Composite Event Specification in Active Databases: Model & Implementation

N. H. Gehani
H. V. Jagadish
O. Shmueli

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Active database systems require facilities to specify triggers that fire when specified events occur. We propose a language for specifying composite events as *event expressions*, formed using *event operators* and events (primitive or composite). An event expression maps an *event history* to another event history that contains only the events at which the event expression is "satisfied" and at which the trigger should fire. We present several examples illustrating how quite complex event specifications are possible using event expressions.

In addition to the basic event operators, we also provide facilities that make it easier to specify composite events. "Pipes" allow users to isolate sub-histories of interest. "Correlation variables" allow users to ensure that different parts of an event expression are satisfied by the same event, thereby facilitating the coordination of sub-events within a composite event.

We show how to efficiently implement event expressions using finite automata. Each event causes an automaton to change state. When an automaton reaches an accepting state, a composite event of interest is recognized, and the corresponding trigger fired.

Events have attributes. For primitive events, these could be parameters of the activity that caused the event, selected parts of the database state, or functions computed therefrom. For composite events, attributes are derived from the attributes of the constituent primitive events. These attributes can be used in checking conditions, and in any actions triggered. Event expressions can specify values (or sets or ranges of values) for particular attributes, and can even require that some attributes be equal. The composite event specified by the expression does not occur unless the specified condition on attributes is satisfied.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada, 1992

1. INTRODUCTION

Of late, there has been a surge of interest in active databases [2, 5, 5, 8, 12, 14-16]. In an active database, a trigger fires when an event of interest happens and some condition is satisfied. Most efforts have focussed on the trigger firing mechanism and the execution of the triggered action. However, recent work [3, 6, 9], has recognized the importance of event specification. Of special interest is the specification of *composite* events, which are constructed from (simpler) primitive events [5].

We propose a language for composite event specification. Specifying composite events is non-trivial because a composite event refers to events that do not all happen simultaneously and because a composite event can often be caused in more than one way. Composite events are specified as *event expressions*, which are formed using *event operators*. An event expression maps an *event history* to another event history that contains only the events at which the event expression is "satisfied." An event history, or simply a history, is an ordered set of primitive events. Primitive events are database operations of interest such as the update of a data value, the commit of a transaction, etc.

To facilitate writing composite event expressions we provide some additional constructs. "Pipes" allow users to focus only on events of interest. "Correlation variables" allow users to ensure that different parts of an event expression are satisfied by the same event. This facilitates coordination of sub-events within a composite event. One can think of them as "pointers" to specific history events, as free variables in a logic program, or a communication mechanism as in statecharts [10].

Events have attributes. For primitive events, these could be parameters of the activity that caused the event, selected parts of the database state, or functions computed therefrom. For composite events, attributes are derived from the attributes of the constituent primitive events. These attributes can be used in checking conditions, and in any procedure actions triggered. Event expressions can specify values (or sets or ranges of values) for particular attributes, and can even require that some attributes be equal. The composite event specified by the expression does not occur unless the specified condition on attributes, if any, is satisfied. We note a similarity between events with attributes and parametrized states in statecharts [10].

Event expressions have the same expressive power as regular expressions. As such, any mapping from histories to histories that can be specified by an event expression can be executed by a finite automaton. We show explicitly how to construct such an automaton for an event expression. We also show how (multiple) automata can be constructed to handle events with attributes. These automata can directly be translated into code such that an inexpensive state transition table look-up for relevant automata is all that is required each time an event (such as an update) occurs in the database.

The paper is organized as follows. In section 2 we formalize event expressions, introduce a collection of useful operators and present examples. In section 3 we explain how to build finite automata for event expressions. The notion of an occurrence tuple, which is a "justification" for the occurrence of a composite event, is introduced in Section 4. Section 5 treats correlation variables in expressions. Event attributes are examined in section 6. Two comprehensive examples are presented in Section 7. Section 8 concludes.

2. EVENTS

An "event" is a happening of interest. Events, including composite events, happen instantaneously at specific points in time. In object-oriented databases, for example, events are related to object manipulation actions such as creation, deletion, and update or access by an object method (member function). Similarly, in a relational database, events are related to actions such as insert, delete, and update. The events can be specified to happen just prior to or just after the above actions. In addition, events can be associated with transactions and specified to happen immediately after a transaction begins, immediately before a transaction attempts to commit, immediately after a transaction commits, immediately before a transaction aborts, and immediately after a transaction aborts. Events can also be associated with time, for example, clock ticks, and the recording of the passage of a day, an hour, a second, or some other time unit.

Event specification must start with a set of *basic* events, such as the ones mentioned above, which are supported by the database system. *Primitive* events[1] are basic events optionally qualified by a *mask*, which is a predicate used to hide or "mask" the occurrence of an event. For instance, the event "before large withdrawal" can be composed using the basic event "before execution of the method withdrawal" and qualifying it with the mask "withdrawal amount > 1000". We shall use primitive events as the basis of our discussion of composite event specification. We assume that primitive events are mutually exclusive and that their number is finite.

2.1 EVENT OCCURRENCES AND EVENT HISTORIES

An *event occurrence* (informally referred to as an event) is a tuple of the form *(primitive event, event-identifier)*. Event-identifiers (eids) are used to define a total ordering, denoted by <, on event occurrences. An example of an event identifier is a time-stamp specifying the time at which the primitive event occurred.

An *event history*, or simply a *history*, is a finite set of event occurrences in which no two event occurrences have the same event identifier. When this set is empty, the history is called the *null* history. The event occurrences in a history are sometimes referred to as *points*. A special primitive event *start* occurs at the beginning of the system history Ψ. Its eid is less than the eids of all other event occurrences.

2.2 EVENT EXPRESSIONS

An *event expression* E, which specifies a primitive or composite event, is a mapping from (domain) histories to (range) histories:

E: *histories* $\rightarrow$ *histories*

The result of applying an event expression to a history h is itself a history which contains the event occurrences of h at which the events specified by E take place. These intuitive notions are formalized below.

Let E be an event expression. $E[h]$ denotes the application of E to history h. It is always the case that $E[h] \subseteq h$. We say that E *takes place* at event e in h iff $e \in E[h]$. An event occurrence $e \in h$ *satisfies* expression E iff $e \in E[h]$; E is said to be *satisfied* by e.

An event occurrence e_1 takes place after (before) an event occurrence e_2 if the eid of e_1 is larger (smaller) than the eid of e_2. Two event occurrences e_1 and e_2 refer to the same event occurrence if their eids are identical.

An *event expression* is formed using primitive events and the operators (connectives) described below. An event expression can be *NULL*, any primitive event a, or an expression formed using the operators $\wedge$, ! (not), *relative* and *relative*+. The semantics of event expressions are defined as follows (E and F are used to denote event expressions):

1. $E[null] = null$ for any event E, where *null* is the empty history.
2. $NULL[h] = null$.
3. $a[h]$, where a is a primitive event, is the maximal subset of h composed of event occurrences of the form (a, eid).
4. $(E \wedge F)[h] = h_1 \cap h_2$ where $h_1 = E[h]$ and $h_2 = F[h]$.
5. $(!E)[h] = (h - E[h])$.
6. $relative(E, F)[h]$ are the event occurrences in h at which F is satisfied assuming that the history started immediately following *some* event occurrence in h at which E takes place.

