Event Operators: Formalization, Algorithms, and Implementation Using Interval-Based Semantics

Raman Adaikkalavan, Sharma Chakravarthy

Technical Report CSE-2002-3
June, 2002
Event Operators: Formalization, Algorithms, and Implementation Using Interval-Based Semantics

Raman Adaikkalavan and Sharma Chakravarthy

Computer Science and Engineering Department
The University of Texas at Arlington
Arlington, TX 76019, USA
{adaikkal, sharma}@cse.uta.edu

Abstract

Snoop is an event specification language developed for expressing primitive and composite events that are part of Event-Condition-Action (or ECA) rules. Snoop also supports complex event expressions using a set of event operators, such as And, Or, Sequence, Not etc. A detection-based (using the end time of an event occurrence on the time line) semantics was provided for all the operators in various contexts (such as Cumulative and Continuous) and were implemented in Sentinel – an active object-oriented DBMS. An event was defined to be an instantaneous, atomic (happens completely or not at all) occurrence of interest and the time of occurrence of the last event in an expression was used as the time of occurrence for the entire event expression.

The above detection-based semantics does not recognize multiple compositions of some operators – especially Sequence – in the intended way. In order to recognize all the Snoop operators in all the contexts in the intended way, the semantics need to include start time as well as end time for a composite event (i.e., interval-based semantics). In this paper, we formalize the occurrence of Snoop event operators and expressions using interval-based semantics for the recent context. The algorithms for the detection of events using interval-based semantics introduce some challenges, as not all the events are known (especially their starting points). We present a few representative algorithms to detect some operators conforming to the interval-based semantics. Finally, we comment on their implementation in the context of Sentinel.

1 Introduction

There is consensus in the community on the Event-Condition-Action rules (or ECA) as the most general format for expressing rules in an active database management system (ADBMS). As the event component was the least understood (conditions correspond to queries, and actions correspond to transactions) part of the ECA rule, there is a large body of work on the language for event specification. Snoop [1, 2] was developed as the event specification component of the ECA rule formalism used as a part of the Sentinel project [3-6] on active object-oriented DBMS. Snoop supports expressive ECA rules that include coupling modes and parameter (or event consumption) contexts.

The rest of the paper is organized as follows. Section 2 highlights the need for interval-based semantics and why the distinction between detection and occurrence semantics is important. This section also refers to related work on event specification without going into the details as all of them use detection-based semantics. The reader is referred to [7] for a good description of the differences between the AI and database approaches. Section 3 explains Snoop operators and their semantics in the unrestricted context using interval-based occurrences. Section 4 extends the above to recent context. Section 5 provides an illustrative example of event detection in interval-based semantics in recent context. Section 6 discusses algorithms and implementation issues. Section 7 has conclusions and future work.

2 Need for Interval-Based Semantics

The detection-based semantics typically used by all event specification languages used in Active DBMSs (Snoop [1, 2], COMPOSE [8, 9], Samos [10, 11], ADAM [12, 13], ACOOD [14, 15], event-based
conditions [16], and Reach [17-19]) do not differentiate between event occurrence and event detection. Typically, an event is, or can be detected at the end of the interval over which it occurs. However, the event itself occurs over an interval although it is typically detected at the end of the interval. Also, from a detection viewpoint, the start of the event interval is not known until the event is detected. The occurrence and detection semantics are not differentiated in the above event specification languages as pointed out by [7] which leads to some unintended semantics, when certain operators, such as Sequence, is composed more than once.

For example, in \( E_1;E_2 \) (";' refers to the sequence operator, \( E_1 \) and \( E_2 \) refers to event types), \( E_1 \) is defined to occur earlier than \( E_2 \). Using detection-based semantics, \( E_1;E_2 \) is detected as long as the end time of an instance of \( E_1 \) is less than the end time of an instance of \( E_2 \). Composite event \( E_1;E_2 \) is detected at the point when the last constituent event (i.e., \( E_2 \)) of the composite event is detected. Because of the detection (not the occurrence) semantics, the start times of event instances are not considered. This will lead to the following problem.

