Increasing Expressiveness of Composite Events Using Parameter Contexts ⋆

Indrakshi Ray and Wei Huang

Department of Computer Science
Colorado State University
Fort Collins CO 80528-1873

Abstract. The event-condition-action paradigm (also known as *triggers* or *ECA rules*) gives a database "active" capabilities – the ability to react automatically to changes in the database or the environment. Events can be primitive or composite. Primitive events cannot be decomposed. Different types of primitive events can be composed using event composition operators to form composite events. When a composite event occurs, it is possible that many instances of some constituent primitive event occurs. The *context* determines which of these primitive events should be considered for evaluating the composite event. Researchers have identified four different contexts, namely, *recent*, *chronicle*, *continuous*, and *cumulative* that can be associated with a composite event. Associating a single context with a complex composite event is often times not adequate. Many real-world scenarios cannot be expressed if a composite event is associated with a single context. To solve this problem, we need to associate different contexts for the various constituent primitive events. We show how this can be done by providing a formal semantics for associating contexts with primitive events. Finally, we give algorithms for event detection that implement these semantics.

1 Introduction

Traditional database management system are *passive*: the database system executes commands when requested by the user or application program. However, there are many applications where this passive behavior is inadequate. Consider for example, a financial application: whenever the price of stock for a company falls below a given threshold, the user must sell his corresponding stocks. One solution is to add monitoring mechanisms in the application programs modifying the stock prices that will alert the user to such changes. Incorporating monitoring mechanisms in all the relevant application programs is non trivial. The alternate option is to poll periodically and check the stock prices. Polling too frequently incurs a performance penalty; polling too infrequently may result in not getting the desirable functionalities. A better solution is to use active databases.

Active databases move the reactive behavior from the application into the database. This reactive capability is provided by *triggers* also known as *event-condition-action rules* or simply *rules*. In other words, triggers give active databases the capability to

⋆ This work was supported in part by AFOSR under contract number FA9550-07-1-0042.

monitor and react to specific circumstances that are relevant to an application. An active database system must provide trigger processing capabilities in addition to providing all the functionalities of a passive database system.

Different types of events are often supported by a database application: (i) data modification/retrieval events caused by database operations, such as, insert, delete, update, or select, (ii) transaction events caused by transaction commands, such as, begin, abort, and commit, (iii) application-defined events caused by application programs, (iv) temporal events which are raised at some point of time, such as December 25, 2007 at 12:00 p.m. or 2 hours after the database is updated, and (v) external events that occur outside the database, such as, sensor recording temperature above 100 degrees Fahrenheit. Various mechanisms are needed for detecting these different types of events. The types of events that are of interest depend on the specific application and the underlying data model. We do not distinguish between these different types of events in the rest of the paper.

An application may be interested in an occurrence of a single event or in a combination of events. The occurrence of a single event is referred to as a *primitive event*. Primitive events are atomic in nature – they cannot be decomposed into constituent events. Example of a primitive event is that the temperature reaches 100 degrees Fahrenheit. Sometimes an application is more interested in the occurrence of a combination of primitive events. Such an event that is formed by combining multiple primitive events using event composition operators is termed a *composite event*. For example, an application may be interested in the event that occurs when the stock prices of IBM drop after that of Intel. This is an example of a composite event that is made up of two primitive events: stock price of IBM drops and stock price of Intel drops. The event composition operator in this case is the temporal operator "after".

Many instances of a constituent primitive event may occur before the occurrence of a composite event. In such cases, we need to identify which primitive event(s) to pair up to signal the occurrence of the composite event. An example will help to explain this. Say, that we are interested in detecting the occurrence of the composite event e defined as follows: $e = e_1; e_2$. This means that event e occurs whenever primitive event e_2 occurs after the primitive event e_1. Suppose e_1 = *Intel stock price drops* and e_2 = *IBM stock price drops*. Assume that the following sequence of primitive events occur: *Intel stock price drops*, *Intel stock price drops*, *IBM stock price drops*. In other words event e_1 occurs twice before the occurrence of e_2. In such cases, which instance(s) of the primitive event e_1 should we consider in the evaluation of the composite event e. Should we consider the first occurrence of e_1, or the most recent occurrence, or both?

Chakravarthy et al. [4] have solved this problem by formalizing the notion of *context*. They proposed four kinds of contexts, namely, *recent*, *chronicle*, *continuous* and *cumulative*, can be associated with composite events. Recent context requires that only the recent occurrences of primitive events be considered when evaluating the composite event. In the previous example, if recent context were to be used, then the last occurrence of e_1 will be paired up with the occurrence of e_2 and the composite event e will be signaled only once. Chronicle requires that the primitive events be considered in a chronological order when evaluating the composite event. In this context, the composite event will consist of the first occurrence of e_1 followed by the occurrence of e_2 and

will be signaled only once. Continuous requires that the primitive events in a sliding window be considered when evaluating the composite event. In this case, the composite event will be signaled twice. In other words, there will be two occurrence of the composite event. In the first case, the first occurrence of e_1 will be paired with e_2. In the second case, the next occurrence of e_1 will be composed with e_2. Cumulative requires that all the primitive events be considered when evaluating the composite event. In this case, only one composite event will be signaled. However, it will take into account both the occurrences of e_1. Although the four contexts proposed by Chakravarthy et al. [4] can model a wide range of scenarios, they fail to model many situations. For example, consider the following ECA rule that is used in a hospital. *event: admit emergency patient and doctor enters emergency department*, *condition: true*, and *action: alert doctor about patients' conditions*. Here *event* is a composite one made up of two primitive events *E1 = admit patient* and *E2 = doctor enters emergency room*. We want this rule to be triggered every time a new patient is admitted and the doctor enters the emergency room. For event *E1* we want to use the cumulative context and for event *E2* we want the recent context. Such possibilities cannot be expressed by Chakravarthy's work because the entire composite event is associated with a single context.

