Event Specification in an Active Object-Oriented Database

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

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The concept of a trigger is central to any active database. Upon the occurrence of a trigger event, the trigger is "fired", i.e, the trigger action is executed. We describe a model and a language for specifying basic and composite trigger events in the context of an object-oriented database. The specified events can be detected efficiently using finite automata.

We integrate our model with O++, the database programming language for the Ode object database being developed at AT&T Bell Labs. We propose a new Event-Action model, which folds into the event specification the condition part of the well-known Event-Condition-Action model and avoids the multiple coupling modes between the event, condition, and action trigger components.

1. INTRODUCTION

Of late, there has been a surge of interest in active databases [2, 6, 6, 15, 18, 20]. Several trigger and constraint specification mechanisms have been proposed, and the use of such mechanisms for various applications has been considered. By and large, the fundamental model underlying this work is the Event-Condition-Action (E-C-A) model first enunciated in [16].

The question of what constitutes an event has not been fully addressed in the literature so far, in our opinion. There are however some considerations of this issue, specifically in [7] and [5]. In this paper, we focus on the different types of events that can occur in an object-oriented database and provide facilities for specifying *composite* events, constructed from (simpler) basic events.

We describe the integration of composite event specification in the context of O++, the database programming language for the Ode object database [1]. We propose a new Event-Action (E-A) model, which is simpler than the E-C-A model. Not only have we eliminated the need for a separate "condition" part in the model — the condition is part of an event specification in our model — we have also eliminated the need to have special types of couplings as proposed in the E-C-A model. In fact, our event specification facility can model not just the E-C-A couplings discussed in the literature, but arbitrary additional types of couplings as well. We construct a formal model and a language for the specification (and detection) of composite events. The language is equivalent, in terms of expressive power, to regular expressions over strings of logical events. The language differs from regular expressions in that it focuses on sequences and (not necessarily consecutive) sub-sequences, rather than on strings and sub-strings. On this basis we are able to compile arbitrary composite event specifications into finite automata, thereby rendering event detection particularly efficient. In many cases, one automaton per class is sufficient and objects need only record their states within this automaton.

The paper is organized as follows. First, in Section 2, we give a summary of Ode. Then in Section 3 we describe events: basic and composite events, and how they can be used in O++ triggers. Our model for event specification is presented in Section 4. In Section 5 we discuss how to implement events. Section 6 addresses basic events and their relationship to transactions while Section 7 contains a discussion of the relationship between the triggered actions and the triggering transactions. Related work is discussed in Section 8. Finally, future work and conclusions are outlined in Section 9.

2. ODE SUMMARY

The work in this paper builds on the trigger facilities in Ode, an object-oriented system being developed at AT&T Bell Labs [1, 8]. The O++ object facility is based on the C++ object facility and is called the *class*. O++ extends C++ by providing facilities to create persistent objects. O++ visualizes memory as consisting of two parts: volatile and persistent. *Volatile* objects are allocated in volatile memory and are the same as those created in ordinary programs. *Persistent* objects are allocated in persistent memory and they continue to exist after the program creating them has terminated. Each persistent object is identified by a *unique* identifier, called the object identity [14]. The object identity is referred to as a *pointer to a persistent object.*

Transactions in O++ have the form

```
trans {  · · ·  }
```

Transactions are aborted using the `tabort` statement.

O++ provides facilities for associating *constraints* and *triggers* with an object. Triggers are specified in the trigger section of a class definition:

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0081...$1.50

```
class name {
      . . .
trigger:
     trigger-list
      . . .
};
```

trigger-list is a list of triggers each of which is specified as[1]

trigger-name (*parameters*) :
    [perpetual] *event* ==> *trigger-action*

The *trigger-action* is any arbitrary O++ statement block to be executed when the trigger is fired. What an *event* is we describe below.

Triggers do not fire unless they are *active*. A trigger is activated by invoking its name, along with parameter values, just as an ordinary member function is invoked. An ordinary trigger is automatically deactivated the moment it fires. On the other hand, a `perpetual` trigger, once activated, remains active forever unless explicitly deactivated.

3. EVENTS

An "event" is a happening of interest. Events happen instantaneously at specific points in time. In object-oriented databases, events are related to actions that happen to objects and the state of the object.

Events have a "scope." In an object-oriented system, most events are local to a particular object. In some cases it may be appropriate to define events over other scopes, such as the database. An example of an event that applies to the database is the creation of object type, i.e., schema modification.

3.1 BASIC EVENTS

Each event specification system must start with an alphabet of *basic* events that the system supports. While this set of basic events could be arbitrary in general, there are some basic events that we consider to be important in the context of an object-oriented database system such as Ode. We list these events below:

1. *Object State Events*:
   a. Immediately after an object is created.
   b. Immediately before an object is deleted.
   c. Immediately before or after an object is updated/read/accessed through a public member function[2].