Consider the composite events \( E_1;(E_2;E_3) \) and \( E_2;(E_2;E_3) \). Intuitively, since ‘;' is sequential composition, we should expect these two to be different as \( E_1 \) strictly precedes \( E_2 \) in the first case and \( E_2 \) strictly precedes \( E_1 \) in the latter case. However, since the detection semantics is used, that subtle difference is lost depending upon the intervals over which \( E_1, E_2 \), and \( E_3 \) occur.

```
1 2 3 4 5 6
E1 E2 E3
⇒ Start Time of the Event
⇒ End Time of the Event
```

Figure 1 Example events

Given the occurrences of \( E_1 [1,2] \) (\( E_1 \) is the event, 1 is the start interval and 2 is the end interval of event \( E_1 \)), \( E_2 [3,4] \), and \( E_3 [5,6] \), as shown in the Figure 1, they satisfy both the event expressions using the detection-based semantics. Both these event expressions are satisfied because the detection time (end time) of event \( E_1 [2,2] \) is less than the detection time (end time of last constituent event) of \( E_2;E_3 [6,6] \) (i.e., \( 2 < 6 \)) in the first case and detection time (end time) of event \( E_2 [4,4] \) is less than the detection time of \( E_1;E_3 [6,6] \) (i.e., \( 4 < 6 \)) in the second case. However, if interval-based definition is used for occurrence, then only the first expression \( (E_1;(E_2;E_3)) \) should be correctly detected and not the second one \( (E_2;(E_1;E_3)) \), since the start times are considered in both the cases. In the first case, detection time (end time) of event \( E_1 [1,2] \) is less than the start time of \( E_2;E_3 [3,6] \) (i.e., \( 2 < 6 \)) and in the second case, detection time (end time) of event \( E_2 [4] \) is not less than the start time of \( E_1;E_3 [1,6] \). Galton [7] has pointed out this discrepancy between database work where detection semantics has been used to define the operators in contrast to work in AI where occurrence has played a dominant role for inference than detection and hence interval semantics has been used [20, 21]. In the rest of the paper, we present interval-based semantics of event occurrences and discuss algorithms for event detection and their implementation using event graphs.

### 3 Interval-Based Semantics of Snoop

For the purpose of this paper, we assume an equidistant discrete time domain having “0” as the origin and each time point represented by a non-negative integer. The granularity of the domain is assumed to be specific to the domain of interest. An event is detected atomically at a point on the time line. In object-oriented databases, interest in events comes from the state changes produced by method executions by an object. Similarly, in relational databases, interest in events comes from the data manipulation operations such as insert, delete, and update. Similar to these database (or domain specific) events there can also be temporal events that are based on time or explicit events that are detected by an application program (outside of a DBMS) along with its parameters. Detection-based semantics was adopted as begin and end events were of significance in most of the database related work.

#### 3.1 Primitive Events

Primitive events are a finite set of events that are pre-defined in the (application) domain of interest. Primitive events are distinguished as domain specific, temporal and explicit events (for more detail refer to [1, 2, 22]). For example, a method execution by an object in an object-oriented database is a primitive event. These method executions can be grouped into before and after events (or event types) based on where they are detected (immediately before or after the method call). An event occurs over a time interval and is denoted by \( E [t_1, t_2] \) (where \( E \) is the event, \( t_1 \) is the start interval of the event denoted by \( \uparrow E \), \( t_2 \) is the end interval of the event denoted by \( \downarrow E \)). In the case of primitive events, the start and the end interval are assumed to be the same (i.e., \( t_1 = t_2 \)). For events that span over an interval, the event occurs over the interval \([t_1, t_2]\) and detected at the end of the interval.
3.2 Event Expressions

For many applications, supporting only primitive events is not adequate. In many real-life applications, there is a need for specifying more complex patterns of events such as, arrival of a report followed by a detection of a specified object in a specific area. They cannot be expressed with a language that does not support expressive event operators along with their semantics. An appropriate set of operators along with the closure property allows one to construct complex composite events by combining primitive events and composite events in ways meaningful to an application interested in situation monitoring. To facilitate this, we have defined a set of event operators along with their semantics. Snoop [1, 2] is an event specification language that is used to specify combinations of events using Snoop operators such as And, Or, Sequence, Not, Aperiodic, Periodic, Cumulative Aperiodic, Cumulative Periodic, and PLUS. The motivation for the choice of these operators and how they compare with other event specification languages can be found in [1, 2].

