_1, p_2, ..., p_n\}$, n>1 be event patterns (either composite or simple.) Then the following are also composite event patterns:

1. $(p_1, p_2, ..., p_n)$: a sequence consisting of p_1, immediately followed by p_2, ..., immediately followed by p_n.
2. $n{:}p_i$: a sequence of n>0 consecutive p_i's.
3. $*{:}p_i$: a sequence of zero or more consecutive p_i's.
4. $[p_1, p_2, ..., p_n]$: $(p_1, *{:}!p_2, p_2, ..., *{:}!p_n, p_n)$.
5. $\{p_1, p_2, ..., p_n\}$ denotes that at least one of the p_i must be satisfied for $(1 \le i \le n)$.

Using the composite patterns we can model complex patterns of events. For instance, we can be interested to "the first sunny day after at least three consecutive raining days." Assuming that we have a "weather" table which is updated every day (the "type" attribute contains the weather type for the specific day.) Our rule will be:

```
[ (3:insert( weather(X), X.type = 'rain')),
  insert( weather(Y), Y.type = 'sun')]
->
write("hurrah, eventually sun on %d\n", Y.evtime).
```

This rule fires even though between the three raining days and the sunny one there are days whose weather type is different from "rain". In case we are interested to "the first sunny day immediately following three or more raining days," we should rewrite the rule as:

```
(3:insert( weather(X), X.type = 'rain'),
 *:insert( weather(Z), Z.type = 'rain'),
 insert( weather(Y), Y.type = 'sun'))
->
write("hurrah, eventually sun on %d\n", Y.evtime).
```

Note the different use of the *relaxed sequence* operator "[...]" in the first case and the *immediate sequence* one "(...)" in the second rule. By combination of the available composite operators, EPL can express very complex patterns of events.

EPL ARCHITECTURE

EPL's architecture combines CLIPS with a DBMS that supports the *active* rule paradigm. The current prototype runs on Sybase [SYBASE] and Postgres [POST] — but porting to other active RDBMS [K90][MS93] is rather straightforward.

Sybase rule system presents some drawbacks like:

- Only one trigger is allowed on each table for each possible event;
- The trigger fires immediately after the data modification takes place;
- Trigger execution is set oriented, which means the triggers are executed only once regardless the number of tuples involved in the transaction;
- Only SQL statements are allowed as actions.

EPL tries to both overcome to such limitations and allow the user to model patterns of meaningful events.

Event Monitoring Mechanism

For each event which needs to be monitored by an EPL rule a trigger and a table (also referred as Event monitor Relation, ER) have to be created on Sybase. The trigger fires on the event which can be either an insert, delete or, update. This trigger will copy the modified tuple(s) into the corresponding ER.

As an example, say that we want to monitor the insertion on the **ACC** relation previously defined. As soon as the EPL monitor is loaded into the system the fact *(universe acc_mon i acc)*, will be asserted into the CLIPS working memory. This new fact triggers a CLIPS rule which creates a new Sybase relation called "ERacc_ins" having the following schema: *(int accno, int balance, int evtime)*. Moreover, a Sybase trigger is created by sending the following command from CLIPS to Sybase:

```
create trigger ETacc_ins on acc for insert as
begin
  declare @d int
  select @d = datediff( second, '16:00:00:000 12/31/1969', getdate())
  insert ERacc_ins select distinct *, @d from inserted
end
```

The event time is computed as the number of seconds since the 4pm of 12/31/1969. This is a standard way to represent time on UNIX systems. Any ER name starts with the prefix "ER" followed by both the monitored table name and the type of event. The correspondent trigger has "ET" as prefix.

As later explained, the module **EPL-Querier** performs the communication between CLIPS and Sybase.

EPL rules as finite-state machines

As previously discussed, any EPL rule is modeled by a finite state automaton which is implemented by a set of CLIPS rules. Transitions between automaton states take place when the following conditions occur:

- a new incoming event satisfies the current pattern;
- a pending clock request reaches its time;
- a predicate is satisfied.

On a transition the automaton can take one of the following actions:

- Move to the next **Acceptance** state where it will wait for the next event;
- Move to a **Fail** state. Here, the automaton instantiation, with relative information, is removed from the memory;
- Move to a **Success** state. Here, the actions specified in the right hand side of the rule are executed.

Each EPL rule is transformed to a set of CLIPS rules which implement its finite state machine. Several instantiations of the same EPL rule can be active at the same time.

Architecture overview

EPL is basically built on top of CLIPS in two ways: 1) Some new functions, implemented in C, have been added to CLIPS in order to support EPL; 2) CLIPS programs have been written to implement the EPL rule execution system. Figure 1 depicts the entire EPL system.

