Temporal Aggregation in Active Database Rules*

Iakovos Motakis† Carlo Zaniolo
Computer Science Department, University of California
Los Angeles, California 90024
motakis@cs.ucla.edu zaniolo@cs.ucla.edu

Abstract

An important feature of many advanced active database prototypes is support for rules triggered by complex patterns of events. Their composite event languages provide powerful primitives for event-based temporal reasoning. In fact, with one important exception, their expressive power matches and surpasses that of sophisticated languages offered by Time Series Management Systems (TSMS), which have been extensively used for temporal data analysis and knowledge discovery. This exception pertains to temporal aggregation, for which, current active database systems offer only minimal support, if any.

In this paper, we introduce the language TREPL, which addresses this problem. The TREPL prototype, under development at UCLA, offers primitives for temporal aggregation that exceed the capabilities of state-of-the-art composite event languages, and are comparable to those of TSMS languages. TREPL also demonstrates a rigorous and general approach to the definition of composite event language semantics. The meaning of a TREPL rule is formally defined by mapping it into a set of $Datalog_{1S}$ rules, whose logic-based semantics characterizes the behavior of the original rule. This approach handles naturally temporal aggregates, including user-defined ones, and is also applicable to other composite event languages, such as ODE, Snoop and SAMOS.

1 Introduction

Recent research in active databases aims at extending the power of active rules to trigger on complex patterns of temporal events. In many applications, in fact, the simple-event detection mechanisms found in commercial systems, such as Ingres, Oracle or Sybase, are not sufficient; complex sequences of events must instead be detected as the natural precondition for taking actions and firing rules [8]. Therefore, several research prototypes now provide this capability; an incomplete list of such systems includes [8, 15, 13, 3, 16].

The *composite event specification languages* supported by these systems suffer from serious limitations in terms of expressive power, since they provide little or no support for temporal aggregation. However, temporal aggregation is very important as demonstrated by its extensive use in time-series management systems (TSMS). Obviously, most patterns of temporally aggregated events that are worthy of off-line detection by TSMS applications, are also worthy of on-line detection by active rules. A powerful temporal aggregation construct can be found in [28], which supports condition-action rules only. Clearly, comparable functionality needs to be provided by languages that support composite event expressions.

The first objective of this paper is to propose a composite event language with temporal aggregation capabilities similar to those offered by database languages for time-series analysis. We present the design of TREPL, a language that allows the specification of complex event patterns and temporal aggregates at a level comparable to that of time-series languages— while preserving the amenability to efficient implementation in an active database framework.

Then, we also present a formal logic-based semantics for TREPL using $Datalog_{1S}$ [2], which is very well-suited to the task, since it can describe in one framework both the temporal and deductive aspects of active rules. A similar approach to the semantics and temporal aggregation is also applicable to other event-based languages for active databases, such as [15, 13, 3], as well as to time-series languages. Thus, the paper also seeks and achieves a conceptual reapproachment of languages in these two fields, that reduces some of the complexities associated with the assortment of different techniques used in their original specifications.

2 Temporal Aggregation & Composite Event Languages

In this section, we discuss the technical framework for temporal aggregation in active databases. First, we present examples of the most important classes of temporal aggregation queries. Then, we discuss some important differences in the computational methods used in time-series analysis and active databases.

For the rest of the paper, we assume an application that monitors stock price changes (*price ticks*), as well as daily *closing prices* for a fixed set of stocks. It also monitors the stock trading transactions for a set of customers. Thus, we have the following database relations:

- `stock_tick(ID, Price)` contains the current prices for

*This work was supported in part by the National Science Foundation under grant IRI-96-32272

†Currently with Cambridge Technology Partners.

SIGMOD '97 AZ,USA

the stocks of interest. A stock price update may occur at any time during a working day.

- `stock_closing(ID, Price)` contains the most recent closing prices for all stocks of interest.
- `owns(Name, ID, Shares)` contains the stock portfolios of the customers. *ID* denotes the stock identifier.

  We assume that closing price updates are obtained in batch form at the end of each day.

2.1 Temporal Aggregation Query Classes

The following query classes were identified by studying several TSMS [25, 11, 18, 4, 10], which routinely employ a variety of aggregation functions for time-series analysis.

Counting Aggregates

Typical queries require counting the number of occurrences of an event pattern within a time interval; other queries recognize and take some action upon the *n-th*, *first n*, or *every n-th* event. Examples follow:

- Report the closing price of a stock for the first 30 days after it raises by more than 20%.
- Report the closing price of 'IBM', every 7 days after it exceeds $100, as long as it remains above $100.

Accumulation and Running Aggregates.

When an accumulation function is applied on an event sequence, an aggregate value is generated for each point in the sequence. This aggregate value depends on all events in the sequence so far. Thus, a sequence of aggregate values is produced. Such temporal aggregates are often called running aggregates.

Examples:

- Compute the sequence of running averages for 'IBM' price ticks.
- Starting now, report every time the closing price of a stock achieves a *new maximum*.

Moving-Window Aggregates

Moving-window functions are frequently used in time-series analysis [25, 4, 27]. The *moving-window average* is the most commonly used among these functions.

As with accumulation functions, moving-window functions also generate an aggregate value at each point in the event sequence they are applied on. The difference from the previous category is that each aggregate value depends on only the *last n* events in the sequence, where *n* is the *size of the moving window*. An example follows:

- For each stock, from the sequence of its price ticks, derive the sequence of its price-tick moving-averages for a window size of *5 price updates*.

A variation of moving-window aggregation is when the size of the moving-window is based on time duration, as in the following query:

- *Time-based Window:* At each 'IBM' stock price update, report its average price in the last two hours.

Temporal Grouping

Time-series analysis often involves transformations to different time granularities. For instance, aggregating the price updates of a stock *by Month* generates a sequence of monthly aggregate values for that stock. In this kind of aggregation, the time line is partitioned into consecutive time intervals, and input events are grouped by and aggregated over these intervals.

The time line partition is often defined by a *calendar* [30, 6]. This allows aggregation by *Day, Month, Year*, etc.

Quantified Temporal Aggregates

Even though this kind of temporal aggregation is not discussed in the model proposals and systems we surveyed, we believe that it is very important for *temporal decision-support* applications. Quantified temporal aggregates is an adaptation of *quantified aggregates* proposed in [17] and in [24]. They typically involve phrases, such as *all, most frequently, more than five, at least twice as many, not all*, etc.. Examples follow:

- For every stock, derive its phases of *monotonically increasing* closing prices, that lasted at least 30 days.
- For the 'IBM' stock, report its price every time when there have been *at least twice as many* increases of its price so far, as decreases.

2.2 Composite Event Languages

Active database languages such as ODE [15], Snoop [3], SAMOS [13] and EPL [23] support the specification of rules that are fired when complex patterns of events are detected. For this purpose, they provide constructs to express:

Qualified Basic Events, i.e., basic events such as database relation modifications in relational systems, or method invocation commands in object-oriented systems, often qualified by conditions that the attributes of the affected tuples or objects must satisfy.