3.3 Composite Events

Composite events are constructed using primitive events and event operators in a recursive manner. A composite event consists of a number of primitive events and operators; and the set of primitive events of a composite event are termed as constituent events of that composite event. A composite event is said to occur over an interval, but is detected at the point when the last constituent event of that composite event is detected. The detection and occurrence semantics is clearly differentiated and the detection is defined in terms of occurrence as shown in [7]. Note that occurrence of events cannot be defined in terms of detection which was the problem with the earlier detection-based approaches.

We introduce the notion of an initiator, detector, and terminator for defining event occurrences. A composite event occurrence is based on the initiator, detector and terminator of that event which in turn are constituent events of that composite event. An initiator of a composite event is the first constituent event whose occurrence starts the composite event. Detector of a composite event is the constituent event whose occurrence detects the composite event, and terminator of a composite event is the constituent event that is responsible for terminating the composite event. For some operators, the detector and terminator are different (e.g., Aperiodic). For many operators, the detector and terminator are the same (e.g., Sequence).

A Composite event E occurs over a time interval and is defined by E [t1, t2] where E is a composite event, t1 is the start time of the composite event occurrence and t2 is the end time of composite event occurrence (t1 is the starting time of the first constituent event that occurs (initiator) and t2 is the end time of the detecting or terminating constituent event (detector or terminator) and they are denoted by ↑E and E↓ respectively).

End of an event: O (E↓, t) ≡ ∃t' ≤ t (O (E, [t', t]))
Start of an event: O (↑E, t) ≡ ∃t' (t ≤ t' ∧ O (E, [t, t']))

Below, we define Snoop operators intuitively and then provide a formal definition using the interval-based semantics. The formal definitions are reproduced from [7] except for the Plus operator.
3.3.1 Overlapping Event Combinations:

AND (\(\Delta\)): Conjunction of two events \(E_1\) and \(E_2\), denoted by \(E_1 \Delta E_2\), occurs when both \(E_1\) and \(E_2\) occur, irrespective of their order of occurrence.

For the event occurrences in Figure 2, the event \(E_1\) is combined with the following occurrences \(E_2\) for the “\(\Delta\)” event: 1,2,4,5,6,8,9,10,11,12,13

SEQUENCE (\(\&\)): Sequence of two events \(E_1\) and \(E_2\), denoted by \(E_1; E_2\), occurs when \(E_2\) occurs provided \(E_1\) has already occurred. This implies that the end interval time of occurrence of \(E_1\) is guaranteed to be less than the start interval time of occurrence of \(E_2\).

For the event occurrences in Figure 2, the event \(E_1\) is combined with the following occurrences \(E_2\) for the “;” event: 4.

OR (\(\nabla\)): Disjunction of two events \(E_1\) and \(E_2\), denoted by \(E_1 \nabla E_2\), occurs when \(E_1\) occurs or \(E_2\) occurs.

For the event occurrences in Figure 2, the following \(\nabla\) events are detected: 1,2,3,4,5,6,7,8,9,10,11,12,13.

NOT (\(\neg\)): The NOT operator, denoted by \(\neg (E_3)\), detects the non-occurrence of the event \(E_3\) in the closed interval formed by \(E_1\) and \(E_2\).

For this Snoop event operator, for the event \(E_1\), the possible combinations of occurrences of \(E_2\) is only 4 (from Figure 2) and there is no occurrence of \(E_3\) in the interval formed by \(E_1\) and \(E_2\).