   Formally, $relative(E, F)[h]$ is defined as follows. Let $E^i[h]$ be the i^{th} event occurrences in $E[h]$; let h_i be obtained from h by deleting all event occurrences whose eids are less than or equal to the eid of $E^i[h]$. Then $relative(E, F)[h] = \bigcup_i F[h_i]$,
   where i ranges from 1 to the cardinality of $E[h]$.

1. Primitive events are referred to as *logical* events in [9].

7. $relative+(E)[h] = \bigcup_{i=1}^{\infty} relative^i(E)[h]$ where
$relative^1(E) = E$ and
$relative^i(E) = relative(relative^{i-1}(E), E)$.

Regular expressions are widely used for specifying sequences The above event expression language has the same expressive power as regular expressions [9].[2] It can be shown that the operators $\wedge$, !, *relative*, and *relative*+ constitute a minimal operator set; reducing it will make the expressive power less than that of regular expressions.

2.3 MORE OPERATORS FOR EVENT EXPRESSIONS

We present some additional operators (connectives) that make composite events easier to specify. These operators do not add to the expressive power provided by the operators introduced in the previous section.

Let h denote a non null history, and E, F, and E_i denote event expressions. The new operators are

a. $E \vee F = !(!E \wedge !F)$.

b. *any* denotes the disjunction of all the primitive events except for *start*.

c. $prior(E, F)$ specifies that an event F that takes place after an event E has taken place. E and F may overlap. Formally, $prior(E, F) = relative(E, any) \wedge F$.

d. $prior(E_1, \ldots, E_m)$ specifies occurrences, in order, of the events $E_1, E_2, \ldots, E_m$.
$prior(E_1, \ldots, E_m) =$
$prior((prior(E_1, \ldots, E_{m-1}), E_m)$.

e. $sequence(E_1, \ldots, E_m)$ specifies immediately successive occurrences of the events $E_1, E_2, \ldots, E_m$:

   1. $sequence(E_1, \ldots, E_m) =$
   $sequence((sequence(E_1, \ldots, E_{m-1}), E_m)$.

   2. $sequence(E_1, E_2) =$
   $relative(E_1, !(relative(any, any))) \wedge E_2$.

   The first operand of the conjunction specifies the first event following event E_1. The second operand specifies that the event specified by the complete event expression must satisfy E_2.

f. *first* identifies the first event in a history.
$first = !relative(any, any)$.

g. $(E|F)[h] = F[E[h]]$; i.e., F applied to the history produced by E on h. Operator | is called *pipe*, with obvious similarity to the UNIX® operator.

h. $(<n>E)$ specifies the nth occurrence of event E. Formally,
$(<n>E) = ((E|seq(any_1, any_2, \ldots, any_n))|first)$,
where each any_i is simply *any*.

i. $(every <n>E)$ specifies the nth, 2nth, ..., occurrences of event E. Formally,
$(every <n>E) = (E|relative+(<n>any))$.

j. $(F / E)[h] = F[h']$ where h' is *null* if $E[h] = null$ and otherwise h' is the history obtained from h by eliminating all the event occurrences before and including $(<1>E)[h]$. Formally, $F/E = relative((!prior(E, any) \wedge E), F)$, equivalently, $F/E = relative((E|first), F)$.

k. Suppose that E takes place m times in h. $F /+ E[h] = \bigcup_{i=1}^{m} F[h'_i]$. h'_i, $1 \le i < m-1$, is obtained from h by eliminating all event occurrences before and including event $(<i>E)[h]$ and all event occurrences including and following $(<i+1>E)[h]$. h'_m is obtained from h by eliminating all event occurrences before and including event $(<m>E)[h]$.

   E is used to delimit sub-histories of h, where the "delimiter" are event occurrences at which E takes place. F is applied to each such sub-history, and the results of these applications are combined (unioned) to form a single history.

l. $firstAfter(E_1, E_2, F)[h]$ specifies events E_2 that take place relative to the last preceding occurrence of E_1 without an intervening occurrence of F relative to the same E_1. Formally,

   $firstAfter(E_1, E_2, F) =$
   $(E_2 \wedge !prior(F, any)) /+ E_1$.

m. $before(E) = prior(E, any)$.

n. $happened(E) = E \vee prior(E, any)$.

o. $prefix(E)$ $[h]$ is satisfied by each event occurrence e such that there exists a history h' identical to h up to event occurrence e, and E is satisfied in h' at some event occurrence following e. In other words, $prefix(E)$ is recognized at each event occurrence as long as a possibility exists that an E event will be recognized eventually. This operator is normally used in the form $!prefix(E)$, which occurs as soon as we can be sure that E cannot occur.

p. $E{*}T$ is a series of zero or more E events followed by a T event.
$E{*}T = T \wedge !prior(!E, T)$.

In addition to the operators described above, there may be composite event sub-expressions that are used repeatedly in particular applications. Such sub-expressions may be defined, named and then used in building up larger expressions. We do this in some of the examples later on.

2.4 EXAMPLES

2. In [9] operator $\vee$ is used instead of operator $\wedge$ but the proof is similar.

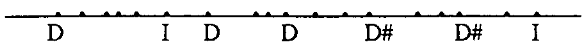
® UNIX is a trademark of USL.

2.4.1 *EVENT EXPRESSIONS:*

1. All occurrences of an event a:
```
a
```
2. The 5th occurrence of `deposit`:
```
(<5>deposit)
```
3. `deposit` followed immediately by `withdraw`:
```
sequence(deposit, withdraw)
```
4. `deposit` followed eventually by `withdraw`:
```
prior(deposit, withdraw)
```
5. `deposit` followed eventually by `withdraw` with no intervening `interest`:
```
relative(deposit, !before(interest))
∧ withdraw
```
6. Event expression that is satisfied when an E occurs provided there is no "non E" event before it. We are essentially recognizing a series of E events:
```
E ∧ !prior(!E, E)
```

2.4.2 DISCOUNT RATE CUT: The United State Federal Reserve Board raises and lowers a key interest rate, called the discount rate, to control inflation and economic growth. Three or more successive discount rate cuts (D) without an intervening discount rate increase (I) is a rare phenomenon and is of interest to the financial community. Many other events can occur, for example, the prime rate may be cut and the stock market can crash, but these events do not interest us here. Our problem is to write an event expression that is satisfied by such cuts in the discount rate.

Here is an example history with the dots marking the events in the history with the discount rate cut events labeled by D (decrease) and increases labeled by I (increase):



The composite event of interest occurs at the last two D events (marked with #).