2. *Method Execution Events*:
   Immediately before or after the specified member function is applied to an object. (Items 1a and 1b become special cases if constructors and destructors have been specified explicitly.)

3. *Time Events*: Time events are specified in O++ as

   at *time-specification*
   every *time-period*
   after *time-period*

   Time can be specified in the format

   time (YR=*year*, MON=*month*, DAY=*day*, HR=*hour*,
        M=*minute*, SEC=*seconds*, MS=*milliseconds*)

   with any of these items possibly being omitted.

   Time events are really global, but are considered events of interest and posted only to the "relevant" objects.

4. *Transaction Events*:
   a. Immediately after a transaction begins.
   b. Immediately before a transaction attempts to commit.
   c. Immediately after a transaction commits.
   d. Immediately before a transaction aborts.
   e. Immediately after a transaction aborts.

   Like time events, transaction events are also really global. However, they are considered local events of interest to exactly the set of objects accessed by the transaction. We will typically not know *a priori* the set of objects that will be accessed by a transaction. So the "after transaction begin" event

   ```
   after tbegin
   ```

   is "posted" to an object only immediately before the object is first accessed by the transaction. Logically, there is no difficulty, since there is no way to control the times at which the "same"

   ```
   after tbegin
   ```

   event is posted at two different objects.

The following keywords, in conjunction with the qualifiers `before` and `after`, are used to specify some basic events in O++:

- `create` (object creation),
- `delete` (object deletion),
- `update` (object update),
- `read` (object read),
- `access` (object access),
- `tbegin` (transaction begin),
- `tcomplete` (execution of the transaction code is complete but the transaction has not as yet attempted to commit),
- `tcommit` (transaction commit), and
- `tabort` (transaction abort).

An example illustrating the use of the above keywords and event qualifiers is

```
after read
```

which specifies an event that occurs immediately after the execution of a public member function that accesses an object for reading only. Similarly,

1. The square brackets [and] are used enclose an optional item; { and } are used to enclose an item that may be repeated zero aor more times.

   Note that O++ originally had a special construct for a timed trigger. In the light of the work described in this paper, it will be clear that timed triggers can be simulated using composite events.

2. We chose not to consider arbitrary object accesses without the use of member functions as events. Such access is not in the spirit of object-oriented programming, which requires the use of methods to preserve encapsulation. Also, implementing each such access as an event might cause each pointer dereference to have some event overhead associated with it. This overhead, however small, would most likely not be acceptable at the granularity of an individual pointer dereference.

`before tcomplete`

specifies an event that occurs just before a transaction attempts to commit after having accessed the object. Note that the specification of the event

`before tcommit`

is not allowed because we cannot be sure that a transaction is going to commit until it actually does so. An event can be scheduled to occur after a specified period (from the current time, when the trigger is armed) has elapsed as follows:

`after time(HR=2, M=30)`

Member function names with parameter declarations can be used to specify events specification as illustrated below:

`after withdraw(Item i, int q)`

Formal parameter declarations help distinguish between different member functions with the same name but with different signatures, i.e., between overloaded functions with the same name and belonging to the same class. These parameters can also be used for defining predicates (called "masks") that can be associated with the basic events (we discuss this below). If `withdraw` is not overloaded and the parameters are not of interest, then we can simply write

`after withdraw`

3.2 LOGICAL EVENTS

Every basic event is a *logical* event. In addition, a basic event qualified with a *mask* is a logical event. A mask is a predicate that is used to hide or "mask" the occurrence of an event. For example, the event specification

`after withdraw(Item, int q) && q>1000`

describes a logical event representing a "large" withdrawal (of more than 1000 units) of an item. The mask here is

`q>1000`

As demonstrated in the example above, predicates associated with a logical event may use the parameters of the basic event being masked. In addition, of course, they may access the state of any object in the database, and this state information is evaluated as of the time at which the basic event occurred. Therefore, when multiple logical events are used to express a composite event (see next section), the predicates associated with individual logical events will be evaluated at different times, corresponding to the occurrences of their respective basic events.

3.3 COMPOSITE EVENTS

Logical events can be combined to create *composite* events using logical operators and special event specification operators. All events occur instantaneously at specific points in time. This is obvious in case of basic and logical events. In case of a composite event, the event is said to occur at the point of occurrence of the last logical event that was needed to make it happen.