We argue that associating a single context with a composite event is often times very restrictive for new applications, such as those executing in the pervasive computing environments. Such applications will have composite events made up of different types of primitive events. Often times, the type of event determines which context should be used. For example, it makes sense to use recent context for events based on streaming data, chronicle context for events involving processing customer orders. Since the constituent primitive events in composite events are of different types, requiring them to be associated with the same context is placing unnecessary restrictions on the composite event and prohibiting them from expressing many real-world situations.

We propose an alternate solution. Instead of associating a single context with a composite event, we associate a context with each constituent primitive event. This allows different types of primitive events to be combined to form a composite event. It also allows us to express many real-world situations, such as the healthcare example stated above. We discuss how this can be done and provide the formal semantics of associating contexts with primitive events for each event composition operator. We prove that our approach is more expressive than Chakravarthy et al.[4]. We give algorithms showing how composite event detection can take place when the primitive events have varying contexts.

The rest of the paper is organized as follows. Section 2 describes few related work in this area. Section 3 illustrates our notion of contexts. Section 4 formally defines our notion of contexts. Section 5 gives algorithms for detecting composite events that use our definition of contexts. Section 6 concludes the paper with pointers to future directions.

2 Related Work

A number of works [1–8] have been done in event specification and detection in active databases. Some active databases detect events using detection-based semantics [3, 4];

others use interval-based semantics [1, 2]. But not much work appears in the area of parameter contexts.

In COMPOSE [6, 7] and SAMOS [5] systems, the parameters of composite events detected are computed using the unrestricted context. In the unrestricted context, the occurrences of all primitive events are stored and all combinations of primitive events are used to generate composite events. For instance, consider the composite event $E = P; Q$; Let $p1, p2$ be instances of P and $q1, q2$ be instances of Q. Consider the following sequence of events: $p1, p2, q1, q2$. This will result in the generation of four composite events in the unrestricted context: $(p1, q1)$, $(p2, q1)$, $(p1, q2)$, and $(p2, q2)$. Unrestricted contexts have two major drawbacks. The first is that not all occurrences may be meaningful from the viewpoint of an application. The other is the big computation and space overhead associated with the detection of events.

The SNOOP system [1–4] discusses an event specification language for active databases. It classifies the primitive events into database, temporal, and explicit events. Composite events are built up from these primitive events using the following five event constructors: *disjunction*, *sequence*, *any*, *aperiodic*, *cumulative aperiodic*, *periodic* and *cumulative periodic* operators. One important contribution of SNOOP is that it proposes the notion of parameter contexts. The parameter context defines which instance of the primitive events must be used in forming a composite event. The authors propose four different parameter contexts which were discussed earlier. Although the SNOOP System discussed how to specify consumption of events in the four different parameter contexts, a parameter context can be specified only at the top-level event expression, which means that the entire composite event is associated with a single context. We argue that associating a single context with a composite event is often times not very meaningful. This is because the type of event often determines which context should be used. Since the constituent primitive events in a composite event are of different types, requiring them to be associated with the same context is placing unnecessary restrictions on the composite event and prohibiting them from expressing many real-world situations.

Zimmer and Unland [9] provides an in-depth discussion of the semantics of complex events. They provide a meta-model for describing various types of complex events. Each event instance belongs to a type. In a complex event, it is possible that many event instances belonging to a particular type occurs. The event instance selection decides which instance to consider in the composite type. They have the operators *first*, *last*, and *cumulative*. Event instance consumption decides whether an event instance can be reused in the composite event or consumed by the composite event. Event instance consumption can be of three types: *shared*, *exclusive*, and *ext-exclusive*. The shared mode does not delete any instance of the event. The exclusive mode deletes only those event instances that were used in triggering the composite event. The ext-exclusive deletes all occurrence of the event. Although the authors provide many different possibilities using the combinations of event instance selection and event instance consumption, their formal semantics are not presented. Moreover, it is hard to understand the impact of these semantics on the implementation. For instance, since shared events are never consumed, it is unclear as to how long they must be stored.

3 Our Model

Time in our model is represented using the totally ordered set of integers. We represent time using temporal intervals and use an interval-based semantics to describe our work. We begin by giving a definition of event occurrence and event detection. The occurrence of an event typically occurs over a time interval. The detection of an event occurs at a particular point in time.

Definition 1. [Occurrence of an event] *The* occurrence of an event E_i *is denoted by the predicate* $O(E_i, [t_{si}, t_{ei}])$ *where* $t_{si} \leq t_{ei}$*, and* t_{si}*,* t_{ei} *denote the start time, end time of* E_i *respectively. The predicate has the value true when the event* E_i *has occurred within the time interval* $[t_{si}, t_{ei}]$ *and is false otherwise. Primitive events are often instantaneous – in such cases the start time* t_{si} *equals the end time* t_{ei}*, that is,* $t_{si} = t_{ei}$*.*

Thus, in our model, events occur over a temporal interval. Since it is not always possible to define a total order on temporal intervals, we propose the notion of overlapping and non-overlapping events.