Composite Events. These are specified by combining qualified basic events, using the language constructs. Composite event patterns that can be expressed in one or more of these languages include: (a) Finite sequences of events in an assigned temporal relationship, such as $E2$ *immediately following* $E1$ or $E2$ *eventually following* $E1$, (b) Disjunction of events, (c) Conjunction of simultaneously occurring events, and (d) (Possibly infinite) recursive patterns of events.

2.3 From Time Series to Composite Events

The temporal languages used to query time-series are employed in a computational context that is different, and normally simpler, than that of active databases. Some of the main differences are:

- *On-line versus off-line reasoning.*

  Time-series languages normally assume that time-series are stored, and thus, several passes can be performed over them. For instance, the event values in a time interval of interest can be extracted during the first pass, and aggregates can be computed during one or more additional passes. Active databases instead, reason upon the incoming events as they occur and their

response must be immediate. Thus, both the event pattern detection and the generation of temporal aggregate values must be done *on-the-fly*.

The advantages of *on-line temporal reasoning* are:

1. The response of the system is quicker.
2. Unless needed in the future, the incoming event sequences do not need to be stored.

In the sequel, and for clarity reasons, we will use the terms *time-series* and *event sequence*, when we refer to TSMS and active databases, respectively.

- *Time-Series for Objects versus Event Sequences for Classes*

  TSMS often assume that time-series are arranged by objects and their surrogates, rather than by content-defined classes of events. For instance, in stock analysis applications, stock price updates are typically arranged by their identifiers in separate temporal sequences; one sequence for each stock identifier. In an active database environment, however, all price updates for all monitored stocks would typically form a single input event sequence.

- *Irregular versus Regular* events.

  In an *irregular* time-series, the time-points do not obey any particular structure. For instance, the time-series for 'IBM' price ticks is irregular, since a price tick can occur at any point in time during a working day.

  On the other hand, the time-series for the *'IBM' closing prices* contains a value (price) for each time point in the associated *calendar Working Day*. This calendar represents the sequence of working days and defines its periodicity, and its exceptions (holidays). For more information on the definition and representation of calendars, the reader is referred to [30, 6].

  For both regular and irregular series, multiple *simultaneous* events must also be handled. For instance, daily updates of stock closing prices are updated with the same *timestamp*. This can be either a *valid timestamp*, this working day, or a *transaction-time timestamp*, containing the commit time of the transaction that performed all these updates.

  Most methods for analyzing time-series can be applied to regular ones only [25]. Active databases can handle equally well both kinds of event sequences.

¿From the above analysis, it is clear that for time-series manipulation, active databases are often advantageous over traditional TSMS. These advantages are related to storage space, language complexity, system maintenance, and processing time.

In fact, it seems that the only limitation for using active databases for this task is the rate of incoming events, which must be low enough for the rule manager to handle without causing significant performance degradation in the DBMS.

3 The TREPL Language

We now present the design of TREPL (Temporal Reasoning Event Pattern Language), which provides powerful temporal aggregation capabilities. TREPL is currently being implemented as an extension to EPL [16]; EPL is a composite event language previously developed at UCLA, as a portable front-end to active relational databases, such as commercial systems that only support only simple-event detection.

3.1 TREPL Programs

A TREPL program is organized in *modules*, which can be compiled and enabled independently. The *events declaration section* of a module defines the set of relevant basic event types monitored by the module. This is the *universe of events*, under which the module's rules are evaluated. A *basic event type* is either:

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

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

TREPL rules are specified in a module's *rules section*. Each rule has a name, which is unique within its module. A rule's body (head) corresponds to an event (action). Example 1 demonstrates an EPL rule within a module.

Example 1 *A TREPL rule that keeps track of big stock buys of customer "Jones".*

```
Begin Module OwnsMod
monitor update(owns);
BigBuy:
  update(owns(X), X.Name = 'Jones',
         X.new_Shares - X.old_Shares > 1000 )
  -> write("Big Buy of %s from Jones at time %s\n",
           X.ID, asctime( X.Timestamp) ).
End Module.
```

Rule *BigBuy* specifies a *qualified basic event*. Such an event has the form:

`evtkind( Rname(X),` < `condition-expression` > `)`,

where `X` is a variable, denoting the tuple of relation `Rname`, that has been inserted, deleted or updated. For *update* events, TREPL makes available to the programmer both the new and the old contents of the updated tuple. The prefixes `new` and `old` are used to distinguish between them. When no prefix is specified, `new` is assumed.

The < condition-expression > is built using the standard arithmetic and comparison operators and the logical connectives AND (*or comma*), OR and NOT.

Timestamps: Each event is attached a *transaction-time Timestamp*, which contains the date and time the event was recorded in the database. The format for timestamps adheres to the SQL-92 standard, thus, an example timestamp value is '1996-08-21 09:13:27'. We also assume the *gregorian calendar*, and we support the SQL-92 time literals *Date, Time, Year, Month, Day, Hour, Minute, Second*. These literals can be used instead of *Timestamp* in an expression. For instance, if *X.Timestamp = '1996-08-21 09:13:27'*, then *X.Time = '09:13:27'*, *X.Day = '21'* and *X.Month + 1 = '09'*.

In some cases, in addition to the transaction-time timestamp, a *valid-time timestamp* might also be part of the modified tuples. This attribute stores the logical time of the corresponding events, and its granularity depends on the application. For the *stock_closing* relation for instance, the assumed time granularity is *Day*.

Actions and Derived Events: Two kinds of *explicit* actions are supported by TREPL: (a) *SQL commands*, and (b) *Operating system calls*, which allow TREPL to communicate with the user, or other applications that run concurrently. An example of the second kind is the UNIX *write* system call. *Multiple actions* may be specified for a TREPL rule.

An action may also generate a *derived* event. Such an event does not result in any change of the database state, but it can be used in the event part of TREPL rules, similarly to basic events. An example of a derived event is shown in Example 2. Observe that in the *monitor* statement of this example, the event update(stock_tick) has been given the alias st.

Example 2 *Isolate 'IBM' stock ticks.*

```
Begin Module MyModule
monitor update(stock_tick) as st,
        ins(owns), upd(owns);
derived ibm_tick(ID, Price);

IBM_StockTicks:
    st(X, X.ID = 'IBM') )
    -> ibm_tick(X.ID, X.Price);
End Module.
```

This rule can be used for isolating 'IBM' stock ticks from other events monitored by the module, so that they can be processed separately by other TREPL rules. ibm_tick is a *derived event*, and it is declared in the events declaration section of a module. It contains attributes, which become bound every time the event occurs.

Derived events facilitate the interaction of TREPL rules, which makes it possible to express very complex event patterns. Examples are shown later.

Simple Events: A simple event is either a basic event, or a qualified basic event.

In the rest of this section, we first introduce the composite event constructs of TREPL (same as in EPL), and then we describe its new features.

3.2 Composite Event Constructs

The TREPL language allows the specification of *composite events*, that describe patterns of *simple events*. We distinguish between *event expressions* (also called *event types*) and *event instances*. An event expression E defines an event pattern of interest, whereas an instance of E consists of a set of simple event occurrences that satisfies this pattern. In the sequel, we will simply refer to *events*, when the distinction is clear from the context.

We will also use the term *event occurrence* to refer to the time instant when an event expression is *satisfied*, i.e., an instance of this event expression is completed. Finally, we will refer to the *universe of events*, with the understanding that this is with respect to a particular module, where the event expression appears.