An optional *mask* predicate can be applied to a composite event to obtain a *logical-composite event.* This is our general notion of an event. A composite event has no parameters even if its constituent basic events do. Any mask predicate applied to a composite event, unlike the mask predicates of logical events, can only be evaluated in terms of the "current" state of the database. The following BNF summarizes our event composition mechanisms:

logical-composite-event = *composite-event* [`&&` *mask*]

composite-event = *logical-event*
| (*composite-event*)
| *composite-event* `&` *composite-event*
| *composite-event* `|` *composite-event*
| `!`*composite-event*
| `relative` (*composite-event-list*)
| `relative+` (*composite-event*)
| `relative` *const-integer-expression* (*composite-event*)
| `prior` (*composite-event-list*)
| `prior` *const-integer-expression* (*composite-event*)
| *composite-event*`;` *composite-event*
| `sequence` (*composite-event-list*)
| `sequence` *const-integer-expression* (*composite-event*)
| `choose` *const-integer-expression* (*composite-event*)
| `every` *const-integer-expression* (*composite-event*)
| `fa` (*composite-event*, *composite-event*, *composite-event*)
| `faAbs` (*composite-event*, *composite-event*, *composite-event*)

composite-event-list=*composite-event* { , *composite-event* }

logical-event=*basic-event* [`&&` *mask*]

When we simply say *event*, we always mean a logical composite event. *mask* is a Boolean-valued expression, *const-integer-expression* is an expression that can be evaluated at compile time, The symbol `|` denotes union, the symbol `&` denotes intersection, and `&&` denotes logical conjunction of O++ masks. *basic-event* is an event listed in Section 3.1. A method name, say *f*, can be used as a shorthand to denote the composite event

`(before` *f* `| after` *f*`)`

For example, the expression

`!deposit`

is a shorthand for

`!(before deposit | after deposit)`

The keywords `relative`, `prior`, `sequence`, `choose`, `every`, `fa`, and `faAbs` refer to event composition operators, which are discussed later.

A particularly important composite event is when an object reaches a specified state, described as

`(after update | after create) &&`
Boolean-expression-specifying-object-state

Since we expect this form of event to be used often, we allow it to be specified simply as

Boolean-expression-specifying-object-state

Indeed, this is the only sort of event allowed in Ode prior to the work described in this paper. Here is an example of an event that occurs when the `balance` (say of a bank customer's account) falls below 500 dollars:

```
balance < 500.00
```

3.4 EVENT HISTORY & EVENT COMPOSITION OPERATORS

An *event history* (or simply a history) is associated with every object; it is an ordered set of logical events that were posted to the object[3]. Each logical event "carries" its position in the ordered set. We can use set operations, e.g., union, on histories with the understanding that logical events with distinct history positions are distinct elements. Thus the resulting sets are also ordered.

One or more logical events determine a composite event. A composite event is associated in this history with the *last* logical event required to make it happen. This is a point in the history when we can recognize that this composite event has occurred. We say that this is the point in history at which a composite event occurs. There is also a unique "first" logical event, called `start`, that participates in the determination of a composite event; this event is placed at the beginning of the history just prior to the first user specified logical event that is "posted."

An important event composition operator is sequencing for specifying the order in which constituent events occur. For example, a simple sequencing operator that we would like to provide is one that specifies a composite event in which one component event occurs after another. With logical events (as opposed to composite events) this is straightforward. For example, one can use the notation

```
E1 .* E2
```

to specify a composite event that consists of the logical event `E1` followed, later on at some point, by the logical event `E2`.

However it is not easy to use this notation when specifying composite events that are composed of composite events themselves. For example, suppose that the composite events `E` and `F` are defined as

```
E = E1 .* E2
```

and

```
F = F1 .* F2
```

where `E1`, `E2`, `F1`, and `F2` are logical events.

Now suppose that we specify a composite event `G` that consists of the event sequence "`E` followed by `F`" as

```
G = E .* F
```

What do we mean? Clearly, `G` has occurred at the `F2` point if the sequence of logical events

```
E1 E2 F1 F2
```

has occurred. But what if the sequence of logical events that occurs is

```
F1 E1 E2 F2
```

3. We assume for now that logical events are disjoint, i.e., cannot happen at the same time, so that a sequence is well-defined. We show in Section 5 how to get around this restriction.

`E` occurs coincident with `E2` and `F` occurs coincident with `F2`, these being the last logical events required to determine the respective composite events. So `E` occurs before `F`. We could argue that `G` has once again occurred at point `F2`. But note that `F1`, part of the composite event `F`, occurs before `E`.

We provide two distinct sequencing operators to resolve this issue. One operator is called `prior` and the other is called `relative`. The two operators have identical semantics when applied to logical events. When applied to composite events, `prior(E,F)` holds if `E` occurs before `F` (that is, if the last logical event of `E` occurs before the last logical event of `F`). The order in which the other events occur is immaterial. On the other hand, `relative(E,F)` requires that the last logical event of `E` occur prior to the first logical event of `F`. Thus the event `prior(E,F)` occurs at `F2` when the event history is

```
F1 E1 E2 F2
```

but the event `relative(E,F)` does not occur.

Conceptually, a composite event is recognized in the context of a history. The history has a "starting point." The `relative` operator shifts this starting point to a new place. The event of interest must occur in this truncated history. If the event of interest is composite, then all its constituent logical events must occur in the truncated history.