We will now give an event expression that specifies a composite event satisfied when three or more successive discount rate cut events D take place without an intervening rate increase event I. We specify this composite event in steps. First, the event expression
```
prior(I, D)
```
specifies D events that are preceded by an I event. Expression
```
!prior(I, D)
```
specifies all events except the occurrences of D that are preceded by I. Expression
```
!prior(I, D) ∧ D
```
specifies D events that are not preceded by an I event.

The expression
```
relative(D, !prior(I, D) ∧ D)
```
specifies a D event followed eventually by another D event with no intervening I events. This expression gives us a pair of D events with no intervening I events. Note that in this case, the `relative` operator is used to look at the history starting after a D event.

Finally, the event that we are interested in can be specified as
```
relative (relative (D, !prior(I, D) ∧ D),
         !prior(I, D) ∧ D)
```
The outermost `relative` finds another D without a preceding I giving us three D events without an intervening I event.

Using the pipe operator, we can write the composite event for the three successive discount rate cuts simply as
```
(I ∨ D) | sequence(D, D, D)
```

3. AUTOMATA CONSTRUCTION

In the previous section we developed a powerful mechanism to specify composite events. In a real database, we have to be able to detect these occurrences on the fly, for instance as a consequence of some update. Clearly, it is impractical to perform a complete check for each composite event every time a primitive event occurs in a database. We present an incremental detection technique in this section.

Since event expressions are equivalent to regular expressions, except for ε which is not expressible using event expressions [9], it is possible to "implement" event expressions using finite automata. The history in the context of which an event expression is evaluated provides the sequence of input symbols to the automaton implementing the event expression. The automaton is fed as input the primitive event components from event occurrences in the history, one at a time, (in eid order) as they occur. If, after reading a primitive event, the automaton enters an accepting state, then the event implemented by the automaton is said to take place at the primitive event just read. The history need not be known in its entirety *a priori*, and the automaton can be used, in "real time", as a triggering device.

Given an event expression, E, we now show how to build an automaton M_E. The construction resembles that of an automaton for a regular expression. All machines have a non-accepting start-state. Let ε denote an empty event which can be used in automata transitions; it is similar to the empty string in automata theory. We consider the possibilities for E:

1. *NULL*: M_E has a start-state and an additional state p (non-accepting). On all primitive events the move from the start-state is into p; p self-loops on all primitive events.

2. a, where a is a primitive event: M_E has three states: the start state (non-accepting), p (accepting) and q (non-accepting). On a the move from the start-state is into p. On all other primitive events the move is into q. On a both p and q move into p and on all other primitive events both p and q move into q.

3. $E_1 \wedge E_2$: Using inductively M_1 for E_1 and M_2 for E_2, build an automaton M that is the product of the two, with each state in M corresponding to a pair of states, one in M_1 and one in M_2. A state in M is marked accepting if and only if the states of M_1 and M_2 in the corresponding

pair are both accepting. All other states in M are non-accepting. The unique start state of M is the state corresponding to the pair of start states of M_1 and M_2. Let Δ_i be the transition relation of M_i, $i=1,2$. From state (p, q) on a M moves to (r, s) such that $r \in \Delta_1(p, a)$ and $s \in \Delta_2(q, a)$.

4. $!E$: Let M_E be a deterministic automaton for E. The automaton for $!E$ is obtained from M_E by switching the roles of accepting and non-accepting states, except for the start-state which remains non-accepting.
5. *relative*(E, F): Build two automata M_E and M_F for E and F, respectively. Connect the accepting states of M_E with ε (empty event) transitions to the start state of M_F. The accepting states are the accepting states of M_F and the start state is that of M_E. Note that M_E may be non-deterministic.
6. *relative*$+(E)$: Build an automaton M_E for E. Connect the accepting states of M_E with ε (empty event) transitions to the start state of M_E. The accepting states of the newly constructed automaton are those of M_E, and the start state is the start state of M_E. Note that the new automaton may be non-deterministic.

Theorem 1: Let h be a history. Let E be an event expression and M_E the constructed automaton. Let $M_E{}^d$ be a deterministic automaton accepting the language accepted bt M_E. Then, the last event occurrence of h is in $E[h]$ iff $M_E{}^d$ is in an accepting state after scanning h.

Proof follows from the construction, inductively.

Corollary 1: Let h, E, and $M_E{}^d$ be a deterministic automaton as above. Then $E[h]$ contains exactly the event occurrences o such that when scanning h, $M_E{}^d$ is in an accepting state immediately after scanning o.

Automata can be derived for the additional operators by using their translation into basic operators. While more efficient direct constructions may exist, we shall not present these, for brevity. The only exception are the $|$(pipe), $/+$ and the *prefix* operators, items g, k and o respectively in Sec. 2.3, for which we have not provided a translation into basic operators so far.

Consider an expression $E|F$ employing the pipe operator. Let, inductively, M_E (respectively, M_F) be a finite automaton for E (respectively, F). Without loss of generality, both automata are deterministic with transition functions Δ_E and Δ_F. Form a cross-product machine M from M_E and M_F. The start-state of M is the state composed of the two start states of M_E and M_F. Suppose M is in state (p, q) scanning a. If $\Delta_E(p, a)=f$ is an accepting state of M_E then the new state of M is $(f, \Delta_F(q, a))$, otherwise it is (f,q). Intuitively, the effect is that M_F only "sees" occurrences which M_E declares to be in its output history. The accepting states of M are those (p, q) where p is an accepting state of M_E and q is an accepting state of M_F.

Consider an expression $F/+E$. Using inductively (w.l.o.g the deterministic) finite automata M_E for E and M_F for F, build an automaton M that is the product of the two, with each state in M corresponding to a pair of states, one in M_E and one in M_F. A state in M is accepting if and only if the component state of M_F is accepting. The unique start state of M is the state corresponding to the pair of start states of M_E and M_F. Let Δ_E (Δ_F) be the transition function of M_E (M_F). We now explain how M moves when in state (p, q). On reading a, the move is into (r, s) where $\Delta_E(a)=r$ and $\Delta_F(a)=s$, unless r is an accepting state of M_E. If r is an accepting state of M_E, then instead of moving into (r, s), move into $(r, Start_F)$. Finally, construct the desired machine M' as a combination of M_E and M, with the start state of M' being the start state of M_E, the accepting states of M' being the accepting states of M, and all transitions out of all accepting states of M_E are replaced by ε (empty event) transitions to the start state of M.

Consider an expression *prefix*(E). Using inductively automaton M_E for E, build an automaton M for *prefix*(E) as follows. M is the same as M_E except for the specification of accepting states. A state q in M is an accepting state iff there exists a sequence of primitive events taking q to a state which is an accepting state in M_E.

4. OCCURRENCE TUPLES

An *occurrence* tuple encodes a derivation tree showing why sub-expressions of an event expression are "satisfied". When a composite event occurs, it is possible to derive one or more occurrence tuples for it to explain why it occurred, and to identify particular events in the history that were relevant to its occurring. The identification of an occurrence tuple from a history in an active database is akin to parsing a string in a compiler.