Composite event expressions are defined as follows:

Definition 1 *Let $E_1, E_2, ..., E_n$, $n > 1$, be TREPL event expressions (maybe composite themselves). The following are also TREPL event expressions :*

1. *$(E_1, E_2, ..., E_n)$: a sequence consisting of an instance of E_1,* immediately *followed by an instance of E_2, ...,* immediately *followed by an instance of E_n.*
2. *$(E_1$ & E_2 & ... & $E_n)$: A conjunction of events. It occurs when all events $E_1, \ldots, E_n$ occur simultaneously.*
3. *$\{E_1, E_2, \ldots, E_n\}$: A disjunction of events. It occurs when at least one event among $E_1, \ldots, E_n$ occurs.*
4. *$!E$: It occurs when any basic event in the universe occurs, but E is not satisfied.*
5. *$*:E$: a sequence of one or more consecutive instances of E.*

A number of additional (derived) constructs may be defined in terms of the basic ones (see also [14]). Some of these are :

- *any* $\equiv$ *The disjunction of all basic events in the universe. It occurs every time such an event occurs.*
- *$[E_1, E_2, ..., E_n] \equiv (E_1, *:any, E_2, *:any, \ldots, *:any, E_n)$. Relaxed sequence. It consists of an instance of E_1, followed* later *by an instance of $E_2, \ldots,$ followed* later *by an instance of E_n.*
- *$prior(E_1, E_2) = [E_1, any]$ & E_2. An occurrence of E_2 follows an occurrence of E_1, i.e., an instance of E_1 is completed prior to the completion of an instance of E_2.*
- *$first(E) \equiv (E$ & $![E, any])$: It occurs when the first instance of E occurs.*

The difference between $[E_1, E_2]$ and $prior(E_1, E_2)$ is that in the first case, the first basic event in an instance of E_2 must follow the occurrence (i.e., completion of an instance) of E_1, whereas in the second case, it is only required that an occurrence of E_2 must follow an occurrence of E_1.

In addition to the above, a composite event may have attributes, which are derived from the attributes of its component basic events. Attribute semantics and scope rules are described in section 5.5.

Example 3 Relaxed Sequence: *Identify all customers who bought a stock back at a greater quantity than they previously sold the same stock.*

```
monitor update(owns)
BoughtBack:
  [ update(owns(X), X.new_Shares < X.old_Shares),
    update(owns(Y), Y.Name = X.Name, Y.ID = X.ID,
           Y.new_Shares - Y.old_Shares >
           X.old_Shares - X.new_Shares)
  ]
  -> write("Customer %s back on %s\n",Y.Name,Y.ID)
```

For a detailed discussion of the TREPL language and its comparison to other composite event languages, the reader is referred to [21].

3.3 Implicit Recursion and Temporal Aggregates

The star construct '*', that was used in EPL to express implicit recursion has been extended in TREPL to express temporal aggregation. Although TREPL also supports explicit recursion via derived events, implicit recursion is preferable in most situations, because of its simplicity and amenability to efficient implementation. For instance, consider the following example:

Example 4 *If the closing price of a stock raises by more than 20%, to a price, say P, report its price on every day following that raise, as long as it remains above P.*

```
monitor update(stock_closing) as st_cl
StarRule:
  ( st_cl(X, X.new_Price > 1.2 * X.old_Price),
    * st_cl(Y, Y.ID = X.ID, Y.Price >= X.new_Price)
  )
  -> write(Y.ID, Y.Price)
```

The first event occurs when a stock's closing price raises by more than 20% to a price P. Then, the star event occurs on *every following working day*, as long as the stock's closing price remains above P. The star sequence is completed when at least one of the associated conditions fails, i.e., either there is no closing price for that stock in the next working day's batch, or its price goes down below P.

The use of tuple variable `Y` inside the star expression refers to the most recent event (last closing price) in the sequence. Also, observe that the condition `Y.ID = X.ID` in the star expression achieves the desired effect that for each stock, the sequence of its closing price updates is processed separately.

The unfolding of implicit recursion to explicit one elucidates the semantics of the star. For instance, the previous rule can be expanded as follows:

Example 5 *Expanding the implicit recursion in example 4 into recursive rules.*

```
monitor update(stock_closing) as st_cl;
derived star(FirstPrice, ID, Price);

StarRule1:
  ( st_cl(X, X.new_Price > 1.2 * X.old_Price),
    st_cl(Y, Y.ID = X.ID, Y.Price >= X.new_Price)
  )
  -> star(X.new_Price, Y.ID, Y.Price);

StarRule2:
  ( star(X),
    st_cl(Y, Y.ID = X.ID, Y.Price >= X.FirstPrice)
  )
  -> star(X.FirstPrice, Y.ID, Y.Price);

HeadRule:
  star(Z)
  -> write(Z.ID, Z.Price);
```

`StarRule1` is the beginning rule, which produces the first occurrence of the derived event `star`. Then, `StarRule2` produces recurring occurrences of the `star` event, as long as the query's conditions hold. Every time `star` occurs, rule `HeadRule` writes the new closing price out.

The syntax and semantics of temporal aggregates in TREPL follow naturally from viewing the star sequence construct as defining implicit recursion. Say for instance, that in the last example, we also want to compute the *running count*, which tells how many (working) days have passed since the 20% raise. Then, in the TREPL rule in example 4, we can use the *built-in aggregate* `Count` and modify its head as follows:

```
-> write(Y.ID, Y.Price, Count(Y.ID)).
```

`Count(Y.ID)` denotes an aggregation over the events in the star sequence. At its point in this sequence, its value depends on the events *so far*.

The intuitive semantics can again be obtained by unfolding the implicit recursion into an explicit one. In the rules of example 5, we need to add an attribute `Count` to the `star` event, as follows:

```
monitor update(stock_closing) as st_cl;
derived star(FirstPrice, ID, Price, Count);

StarRule1:
  ( st_cl(X, X.new_Price > 1.2 * X.old_Price),
    st_cl(Y, Y.ID = X.ID, Y.Price >= X.new_Price)
  )
  -> star(X.new_Price, Y.ID, Y.Price, 1);

StarRule2:
  ( star(X),
    st_cl(Y, Y.ID = X.ID, Y.Price > X.FirstPrice)
  )
  -> star(X.FirstPrice, Y.ID, Y.Price, X.Count+1);

HeadRule:
  star(Z)
  -> write(Z.ID, Z.Price, Z.Count).
```

Various other aggregates, including the standard SQL ones, *Sum*, *Avg*, *Min*, *Max*, etc., can be used in a similar way, as built-in aggregates.

4 Temporal Aggregation in TREPL

Having described the basic mechanism for temporal aggregation, in this section, we demonstrate our framework and the temporal aggregation capabilities of TREPL in more detail. In order to illustrate the generality of our approach, we present one example for each class of queries discussed in section 2. Also, we give an example of a specialized aggregate, defined via the use of derived events.

In general, *the reference to a temporal aggregate* AggrFunc(X.A) *implies that the built-in aggregation function* AggrFunc *is evaluated over the sequence specified by a star sequence goal with tuple variable* X. At each point in the sequence, a new value for the temporal aggregate is obtained, which depends on the A values in the *sequence so far* .