Our sequencing operators can accept an arbitrary number of arguments, instead of just two. In such cases, the specified operator is applied recursively to the arguments, using a form of "currying". For instance "`prior(E,F,G)`" is a shorthand for "`prior(prior(E,F),G)`". For completeness, we define "`prior(E)`", "`relative(E)`", etc., to mean simply "`E`".

The composite event `sequence(`$E_1, \ldots, E_n$`)` specifies that the component event E_k occurs immediately (at the next logical event) following the component event E_{k-1} event ($2 \leq k \leq n$). For example,

```
sequence(after tbegin, before access,
      after access, before tcomplete)
```

specifies a transaction attempting to commit after accessing an object, and causing no other events to be posted to the object.

Semicolons can be used instead of the `sequence` operator to specify a sequence of events. For example, the above composite event can be specified alternatively as

```
after tbegin; before access;
  after access; before tcomplete
```

The modifier + can be applied to any of the above operators to indicate repeated application. For instance, "`relative+(E)`" means the infinite disjunction:

```
relative(E) | relative(E, E) |
         relative(E, E, E) | . . .
```

The events `prior+(E)` and `sequence+(E)` are both equivalent to the event `E`. We will explain this for the operator `prior`. `E` is the same as `prior(E)`; `E` must have occurred at the point at which `prior(E,`

E) holds; and so on, the additional disjuncts being specializations of E. Consequently, modifier + is not provided for the operators prior and sequence.[4]

We can specify limited repetition, instead of unlimited repetition, by using an integer constant (literal) as the first argument of the above operators. For example,

```
relative 5 (after deposit)
```

specifies the composite event that consists of the fifth and any subsequent "after deposit" events.

Operator choose is used for specifying which occurrence of an event is to be selected. For example, the event specified by

```
choose 5 (after tcommit)
```

is posted by the commit of the fifth transaction.

The every operator is used for specifying events that occur periodically. For example, the event specified by

```
every 5 (after tcommit)
```

is posted by the commit of the 5th transaction, the 10th transaction, the 15th transaction, and so on.

We will now introduce two operators we find useful: fa and faAbs. Operator fa(E, F, G) is defined as the first occurrence of event F (at some logical event *p*) relative to an event E, with no intervening event G relative to E taking place prior to the occurrence of the logical event *p*. For example, event

```
fa(after tbegin,
   prior(after update, after tcommit),
   (after tcommit | after tabort))
```

specifies the commit of a transaction that updated an object, since there are no intervening aborts or commits after the tbegin.

Operator faAbs(E, F, G) is defined as the first occurrence of event F (at some logical event *p*) relative to an event E, with no intervening event G relative to the whole history taking place prior to logical event *p*. The difference between fa and faAbs is that G is defined relative to E in the first, and relative to the beginning of history in the latter.

3.5 EXAMPLES

Consider an object of type stockRoom, which tracks multiple items. Type stockroom is defined as

4. In general event E may not occur at the point relative(E,E). For instance, let E be the event

```
F & !prior(F,F)
```

Given the sequence of events

```
F F
```

event E occurs at the first F but not at the second. However, relative(E,E) occurs at the second F but not the first.

```
...
class stockRoom {
    ...
    Item items[max];
    int n;
public:
    stockRoom();
    ...
    void deposit(Item i, int q);
    void withdraw(Item i, int q);

    int authorized(UserId);
    void log();
    void order(Item i);
    void printLog();
    int reorder(Item i);
    void report();
    void summary();
    UserId user();
    void updateAverages();
};
```

To illustrate event specification, we use this type definition as a basis. Here is the modified version of class stockroom with triggers specified (member functions are not repeated).

```
...
#define dayBegin   at time(HR=9)
#define dayEnd     at time(HR=17)
#define 5thLrgW  choose 5\
      (after withdraw(i, q)&&q>100)
class stockRoom {
    ...
    Item items[max];
    int n;
public:
    stockRoom();
    ...
trigger:
 T1():perpetual before withdraw &&
        !authorized(user())==>tabort;
 T2():after withdraw(i,q)
        && i.balance<reorder(i)==>
                             order(i);
 T3():perpetual dayEnd==>summary();
 T4():perpetual relative(dayBegin,
       prior(choose 5(after tcommit),
              after tcommit) &
      !prior(dayBegin, after tcommit)
         ) ==> report();
 T5():perpetual every 5(after access)
          ==>updateAverages();
 T6():perpetual after withdraw(i,q)&&
          q > 100 ==> log();
 T7():perpetual fa(dayBegin, 5thLrgW,
          dayBegin) ==> summary();
 T8():perpetual after deposit;
       before withdraw;after withdraw
          ==> printLog()
};
```

Note the use of the C++ #define statement to specify convenient abbreviations. The triggers specified are:

1. Only authorized users can withdraw an item. Otherwise, the transaction is to be aborted.
2. If the item quantity in the stock room falls below a certain amount, say the economic order quantity

for the item, then an order is to be placed to buy more of the item. This trigger must be explicitly reactivated after it has fired.
3. At the end of the day, a summary is to be printed.
4. Every transaction after the 5th transaction within the same day is to be explicitly reported.
5. After every 5 operations, the averages are to be updated.
6. All large withdrawals (quantity > 100) are to be recorded.
7. After the 5th large withdrawal of an item in the same day, print a summary.
8. Print the log when a deposit is immediately followed by a withdrawal.