Definition 2. [Overlap of Events] *Two events* E_i *and* E_j *whose occurrences are denoted by* $O(E_i, [t_{si}, t_{ei}])$ *and* $O(E_j, [t_{sj}, t_{ej}])$ *respectively are said to overlap if the following condition is true:* $\exists t_p$ *such that* $(t_{si} \leq t_p \leq t_{ei}) \wedge (t_{sj} \leq t_p \leq t_{ej})$*. Otherwise the two events are said to be non-overlapping. The overlap relation is reflexive and symmetric but not transitive.*

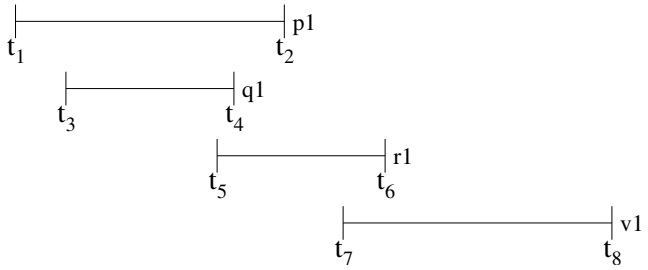

Fig. 1. Example of Event Occurrence

Example 1. Consider the events show in Figure 1. The event $p1$ overlaps with $q1$ and $r1$. $q1$ overlaps with $p1$ and $r1$. $r1$ overlaps with all other events. $v1$ overlaps with $r1$ only. The event $p1$ and $v1$ may correspond to admitting patients *Tom* and *Dick* respectively. The event $q1$ and $r1$ may correspond to alerting the physician and the nurse respectively.

Definition 3. [Detection of an event] *The* detection of an event E_i *is denoted by the predicate* $D(E_i, t_{di})$ *where* $t_{si} \leq t_{di} \leq t_{ei}$ *and* t_{si}*,* t_{di}*,* t_{ei} *represent the start time, detection*

time, end time of event E_i respectively. In almost all events (except the composite ones connected by a ternary operator), the detection time is the same as the termination time, that is, $t_{di} = t_{ei}$.

We assume that the detection time of all events in the system form a total order. This is not an unrealistic assumption since detection of an event takes place at an instant of time.

Definition 4. [Event Ordering] *We define two ordering relations on events. We say an event E_i* occurs after *event E_j if $t_{di} > t_{dj}$. We say an event E_n* follows *event E_m if $t_{sn} > t_{em}$, that is event E_n starts after E_m completes. Note that the* follows *relation can be defined for non-overlapping events but* occurs after *can be defined for any pair of events.*

For the example given in Example 1, $p1$, $r1$, $v1$ occurs after $q1$. $v1$ follows $p1$ and $q1$. For composition operators, such as conjunction, occurs after relation is important. For others, such as sequence, follows on relation is more important.

Next, we give our definitions of primitive and composite events.

Definition 5. [Primitive Event] *A* primitive event *is an atomic event which cannot be decomposed.*

In Example 1, $q1$ (alerting the physician) and $r1$ (alerting the nurse) are examples of primitive events. In general, a primitive event can be a database event, a temporal event, or an explicit event. Examples of database primitive event include database operations, such as, select, update, delete, insert, commit, abort, and begin. Example of a temporal event is 06/31/06 midnight. Example of an explicit event is when the sensor reading equals 100 Celsius.

Definition 6. [Composite Event] *A* composite event *E is an event that is obtained by applying an event composition operator op to a set of constituent events denoted by $E_1, E_2, \ldots, E_n$. This is formally denoted as follows $E = op(E_1, E2, \ldots, E_n)$. The event composition operator op may be binary or ternary. The constituent events E_1, E_2, …, E_n may be primitive or composite event.*

Consider the composite event $E = P;Q$ where P, Q, and E correspond to admitting patient, alerting physician, and treating the patient respectively. In this case, the sequence operator ; is used to compose the primitive events – the physician is alerted after the patient is admitted. A composite event may be associated with different instances of primitive events. Some of these instances have a special significance because they are responsible for starting, terminating or detecting the composite event. These instances, referred to as *initiator*, *terminator* and *detector*, are formally defined below.

Definition 7. [Initiator] *Consider a composite event $E = op(E_1, E_2, \ldots, E_n)$ that occurs over the time interval $[t_s, t_e]$. The first detected constituent event instance e_i ($i \in [1,2,..,n]$) that starts the process of parameter computation for this composite event is known as the* initiator.

Definition 8. [Detector] *Consider a composite event $E = op(E_1, E_2, \ldots, E_n)$ that occurs over the time interval $[t_s, t_e]$. The constituent event instance e_i ($i \in \{1, 2, \ldots, n\}$) that detects the occurrence of the composite event E is called the* detector.

Definition 9. [Terminator] *Consider a composite event $E=E_1\ op_1\ E_2\ op_2\ \ldots\ op_x\ E_n$ that occurs over a time interval $[t_s, t_e]$. The constituent event instance e_i ($i \in [1,2,..,n]$) that ends in time t_e which terminates the composite event is called the* terminator.