This kind of *syntactic encapsulation* has been successfully used before in logic-based languages supporting aggregates. In LDL++ [1] e.g, aggregates can be expressed in the head of a rule. This rule is then compiled by the system into a recursive clique that computes the aggregates.

In TREPL, a temporal aggregate evaluated over a star sequence, can be used in one or more of the following places:

- In the head of the rule.
- In the condition of a goal following the goal of this star sequence.
- In the condition of this star sequence goal.

An example of the first case was described in the last section. The following two examples belong to the second and third cases, respectively.

Example 6 (Running Aggregate) *If the closing price of 'IBM' raises by more than 20%, to a price, say P, report every new maximum in the following working days, provided that its closing price has remained above P.*

```
monitor update(stock_closing) as st_cl
New_Max_So_Far:
( st_cl(X, X.ID = 'IBM',
        X.new_Price > 1.2 * X.old_Price),
  * st_cl(Y, Y.ID = 'IBM', Y.Price >= X.new_Price),
  Y.Price = Max(Y.Price)
)
-> write(Y.ID, Y.Price).
```

The essential difference between this query and the query of Example 4, is that within a recognized star sequence, we want the rule to fire only when the condition *maximum so far within this sequence* is satisfied. This condition is expressed in the last goal of the rule, which is a *condition-only goal.*

It is important to realize that the star sequence for 'IBM' closing prices continues independently of the condition `Y.Price = Max(Y.Price)`, as long as new closing prices remain above the initial value P. Then, at each point in the sequence, this condition is evaluated, and if satisfied (new maximum in the sequence so far), the rule fires and the new maximum is written out.

Example 7 (Quantified Aggregate) *For every stock, derive its phases of monotonically increasing closing prices that lasted at least 10 days.*

```
monitor update(stock_closing) as st_cl
Good_Phase:
  ( * st_cl(X, X.ID = First(X.ID),
            X.Price > Last(X.Price)),
    st_cl(Y, Y.ID = X.ID, Y.Price < X.Price,
          Count(X.ID) >= 30 )
  )
  -> write("Good Phase for %s, from %s to %s\n",
            X.ID, First(X.Day), X.Day).
```

The `First` and `Last` aggregates refer to the first and the last (previous) event in a star sequence instance.

Observe the use of these temporal aggregates inside the goal of the star sequence over which they are evaluated. The condition `X.ID = First(X.ID)` achieves the result that each stock is considered separately, an effect equivalent to that obtained by using *group by X.ID* in SQL.
The condition `X.Price > Last(X.Price)` ensures that the star sequence continues, as long as we are in a phase of monotonically increasing prices. When such a phase ends, the second event occurs, and if `Count(X.ID) >= 30`, the rule fires.

It is also important to realize that in order for the above rule to work correctly, every recognized star sequence instance must be a *maximal* sequence of monotonically increasing closing prices. No subsequences of such a maximal sequence will be recognized separately. This *maximality property* is obtained by the semantics definition for the star sequence construct, as explained in section 5.4.

Finally, we present examples of the other two important query classes, i.e., *moving-window* aggregates and *temporal grouping,*

Example 8 (Moving Average) *¿From the sequence of IBM closing prices, derive the sequence of its moving averages, for a window-size of* 5 working days.

```
Moving_Avg:
  ( any,
    * stock_closing(X, X.ID ='IBM', Count(X) <= 5),
    Count(X) = 5
  )
  -> write(X.Name, X.Day, Avg(X.Price)).
```

The use of *any* as the first event ensures that at *every working day,* the computation of a new moving-average begins. This computation takes place over the sequence of the *next five* 'IBM' closing prices. The new moving-average is generated at the end of the fifth day, as per the condition *Count(X) = 5.*

The case of moving-window aggregation, where the size of the window is based on time duration can be handled similarly, by using explicit time conditions.

Example 9 (Temporal Grouping) *For each month, report the monthly price averages for all stocks.*

```
monitor stock_tick
MonthlyAvg:
   ( stock_tick(X),
     * stock_tick(Y, Y.ID = X.ID,
                     Y.Month = X.Month +1),
     stock_tick(Z, Z.ID = X.ID,
                   Z.Month = Y.Month +1)
     )
   -> write("Average Price for %s in %s/%s = %d\n",
            X.ID, Y.Month, Y.Year, Avg(Y.Price) ).
```

For a particular stock, because of the conditions *Y.Month = X.Month + 1* and *Z.Month = Y.Month + 1*, a different star sequence instance is active for each month. This star sequence instance consists of all events (price ticks) in this month. This achieves the desired effect of aggregating *by Month.*

4.1 Special-Purpose Aggregates

While common built-in aggregates can be easily expressed within star expressions, the user may need to define his/her own temporal aggregates. In fact, the capability to define customized aggregation functions is considered essential for TSMS [11, 18]. By using derived events, specialized temporal aggregates can also be specified in TREPL.

Consider for instance the *quantified* query:

Example 10 *For the 'IBM' stock, report its price every time when there have been at least twice as many increases of its price so far, as decreases.*

This query is hard to express using a single TREPL rule. Instead, we can break it down to multiple rules, as it is shown below. Module `IBMTicks` monitors the `ibm_tick` events ('IBM' price updates), derived as shown in example 2.

```
Begin Module IBMTicks
monitor ibm_tick;
derived chg_count(IC,DC);

InitRule:
    first(ibm_tick)
    -> chg_count(0, 0);

Increase:
    ( chg_count(X),
      ibm_tick(Y, Y.new_Price > Y.old_Price )
    -> chg_count(X.IC+1, X.DC);

Decrease:
    ( chg_count(X),
      ibm_tick(Y, Y.new_Price < Y.old_Price )
    -> chg_count(X.IC, X.DC+1);

Report:
    chg_count(X, X.IC >= 2 * X.DC )
    -> write("Twice as many Incr. as Decr. for IBM",
             X.IC, X.DC, X.Timestamp).
End Module.
```

The sequence of ibm_tick derived events forms an event history, upon which this module's rules are evaluated. A derived event chg_count(IC,DC) is generated every time an ibm_tick occurs, its attribute values IC, DC denoting respectively, the number of increase so far, and the number of decreases so far.

An initial event chg_count(0,0) occurs at the first ibm_tick. After that, every time ibm_tick occurs, either the Increase rule or the Decrease rule is triggered, and a chg_count event with updated IC and DC values is generated. If the condition "twice as many increases as decreases" is satisfied, rule *Report* is triggered and a message is written out.

An alternative way to implement the query of Example 10 would be to use a *temporary relation* chg_count, containing one tuple with the values for attributes IC and DC. Then every time an ibm_tick event occurs, this tuple must be retrieved and updated accordingly. Because of space limitations, we do not discuss this approach any further.

5 Semantics of TREPL

In this section we present the semantics of TREPL, employing the same method we used for its predecessor EPL [23, 22]. First, we introduce the basic concepts, then we focus on *derived events* and *aggregation* within star sequences. For more details and examples about the formalization of event histories and the semantics of the other TREPL constructs, the reader is referred to [22].