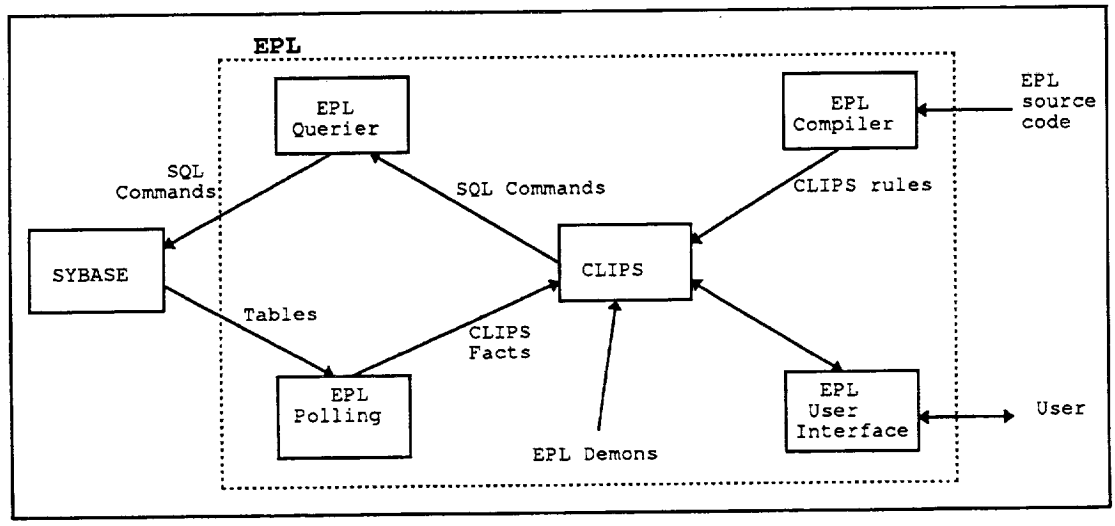

Figure 1. EPL Architecture

EPL-Compiler is a rule translator, it takes an EPL program as input and produces a set of CLIPS rules which implement the EPL program. **EPL-Polling**, at regular intervals, transfers the ERs from Sybase to the CLIPS working memory. This process requires some data type conversion in order to accommodate Sybase tuples as CLIPS facts. The **EPL-Querier** sends SQL commands to Sybase server when either (1) Sybase triggers or ERs have to be created (or removed) or when (2) an SQL command is invoked on the action side of an EPL rule. The **EPL-User-Interface** accepts commands from the user and produces output to either the screen or a file. At low level, EPL commands are executed by asserting a fact in the CLIPS working memory. Such an assertion triggers a rule which executes the desired command. This loose coupling allows an easy design of the user interface whose only task is to insert a new fact depending on the user action.

EPL-Demons is a CLIPS program which implements the EPL rule execution system. Basically this set of CLIPS rules monitors the entire EPL execution. The EPL demons, together with the CLIPS rules produced by the EPL-compiler, form the entire CLIPS program under normal execution time. The CLIPS facts on which these rules work are those periodically produced by the EPL-Polling, and those asserted by the EPL user interface as a consequence of user actions.

Conclusions

This document has described the design and the architecture of EPL, a system which provides sophisticate event pattern monitoring and triggering facilities on top of commercial active databases, such as SYBASE. EPL implementation is based on CLIPS and the design of an interface between SYBASE and CLIPS represented one of the most critical tasks in building EPL. EPL rules are translated into a set of CLIPS rules which implement the finite state machine needed to execute such EPL rules.

This paper provided an overview of the language definition and a brief description of the system implementation neglecting various implementation details for lack of space.

Future work is required to provide language extensions and interfacing with other active database systems.

Acknowledgments. This report builds upon a previous one authored by S. Lau, R. Muntz and C. Zaniolo. The authors would also like to thank Roger Barker for several discussions on EPL.

REFERENCES

[ISO] ISO-ANSI Working Draft of SQL-3.

[COLE] R. Coleman. "Pulling the Right Strings", Database Programming and Design, Sept. 1992, pp. 42-49.

[SYBASE] Sybase Inc. Sybase's Reference Manual.

[Fo82] C.L.Forgy. "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem" on Artificial Intelligence 19, 1982.

[CLIPS] "CLIPS User's Guide", Artificial Intelligence Section.

[K90] G.Koch, "Oracle: the Complete Reference," Berkeley, Osborne, McGraw-Hill, 1990.

[LMM86] J.L Lassez, M.J. Maher, K. Marriot. "Unification Revisited", Lecture Notes in Computer Science vol.306, Springer-Verlag, 1986.

[MS93] J.Melton, A.R.Simon. "Understanding the new SQL: A Complete Guide," San Mateo, California, Morgan Kaufmann Publishers, San Francisco, California, 1993.

[NT89] S.Naqvi, S.Tsur "A Logical Language for Data and Knowledge Bases," Computer Science Press, New York, 1989.

[POST] J. Rhein, G. Kemnitz, POSTGRES User Group, The POSTGRES User Manual, University of California, Berkeley.

[St87] M. Stonebraker, The POSTGRES Storage System, Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[St92] M. Stonebraker, The integration of rule systems and database systems IEEE Transactions on Knowledge and Data Engineering, October 1992.