Aperiodic Operators (\(A\), \(A^*\)): The Aperiodic operator allows one to express the occurrences of an aperiodic event within a closed time interval. There are two versions of this event specification. The non-cumulative aperiodic event is expressed as \(O (A (E_1, E_2, E_3), [t_1, t_2])\). The event \(A\) is signaled each time \(E_2\) occurs within the time interval ended by \(E_3\) and started by \(E_1\).

On the other hand, the cumulative aperiodic event is expressed as \(O (A^* (E_1, E_2, E_3), [t_1, t_2])\). This event occurs only once when \(E_3\) occurs and accumulates the occurrences of \(E_2\) within the interval formed by \(E_1\) and \(E_3\).

Periodic Operators (\(P\), \(P^*\)): A periodic event (\(P\)) is a temporal event that occurs periodically. A periodic event is denoted as \(O (P (E_1, [t], E_2), [t_1, t_2])\). While \(E_1\) and \(E_3\) are events the event \(E\) at \([t]\) should be a time string (temporal event). The Periodic event occurs whenever the time string \([t]\) occurs within the time interval ended by \(E_3\) and started by \(E_1\).

\(P\) has a cumulative variant \(P^*\) expressed as \(O (P^* (E_1, [t], E_3))\). Unlike \(P\), \(P^*\) occurs only once when the event \(E_3\) occurs. It also accumulates the event \(E_2\) occurrences at the end of each period and made available when \(P^*\) occurs.

The above definitions describe the meaning of event operators in the unrestricted (or general) context. This means events, once they occur, cannot be discarded at all. For example, for a “;”, all event occurrences that occur later than an event will get paired with that event as per the semantics. In the absence of any mechanism for restricting the event usage (or consumption), events need to be detected and the parameters for those composite events need to be computed using the unrestricted context definitions of the Snoop event operators. However, the number of events produced by the above definitions (in the unrestricted context) can be large and not all event occurrences may be meaningful for an application. In addition, detection of these events has substantial computation and space overhead, which may become a problem for situation monitoring applications.

3.3.2 Semantics of Event Operators:

Primitive event: \(O (E [t, t'])\)

An event \(E\) occurs over the interval \([t, t']\) and are predefined in the subsystem.

- \(t\) is the Start time of an event
- \(t'\) is the End Time of an event

AND event: \(E = O (E_1 \Delta E_2, [t_1, t_2])\)

\(E\) is the AND event
\(E_1\) can be either \textit{initiator} or \textit{terminator}
\(E_2\) can be either \textit{initiator} or \textit{terminator}

Definition: \(O (E_1 \Delta E_2, [t_1, t_2]) \triangleq \exists_{t, t'} (t_1 \leq t \leq t' \land t_1 \leq t' \leq t_2) \land ((O (E_1, [t_1, t]) \land O (E_2, [t', t_2])) \lor (O (E_1, [t', t_2]) \land O (E_2, [t_1, t])))\)

Sequence Event: \(E = O (E_1; E_2, [t_1, t_2])\)

\(E\) is the sequence event
\(E_1\) is the \textit{initiator} of the sequence event
\(E_2\) is the \textit{terminator} of the sequence event

Definition: \(O (E_1; E_2, [t_1, t_2]) \triangleq \exists_{t, t'} (t_1 \leq t < t' \leq t_2) \land ((O (E_1, [t_1, t]) \land O (E_2, [t', t_2])) \lor (O (E_1, [t', t_2]) \land O (E_2, [t_1, t])))\)

Or Event: \(E = O (E_1 \nabla E_2, [t_1, t_2])\)

\(E\) is the \textit{Or} event
E₁ & E₂ is the initiator as well as the terminator

Definition: \( O (E₁ ∨ E₂, [t₁, t₂]) \triangleq O (E₁, [t₁, t₂]) ∨ O (E₂, [t₁, t₂]) \)

In the following operators, we use the start and end of an event defined earlier. To enable us to express this more concisely the predicate \( O_m \) is defined as follows [7].

\[
O_m (E [t₁, t₂]) \triangleq \exists t₁', t₂' \ (t₁ ≤ t₁' ≤ t₂' ≤ t₂ ∧ O (E, [t₁', t₂']))
\]

**Not Event:** \( E = O (¬ (E₃)[E₁; E₂], [t₁, t₂]) \)

- \( E \) is the Not event
- Detects the non-occurrence of the event \( E₃ \) in the closed interval formed by \( E₁ \) and \( E₂ \).