5.1 Event Histories

In order to define the semantics of TREPL expressions, we need to introduce the notion of *event histories*, against which the TREPL expressions are evaluated. Our starting point is the *event tables* that log the occurrences of the various basic events. According to the practice in most commercial databases, each *event table* corresponds to a particular *basic event type*, and accumulates the *time-stamped* occurrences of this event type. For inserts into the *owns* table for instance, we have the table

```
ins_owns(Name, ID, Shares, Timestamp).
```

The *del_owns* table has a similar format, while for updates of *owns*, we must record both the old and new values:

```
upd_owns(Name_old, Name_new, ID_old, ID_new, ...
                    Timestamp).
```

Furthermore, we assume a table evt_monit(ModuleName, EventType) , which records the basic event types each module monitors.

Using these system tables, an *event history* can be obtained for each module of interest. An event history contains the occurrences of monitored event types, ordered by their timestamps.

Example 1 shows a short event history for module MyModule in Example 2. The derived events ibm_tick are ignored for now, but they are discussed in section 5.3.

Observe that a sequence number called *stage* has been introduced. The stage sequence defines an ordered structure on the distinct timestamps, that allows us to express properties of composite events that are based on the *relative order* of occurrence of their component basic events, as opposed to absolute time properties. Absolute time properties of events can also be expressed using their timestamps.

Example 11 *An brief example event history for module* MyModule *in Example 2. For the sake of clarity, the stock names for the price ticks are shown in parentheses.*

hist_MyModule		
EventType	*Timestamp*	*Stage*
—	0000	0
upd_stock_tick(TRD)	1423	1
upd_stock_tick(IBM)	1425	2
upd_owns	1430	3
upd_stock_tick(WER)	1502	4
upd_stock_tick(IBM)	1510	5
ins_owns	1536	6
upd_stock_tick(IBM)	1538	7
upd_stock_tick(TYG)	1549	8

Table 1: A sample event history for module MyModule.

Different *event occurrence granularities* can be handled. At the "smallest database operation granularity", every new insertion, deletion, or update creates a new stage. However, if "transaction boundaries granularity" is assumed, then each committed transaction creates a new stage, and all the basic events that occurred within this transaction are recorded in its stage, timestamped with the transaction's commit time. Basic events that share a stage number are considered to be *simultaneous*.

These observations lead naturally to the use of $Datalog_{1S}$ as a means for defining the semantics of EPL rules. $Datalog_{1S}$ [2] is a temporal language that extends *Datalog*, by allowing every predicate to have at most one temporal parameter (constructed using the unary successor function s), in addition to the usual data parameters. The temporal argument in our case is the *stage* parameter.[1] As discussed in [2], $Datalog_{1S}$ has interesting temporal properties, such as periodicity, that can be useful in the analysis of active rule behavior [33]. These properties, however, are not the primary interest of this paper. In fact, those properties no longer hold if the original TREPL rule contains arithmetic or aggregates; then the *Datalog* program produced by our syntax-directed translation contains other arithmetic functions, besides the successor function. The primary goal of this paper is a definition of the semantics of TREPL; toward this goal, the query-oriented formal semantics of $Datalog_{1S}$—proper, or extended with arithmetic, much in the same way as Datalog is—provides a very natural framework.

5.2 Event Satisfaction

Each TREPL event expression is evaluated with respect to the event history of its module. In this section we show how an incremental evaluation scheme can be described using $Datalog_{1S}$.

First, for each *basic event type* monitored by a module, we define a predicate that describes the history of the event's occurrences within this module. For each occurrence of that event, this predicate contains a tuple with its attribute bindings and the stage of the occurrence. For instance, for the basic event type ins_owns in module MyModule, we get:

```
ins_owns_MyModule(Name, ID, Shares, Time, J) ←
                    hist_MyModule(ins_owns, Time, J),
                    ins_owns(Name, ID, Shares, Time).
```

[1] Note that the *Timestamp parameter* is supplied by the database system, and thus, it is treated as a data parameter.

hist_MyModule			hist_IBMTicks		
EventType	*Timestamp*	*Stage*	*EventType*	*Timestamp*	*Stage*
—	0000	0	—	0000	0
upd_stock_tick(TRD)	1423	1			
upd_stock_tick(IBM)	1425	2			
ibm_tick	1425	2	ibm_tick	1425	1
			chg_count	1425	1
upd_owns	1430	3			
upd_stock_tick(WER)	1502	4			
upd_stock_tick(IBM)	1510	5			
ibm_tick	1510	5	ibm_tick	1510	2
			chg_count	1510	2
ins_owns	1536	6			
upd_stock_tick(IBM)	1538	7			
ibm_tick	1538	7	ibm_tick	1538	3
			chg_count	1538	3
upd_stock_tick(TYG)	1549	8			

Table 2: Brief event histories for module MyModule in example 2, and for module IBMTicks in Example 10. Derived events are indented.

We can now assume that we have a single module of discourse, and concentrate on defining the meaning of general TREPL event expressions. First, we introduce the notion of *satisfaction predicates.* For a particular *event type*, simple or composite, its *satisfaction predicate* describes the history of the event's occurrences.

We start with qualified basic events. By the notation used so far, a qualified basic event is represented as

$$\texttt{E} = \texttt{evtkind(R(X),\ q(X))}$$

where evtkind is ins, del or upd, R is a relation name, and q denotes the event's condition expression, which can refer to the attribute values of tuple variable X [2]. The rule template for the *satisfaction predicate* of such an event is:

$$\texttt{sat}_{\texttt{E}}\texttt{(X, J)} \leftarrow \quad \texttt{evtkind\_R(X, J),\ q(X).}$$

where the predicate evtkind_R(X, J) is defined as shown in the last rule. For instance, the satisfaction predicate for the event E = ins(owns(X), X.ID = ''IBM") is:

$\texttt{sat}_{\texttt{E}}$(Name, ID, Shares, Timestamp, J) ←
ins_owns(Name, ID, Shares, Timestamp, J),
ID = 'IBM'.

The concept of "an event *immediately following* another event" can also be expressed. Take for instance the following example, where F represents an immediate sequence:

Example 12 *The immediate sequence*

F = (upd(owns(X), q_1(X)), upd(owns(Y), q_2(X, Y)))

Its semantics is defined by the following three Datalog$_{1S}$ rules:

$\texttt{sat}_1\texttt{(X, J)} \leftarrow$ upd_owns(X, J), q_1(X).
$\texttt{sat}_2\texttt{(X, Y, s(J))} \leftarrow$ upd_owns(Y, s(J)), $\texttt{sat}_1$(X, J), q_2(X, Y).
$\texttt{sat}_{\texttt{F}}\texttt{(X, Y, J)} \leftarrow$ $\texttt{sat}_2$(X, Y, J).

The first qualified basic event occurs at stage J, if an update on relation ACC is recorded at this stage and condition q_1 is satisfied. The second update on ACC must then occur at the next stage s(J) and condition q_2 must be satisfied. Observe that q_2 can refer to the tuple variable X defined by the first basic event, in addition to Y. The third rule is a *copy* rule, inasmuch as the satisfaction of composite event F coincides with that of $\texttt{sat}_2$.

[2] From now on, unless otherwise indicated, variables will denote tuples.

5.3 Derived Events and Rule Interaction

Consider example 2 again. A sample event history for module MyModule was shown in Table 1.