Consider the composite event $P;Q$ where P and Q correspond to admitting patient and alerting physician respectively. The instance of event P that starts the parameter computation for the composite event will be called the the initiator. An instance of Q that occurs after the initiator will cause the composite event to be detected. This instance of Q will be called the detector. The same instance of Q also terminates the composite event and hence also acts as the terminator.

In this paper, we focus our attention to binary event composition operators. The same event instance is the *detector* as well as the *terminator* for these binary operators. Hence, in our discussions we will use the terms initiators and terminators only.

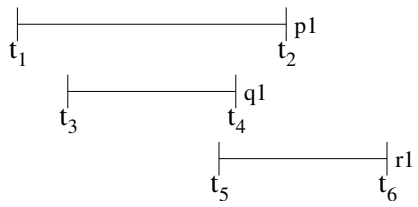

Fig. 2. Example of Initiator and Detector in Example 2

Example 2. Consider the following composite event $E = P \wedge Q \wedge R$. In the hospital example, the events P, Q and R may correspond to admitting the patient, alerting the doctor and alerting the nurse respectively. Let $p1$, $q1$ and $r1$ represent the event instances of P, Q and R respectively. The occurrences of these event instances are shown in Figure 2. The detection and termination occur at the same time instance for all of these events. $q1$ is the initiator and $r1$ is the terminator for event E. Note that, although $p1$ is started before $q1$, $p1$ is not considered to be the initiator. This is because $q1$ is detected before $p1$ and is therefore responsible for starting the parameter computation of the composite event. The occurrence time of E is denoted by the interval $[t_{sp}, t_{er}]$.

The initiator of each event can be associated with different contexts. The contexts specify which instance(s) of the initiator should be paired with a given terminator instance. The terminator instance determines whether the same instance can be used with one or more terminators. The formal definition of initiator and terminator contexts are given below.

Initiator in Recent Context An instance of the initiator event starts the composite event evaluation. Whenever a new instance of the initiator event is detected, it is used for this composite event and the old instance is discarded. An instance of the initiator event is used at most once in the composite event evaluation. After an instance of initiator event has been used for a composite event, it is discarded.

Consider the composite event $T;M$ where T signifies that the patient's temperature rises above 102 degrees fahrenheit, M denotes administering treatment to reduce the fever. If we are only concerned about the latest temperature of the patient, we would use initiator in recent context.

Initiator in Chronicle Context Every instance of the initiator event starts a new composite event. The oldest initiator event is used for the composite event. The instance of initiator event is discarded after using it for composite event calculation.

Consider an out-patient check clinic where the patients come for routine check-up. An example composite event is $P;Q$ where P represents the event of patients getting admitted and Q denotes the availability of the health care professional to serve the patient. Since the patients must be served in the order in which they arrive, we will use the initiator in chronicle context.

Initiator in Continuous Context Every new event instance of the initiator starts a composite event after discarding the previous instance of the initiator. An instance of the initiator event can be used multiple times in the composite event evaluation. The same initiator can be paired with multiple terminators. The initiator is discarded only after another initiator event occurs.

Consider, for example, the composite event $E = P;Q$ where events P, Q and E correspond to doctor entering the emergency room, admitting a patient, and treating the patient respectively. The initiator event corresponds to the doctor entering the emergency room. Since we are interested in the latest occurrence of this event, we can use either recent or continuous context. However, since the same initiator may have to be paired with multiple terminators, we need to use the continuous context.

Initiator in Cumulative Context The first instance of the initiator event starts the composite event. The subsequent occurrences of the initiator events will all be used in this same composite event. The instances of initiator events are discarded after use.

Consider, for example, the composite event $E = P;Q$ where events P, Q, and E correspond to admitting a patient, doctor entering the emergency room, and notifying the doctor about the condition of the most critical patient. Here, we would like to use the initiator in the cumulative context because the condition of all the patients must be evaluated to find the most critical patient.

Terminator in Continuous Context Each terminator can be used multiple times in the composite event evaluation. That is, a terminator can be paired with multiple initiators.

Consider, for example, the composite event $E = P;Q$ where events P, Q, and E correspond to admitting a patient, doctor entering the emergency room, and notifying the doctor about the condition of each patient. The doctor entering the emergency room is the terminator event. Since this terminator may have to be paired with multiple initiators corresponding to the different patients admitted, we would need to use continuous context for the terminator.

Terminator in Chronicle Context Each terminator is used only once in the composite event evaluation. A terminator can be paired with only one initiator. Here the different contexts are not distinguished. This is because the very first occurrence of the terminator will terminate the event.

Consider the composite event $E = P;Q$ where events P, Q, and E correspond to doctor entering the emergency room, admitting a patient, and administering treatment respectively. Since each terminator event of admitting a patient will be used only once, we need to use chronicle context for this.

In this paper, we consider only three binary event composition operators, namely, *disjunction*, *sequence*, and *conjunction*.

Disjunction $E_1 \vee E_2$

This event is triggered whenever an instance of E_1 or E_2 occurs. Since only one event constitutes this composite event, there is no context associated with this single event. This is because the very first instance of E_1 or E_2 will be the initiator as well as the terminator of this composite event.