Definition: \( O (¬ (E₃)[E₁, E₂], [t₁, t₂]) \triangleq \)

\[
O (E₁↓, t₁) ∧ O (↑E₃, t₂) ∧ ¬O_m (E₂, [t₁, t₂])
\]

**Aperiodic Event:** \( E = O (A (E₁, E₂, E₃), [t₁, t₂]) \)

- \( E \) is the Aperiodic event
- \( E₁ \) is the initiator
- \( E₂ \) is the detector
- \( E₃ \) is the terminator

Definition: \( O (A (E₁, E₂, E₃), [t₁, t₂]) \triangleq \)

\[
O (E₂, [t₁, t₂]) ∧ \exists t < t₁ \ (O (E₁↓, t) ∧ ¬O_m (E₃, [t + 1, t₂]))
\]

The occurrence time of the event \( A \) is the occurrence time for \( E₂ \); an occurrence of the event \( A \) is an occurrence of \( E₂ \) and is determined by \( E₁ \) and \( E₃ \). The rest of the condition specifies the context. There must be no occurrence of \( E₃ \) wholly within the interval between the occurrence of \( E₁ \) and the occurrence of \( E₂ \).

**Periodic Event:** \( E = O (P (E₁, n, E₃), [t]) \)

- \( E \) is the Periodic event
- \( E₁ \) is the initiator
- \( E₂ \) is the detector
- \( E₃ \) is the terminator

Definition: \( O (P (E₁, n, E₃), [t]) \triangleq \)

\[
\exists t' < t ∧ \exists i ∈ ℤ^+ \ (t = t' + ni ∧ O (E₁↓, t') ∧ ¬O_m (E₃, [t' + 1, t]))
\]

The occurrence time of the periodic event is the time of occurrence of the event \( E₂ \), which is the time string. The periodic event occurs after the time \( [t] \) specified by the time string, after the occurrence of the event \( E₁ \).

**Plus Event:** \( E = O (Plus (E₁, n) [t]) \)

- \( E \) is the Plus event
- \( E₁ \) is the initiator
- \( n \) is the terminator

Definition: \( O (Plus (E₁, n) [t]) \triangleq \exists t' < t \ (O (E₁↓, t') ∧ t = t' + n) \)

A Plus operator is used to specify a relative time event [23]. A Plus operator combines two events \( E₁ \) and \( E₂ \) where \( E₁ \) can be any type of event and \( E₂ \) is a time string \([t]\). The Plus event occurs after time \([t]\), after the event \( E₁ \) occurs. The definition for the Plus operator for the unrestricted context is shown above.

### 3.4 Disjoint Event Combinations:

Another aspect of event occurrences of the constituent events of a composite event is that they can be either overlapping or disjoint. When the events are allowed to overlap, all combinations in which two events can occur are shown in Figure 3.

When events are not allowed to overlap, we have fewer combinations. This may be meaningful for many applications where the same event should not participate in more than one composite event or only one of the overlapping events is of interest. The possible disjoint event combinations are shown in Figure 3.

[Figure 3 Disjoint event combinations]

In this paper, we assume that constituent events can overlap. The number of events that take part in the detection of the composite event depends on the semantics of Snoop operators.