Let E be an event expression and let $E_1, \ldots, E_k$ be its sub-expressions (including itself as E_1). These sub-expressions are determined by inductively decomposing the event expression to form a "parse-tree". These sub-expressions can be (uniquely) ordered by a (pre-order) traversal of this tree. For example, if $E = relative(a, relative+(a))$, then the sub-expressions of E are E, a (first), *relative*$+(a)$, and a (second, inside the *relative* $+$).

An *occurrence tuple* for E (for a history h) is a tuple of the form $(origin, f_1, e_1, \ldots, f_k, e_k)$ where *origin* is an event occurrence in h with eid less than or equal to the eid of each e_i or f_i. Each e_i, for $i=1, \ldots, k$, is an event occurrence in h, at which the sub-expression E_i is "satisfied". Each f_i is an event occurrence whose eid is less than or equal to the eid of e_i, $i=1,\ldots,k$; it indicates "from" where in the history the satisfaction of the i'th sub-expression starts, $origin=f_1$. So, the i'th sub-expression is satisfied on the history segment from f_i till e_i.

If the sub-expression E_i consists of the operator *relative*$+$ and its argument, then e_i is not a single event occurrence but rather is a sequence of one or more occurrence tuples, f_i's eid must be less than or equal to the eids in these tuples. Since E_1 is always the event E itself, the eid of its occurrence point e_1 must be greater than or equal to the eids of all other e_i.

A sub-expression E_i *terminates* at event occurrence e relative to an occurrence tuple t if either e_i, its entry in t, is e, or E_i is of the form *relative*$+(F)$ and e is the *effective* e_i which is defined as the largest eid in the last occurrence tuple contained in e_i.

See the appendix for a formal definition of an occurrence tuple for an event expression.

5. CORRELATION VARIABLES

Correlation variables are used to refer to the same event in the history in different parts of an event expression. Consider the following event expression E that contains the correlation variable x:

$$E = \exists x\, prior(b{=}x,\ c) \wedge\ !relative(x,\ prior(a,\ c)) \wedge relative(x,\ prior(d,\ c))$$

Consider the following histories (h_1 is a prefix of h_2 which is a prefix of h_3):

$h_1 = e\ b\ a\ c$
$h_2 = e\ b\ a\ c\ d\ b\ c$
$h_3 = e\ b\ a\ c\ d\ b\ c\ d\ b\ c$

We want to determine if E can be satisfied (will trigger) at the last event, a c event, in the above histories. When determining the points at which E can be satisfied in the above histories, the correlation variable x will be associated with a specific b event in each history. In case of h_1, x must be associated with the only b present; E will not trigger at c because $!relative(x,\ prior(a,\ c))$ is not satisfied. In case of h_2, there are two b events. The first has the same problem as in h_1. If we associate x with the second b in of h_2, then $relative(x,\ prior(d,\ c))$ is not satisfied. In case of h_3, there are three choices of b with which to associate x. If we choose the first, $!relative(x,prior(a,\ c))$ is not satisfied. If we choose the third, $relative(x,\ prior(d,\ c))$ is not satisfied. However, if we choose to associate x with the second b, then E will trigger at the last c.

To appreciate the role played by x, consider the event expression E', given below, which is the same as E except that the last occurrence of x has been replaced by b.

$$E' = \exists x\, prior(b{=}x,\ c) \wedge\ !relative(x,\ prior(a,\ c)) \wedge relative(b,\ prior(d,\ c))$$

E' triggers on h_3 in the same way as E. However, it also triggers on h_2, where x is associated with the second b. $relative(b,prior(d,\ c))$ is satisfied now on account of the first b, which does not have to be associated with x.

Finally, the event expression E'', without correlation variables, given below, does not trigger on h_1, h_2, h_3, or any other history of which h_1 is a prefix. The reason is that h_1 has in it the sequence $b\ a\ c$ guaranteeing that the clause $!relative(b,\ prior(a,\ c))$ can never be satisfied.

$$E'' = prior(b,\ c) \wedge\ !relative(b,\ prior(a,\ c)) \wedge relative(b,\ prior(d,\ c))$$

We shall use the notation E to denote an event expression without any correlation variables, $E{<}x_1,\ x_2,\ \cdots\ x_n{>}$ to denote an event expression E with a "free" set of correlation variables $\{x_1,\ x_2,\ \cdots\ x_n\}$, $n \geq 0$, and the notation $E{<>}$ to denote an event expression E with correlation variables none of which are "free".

We use the quantifier $\exists$ to specify the scope of correlation variables. The syntax of event expressions with correlation variables, and the concepts of free and bound variables, are defined inductively below. The definition of satisfaction of an event expression by an occurrence tuple is extended.

1. E: E has no free variables.
2. $relative(E{<}x_1,x_2,\ldots,x_m{>},\ F{<}y_1,y_2,\ldots,y_n{>})$: The free variables are $\{x_1,\ x_2,\ \cdots,\ x_m\} \cup \{y_1,\ y_2,\ \cdots,\ y_n\}$.
3. $relative+(E{<>})$: Operator $relative+$ cannot be applied to an event expression with free correlation variables.
4. $E{<}x_1,\ x_2,\ \cdots,\ x_m{>} \wedge F{<}y_1,\ y_2,\ \cdots,\ y_n{>}$: The free variables are $\{x_1,\ x_2,\ \cdots,\ x_m\} \cup \{y_1,\ y_2,\ \cdots,\ y_n\}$.
5. $!E{<}x_1,\ x_2,\ \cdots,\ x_m{>}$: The free variables are $\{x_1,\ x_2,\ \cdots,\ x_m\}$.
6. $\exists\, w\ E{<}x_1,x_2,\cdots,\ x_n,w{>}$: All variables in $\{x_1,\ x_2,\ \cdots,x_n\}$ are free in this expression, w is bound.

   An occurrence tuple t *satisfies* a sub-expression of the form $\exists\, w\ A{<}x_1,\ x_2,\ \cdots,\ x_m{>}$ of E, if it satisfies $A{<}x_1,\ x_2,\ \cdots,\ x_m,\ w{>}$ and all sub-expressions, appearing in $A{<}x_1,\ x_2,\ \cdots,\ x_m,\ w{>}$, which are equated with w are satisfied, and terminate at the same event occurrence.
7. $E{<}x_1,\ x_2,\ \cdots,\ x_n{>} = w$, where w is not in $\{x_1,\ x_2,\ \cdots\ ,\ x_n\}$. The free variables are $\{x_1,\ x_2\ ,\cdots,\ x_n,w\}$.

   An occurrence tuple t *satisfies* the sub-expression $A{<}x_1,\ x_2,\ \cdots,\ x_m{>} = w$ of E, if t satisfies $A{<}x_1,\ x_2,\ \cdots\ ,\ x_m{>}$.

Finite automata construction for expressions containing correlation variables is given in the Appendix.