All the triggers, except the second, are `perpetual` triggers; so that they are not deactivated after they fire.

Triggers must be activated explicitly. Once the trigger event is satisfied, the trigger "fires", that is, the trigger action is scheduled for execution. The occurrence of a single logical event may cause multiple triggers to fire. The order in which these actions are executed is not specified.

The initial activation can be specified in the constructors of class `stockRoom` which are automatically invoked when an object of the associated type is created. For example, here is the body of the class `stockItem` constructor:

```
stockRoom::stockRoom()
{
    ...
    //activate the triggers
    T1(); T2(); T3(); T4(); T5();
    T6(); T7(); T8();
}
```

As another example (from process control), we will specify the composite event consisting of a pressure drop (pressure falls below the specified low limit) followed by a valve open which is the composite event consisting of the completion of the method `motorStart` followed by the completion of the method `motorStop`:

```
#define pDrop (pressure<low_limit)
#define valveOpen relative(\
    after motorStart,after motorStop)
class vessel {
    ...
    float low_limit;
public:
    ...
    float pressure;
    motorStart();
    motorStop();
trigger:
    T(): relative(pDrop,valveOpen)
          ==> check pressure;
};
```

4. MODEL

For any given event specification, there are likely to be multiple points in an event history at which the specified event occurs. When dealing with logical events, this is not an issue since such an event occurs at and depends upon a single point in the history. With composite event specification, there could in general be multiple prior occurrences of events upon which it may depend. For instance, returning to the first example in the event history sub-section, while the event

`relative(E, F)`

does not occur at `F2` for the event history sequence

`F1 E1 E2 F2`

it is still possible for this event to occur if the event `E` had also occurred at some earlier point in the history. In fact, if there are multiple prior occurrences of `E` in the history, there are multiple composite events `relative(E, F)` that occur at `F2`, each with its own first logical event. However, the system only takes cognizance of the occurrence of this event once, just as a disjunctive Boolean predicate is considered no more true if multiple disjuncts are true than if just one is true.

While we have introduced a rich set of event composition operators into O++ for ease of expression, not all of them are necessary in terms of expressive power. For example, the curried operators are not necessary.

We present now a formal definition of a "core" event specification language for specifying the event. Other O++ event specification facilities can be derived from the operators present here. This claim can be proved either by simple re-writing, or by relying on an explicit finite automata construction and the equivalence between event expressions and regular expressions [10].

Recall that an event history H is an ordered set of logical events (points). Composite events are specified as event expressions which are evaluated in the context of a history. An event expression E evaluated in the context of a history H, denoted as $E[H]$, specifies a subset (sub-sequence) of H. An event expressions is one of

1. ϕ; denotes the empty set of logical events.
2. a, where a is a logical event; denotes the set of all points in H, that are the a logical event.
3. `relative`(F_1, F_2), where F_1 and F_2 are event expressions and `relative` is defined as follows:

   Let G = $\{H' \mid H'$ is a suffix of H got by deleting some F_1 in H and all logical events prior to it $\}$

   Then `relative`(F_1, F_2) is equal to $\bigcup_{H' \in G} F_2[H']$, i.e., the union of the sets of points specified by the event expression F_2 in the context of the histories in G
4. F_1 `&&` F_2 is the intersection of the sets of event points specified by F_1 and by $F2$;
5. $!F_1$ is the complement, with respect to the set of all points in H, of the set of points specified by F_1.
6. `relative+`(F) specifies the set of points

$\{p \mid$ there exists a sequence of history points, in ascending order, $h_1, h_2, \ldots, h_k = p$, $k \geq 1$ such that h_1 is in $F[H]$ and h_{i+1}, $1 \leq i < k$ is in F evaluated in the context of the history obtained from H by deleting all logical events up to and including h_i.

Take the history associated with some object in the database at a particular point in time. The empty event set "labels" no points. A logical event *a* "labels" all points in the history where *a* occurred. Operators (items 3-6) are used to manipulate such sets of "labeled points".

Consider an evolving history at a particular point in time and an event expression *E*. Evaluate the expression *E* according to the semantics assigned to operators above. Eventually, a final set of points in the history is labeled. If the rightmost history symbol is labeled, then the specified event has just occurred.