### 4 Parameter Contexts

A large number of events are generated when unrestricted context is used. When we studied many application domains, it turned out that these application domains may not be interested in the unrestricted context all the time but need mechanisms to tailor the semantics of event expression to their domain needs. In order to provide more meaningful event occurrences to match application needs, Snoop introduced several parameter contexts: Recent, Chronicle, Continuous, and Cumulative. The idea behind the parameter contexts is to filter the events (or the history) generated by the unrestricted context in various ways to reduce the number of events generated. The ideal situation is to allow the user to roll his/her own context as needed. We briefly describe below the motivations for the introduction of contexts. It is also the case that each context defined below generates a subset of events generated by the unrestricted context.
**Recent Context:** In applications where events are happening at a fast rate and multiple occurrences of the same event only refine the previous value can use this context. Only the most recent or the latest initiator for any event that has started the detection of a composite event is used in this context. This entails that the most recent occurrence just updates (summarizes) the previous occurrence(s) of the same event type. In this context, *not all occurrences* of a constituent event will be used in the composite event detection. An initiator will continue to initiate new event occurrences until a *new initiator* or a *terminator* occurs. Binary Snoop operators use only detectors. This implies that the initiator will continue to initiate new event occurrences until a new initiator occurs. On the other hand, ternary Snoop operators contain both detectors and terminators, which implies that the initiator will continue to initiate new event occurrences until a new initiator occurs or until a terminator occurs. Once the composite event is terminated, all the constituent events of that composite event will be deleted.

**Chronicle Context:** In applications where there is a correspondence between different types of events and their occurrences, and this correspondence needs to be maintained, chronicle context is useful. In this context, for a composite event occurrence, the initiator and terminator pair is unique (oldest initiator is paired with the oldest terminator; hence the name). The detector and the initiator in this context can take part in more than one event occurrence (e.g., Aperiodic), but the terminator does not take part in more than one composite event occurrence. For binary Snoop operators, both the detector and terminator are the *same*, so once detected the entire set of participating constituent events (initiator, detector and terminator) are deleted. For ternary Snoop operators, detectors and terminators are *different*, so once detected (e.g., Aperiodic) the detectors are deleted, and when terminated (e.g., Aperiodic*) only the corresponding initiator and terminator pairs are deleted and the constituent events (except the initiators and terminators) that can be used in future events are preserved. Future events are those that can be the constituent event for some future event will be preserved.

**Continuous Context:** In applications where event detection along a moving time window is needed, continuous context can be used. In this context, each initiator starts the detection of that composite event and a single detector or terminator may detect one or more occurrences of that same composite event. An initiator will be used *at least once* to detect that event. For binary Snoop operators, all the constituent events (initiator, detector and/or terminator) are deleted once the event is detected. For ternary Snoop operators detector and terminator are different, so once detected (e.g., Aperiodic) the detectors are deleted and when terminated (e.g., Aperiodic*) only the corresponding initiator and terminator pairs are deleted and the constituent events (except the initiators and terminators) that can be used in future events are preserved. Future events are the events that are initiated by the initiators that are not paired with this terminator and which can include these constituent events at the time of their detection.

**Cumulative Context:** Applications use this context when multiple occurrences of constituent events need to be grouped and used in a meaningful way when the event occurs. In this context, all occurrences of an event type are accumulated as instances of that event until the event is terminated. An event occurrence does not participate in two distinct occurrences of the same composite event. In both binary and ternary operator, detector and terminator are same and once detected and terminated all constituent event occurrences that were part of the detection are deleted. Other events that can be the constituent event for some future event will be preserved.

### 4.1 Operator Semantics in Recent Context

In this section, we extend the formal semantics to recent context. We describe all the operators excluding the periodic operators in the recent context. In addition, we provide some algorithmic and implementation details with respect to the event detection in recent context. For a full description, please refer to [24]. Below, “O” represents the occurrence-based Snoop semantics.

### 4.2 Event Histories

The above intuitive explanations of contexts are based on the event occurrences over a time line. In this section, using the notion of *event histories*, we formalize these definitions to take the parameter contexts into account. An event history maintains a chronological history of event occurrences up to a given point in time. Suppose $e_i$ is an event instance of type $E_1$ then $E_1 [H]$ represents the event history that stores all the instances of the event $E_1$ (namely $e_{ij}$). In order to extend these definitions to parameter contexts the following notation is used.