Let us examine another example:

$$E = \exists\, x\ relative(a,relative(b{=}x,c)) \wedge\ !relative(x,\ prior(d,\ any))$$

Consider the history $h = a\ b\ d\ c\ a\ b\ c$. For the last c to satisfy E we must identify a point x. The first b, namely $a\ (b{=}x)\ d\ c\ a\ b\ c$ is problematic, because in $d\ c\ a\ b\ c$ the last c is preceded with a d. Choosing x to label the second b, we get $a\ b\ d\ c\ a\ (b{=}x)\ c$, now we can satisfy c in the first conjunct of E and there is no satisfaction of $prior(d,\ any)$ in the history c and so the expression E is satisfied by the last c in h.

In the same spirit we can introduce correlation variables into the additional operators that we have presented in Section 2.3. For example, operator "/+" was explained from first principles since, in Section 2.3, it was too cumbersome to define in terms of event expressions. However, with the help of correlation variables, such definition becomes easy:

$$F\ /{+}\ E = relative(E{=}x,\ F) \wedge\ !prior(prior(x,\ E),any) \wedge\ !E.$$

Finally, continuing our discount rate example from Section 2, we will now use correlation variables to specify three or more successive cuts in the discount rate without any intervening increases:

```
relative(D, prior(D, D=d3) ∧ !prior(I, d3))
```

Correlation variables make it simpler to write the expression by giving us a handle with which to refer to specific instances of events.

6. EVENTS WITH ATTRIBUTES

A set of attributes can be associated with each primitive event. These attributes can carry information about the action that caused the event to occur, such as the transaction id, the issuing user, and so on. The attributes can also record information about the state of the database (as visible to the transaction causing the event to occur) at the time the event occurs. This information can be used when a composite event occurs (at a later time), typically by routines executed in the action part when a trigger fires.

For example, the event *hire* might have three positional attributes which semantically refer to the *name*, *age* and *sex* of the employee. Each event occurrence will have attribute values associated with these attributes. For example, *hire*(*smith*, 27, *m*). Another example is an event *fire* that has one attribute: the *name* of an employee. An example of a *fire* event is *fire*(*smith*).

Composite events "inherit" their attributes from constituent simpler events. Consider the following example:

immediate_re_hire(X) = *sequence*(*fire*(X), *hire*(X, Y, Z))
  // attribute specification for a composite event.

The event *immediate_re_hire* is the a hiring of an employee immediately after firing the employee: the attribute of interest is the name of the employee.

6.1 ATTRIBUTE COLLECTION

Given a composite event specification of the form *composite_event*($\bar{X}$)= E, and an automaton implementing E (ignoring the attributes), one may ask for the values of the attributes each time the automaton reaches an accepting state. A difficult issue in attribute value collection is the multiplicity of ways (parsings) in which an event occurrence may be declared as part of the output history, i.e., the multiplicity of occurrence tuples that could justify a particular composite event occurrence.

Each occurrence tuple specifies a different interpretation of the history points where sub-expressions are satisfied, and hence a different set of attribute values will in general be collected. Such a set of attribute values is called a *tuple of compatible attribute values*. For example, consider the composite event *fire_after_hire* defined as

fire_after_hire(X, W) = *prior*(*hire*(X, Y, Z), *fire*(W))

Once this expression is satisfied, the value, w, of the fired employee, W, is uniquely determined. There may be many *hire* event occurrences preceding this *fire* event occurrence. Each such *hire* occurrence may supply a different value, x, for X resulting in a different tuple (x, w) of compatible values.

One way is to legislate a preference. Natural candidates are the "most recent" or "earliest" sets of satisfying events. This seems to be an ad-hoc choice as there might be other interesting criteria related to, say, the value of the attributes. So, we shall break the problem into two parts. The first part is the generation of *all* possible sets of compatible tuples of attribute values. The second part is choosing attribute values of interest from within this collection. The second part can be thought of as asking a query against a relation, which is the set of compatible tuples.

There are two main steps in collecting attributes values. One component is an annotated version of the history that we call *the annotated history*; it contains the necessary information to form all possible ways of satisfying an expression. The second component is an *annotated* automaton that is used to produce the annotated history. When tuples of compatible attribute values are needed, they can be generated easily from the annotated history.

This technique creates all possible compatible tuples of attribute values for a composite event occurrence. Typically, one is interested in applying a (selection or aggregation) query to this set of tuples. Optimization issues with regard to how such selections and/or aggregations can be moved in are a topic for further research. A detailed description of this technique, including possible optimizations, will be given in a future paper.

6.2 EVENT MODIFICATION BY ATTRIBUTE SPECIFICATION

If the same variable appears in multiple places in an expression, it constrains the corresponding values to be identical. We shall illustrate this concept using an example.

Consider a brokerage system with accounts 1, $\cdots$, m. Suppose there are two kinds of primitive events *order* and *perform*. The declarations are as follows:

Primitive Events:
order(*account_num*, *ask_quantity*)
perform(*account_num*, *actual_quantity*)
Composite Events:
complete(I)=*prior*(*order*(I, Q) , *perform*(I, A))

So, the *complete* event checks for a *perform* event occurrence which follows an *order* occurrence for the *same* account number. Formally, this expression is a shorthand for the following set of expressions:

{ *complete*(i)=*prior*(*order*(i, Q) , *perform*(i, A)) :
  $i \in$ *Accounts*}

where *Accounts* is the set of all possible account numbers.

Thus, we can express a set of expressions (one for each value of i in the above example), while writing a single expression, via attributes. This, in effect, makes the alphabet potentially infinite and takes us out of the realm of finite automata.

In general, there is no need to restrict constraints between attribute values just to the simple equality between attributes discussed above. One could specify arbitrary constraints among attribute values. For instance, an event expression may be conjoined with an expression of the form $X \neq Y$, indicating that the values of attributes X and Y need be distinct; or of the form X = *constant*, constraining the value of attribute X to equal *constant*. If no constraint is specified, directly or transitively, for X and Y then they specify values that may be equal or not equal.

In the Appendix we provide a construction of (multiple) automata to implement events modified by attribute values.

7. EXAMPLES

Our composite event specification facilities provide a convenient mechanism for specifying complex situations. The declarative nature of these facilities makes complex situations relatively easy to specify when compared to how one would write code to detect these situations in an imperative language such as C++ or C.

We will consider two examples: specification of the end of a point in the game of racketball, and detection of plane landings at an airport. Writing event specifications, particularly when these specifications are declarative, clarifies the conditions required to make an event happen.

7.1 RACKETBALL

Racketball is a ball game that typically is played by two players in an enclosed room. The ball can hit the walls, the ceiling, or the floor. Each of these is significant in that there are rules as to when the ball can hit them. For the purpose of this example, we ignore the lines on the floor and on the back wall, and allow the players only one serve per point.