When rule IBMStockTicks fires, a *derived event* ibm_tick is produced. This event becomes part of the module's event history, and is recorded at the stage that the event part of the rule was satisfied, and the rule fired. Thus, the new event history for this module, that includes the ibm_tick derived events is shown on the left hand side of Table 2.

Now, let us focus on module IBMTicks in example 10, the event history of which is shown on the right hand side of Table 2. This history consists of the ibm_tick occurrences, plus the occurrences of derived events chg_count.

The rules in this module are evaluated with respect to the event history hist_IBMTicks. For instance, the translation for the last two rules of this module follows:

Decrease :
$\texttt{sat}_1\texttt{(X, J)} \leftarrow$ chg_count(X, J).
$\texttt{sat}_2\texttt{(X, Y, s(J))} \leftarrow$ ibm_tick(Y, s(J)), $\texttt{sat}_1$(X, J), Y.new_Price < Y.old_Price.
chg_count(X.IC + 1, X.DC, J) ← $\texttt{sat}_2$(X, Y, J).

Report :
$\texttt{sat}_{\texttt{R}}\texttt{(X, J)} \leftarrow$ chg_count(X, J), X.IC >= 2 * X.DC

5.4 Star Sequences and Built-in Aggregates

Consider the following event expression, which is similar to the event part of Example 6, but somewhat simpler than that.

Example 13 *A star sequence with temporal aggregation.*

```
monitor update(stock_closing) as st_cl
   ( st_cl(X, X.ID = 'IBM', X.Price > 50),
     * st_cl(Y, Y.ID = 'IBM', Y.Price > 40),
     Y.Price = Max(Y.Price)
   )
```

For this event expression, a star sequence instance begins at the *first time* when the closing price of a stock exceeds 50, and it continues, as long as it remains above 40. When it drops below 40, the next instance will start the next time it raises again above 50, and so on.

The semantics of this event expression is captured by the following rules:

```
sat1(X, J) ←                    st_cl(X, J),
                                X.ID = 'IBM', X.Price > 50.

sat2(X, Y, Y.Price, s(J)) ←     st_cl(Y, s(J)), sat1(X, J),
                                ¬sat2(_, _, _, J),
                                Y.ID = 'IBM', Y.Price > 40.

sat2(X, Y, MaxPrice, s(J)) ←
                    st_cl(Y, s(J)),
                    sat2(X, Y1, MaxPrice1, J),
                    Y.ID = 'IBM', Y.Price > 40,
                    max(Y.Price, MaxPrice1, MaxPrice).

sat3(Y, J) ←                    sat2(X, Y, MaxPrice, J),
                                Y.Price = MaxPrice.

max(A, B, B) ←                  A <= B.
max(A, B, A) ←                  A > B.
```

The first rule for sat_2 recognizes the beginning of a star sequence instance. The negated goal $\neg sat_2(\_, \_, J)$ ensures the *left maximality* of such an instance.

The recursive rule for sat_2 recognizes the rest of the events in the sequence, and guarantees its *right maximality*. At the same time, the aggregate `Max(Y.Price)` (maximum on attribute *Price*) is *incrementally* evaluated within a star sequence instance. This is accomplished by introducing an extra column (which we arbitrarily named `MaxPrice`) in predicate sat_2.

The rule for sat_3 corresponds to the goal `Y.Price = max(Y.Price)`. Observe, that since this is a condition-only goal, the stage argument J is not incremented.

In general, as explained in Section 3.3, a temporal aggregate `AggrFunc(X.A)`, where `AggrFunc` is a built-in aggregation function and `X` is the tuple variable of a star sequence goal, may be used in (i) a condition in the star expression, or (ii) a condition in a goal following the star sequence goal, or (iii) in the head of the rule. The compiler recognizes all these cases, and for each such aggregate, an extra column is added in the satisfaction predicate for the corresponding star sequence. Furthermore, the recursive rules defining this star sequence expression are extended with additional arguments and goals that allow the incremental evaluation of aggregates. The rules for `max` were shown in the last example. Similar templates can be defined for all aggregates.

5.5 From Syntax to Semantics

There exists a natural mapping from TREPL expressions to $Datalog_{1S}$, which is defined through a procedure that derives an equivalent set of $Datalog_{1S}$ rules for that expression. The resulting set of rules has a well-established formal semantics (model-theoretic and fixpoint-based) [2]. To formalize the translation, we represent TREPL expressions by their parse trees, using the following prefix notation:

1. $seq(E_i, E_j) \equiv (E_i, E_j)$. [3]
2. $*seq(E_i, E_j) \equiv (*E_i, E_j)$. [4]
3. $and(E_i, E_j) \equiv E_i \ \& \ E_j$.
4. $or(E_i, E_j) \equiv \{E_i, E_j\}$.
5. $neg(E) \equiv !E$.

[3] $(E_1, E_2, \ldots, E_n) = seq(E_1, seq(E_2, \ldots, seq(E_{n-1}, E_n)\ldots))$. Similarly for the relaxed sequence, the conjunction and the disjunction constructs.

[4] We use the binary construct *seq in place of the * TREPL construct, so that the representation is more compact and easier to follow. This is not restrictive, since $* : E \equiv *seq(E, \epsilon)$, where $\epsilon \equiv$ no event.

Example 14 *A TREPL expression with Immediate and Star Sequences.*

```
( upd(A(X), qa(X)),
  * (* ins(B(Y), qb(X,Y)), del(C(Z), qc(X,Z))),
  upd(D(V), qd(X,V)) )
)
```

The parse tree for the expression of Example 14 is shown in Figure 1. The nodes of the tree are numbered according to the postorder traversal sequence.

Each node i corresponds to a subevent E_i, e.g., $E_4 = *seq(\ ins(B(Y), q_b(X,Y)),\ del(C(Z), q_c(X,Z))\ )$.

The *satisfaction predicate* of E_i is denoted as sat_i. Each tuple of sat_i contains the variable bindings for an instance of E_i. In particular, at each stage in the event history, sat_i contains one tuple for each instance of E_i that can still contribute towards the recognition of an instance of the root event E.

As demonstrated by Example 14, the $Datalog_{1S}$ rules derived from the translation of a composite TREPL expression must model (i) the transmission of variable bindings according to the scopes of the various TREPL constructs, so that variables can be matched and conditions can be checked, and (ii) the temporal precedences among the various subevents.

Table 3 describes how this information is derived for each basic TREPL construct. Formally, it defines a simple attribute grammar [20, 9].

For each subevent Q of an TREPL event E, the second column in Table 3 defines the *Possible Predecessors Set* of Q, denoted as PPS(Q). A subevent P is a *possible predecessor* of Q within E, if in an instance of E, the satisfaction of P can *immediately* precede the first basic event of an instance of Q (i.e., the instance of Q can begin at the next stage). Because of disjunctions and the star operator, a particular subevent may have many possible predecessors.

For example, consider the immediate sequence construct `E = seq(F,G)`. F is the only possible predecessor of G; but the set of possible predecessors of F depends on which events may precede E—i.e., F inherits E's possible predecessors.