Sequence $E_1;E_2$

This event is triggered whenever an instance of E_2 follows an instance of E_1. In this case, an instance of E_1 will be the initiator and an instance of E_2 will be the terminator. Since there may be multiple instances of initiators involved, context determines which instance of E_1 gets paired with which instance of E_2. To illustrate our ideas, we use an example of composite event *E = A; B*. The events instances are detected as follows: *b1, a1, a2, b2, b3, a3, b4*. The title of the rows is the context of event *A*, while the title of the columns is the context of event *B*. We use the following abbreviations to refer to parameter contexts: R – Recent Initiator Context, C – Chronicle Initiator Context, O – Continuous Initiator Context, U – Cumulative Initiator Context, TC – Chronicle Terminator Context and TO – Continuous Terminator Context.

A ; B	TC	TO
R	(a2,b2) (a3,b4)	(a2,b2) (a3,b4)
C	(a1,b2) (a2,b3) (a3,b4)	(a1,b2) (a2,b2) (a3,b4)
O	(a2,b2) (a2,b3) (a3,b4)	(a2,b2) (a2,b3) (a3,b4)
U	(a1,a2,b2) (a3,b4)	(a1,a2,b2) (a3,b4)

Table 1. Detection of "$A;B$"' in Occurrence of "*b1, a1, a2, b2, b3, a3, b4*" in Our Approach

Note that not all these cases can be modeled using SNOOP. For instance, the case for continuous initiator and chronicle terminator cannot be expressed by SNOOP. A real-world example rule in an Information Technology Department will motivate the need

for this case: *Event E:* A service call comes after an operator enters the on-call status, *Condition C:* true, *Action A:* provide the service. In such cases, the event $E = E_1;E_2$ where $E_1 =$ operator enters on-call status and $E_2 =$ service call comes.

Conjunction $E_1 \wedge E_2$

This event is triggered whenever both instances of E_1 and E_2 occur. For composite event with "and" operator E($A \wedge B$), either event A or event B can be an initiator event or a terminator event. When no instance of event B occurs before any instance of event A, event A is considered as an initiator event and event B is considered as a terminator event. When event B occurs before event A, event B is considered as an initiator event and event A is considered as a terminator event.

Table 2 and Table 3 show the result of *"A∧B"* in different combinations of contexts in our approach. The column heading R/TC indicates that when event B is the initiator it uses "Recent Initiator Context" but when it is the terminator it uses "Chronicle Terminator Context". The other row and column headings are interpreted in a similar manner. Many of the cases shown in Tables 2 and 3 are needed in real-world scenarios and cannot be modeled by SNOOP. For instance, consider the following rule that is used in a hospital: *Event E:* admit emergency patient and doctor enters emergency department, *Condition C:* true, *Action A:* alert doctor about patients' conditions. Here $E = E_1 \wedge E_2$ is a composite event where $E_1 =$ *admit patient* and $E_2 =$ *doctor enters emergency room*. For event E_1 we want to use Cumulative Initiator/Chronicle Terminator context. That is, when an instance of event E_1 is the initiator we want to use the cumulative context and when it is the terminator we want to use the chronicle context. For event E_2, we want to use the Recent Initiator/Chronicle Terminator context. Such possibilities cannot be expressed by SNOOP because the entire composite event is associated with a single context.

4 Formal Semantics

In this section, we give the formal definition of each operator when different contexts are associated with each constituent event. For lack of space, we do not provide the formal semantics for the conjunction operator.

Disjunction operator E1 ∨ E2

For the disjunction operator, the context of the operand is not taken into account for defining the event. This is because the very first event occurrence of either of the events is the initiator as well as the terminator. This operator is commutative.

$$O(E_1 \vee E_2, [t_s, t_e]) = (O(E_1, [t_{s1}, t_{e1}]) \wedge (t_s \leq t_{s1} \leq t_{e1} \leq t_e)) \vee (O(E_2, [t_{s2}, t_{e2}]) \wedge (t_s \leq t_{s2} \leq t_{e2} \leq t_e))$$

Sequence Operator E1; E2

Different contexts will result in different semantics for the sequence operator. Since an initiator can be associated with 4 different contexts and a terminator can be associated with 2 contexts, we have 8 different possibilities. Each event occurrence associated with the context is described using the following notation: $O(E1_{Context}; E2_{Context}, [t_{s1}, t_{e2}])$. Their formal definitions are given below:

$$O(E1_R; E2_{TC}), [t_{s1}, t_{e2}]) = \exists t_{e1}, t_{s2}(t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2} \wedge O(E1_R, [t_{s1}, t_{e1}]) \wedge O(E2_{TC}, [t_{s2}, t_{e2}]))$$