An event specification can also be thought of as prescribing a set of sequences of logical events (i.e. a set of strings over the alphabet of logical events). The event has just occurred (at some object) *iff* the current history (considered as a sequence) is in this set of sequences. Identifying interesting sequences of symbols from some alphabet is exactly what grammars are used for. So the natural question is how does the expressive power of our operators fit into the expressiveness hierarchy of string grammars? We selected the operators carefully so that the expressive power is exactly the same as that of *regular* grammars. A proof is provided in [10].

5. IMPLEMENTATION

Since composite events can alternatively be expressed as regular expressions, their occurrence can be detected using finite automata. An automaton can be defined for each event, which reaches an accepting state exactly whenever the event occurs. The input to the automaton is the sequence of logical events constituting the event history for the object with which the automaton is associated.

Since the set of possible logical events constitutes the alphabet of input symbols to the automaton, we require that the logical events used in a particular trigger definition all be disjoint so that no two logical events occur simultaneously (this is because a history is a sequence, and simultaneous logical events make it difficult to decide the order of sequence events). We ensure that the masks for the basic events are disjoint. If the masks are not disjoint, their Boolean combinations must be disjoint, and we define new logical events using these Boolean combinations. For instance, suppose a trigger has the event

```
sequence(before log && a > 0,
              before log && b > 0)
```

Suppose we do not know that `a > 0` and `b > 0` are disjoint. The composite event can be stated in the following logically equivalent form

```
sequence(before log && a>0 && b>0 |
          before log && a>0 && !(b>0),
         before log && a>0 && b>0 |
          before log && !(a>0) && b>0)
```

Each disjunct in this restatement is a disjoint logical event. While it is true that the sort of rewriting we require could cause a combinatorial explosion, in practice we do not expect to see enough such overlap for this explosion to be a worry. Moreover, it is straightforward to perform such an expansion in the compiler.

In an object-oriented system, all objects in a class must define the same set of events and therefore can use *identical automata*. For each trigger definition, the transition table of the trigger automaton is kept once (for the class), and for each object, for which the trigger has been activated, the state of the trigger automaton may be stored with the object. Only a single (integer) variable is required for storing the state. Thus the extra storage required for storing the trigger state is small — one word per active trigger per object.[5]

Events are monitored as follows. Whenever a basic event (with any associated parameters) is posted to an object, we check the active triggers to determine whether or not any logical events have occurred. If so, for each active trigger for which a logical event has occurred, we move the automaton to the next state. We determine all the trigger events that have occurred, and then we fire the triggers. If the posting of a logical event leads to the firing of multiple triggers, then the order in which the triggers are fired is implementation dependent.

The action associated with the trigger is executed as part of the same transaction as the one that detected the event, and is executed immediately. Two basic events, "`after tabort`" and "`after tcommit`", require special handling. By definition, the transaction responsible for these events has completed just prior to these events occurring. Consequently, the events must be posted by a special "system" transaction, and if a trigger fires, the action part is executed as part of this "system" transaction.

6. DEALING WITH TRANSACTIONS

An important property of transactions is *atomicity*, i.e., either the transaction commits and all its effects are reflected in the database or it aborted and none of its effects are in the database. Traditionally, anything a transaction could do was modeled by its effects on shared data. The question is then, are logical events that correspond to the activity of an aborted transaction to be viewed as part of the event history?

One can find justifications for answering this question both positively and negatively. The "true" history contains operations on behalf of aborted transactions which may be useful in dynamically modifying system parameters: for example, "if the ratio of aborts to commits exceeds *q* then reduce the number of concurrent transactions allowed". On the other hand, a case may be made for viewing the history as comprised of actions on behalf of only committed transactions.

5. The above description assumes one automaton definition per trigger. In many cases such automata may be combined into one, resulting in a more efficient monitoring; we regard this here as merely one of many possible optimizations.

For example, at the object level one would like to see the trail of "real" actions and not ones that were "undone".

It seems that one should be able to state composite events from either of these two viewpoints and that both options should be available. To implement the committed version, one can consider an automata implementation of event monitoring in which the automaton state is considered part of the object data structure and hence will be restored correctly upon abort. To implement the complete history (including the effects of aborted transactions) we use an automaton whose state is not part of the object and hence not restored upon abort.

Assuming object level locking, we have the following:

Claim: Any event expression E made with respect to operations of only committed transactions, with an object scope, can be converted into an event expression with respect to the whole history, including the operations of aborted transactions.