The remaining two columns of Table 3 describe the scope rules for variables in TREPL. The third column shows the set of *exported variables* of an TREPL expression. These are variables defined in the expression (variables appearing in basic events within this expression), whose scopes extend past the satisfaction of the expression. The fourth column contains for each subevent Q of an TREPL expression, the set of variables *imported* into Q. These are variables defined outside Q, whose scopes extends to Q.

Again, for `E = seq(F,G)`, the set of variables exported from E is the union of the variables exported from F and G. On the other hand, E might have imported some variable names from previous events, and if so, these are also passed down to F and G. In addition to variables inherited by E, variables imported into G include those exported from F.

Consider now the third row, for `E = *seq(F,G)`. As mentioned in Section 5.4, for each temporal aggregate evaluated over a star sequence `*F(X)`, an attribute is added to the satisfaction predicate for this star sequence goal. The set of these attributes is denoted as `Aggr(F)` and is exported to following goals. On the other hand, variables defined in a star sequence goal are normally not exported to following goals. If however, this is desired, as for instance, in Example 13, this can be modeled by a pseudo-aggregate `this`,

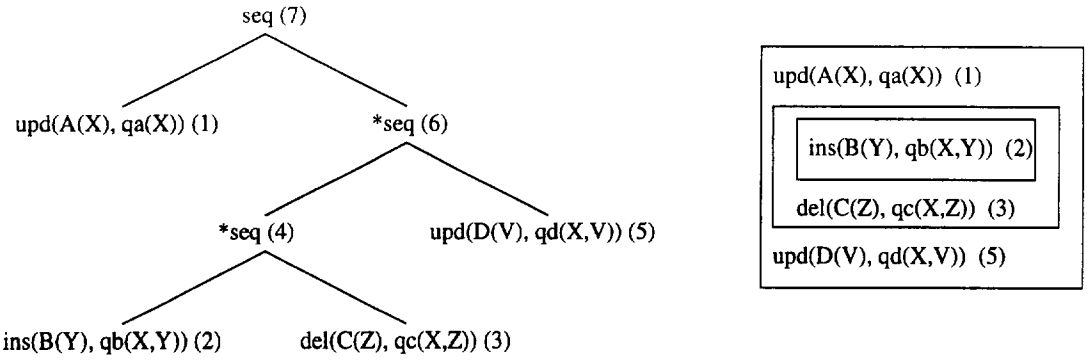

Figure 1: The parse tree for Example 14

EvtType E	*PPS*	*EVar(E)*	*IVar*
evt(R(X))	—	$\{X\}$	—
seq(F,G)	$PPS(F) = PPS(E)$ $PPS(G) = \{F\}$	$EVar(F) \cup EVar(G)$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E) \cup EVar(F)$
**seq(F,G)*	$PPS(F) = \{F\} \cup PPS(E)$ $PPS(G) = \{F\}$	$Aggr(F) \cup EVar(G)$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E) \cup EVar(F)$
or(F,G)	$PPS(F) = PPS(E)$ $PPS(G) = PPS(E)$	$\emptyset$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E)$
and(F,G)	$PPS(F) = PPS(E)$ $PPS(G) = PPS(E)$	$EVar(F) \cup$ $EVar(G)$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E)$
neg(F)	$PPS(F) = PPS(E)$	$\emptyset$	$IVar(F) = \emptyset$

Table 3: An attribute grammar for syntax-directed translation from TREPL to $Datalog_{1S}$.

which returns the last value (tuple) in the event sequence represented by a star sequence goal.

Using this attribute grammar, the generation of the actual rules is simple, as shown in Table 4. Observe that except for basic events, `X` and `Y` denote *sets* of exported variables defined in various subevents, and `IV` denotes the *set* of imported variables into a particular event type `E`. The anonymous variable `_` has replaced all variables that must be kept local.

The first row in Table 4 defines directly the event **any** (see Section 3.2), using the event history relation for the module of discourse **Mod**.

The second row deals with *simple events* having some possible predecessors [5]. Such an event `E` is satisfied at stage `s(J)`, when: (1) a possible predecessor of E was satisfied at stage `J`, (2) `E` occurs at stage `s(J)`, and (3) the condition of `E` is satisfied. Example 12 illustrates this translation.

The rules that define the satisfaction of an immediate sequence or a star sequence event are copy rules, that simply express the fact that such a sequence is satisfied, when the last component event is satisfied — see for instance the third rule in Example 12.

The rules for disjunction and conjunction are straightforward. Observe that in a conjunction, all the variables defined in its conjuncts are exported, whereas in a disjunction, none of the variables defined in its disjuncts is exported. The rule for *negated events* expresses the fact that $\neg F(X)$ is satisfied at every stage following the satisfaction of one of its predecessor events, provided that $F(X)$ is not satisfied at that stage.

[5] The translation for a simple event with no predecessors is exemplified at in the beginning of Section 5.2

Note also, that the variables of a satisfaction predicate consist of the union of its exported variables, plus the variables imported into it.

6 Related Work and Conclusions

In this paper, we proposed temporal aggregation primitives for active database rules triggered by composite events. The capabilities of TREPL in this domain extend substantially those of previous systems, such as ODE [14], Snoop [3] and SAMOS [13]. Basically, these systems can express only *counting aggregates*, such as *first, n-th* and *every n-th.* Snoop and SAMOS also provide constructs for collecting the parameter bindings of all occurrences of a particular event type, within a time interval. The composite event is signaled at the end of this interval, and its set of parameter bindings can be used for computing aggregates. However, the languages themselves do not provide any aggregation features, and thus, the signaled application has to perform the necessary aggregations, using the returned parameter bindings.

A powerful temporal aggregation construct for active rules has been recently proposed by Sistla and Wolfson [28]. This system supports *condition-action* rules only, which are expressed using a variation of Past Temporal Logic (PTL) that includes an assignment operator. However, this system cannot handle event-based expressions, which prevail in active databases.

Our technical requirements in defining temporal aggregation features have been inspired by database-centered TSMS. Traditionally, time-series analysis applications have been developed in the framework of statistical packages, e.g. [12].

Event Type E	$Datalog_{1S}$ Rule Templates
Any Basic Event	$any(J) \leftarrow hist\_Mod(\_, \_, J)$
Simple Event *evt(R(X), Cond)*	*for each* $P \in PPS(E)$ $sat_E(IV, X, s(J)) \leftarrow evt\_R(X, s(J)),\ sat_P(IV, \_, J),$ $Cond(IV, X)$
seq(F(X),G(Y))	$sat_E(IV, X, Y, J) \leftarrow sat_G(IV, X, Y, J)$
**seq(F(X),G(Y))*	$sat_E(IV, Y, J) \leftarrow sat_G(IV, Y, J)$
or(F(X),G(Y))	$sat_E(IV, J) \leftarrow sat_F(IV, X, J)$ $sat_E(IV, J) \leftarrow sat_G(IV, Y, J)$
and(F(X),G(Y))	$sat_E(X, Y, IV, J) \leftarrow sat_F(IV, X, J),\ sat_G(IV, Y, J)$
neg(F(X))	*for each* $P \in PPS(E)$ $sat_E(IV, s(J)) \leftarrow any(s(J)),\ sat_P(IV, \_, J),$ $\neg sat_F(\_, s(J))$
ϵ	*for each* $P \in PPS(E)$, $sat_E(IV, J) \leftarrow sat_P(IV, \_, J)$

Table 4: $Datalog_{1S}$ rule templates for the basic constructs of TREPL.