A ∧ B	R/TC	C/TC	O/TC	U/TC
R/TC	(a2,b1) (b4,a3)	(a2,b1) (b2,a3) (b3,a4)	(a2,b1) (b4,a3) (b4,a4)	(a2,b1) (b2,b3,b4,a3)
C/TC	(a1,b1) (a2,b2) (b4,a3)	(a1,b1) (a2,b2) (b3,a3) (b4,a4)	(a1,b1) (a2,b2) (b4,a3) (b4,a4)	(a1,b1) (a2,b2) (b3,b4,a3)
O/TC	(a2,b1) (a2,b2) (a2,b3) (a2,b4)	(a2,b1) (a2,b2) (a2,b3) (a2,b4)	(a2,b1) (a2,b2) (a2,b3) (a2,b4)	(a2,b1) (a2,b2) (a2,b3) (a2,b4)
U/TC	(a1,a2,b1) (b4,a3)	(a1,a2,b1) (b2,a3) (b3,a4)	(a1,a2,b1) (b4,a3) (b4,a4)	(a1,a2,b1) (b2,b3,b4,a3)
R/TO	(a2,b1) (b4,a3)	(a2,b1) (b2,a3) (b3,a3) (b4,a3)	(a2,b1) (b4,a3) (b4,a4)	(a2,b1) (b2,b3,b4,a3)
C/TO	(a1,b1) (a2,b2) (b4,a3)	(a1,b1) (a2,b2) (b3,a3) (b4,a3)	(a1,b1) (a2,b2) (b4,a3) (b4,a4)	(a1,b1) (a2,b2) (b3,b4,a3)
O/TO	(a2,b1) (a2,b2) (a2,b3) (a2,b4)	(a2,b1) (a2,b2) (a2,b3) (a2,b4)	(a2,b1) (a2,b2) (a2,b3) (a2,b4)	(a2,b1) (a2,b2) (a2,b3) (a2,b4)
U/TO	(a1,a2,b1) (b4,a3)	(a1,a2,b1) (b2,a3) (b3,a3) (b4,a3)	(a1,a2,b1) (b4,a3) (b4,a4)	(a1,a2,b1) (b2,b3,b4,a3)

Table 2. Detection of "$A \wedge B$" in Occurrence of "*a1, a2, b1, b2, b3 , b4, a3, a4*" in Our Approach

$$
\begin{aligned}
&= \exists t_{e1}, t_{s2} (t_{s1} \le t_{e1} < t_{s2} \le t_{e2} \wedge O(E1, [t_{s1}, t_{e1}]) \wedge O(E2, [t_{s2}, t_{e2}]) \\
&\quad \wedge (\neg \exists (O(E1', [t'_{s1}, t'_{e1}]) \wedge (t_{s1} \le t'_{s1} \le t'_{e1}) \wedge (t_{e1} < t'_{e1} < t_{s2}))) \\
&\quad \wedge (\neg \exists (O(E2', [t'_{s2}, t'_{e2}]) \wedge (t_{e1} < t'_{s2} \le t'_{e2}) \wedge (t'_{e2} < t_{e2}))))
\end{aligned}
$$

$O((E1_R; E2_{TO}), [t_{s1}, t_{e2}])$

The same as $O(E1_R; E2_{TC}), [t_{s1}, t_{e2}])$

$O(E1_C; E2_{TC}, [t_{s1}, t_{e2}])$

$$
\begin{aligned}
&= \exists t_{e1}, t_{s2} (t_{s1} \le t_{e1} < t_{s2} \le t_{e2} \wedge O(E1_C, [t_{s1}, t_{e1}]) \wedge O(E2_{TC}, [t_{s2}, t_{e2}])) \\
&= \exists t_{e1}, t_{s2} (t_{s1} \le t_{e1} < t_{s2} \le t_{e2} \wedge O(E1, [t_{s1}, t_{e1}]) \wedge O(E2, [t_{s2}, t_{e2}]) \\
&\quad \wedge (\neg \exists (O(E1', [t'_{s1}, t'_{e1}]) \wedge (t_{s1} \le t'_{s1} \le t'_{e1}) \wedge (t'_{e1} < t_{e1}))) \\
&\quad \wedge (\neg \exists (O(E2', [t'_{s2}, t'_{e2}]) \wedge (t_{e1} < t'_{s2} \le t'_{e2}) \wedge (t'_{e2} < t_{e2}))))
\end{aligned}
$$

A \ B	R/TO	C/TO	O/TO	U/TO
R/TC	(a2,b1) (b4,a3)	(a2,b1) (b1,a3) (b2,a4)	(a2,b1) (b4,a3) (b4,a4)	(a2,b1) (b1,b2,b3,b4,a3)
C/TC	(a1,b1) (a2,b1) (b4,a3)	(a1,b1) (a2,b1) (b1,a3) (b2,a4)	(a1,b1) (a2,b1) (b4,a3) (b4,a4)	(a1,b1) (a2,b1) (b1,b2,b3,b4,a3)
O/TC	(a2,b1) (a2,b2) (a2,b3) (a2,b4) (b4,a3)	(a2,b1) (a2,b2) (a2,b3) (a2,b4) (b4,a3)	(a2,b1) (a2,b2) (a2,b3) (a2,b4) (b4,a3)	(a2,b1) (a2,b2) (a2,b3) (a2,b4) (b4,a3)
U/TC	(a1,a2,b1) (b4,a3)	(a1,a2,b1) (b1,a3) (b2,a4)	(a1,a2,b1) (b4,a3) (b4,a4)	(a1,a2,b1) (b1,b2,b3,b4,a3)
R/TO	(a2,b1) (b4,a3)	(a2,b1) (b1,a3) (b2,a3) (b3,a3) (b4,a3)	(a2,b1) (b4,a3) (b4,a4)	(a2,b1) (b1,b2,b3,b4,a3)
C/TO	(a1,b1) (a2,b1) (b4,a3)	(a1,b1) (a2,b1) (b1,a3) (b2,a3) (b3,a3) (b4,a3)	(a1,b1) (a2,b1) (b4,a3) (b4,a4)	(a1,b1) (a2,b1) (b1,b2,b3,b4,a3)
O/TO	(a2,b1) (a2,b2) (a2,b3) (a2,b4) (b4,a3)	(a2,b1) (a2,b2) (a2,b3) (a2,b4) (b4,a3)	(a2,b1) (a2,b2) (a2,b3) (a2,b4) (b4,a3) (b4,a4)	(a2,b1) (a2,b2) (a2,b3) (a2,b4) (b4,a3)
U/TO	(a1,a2,b1) (b4,a3)	(a1,a2,b1) (b1,a3) (b2,a3) (b3,a3) (b4,a3)	(a1,a2,b1) (b4,a3) (b4,a4)	(a1,a2,b1) (b1,b2,b3,b4,a3)