A point starts when a player, say player 1, starts off by "serving" the ball, the ball must hit the "front" wall directly, and then player 2 must hit the ball. Player 2 can hit the ball before it lands on the floor the second time (no landing on the floor or one landing on the floor is okay; the ball can hit the side walls or the back wall in between, but must not hit the front wall again before the hit). Player 2's hit must take the ball to the front wall but it can hit the other walls in the process. Players 1 and 2 now alternate in hitting the ball. The point terminates whenever the above rules are violated.

We wish to write an event expression to detect the end of a point. First, we define the primitive events:

1. Player 1 serves: s_1
2. Player 2 serves: s_2
3. Player 1 hits: h_1
4. Player 2 hits: h_2
5. Ball hits floor: `floor`
6. Ball hits (touches) front wall: `front`
7. Ball hits (touches) a side wall, the ceiling, or the back wall: `wall`

A point ends as a result of a bad serve, a bad hit, or because the player was unable to return the ball. We now specify each of these events:

1. *Bad serve*: A serve is bad if it does not hit the front wall directly.

   H1 = sequence($s_1 \vee s_2$, !front)

2. *Bad hit*: Ignoring the ball hitting the other walls, the next event after a hit must be the ball hitting the front wall; otherwise, the point ends:

   H2 =
   (!wall) | ((first $\wedge$!front) /+ ($h_1 \vee h_2$))

3. *Unable to Return*: First, we define `Twice` as the ball hitting the floor or the front wall twice, or hitting the front wall immediately after hitting the floor:

   ```
   Twice = sequence(floor, floor) v
           sequence(front, front) v
           sequence(floor, front)
   ```

   We now want to specify the end of a point resulting from the event `Twice` (ignoring the ball hitting the side walls). This means that the other player was not able to hit the ball back in time:

   `!wall | Twice`

   The above event expression may be satisfied (triggered) by several events after a player is unable to return. For example, if the ball bounces ten times on the floor after hitting the front, then the expression will be triggered by every bad bounce. We can refine this expression to catch only the first error with:

   H3 = !wall |
   (Twice | first) /+ ($h_1 \vee h_2 \vee s_1 \vee s_2$)

The end of a point is simply the disjunction of the above three expressions dealing with the different ways in which a point ends:

`H1 v H2 v H3`

In writing an event expression to detect the end of a point, we have written event expressions that isolate bad hits. An alternative strategy would be to write an expression that is satisfied when the point can no longer be continued because it does not satisfy the rules for continuing a point.

We write an event expression to detect the end of a point assuming that player 1 is the server. In this example, we use the event operator `prefix` (E).

Player 1 starts off by serving, and the ball must hit the front wall:

sequence(s_1, front)

A player can return the ball before it hits the floor or after the ball hits the floor once but before it hits the floor a second time. Ignoring the ball hitting the wall, a valid exchange between players 2 and 1 takes place in one of the following four ways:

1. one = sequence(h_2, front, h_1, front)
2. two = sequence(floor, h_2, front, h_1, front)
3. three = sequence(h_2, front, floor, h_1, front)
4. four = sequence(floor, h_2, front, floor, h_1, front)

A `rally` between two players is a series of one or more exchanges, each of one of the types above:

`rally = rel+ (one v two v three v four)`

A point ends when the sequence of events is not a prefix of rally:

```
(!wall | !prefix(rally)) /+ sequence(s_1, front)
```

A similar expression can be specified for a point beginning with a serve by player 2. The final expression is the disjunction of the two expressions.

One benefit of writing events formally is semantic precision. For example, consider a situation, when the ball hits the front wall twice without touching the floor and without a player hitting the ball. Although such a situation is unlikely, it is conceivable that a "strong" person could serve or hit the ball so that it hits the front wall as described. In our specification such a situation ends the point. (We could not find an answer to this question from our racketball playing colleagues. Also, we do not know who gets the point.)

7.2 AIRPORT WITH TWO RUNWAYS

A general aviation airport, with two runways A and B, is used by many planes. It has limited communication facilities and only visual landings take place. Before takeoff, the pilot of each plane informs the airport at which runway the plane will land. Once in a while, because of wind patterns or some other emergency, the pilot lands a plane at a runway other than the one previously declared.

We will write event expressions based on the following primitive events:

1. EA (*i*) : Plane *i* expected to land at A.
2. EB (*i*) : Plane *i* expected to land at B.
3. LA (*i*) : Plane *i* landed at A.
4. LB (*i*) : Plane *i* landed at B.

Here are some event expressions that specify events of interest:

1. Event AT_A(I) denotes the event "plane I is expected to land at A but has not landed there as yet":

   ```
   AT_A(I) = !happened(LA(I)) /+ EA(I)
   ```

2. Event LAST_B(I) denotes the event "last planned landing of plane I at B":

   ```
   LAST_B(I) =
     ((LB(I) ∧ !before(LA(I))) | first) /+ EB(I)
   ```

   The right operand of /+

   ```
   EB(I)
   ```

   specifies that we look at events after the last declaration of expected landing at B of plane I. The left hand operand

   ```
   LB(I) ∧ !before(LA(I)) | first
   ```

   specifies the landing of plane I at B and that it has not landed at A before that landing. Had such a landing taken place, we would no longer be expecting the plane to land at B. The right operand of the pipe operator ensure that we only look at the first planned landing at B.

3. UB(I) denotes an unexpected landing by plane I at B:

   ```
   UB(I) =
     ((LB(I) ∧ !before(LA(I))) | first) /+ EA(I)
   ```

4. ELAB denotes a landing expected in A that happened at B:

   ```
   ELAB(I) = AT_A(I) ∧ UB(I)
   ```

8. CONCLUSION

We propose a language for specifying composite events in an active database and provide procedures for compiling the language expressions using finite automata (and extensions thereof). The language syntax includes primitive event symbols and event (temporal) operators. Formally, an event expression is a function which is applied to an event history and "produces" another event history. The produced history represents the points in the argument history at which the specified composite event which is specified by the expression occurs. We provide a procedure for constructing a finite automaton corresponding to an event expression. This automaton scans the event history in event order, and enters an accepting state each time the event just scanned should be in the produced output history. When used as a triggering device, the automaton triggers each time it enters an accepting state. Thus complex event combinations can be used to fire triggers in an active database.

We presented a number of extensions. An important extension is that provided by the pipe operator | . Pipes allow the specifier to extract from the real history a hypothetical history on which event detection is more convenient. We introduced "correlation variables", which are used to indicate that distinct sub-expressions are to be satisfied simultaneously. Pipes and correlation variables permit a more convenient specification of event expressions without increasing their expressive power.

Events can have attributes. There are three, in some sense orthogonal, aspects associated with attributes. First, event attributes can be used to constrain the occurrence of events. Second, event attributes can supply parameters to trigger actions when an event does take place. Third, the attributes of constituent events can be "collected" for a composite event, in the form of a relation to which various queries may be applied. Of course, an implementation may provide a unified view of these aspects.