Proof: Consider such an expression E defined with respect to logical events only on behalf of committed transactions. Let A be a finite automaton for detecting E events. Convert A into A' that will view the whole history as follows. Each state of A' is a pair (a, b) where both a and b are states of A. Intuitively, a is the state A is really in and b is the state A was in before seeing the most recent "`after tbegin`" basic event. The automaton A' in state (q, p), upon seeing the "`after commit`" basic event, moves to state (r, r), where r is the state A would have moved to had it been in state q. Upon encountering the basic event "`after tabort`", A' in state (q, p) moves to state (p, p). The transitions out of a state (q, p) on all other logical events are to states (r, p) where r is the state A would have moved to had it been in state q. Since A' can be converted into an equivalent regular expression and that expression into an equivalent event expression, the proof is complete. □

In fact, the inclusion of transaction events as basic events greatly adds to the expressive power of our event expressions, as we demonstrate in the next section. One point that requires clarification here is event `before tcomplete`. This event occurs when a transaction "thinks" it has completed execution successfully, and is ready to commit. The occurrence of this event could cause some triggers to fire, requiring some additional work on the part of the transaction. When all this work is done, another `before tcomplete` event occurs. This process goes on until no triggers fire in response to a `before tcomplete` event. At this point, the transaction may actually perform its commit. Thus the `before tcomplete` event can be posted to an object multiple times within a transaction. Also, its occurrence is no guarantee that the transaction will commit.

7. RELATIONSHIP BETWEEN TRIGGERED ACTIONS & TRIGGERING TRANSACTIONS

The most quoted model for describing trigger operation is the E-C-A model [6, 16]. There are four common types of couplings possible between each of E (first part) - C (second part) and C (first part) - A (second part). These are

1. *immediate*: second part is executed immediately after the first part (in the same transaction).
2. *deferred*: second part is executed just prior to commit of the transaction executing the first part.
3. *separate dependent*: second part is executed as a separate transaction, with commit dependency,[6] after the commitment of the transaction executing the first part.
4. *separate independent*: second part is executed as a separate transaction, with no dependency, after the triggering (first part) transaction has committed or aborted.

Thus there are 16 (4 × 4) possible combinations of couplings, each of which the user has to be able to express and which the system has to implement. Even worse, the types of couplings considered in the model may not constitute an exhaustive list. For instance, the original E-C-A work only discussed three types of couplings, with the last two types folded into a single type *decoupled*. The need to specify the presence or absence of commit dependency is evident only in more recent work.

Given our powerful event specification facilities, it is not necessary to define such a list of couplings. Any coupling desired can be implemented by selecting an appropriate event specification, incorporating the required transaction events. Furthermore, the condition evaluation can naturally be folded into the event specification, resulting in an E-A model with only one kind of coupling.

Let E be a (composite) event expression, C be a condition (predicate) to be evaluated when e occurs, and A be the action to be executed if the condition evaluates to true (the trigger fires). Here are different coupling possibilities expressed as O++ trigger events:[7]

1. Immediate-Immediate:

   E&&C ==> A

2. Immediate-Deferred:

   ```
   fa(E&&C, before tcomplete,
                after tbegin) ==> A
   ```

3. Immediate-Dependent:

   ```
   fa(E&&C, after tcommit,
                after tbegin) ==> A
   ```

4. Immediate-Independent:

   ```
   fa(E&&C,after tcommit|after tabort
                after tbegin) ==> A
   ```

5. Deferred-Immediate or Deferred-Deferred:

6. If transaction t_2 is commit dependent on t_1, then t_2 is not allowed to commit until t_1 has. By extension, if t_1 eventually aborts, so must t_2 since it can never hope to obtain the permission to commit.

7. The terms used for the various coupling modes were defined at the beginning of this section.

```
fa(E,before tcomplete,after tbegin)
                       &&C ==> A
```

6. Deferred-Dependent:

```
fa(fa(E, before tcomplete,
       after tbegin) && C,
   after tcommit,after tbegin)==>A
```

7. Deferred-Independent:

```
fa(fa(E, before tcomplete,
       after tbegin) && C,
   after tcommit | after tabort,
   after tbegin) ==> A
```

8. Dependent-Immediate:

```
fa(E, after tcommit, after tbegin)
                  && C  ==> A
```

9. Independent-Immediate:

```
fa(E, after tcommit | after tabort,
   after tbegin) && C ==> A
```

Once the (side-effect-free) condition evaluation is in a separate transaction, there does not appear to be much point in differentiating between execution of the action immediately after condition evaluation, at the end of the transaction, or even in a new transaction. As such, these alternatives have not been addressed above. From the spirit of the expressions written above, it should be clear to the reader that expression of such semantics is possible, if desired.

8. RELATED WORK

Behavior specification using regular expressions and finite state machines is by no means a new idea. In the context of active databases they appear in [7, 19]. Other well-known examples are path expressions and their derivatives [3, 4, 12], which are used to specify process synchronization in concurrent programs. Path expressions are associated with objects. An object method can be executed only if it "satisfies" the path expressions associated with the object. Otherwise, execution of the operation (the process) is delayed until the path expressions can be satisfied.