$E_i [H] \Rightarrow$ Event history for event $e_i$

$\begin{align*}
t_e & \equiv \text{Starting time of an event} \\
t_o & \equiv \text{Ending time of an event}
\end{align*}$

Below, we first describe the history-based event occurrences intuitively before defining them formally.
4.2.1 Occurrence Semantics in Recent Context

Sequence operator in Unrestricted Context: Below we illustrate how event histories can be used for the detection of the ";" operator defined in Section 3.3.

\[ E_1 [H] = \{(3, 5), (4, 6), (8, 9)\} \]
\[ E_2 [H] = \{(1, 2), (7, 10), (11, 12)\} \]

In this context only the most recent initiator is used (see section 4). In the above example, when the event \( e_2^1 \) occurs over the interval [1, 2] there is no event in the event history of \( E_1 \) that satisfies the ";" operator condition. Event \( e_2^2 \) occurs over the interval [7, 10]. It is not paired with event \( e_1^1 \) because there is an occurrence of event \( e_1^2 \) [3, 5] in the interval formed by the end time of event \( e_1^1 [3, 5] \) and start time of event \( e_2^2 [7, 10] \), which does not satisfy the condition given above. Event \( e_2^3 \) does not detect the sequence event in the recent context with the event \( e_1^2 \) since event \( e_1^3 \) is the recent initiator. Event \( e_1^3 \) cannot pair with the event \( e_2^2 \) since it does not satisfy the ";" semantics. Similarly, when the event \( e_2^3 \) occurs it detects the recent event with the event pair \( (e_1^3, e_2^3) \) over the interval [8, 12].

Events in recent context: \( \{(e_1^3, e_2^3) [8, 12]\}. \)

Plus operator in Unrestricted Context: Plus event occurs only once after the time interval specified by 'n' after the event \( E_1 \) occurs and denoted by \( (\text{Plus} (E_1, n [t]) \).

For the events shown in Figure 5, Plus event defined in Section 3.3 generates the following pairs of events in the unrestricted context: \( \{(e_1^1, 4) [9, 9], (e_1^2, 4) [10, 10], (e_1^3, 4) [16, 16]\} \).

Sequence operator in Recent Context: The ";" operator in recent context is formally defined as follows:

\[ O (E_1; E_2, [t_{s1}, t_{e2}]) \triangleq \forall e_1 \in E_1 [H] \wedge \forall e_2 \in E_2 [H] \wedge\]
\[ \{(O (e_1, [t_{s1}, t_{e1}])) \land O (e_2, [t_{s2}, t_{e2}]) \land (t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2})) \} \land (\exists e_1 \in [t_{start}, t_{end}] \land (t_{e1} < t_{e2}) \land \land e_2 \in E_1 [H]) \}

In order to formally define the Sequence event in the recent context, take an event pair \( E_1 \) and \( E_2 \) from the event histories \( E_1 [H] \) and \( E_2 [H] \) respectively. For this event pair to be a Sequence event in the recent context, there should not be an occurrence of any other instance of event \( E_1 \) from the event history \( E_1 [H] \) in the interval formed by this event pair. Formalization of the sequence operator in the recent context is explained below using the example shown in the Figure 4.

\[ E_1 [H] = \{(3, 5), (4, 6), (8, 9)\} \]
\[ E_2 [H] = \{(1, 2), (7, 10), (11, 12)\} \]
Formal definition given above is explained with the events shown in Figure 5. Event $e_1 \uparrow$ initiates a Plus event at the time point 5. When the event $e_2 \uparrow$ occurs it initiates a new Plus event and terminates the Plus event that was initiated previously. At the time point 9 Plus event is detected in the recent context. Similarly event $e_3 \uparrow$ detects a Plus event at the time point 16. Events in recent context: {(e_1^2, 4) [10,10], (e_1^3, 4) [16,16]}.