Table 3. Detection of "$A \wedge B$" in Occurrence of "*a1, a2, b1, b2, b3 , b4, a3, a4*" in Our Approach

$O(E1_C; E2_{TO}, [t_{s1}, t_{e2}])$
$= \exists t_{e1}, t_{s2}(t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2} \wedge O(E1_C, [t_{s1}, t_{e1}]) \wedge O(E2_{TO}, [t_{s2}, t_{e2}]))$
$= \exists t_{e1}, t_{s2}(t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2} \wedge O(E1, [t_{s1}, t_{e1}]) \wedge O(E2, [t_{s2}, t_{e2}])$
$\wedge(\neg\exists(O(E2', [t'_{s2}, t'_{e2}]) \wedge (t_{e1} < t'_{s2} \leq t'_{e2}) \wedge (t'_{e2} < t_{e2}))))$

$O(E1_O; E2_{TC}, [t_{s1}, t_{e2}])$
$= \exists t_{e1}, t_{s2}(t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2} \wedge O(E1_O, [t_{s1}, t_{e1}]) \wedge O(E2_{TC}, [t_{s2}, t_{e2}]))$
$= \exists t_{e1}, t_{s2}(t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2} \wedge O(E1, [t_{s1}, t_{e1}]) \wedge O(E2, [t_{s2}, t_{e2}])$
$\wedge(\neg\exists(O(E1', [t'_{s1}, t'_{e1}]) \wedge (t_{s1} \leq t'_{s1} \leq t'_{e1}) \wedge (t_{e1} < t'_{e1} < t_{s2}))))$

$O(E1_O; E2_{TO}, [t_{s1}, t_{e2}])$
The same as " $O(E1_O; E2_{TC}, [t_{s1}, t_{e2}])$ "

$O(E1_U; E2_{TC}, [t_{s1}, t_{e2}])$
$= \exists t_{e1}, t_{s2}(t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2} \wedge O(E1 - U, [t_{s1}, t_{e1}]) \wedge O(E2 - TC, [t_{s2}, t_{e2}]))$
$= \exists t_{e1}, t_{s2}(t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2} \wedge (\forall [t'_{s1}, t'_{e1}](O(E1, [t'_{s1}, t'_{e1}]) \wedge (t_{s1} \leq t'_{s1} < t'_{e1} \leq t_{e1})))$
$\wedge O(E2, [t_{s2}, t_{e2}]) \wedge (\neg\exists(O(E2', [t'_{s2}, t'_{e2}]) \wedge (t_{e1} < t'_{s2} \leq t'_{e2}) \wedge (t'_{e2} < t_{e2}))))$

$O(E1_U; E2_{TO}, [t_{s1}, t_{e2}])$
The same as $O(E1_U; E2_{TC}, [t_{s1}, t_{e2}])$.

5 Algorithms for Event Detection

Definition 10. [Event Tree]: *An event tree* ET_e=(N, E) *corresponding to a composite event type* E *is a directed tree where each node* E_i *represents an event type. The root node corresponds to event type* E*, the internal nodes represent the constituent composite event types and the leaf nodes correspond to primitive event types that make up* E*. The edge* (E_i, E_j) *signifies that node* E_i *is a constituent of the composite event* E_j*.* E_i *in this case is referred to as the child node and* E_j *as the parent.*

An event can trigger multiple rules. Thus, different event trees can have nodes corresponding to the same event. In such cases, to save storage space, the event trees can be merged to form an event graph. An event graph may contain events belonging to different rules. The nodes corresponding to events which fire one or more rules are labeled d with the rule-ids of the corresponding rule.

Definition 11. [Identical Composite Event Types] *Two composite event types* $C = op(E1_{con1}, E2_{con2}, \ldots, En_{connn})$ *and* $C' = op'(E1'_{con1'}, E2'_{con2'} \ldots, En'_{connn'})$ *are said to be identical if they satisfy the following conditions:*

1. *the constituent events and their associated contexts are identical in both the cases, that is,* $Ei_{coni} = Ei'_{coni'}$ *for* $i \in \{1, 2, \ldots, n\}$.
2. *the constituent events are composed using the same event composition operator, that is,* $op = op'$.

Two or more event trees can be merged to form an event graph if they have any common nodes.

Definition 12. [Event Graph] *An event graph EG=(N,E) is a directed graph where node E_i represents an event E_i and edge (E_i,E_j) signifies that the event corresponding to node E_i is a constituent of the composite event corresponding to node E_j. Each node E_i is associated with a label $label_{E_i}$. $label_{E_i}$ is a set of rule-ids (possibly empty) that indicate the rules that will fire when the event corresponding to node $label_{E_i}$ happens.*

In the following table, Table et_{e_i} stores the parameters of composite event e_i instance. Table let_{e_j} stores the parameters of left event instance of event e_j. Table ret_{e_j} stores the parameters of right event instance of event e_j.