Another well known example of the use of behavior specification using finite automata are statecharts [11]. Statecharts are a mechanism for specifying the input and output of a system interacting with the environment, such a system is called *reactive*. Statecharts specify a finite state (Mealy) machine in an "economical" graphical way, by taking advantage of hierarchical relationships in terms of operations among sets of states, and avoiding combinatorial explosion in the number of states by independently specifying "orthogonal" components (cross-product operands). In addition, the system can generate events (basic symbols in statecharts terminology) which can cause it to change states. Other mechanisms provided include a history, correlating states in orthogonal components, default and other entry points. Statecharts thus *specify* how a deterministic system should behave and how it should react to the environmental inputs.

SQL also includes triggers, these are applied to specific tables (relations), they trigger on update/insert/delete to the table, a search condition (mask) may also be specified (it is an SQL search condition), and the actions are also update/insert/delete statements [17]. Using the concept of "delta", which are uncommitted updates to relations, Hull and Jacobs [13] are capable of capturing the semantics of trigger applications based on changes in relations.

Composite events were proposed for the first time in [7], but few specifics were provided. The only detailed work dealing with composite events is [5]. Here a syntax and semantics for composite event specification is proposed, along with implementation mechanisms. However, the set of event operators selected seems arbitrary, and is not integrated with any existing database programming language.

9. CONCLUSION & FUTURE WORK

In this paper, we have described how to specify composite trigger events. The event specification is based on a set notation identical in expressive power to a notation based on regular expressions. These expressions can be translated into finite state automata in an efficient way.

An important motivation for designing a system based on a formal model is that the model clarifies the underlying concepts and eventually leads to a simpler design. In our case, trying to formalize the composite event specification has led us to the simpler E-A model. Our model folds the condition part of the well-known E-C-A model into event specification. In additions, the model avoids the multiple coupling modes between the event, condition, and action trigger components. The E-A model is easier to explain and has simpler semantics than the E-C-A model.

Finally, we explained how triggered actions are scheduled for execution and argued that the lack of attachment mode for condition checking is not a serious deficiency.

A list of subjects for further research includes:

- Understanding the utility of event expressions and triggers to specify and construct reactive systems.
- Further enhancements, both in terms of ease of use and in terms of expressive power, to event expressions.
- Techniques for event expression compilation and implementation of event monitoring.
- Incorporation of transaction and object identity into logical events. If we monitor events within a single object scope, the issue of object identity recording is mute. This is not the case at the system level where a large number of objects need be tracked.
- The incorporation of arguments into composite event specification. Some events carry values with them which may be of use later on. The issue is how to efficiently collect and record these values and how to use them at future events.
- Explicit manipulation of event histories to specify events. The idea is to define "history expressions" and to integrate them with event expressions.

Progress on these subjects is reported in [9].

ACKNOWLEDGMENT

We are grateful to Shaul Dar and Bruce Hillyer for their comments. We thank the referees for their comments and suggestions.

REFERENCES

[1] R. Agrawal and N. H. Gehani, "Ode (Object Database and Environment): The Language and the Data Model", *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989, 36-45.

[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] R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions", in *Lecture Notes in Computer Science*, vol. 16, Springer-Verlag, 1974.

[4] R. H. Campbell and A. N. Habermann, "Path Expressions in Pascal", *Proceedings of the Fourth International Conference on Software Engineering*, 1979, 212-219.

[5] 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.

[6] 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.

[7] 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.

[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, *Composite Event Specification in Active Databases: Model & Implementation*, AT&T Bell Laboratories, 1992.

[10] N. H. Gehani, H. V. Jagadish and O. Shmueli, "Event Specification in an Active Object-Oriented Database", AT&T Bell Labs Technical Memorandum, 1992.

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

[12] M. R. Headington and A. E. Oldehoeft, "Open Predicate Path Expressions and their Implementation in Highly Parallel Computing Environments", *Proceedings of the International Conference on Parallel Processing*, 1985, 239-246.

[13] R. Hull and D. Jacobs, "Language Constructs for Programming Active Databases", *Proc. of the 17th Int'l Conf. on Very Large Databases*, Barcelona, Spain, Sept. 1991.

[14] S. N. Khoshafian and G. P. Copeland, "Object Identity", *Proc. OOPSLA '86*, Portland, Oregon, Sept. 1986, 406-416.

[15] 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.

[16] D. R. McCarthy and U. Dayal, "The Architecture of An Active Database Management System", *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989, 215-224.

[17] J. Melton, (ed.), "(ISO-ANSI Working Draft) Database Language SQL2 and SQL3", ANSI X3H2-90-001, Dec. 1989.

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

[19] A. Skarra, "Concurrency Control for Cooperating Transactions in an Object Oriented Database", *SIGPLAN Notices Notices 24*, 4 (April. 1989), .

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