We believe that the language of event expressions presented here provides a sound basis for specifying quite complex, and useful, event combinations of interest in active databases. We have shown how all our constructs can be reduced to automata, thereby providing an explicit prescription for how code may be written to implement composite event detection. While efficient code can be generated for many event expressions, optimizations are possible, particularly when event attributes are present, and this is a topic for further research.

Although our motivation in investigating composite events is triggers in active databases, event specifications can be used in other contexts. For example, event specifications are used for software configuration management and cooperative work [11, 13], they can be used for sophisticated text searching (where the events are the various characters in the text), and they can also be used to examine histories in the context of historical databases. Event expressions can also be incorporated into query languages such as SQL [4] or LDL [1] by using a relation to record the event and the event order [7].

APPENDIX

1. OCCURRENCE TUPLES

If v is an event occurrence in a history, h, then $succ(v)$ denotes the event occurrence immediately following v in h (i.e. $succ(v)$ is the event occurrence w whose event identifier is larger than that of v and there is no event occurrence u whose event identifier is less than that of w and larger than that of v). If v is the last event occurrence in a history then $succ(v)$ is undefined.

Whether an occurrence tuple $t = (origin, f_1, e_1, \ldots, f_k, e_k)$ satisfies an event expression E on history h can be determined inductively as follows. First, $origin = f_1$ indicates where in h we start, second, e_1 indicates the event occurrence where satisfaction occurs relative to *origin*.

1. Consider the sub-expression a of E. It is satisfied by occurrence tuple t iff a is the i^{th} sub-expression of E, $e_i = f_i$ is an event occurrence in h whose primitive event is a and whose event identifier is greater than or equal to the eid of *origin* and less than or equal to the eid of e_1. (Recall that E_1 is E itself).

2. Consider the sub-expression $relative(A,B)$ of E, w.l.o.g. the k^{th} sub-expression of E. Let A be the i^{th} sub-expression of E and let B be the j^{th} sub-expression of E. An occurrence tuple t *satisfies* this expression if:
   1. t satisfies A and B, separately.
   2. $succ(e_i) = f_j$, i.e. B starts after A.
   3. $f_i = f_k$.
   4. $e_j = e_k$.
   5. e_i and e_j are less than or equal to e_1 in t, where e_1 is the primitive event occurrence at which E is recognized. Similarly, f_i is greater than or equal to f_1 in t.

3. Consider the sub-expression $relative+(A)$ of E, w.l.o.g. the k^{th} sub-expression of E. An occurrence tuple t *satisfies* this expression if
   1. f_k is greater than or equal to f_1.
   2. e_k is a tuple of $n \geq 1$ occurrence tuples $t_1, \ldots, t_n$.
   3. Each t_i satisfies A.
   4. The origin-entry of t_1 is equal to f_k in t, the origin entry of t_i, $i > 1$, equals $succ(e_1$ of tuple $t_{i-1})$.
   5. The e_1 entry of t_n has an eid less than or equal to the e_1 entry of t - it is the "effective e_k" event occurrence (where the k^{th} expression ends).

4. Consider the sub-expression $A \wedge B$ of E, w.l.o.g. the k^{th} sub-expression of E. Let A be the i^{th} sub-expression of E and let B be the j^{th} sub-expression of E. An occurrence tuple t *satisfies* this sub-expression if
   1. it separately satisfies both sub-expressions for A and B.
   2. $f_i = f_j = f_k$.
   3. $e_i = e_j = e_k$
   4. e_k is less than or equal to e_1 in t, where e_1 is the primitive event occurrence at which E is recognized. Similarly, f_k is greater than or equal to f_1 in t.

5. Consider the sub-expression $!A$ of E, w.l.o.g. the k^{th} sub-expression of E. An occurrence tuple t *satisfies* this sub-expression if there exists no occurrence tuple t' for the sub-history from f_k till e_k which satisfies the sub-expression A.

Finally, an expression E is satisfied in h at event occurrence o if there is an occurrence tuple satisfying E whose origin-entry is $first[h]$ (the first event occurrence in the history h) and its E entry is e. $E[h]$ is defined to be the set of all event occurrences in h satisfying E.

2. CORRELATION VARIABLES

We describe here how to build an automaton to determine if the current event satisfies an expression E that contains correlation variables. Such an automaton will be used to determine if the current event is in $E[h]$. The difference between this construction and the one for expressions without variables is that the states of the automata we construct may be *marked*. An automaton entering a state marked with x after scanning h means that there is a way for the last event in h to satisfy *all* sub-expressions equated with x. In the construction, M_E will denote the marked automaton for E.

We describe here the automata construction for a restricted case in which no correlation variables appear within negated sub-expressions. This construction is as follows:

1. For an event expression E without variables, the machine is M_E as constructed in Sec. 3: no states are marked.

2. Consider the event expression $E<x,y,\cdots> = w$ and the machine $M_{E<x,y,\cdots>}$, let M' be obtained from $M_{E<x,y,\cdots>}$ as follows. Introduce a new state q, connecting via empty event transitions all accepting states of $M_{E<x,y,\cdots>}$ to q, making q accepting and all states of $M_{E<x,y,\cdots>}$ non-accepting. State q is marked with w. All transitions out of q are into a new non-accepting state p; p self-loops on all transitions.

3. $relative(A<x, y, \cdots>, B<z, w, \cdots>)$. If there exists a variable w appearing in both A and B then the expression can not be satisfied and the resulting machine is that for *NULL* in Sec. 3, no states are marked. Otherwise, use $M_{A<x,y,\cdots>}$ and $M_{B<z,w,\cdots>}$ and construct a non-deterministic machine by connecting the accepting states of $M_{A<x,y,\cdots>}$ to the start state of $M_{B<z,w,\cdots>}$ via ε transitions. The start state is that of $M_{A<x,y,\cdots>}$. The accepting states are those of $M_{B<z,w,\cdots>}$. Marks are preserved from the constituent machines.

4. $A<x, y, \cdots> \wedge B<z, w, \cdots>$. Form a cross product machine as for $A \wedge B$ in the construction for expressions without variables. Marks are determined for each state and each variable as follows. If x is a variable appearing in A (respectively, in B) but not in B

(respectively, not in A), then (p,q) is marked with x in the cross product machine iff p (respectively, q) is marked with x. If x is a variable appearing in both A and B then (p,q) is marked in the cross product machine iff both p and q are marked with x.

5. $\exists\ w\ E<x,y,\cdots,w>$. First construct two copies $M1$ and $M2$ of $M_{E<x,y,\cdots>}$. Make all states of $M1$ non-accepting. For all states q marked with w in $M1$, add a transition on ε from q in $M1$ to q in $M2$, and erase the w mark from each state q of $M1$ and $M2$. Let the start state of the combined machine be that of $M1$.

The intuition behind the construction is as follows. Let x be a correlation variable. The machines we construct are generally non-deterministic. Once a computation thread enters a state marked with x, it cannot in the future enter any state marked with x, this is ensured inductively by the construction. Also, if a computation thread enters a state marked with x, the only way for it to parse the history is that all sub-expressions equated with x are simultaneously satisfied at the event occurrence upon which the state was entered. What remains is to "force" all computation threads to pass through an x marked state. This is achieved by having two machine copies and "jumping" from one copy to the other. This ensures that any accepting computation thread will pass through an x marked state.