Algorithm 1

Processing Event e_i for Rules

Input: *EG* - event graph.(2) e_i - event that has occurred, (3)**ET** - Set of event tables where each table corresponds to an event in *EG*

Output: ET - Set of updated event tables for each event in *EG*.

Procedure $ProcessEvent(e_i, \mathbf{ET}, EG)$
begin
  For each rule-id r $\in label_{e_i}$
    signal the rule and send event parameters to condition evaluator
  For all outgoing edges (e_i,e_j) from e_i in *EG*
    begin
      /*propagate parameters in node e_i to the parent node e_j
      and detect composite event */
      $DetectCompositeEvent(EG, \mathbf{ET}, e_i, param_i, e_j, ct\_l, ct\_r)$
    end
    Delete row containing parameters of instance e_i from table et_{e_i}
end

The above algorithm *ProcessEvent* describes the actions taken when an event e_i has been detected. Recall that $label_{e_i}$ contains the set of rules that are triggered by e_i. The parameters of e_i are then passed on to the condition evaluator which determines whether the rules listed in $label_{e_i}$ can be triggered or not. The node e_i is marked as occurred indicating that this event has taken place. The parameters of event e_i are then passed onto its parents and the *DetectCompositeEvent* procedure is called. The parameters of event e_i are then removed from the event table et_{e_i}.

Algorithm 2

Detecting Composite Event e_i for Rules

Input: (1) *EG* – event graph for security level L, (2) **ET** – S et of event tables where each table corresponds to an event in *EG*, (3) e_i – chil d event that has occurred, (4) $param_i$ – parameters of event e_i, (5) e_j - - parent event, (6)ct_l – context type required for the left child of e_j, and (7) ct_r – context type required for the right child of e_j.

Output: ET – Set of updated event tables for each event in *EG* .

Procedure $DetectCompositeEvent(EG, \mathbf{ET}, e_i, param_i, e_j, ct\_l, ct\_r)$
begin

```
case node e_j
    ∨ : store param_i in a new row in composite event table et_{e_j}
        ProcessEvent(e_j, ET, EG)
    ; : if e_i =left child of e_j
        case ct_l
            Recent or Continuous:
                if table let_{e_j} is empty
                    insert param_i in a new row in table let_{e_j}
                else
                    Replace existing row in table let_{e_j} with param_i
            Chronicle:
                store param_i in a new row in table let_{e_j}
            Cumulative:
                store param_i in the same row in table let_{e_j}
        end case
    if e_i =right child of e_j
        if left child of e_j is marked as occurred
        begin
            if (ct_r==Continuous)
            begin
                replace the row with param_i from table ret_{e_j}
                for each row in table let_{e_j}
                    begin
                        copy the rows from let_{e_j} and ret_{e_j} into composite table et_{e_j}
                        ProcessEvent(e_j, ET, EG)
                    end
                if (ct_l!=Continuous)
                    Delete the row from the left event table let_{e_j}
            end
            else
                begin
                    store param_i in a row in table ret_{e_j}
                    copy the first row from let_{e_j} and the row from ret_{e_j} into table et_{e_j}
                    Delete the used row from ret_{e_j}
                    ProcessEvent(e_j, ET, EG)
                    if (ct_l!=Continuous)
                        Delete the used row from left event table let_{e_j}
                end
        end
    ∧: /* Details omitted */
        end case
end
```

6 Conclusion

Many real-world event processing applications often require processing composite events. Composite events are made by applying event composition operator to primitive events. Typically many instances of a primitive event occur before a composite event can be detected. The issue is which primitive event(s) should we consider while computing the composite event. Earlier work on event contexts identified four different types of contexts that can be associated with a primitive event. Associating a single context with a composite event is often not adequate for expressing many real-world scenarios. We have shown how different contexts can be associated with the individual primitive events in a composite event to overcome this situation. We have also provided a formal semantics for event composition and algorithms to enforce it. A lot of work remains to be done. We have focussed only on binary operators for event composition. We would like to extend this work to other operators as well. Although our event composition uses an underlying notion of time, we have not explicitly modeled it. In future, we would like to model time and consider temporal operators as well. We would also like to address how time synchronization can be achieved in a distributed environment.

References

1. R. Adaikkalavan and S. Chakravarthy. Formaliztion and Detection of Events Using Interval-Based Semantics. In *Proceedings of the 11th International Conference on Management of-Data*, pages 58–69, Goa, India, January 2005.
2. R. Adaikkavalan and S. Chakravarthy. SnoopIB: Interval-based Event Specification and Detection for Active Databases. *Data and Knowledge Engineering*, 59(1):139–165, 2006.
3. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, Santiago de Chile, Chile, September 1994. Morgan Kaufmann.
4. S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(1):1–26, 1994.
5. S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *Proceedings of the 4th International Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 2–9, Houston, TX, February 1994.
6. N. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Active Object-Oriented Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 81–90, San Diego, CA, June 1992. ACM Press.
7. N. Gehani, H. V. Jagadish, and O. Shmueli. COMPOSE: A System for Composite Specification and Detection. In N. R. Adam and B. K. Bhargava, editors, *Advanced Database Systems*, volume 759 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 1993.
8. N. W. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, 1999.
9. D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the 15th International Conference on Data Engineering*, pages 392–399, Sydney, Australia, March 1999. IEEE Computer Society Press.