More recently however, database systems supporting time-series have appeared. In particular, Illustra [18] provides *abstract data types* for time-series management and analysis. Associated methods provide primitive time-series functions, and these methods can be combined to perform more complicated tasks [4]. CALANDA [10] is a special purpose object-oriented DBMS for time-series management. In contrast to these systems, which represent a time-series as a 'blob' managed by the system, SEQ [26, 27] is a relational system that stores a time-series as an ordered collection of tuples. SEQ provides a declarative sequence query language based on an algebra of query operators, and therefore, it is more conducive to algebraic query optimization and evaluation.

In this paper, we have shown that temporal aggregation capabilities as powerful as those offered by TSMS can be easily added to active rule systems that support composite events. For TREPL, this was achieved by simply enhancing the star construct that was already present in EPL. Other composite event languages such ODE [14], Snoop [3] and SAMOS [13] also provide constructs and functionality that are closely related to EPL's star construct. Thus, these languages could also be enhanced with temporal aggregation capabilities, and this provides a topic for future research.

The lessons learned during the design of TREPL are also significant for TSMSs languages, which despite providing powerful temporal aggregation capabilities, they do not match the ability of active database languages in describing complex patterns of events. For instance, they do not provide linear recursion operators for recognizing repeating event patterns. Here instead, we have shown that both kinds of temporal reasoning primitives can be naturally supported within one language.

Moreover, TSMS languages often rely on the use of multiple passes through the stored data for specifying complex queries. In active databases instead, powerful queries can be expressed by reasoning upon the incoming event sequence on-the-fly. Furthermore, it is not necessary to physically store the event-series. This corresponds to a stream-oriented processing strategy which is known to be efficient and desirable in databases.

Our plan for future research is to pursue the opportunities outlined above and add further improvements to TREPL. Among these, there is *user-defined extensible calendars*, where sequential time structures different than the ones associated with the standard gregorian calendar can be defined. Another possible improvement is support for *cross-sectional analysis*, where multiple event streams are cross-analyzed and aggregated over dimensions other than time. Another issue that is worth exploring is the design of general rule templates that will make it easier for the user to define new specialized aggregates.

In general, TREPL can be viewed as a first step towards the unification, or, at least, consolidation of languages and models used by composite-event active rule languages and time-series languages. The objective of providing a unifying paradigm for different modes of temporal and event-based reasoning suggests interesting and useful problems for further work.

Acknowledgements

We would like to thank Reza Sadri and the reviewers for their comments and suggested improvements.

References

[1] N. Arni, K. Ong, S. Tsur, and C. Zaniolo. LDL++: A second generation deductive database system. submitted for publication.

[2] M. Baudinet, J. Chomicki, and P. Wolper. Temporal Deductive Databases. In Tansel et al. [31], chapter 13, pages 294–320.

[3] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th VLDB Conference*, pages 606–617, September 1994.

[4] R. Chandra and A. Segev. Managind temporal financial data in an extensible database. In *Proc. of the 19th VLDB*, pages 302–313, 1993.

[5] R. Chandra and A. Segev. Active Databases for Financial Applications. In *Proceedings of the 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 46–52, 1994.

[6] R. Chandra, A. Segev, and M. Stonebraker. Implementing Calendars and Temporal Rules in Next Generation Databases. In *Proc. of the 10th IEEE Intl. Conference on Data Engineering*, pages 264–273, 1994.

[7] J. Chomicki. History-less checking of dynamic integrity constraints. In *Proc. of the Intl. Conf. on Data Engineering*, pages 557–564, 1992.

[8] U. Dayal, A. Buchmann, and S. Chakravarthy. The HiPAC Project. In J.Widom and S.Ceri [19], chapter 7, pages 177–206.

[9] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars : definitions, systems, and bibliography.* Springer-Verlag, 1988.

[10] W. Dreyer, A. Kotz Dittrich, and D. Schmidt. Using the CALANDA Time Series Management System. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1995.

[11] D. Schmidt et al. Time Series, a Neglected Issue in Temporal Database Research ? In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, Workshops in Computing Series, pages 214–232. Springer, 1995.

[12] FAME Software Corporation. *User's Guide to FAME*, 1990.

[13] S. Gatziu and K. R. Dittrich. Events in an object-oriented database system. In *Proceedings of the First Intl. Conference on Rules in Database Systems*, pages 23–39, September 1993.

[14] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proceedings of the 18th VLDB International Conference*, pages 327–338, 1992.

[15] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 81–90, 1992.

[16] G. Giuffrida and C. Zaniolo. EPL: Event Pattern Language. In *In the third CLIPS Conference.* NASA's Johnson Space Center, September 1994.

[17] P. Y. Hsu and D. S. Parker. Improving SQL with Generalized Quantifiers. In *Proceedings of the 10th Intl. Conf. on Data Engineering*, pages 298–305, 1995.

[18] Illustra Information Technologies, Inc. *Illustra Time-Series DataBlade Guide, Release 1.4*, September 1994.

[19] J.Widom and S.Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, 1996.

[20] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968.

[21] I. Motakis. *Temporal Reasoning in Active Databases.* PhD thesis, University of California, Los Angeles, 1997.

[22] I. Motakis and C. Zaniolo. Composite Temporal Events in Active Database Rules: A Logic-Oriented Approach. In *Proceedings of the 4th Intl. Conference on Deductive and Object-Oriented Databases*, 1995.

[23] I. Motakis and C. Zaniolo. Composite Temporal Events in Active Databases: A Formal Semantics. In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, Workshops in Computing Series, pages 332–351. Springer, 1995.

[24] S. G. Rao, A. Badia, and D. van Gucht. Providing better support for a class of Decision Support Queries. In *Proceedings of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 217–227, 1996.

[25] A. Segev and A. Shoshani. A Temporal Data Model Based on Time Sequences. In Tansel et al. [31], chapter 11, pages 248–270.

[26] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 430–441, 1994.

[27] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: Design and implementation of a sequence database system. submitted for publication, 1996.

[28] A. P. Sistla and O. Wolfson. Temporal Conditions and Integrity Constraints in Active Database Systems. In *Proc. of the 1995 ACM SIGMOD Intl. Conference on Management of Data*, pages 269–280, 1995.

[29] R. Snodgrass, S. Gomez, and E. McKenzie. Aggregates in the Temporal Query Language tquel. *IEEE Trans. on Knowledge and Data Engineering*, 5(5):826–842, Oct. 1993.

[30] M. Soo et al. Multiple Calendar Support for Conventional Database Management Systems. Technical Report TempIS Technical Report 07, Computer Science Department, University of Arizona, 1992.

[31] A. Tansel et al., editors. *Temporal Databases: Theory, Design and Implementation.* Benjamin/Cummings, 1993.

[32] C. Zaniolo. A unified semantics for active and deductive databases. In *Proceedings of the 1st International Workshop on Rules in Database Systems*, pages 271–287, 1993.

[33] C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In *Proceedings of the 4th Intl. Conference on Deductive and Object-Oriented Databases*, 1995.