Theorem 2: Let E be an event expression which may contain correlation variables subject to the above restrictions. Let machine M_E be constructed as above. Let M_{E^d} be a deterministic automaton accepting the language accepted by M_E. M_{E^d} enters an accepting state on scanning the last event occurrence in h iff the last event occurrence in h satisfies E.

Proof by induction on the construction. The final step in the construction is to determinize the machine M_E to obtain the machine M_{E^d}.

The general case involves the appearance of correlation variables within negated sub-expressions. The difficulty is that to form a machine for $!E$ we need to determinize the machine for E. In doing so, we have states associated with subsets of states in E and the determination of markings is more complex. Due to its length the full construction is not presented here.

However, it is not too hard to extend the above construction to the case where a correlation variable appears solely within a negated sub-expression (or solely within a `rel+` sub-expression). In such a case, construct the machine for the sub-expression as just shown, and determinize it. Observe that there are no markings in the machine at this stage so that nothing special need be done with respect to markings in the determinization. Thereafter, continue the inductive construction of the machine for the full expression, including the negation (or `rel+`), which is now applied to an event expression with no (free) variables, so that step 1 of the construction applies.

3. EVENT MODIFICATION BY ATTRIBUTE SPECIFICATION

We outline an implementation of attribute modified event expressions. We first consider the sub-class of expressions in which there is a single attribute, say I (which is account# in the following example). We start with a single automaton M_c for the whole set of expressions represented by an attribute event expression. M_c is an incomplete machine in that it may scan symbols for which it has no defined transitions. For such symbols, each state is equipped with a transition on such "unknown alphabet symbols". Upon an attribute event occurrence, e.g. $order(I=121,40)$, if M_c has a transition on $order(I)$ then we create a new automaton M_{121} as a copy of M_c. In this automaton, all transitions on $order(I)$ are replaced with $order(121)$. This automaton starts at the state M_c was in and continues on the event $order(121)$. M_c continues as specified by the unknown alphabet symbol transition from its current state. The machine M_{121} now operates independently. In this way as a new account number (I) is introduced, a dedicated machine for it is spawned. Also, from now on an event such as $order(121)$ is interpreted as an unknown alphabet symbol by M_c.

This implementation idea generalizes to expressions with more than one attribute. To illustrate the idea let us consider the case of two attributes, I and J. Basically, if previously each spawned machine was indexed by a single item, e.g. M_{121}, now each machine will be indexed by either a single value or by two values. M_c spawns as in the case of a single attribute. If M_c spawns a machine for $order(J=121,50)$ this machine will be denoted $M_{J=121}$. This machine may later on spawn a machine for $perform(I=33,8)$, it will be denoted as $M_{I=33,J=121}$. The process of spawning $M_{I=33,J=121}$ from $M_{J=121}$ is similar to that of spawning M_{121} from M_c when we treated the single attribute case. So, in general we shall have machine M_c "looking" for new account numbers, machines that have fixed an account number for either I or J and machines that fixed account numbers for both I and J. Once a "parent machine" spawns a machine for an attribute, it regards future events based on this attribute as an unknown alphabet symbol.

Consider again the case of a single attribute. At first glance, it appears wasteful to spawn new machines each time a new account number is encountered. In fact, one need not duplicate M_c. Each machine, for example, M_{121}, can be represented as an array element, say $a[121]$, whose content is the state M_{121} is in. Some additional bookkeeping information may be required, e.g. a list of "activated" machines.

If the domain of potential values for an attribute is large, one can use a hash table implementation of an array. The method generalizes naturally for the case of more than one attribute. The method can be improved upon in various ways and we leave this topic to future publications.

REFERENCES

[1] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli and S. Tsur, "Sets and Negation in a Logic Database Language", *Proc. 6th Symp. Principles of Database Systems*, San Diego, Calif., March 1987, 21-37.

[2] C. Beeri and T. Milo, "A Model for Active Object Oriented Database", *Proc. of the 17th Int'l Conf. on Very Large Databases*, Barcelona, Spain, Sept. 1991, 337-349.

[3] S. Chakravarthy and D. Mishra, "An Event Specification Language (Snoop) for Active Databases and its Detection", University of Florida CIS Tech. Rep.-91-23, September 1991.

[4] C. J. Date, *A Guide to the SQL Standard*, Addison-Wesley, 1988.

[5] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ladin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny and R. Jauhari, "The HiPAC Project: Combining Active Databases and Timing Constraints", *ACM-SIGMOD Record 17*, 1 (March 1988), 51-70.

[6] U. Dayal, M. Hsu and R. Ladin, "A Transaction Model for Long-Running Activities", *Proc. of the 17th Int'l Conf. on Very Large Databases*, Barcelona, Spain, Sept. 1991, 113-122.

[7] D. Gabbay and P. McBrien, "Temporal Logic & Historical Databases", *Proc. 17th Int'l Conf. Very Large Data Bases*, Barcelona, Spain, 1991, 423-430.

[8] N. H. Gehani and H. V. Jagadish, "Ode as an Active Database: Constraints and Triggers", *Proc. 17th Int'l Conf. Very Large Data Bases*, Barcelona, Spain, 1991, 327-336.

[9] N. H. Gehani, H. V. Jagadish and O. Shmueli, "Event Specification in an Active Object-Oriented Database", *Proc. ACM-SIGMOD 1992 Int'l Conf. on Management of Data*, San Diego, CA, 1992.

[10] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming 8*, (1988), 231-274.

[11] B. Krishnamurthy and D. S. Rosenblum, "An Event-Action Model of Computer-Supported Cooperative Work: Design and Implementation", *Proceedings of the International Workshop on Computer Supported Cooperative Work*, April 1991, 132--145.

[12] G. M. Lohman, B. Lindsay, H. Pirahesh and K. B. Schiefer, "Extensions to Starburst: Objects, Types, Functions, and Rules", *Comm. ACM 34*, 10 (October 1991), 94-109.

[13] D. S. Rosenblum and B. Krishnamurthy, "An Event-Based Model of Software Configuration Management", *Proceedings of the 3rd International Workshop on Software Configuration Management*, June 1991, 94--97.

[14] U. Schreier, H. Pirahesh, R. Agrawal and C. Mohan, "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS", *Proc. of the 17th Int'l Conf. on Very Large Databases*, Barcelona, Spain, Sept. 1991, 469-478.

[15] A. Silberschatz, M. Stonebraker and J. Ullman, "Database Systems: Achievements and Opportunities", *Comm. ACM 34*, 10 (October 1991), 110-120.

[16] M. Stonebraker and G. Kemnitz, "The POSTGRES Next-Generation Database Management System", *Comm. ACM 34*, 10 (October 1991), 78-93.