**NOT operator in Unrestricted Context:** “$\neg$” Operator can be expressed as the Sequence of $E_1$ and $E_2$ where there is no occurrence of the event $E_3$ in the interval formed by these events. Thus, explanation of the “$\neg$” Operator definition is same as the “;” operator with one additional condition. This condition stipulates that there cannot be an occurrence of the event $E_3$ from $E_3[H]$ in the interval formed by the end time of the event $E_1$ and the start time of the event $E_2$. Below we illustrate how event histories can be used for the detection of the NOT operator $\neg (E_3)[E_1, E_2]$, defined in the section 3.3.

$$E_1[H] = \{(3, 5), (4, 6), (8, 9)\}$$
$$E_2[H] = \{(1, 2), (7, 10), (11, 12)\}$$
$$E_3[H] = \{(5, 5)\}$$

Occurrence of event $e_1 \uparrow$ does not detect any event since the event history $E_1[H]$ is empty. Event $e_1 \uparrow$ initiates “$\neg$” event in the recent context, and is terminated by event $e_3 \uparrow$. “$\neg$” Event in the recent context is initiated by event $e_2 \uparrow$. This event is terminated by the occurrence of the event $e_3 \uparrow$, which acts as the most recent initiator. Event occurrence $e_2 \uparrow$ does not pair with the initiator $e_1 \uparrow$ since it does not satisfy the condition ($t_{s_1} \leq t_{e_1} < t_{s_2} \leq t_{e_2}$), where as the event $e_3 \uparrow$ detects a recent event with the initiator $e_3 \uparrow$ since there are no other instances of event $E_1$ occurrence in the interval formed by this event pair and this satisfies both the conditions ($t_{s_1} \leq t_{e_1} < t_{e_2} \leq t_{e_2}$) and ($\not\exists e_1'[t_{start}, t_{end}] | (t_{e_1} < t_{end} < t_{e_2}) \land e_1' \in E_1[H]$).

Events in recent context: {($e_1^3, e_2^3$) [8, 12]}.

**OR event:** The semantics of “$\vee$” does not change with the context as each occurrence is detected individually.

**Aperiodic operator in Unrestricted Context:** Below we illustrate how event histories can be used for the detection of $(A (E_1, E_2, E_3), [t_{s_1}, t_{e_1}])$ in the unrestricted context defined in Section 3.3.

$$E_1[H] = \{(3, 5), (4, 6)\}$$
$$E_2[H] = \{(1, 2), (8, 9), (7, 10), (11, 12)\}$$
$$E_3[H] = \{(5, 5)\}$$

In the unrestricted context, above events shown in Figure 7 generate the following pair of events {($e_1^1, e_2^1$) [3, 9], ($e_1^2, e_2^2$) [4, 9], ($e_1^1, e_2^2$) [3, 10], ($e_1^2, e_3^2$) [4, 10]}. Aperiodic operator in Recent Context: The “$A$” operator in recent context is formally defined as follows:

$$O (\neg (E_3)[E_1, E_2], [t_{s_1}, t_{e_1}]) \triangleq \forall e_1 \in E_1[H] \land \forall e_2 \in E_2[H] \land \forall e_3 \in E_3[H] \land \left\{ (O (e_1, [t_{s_1}, t_{e_1}]) \land O (e_2, [t_{s_2}, t_{e_2}]) \land (t_{s_1} \leq t_{e_1} < t_{s_2} \leq t_{e_2}) \land \neg O_{in} (e_3, [t_{s_1}, t_{e_2}]) \right\} \left\{ \not\exists e_1'[t_{start}, t_{end}] | (t_{s_1} < t_{end} < t_{e_2}) \land e_1' \in E_1[H] \right\}$$

Formally, for an event pair $E_1$ and $E_2$ to be in the recent context, there cannot be an occurrence of any other instance of event $E_1$ from the event history $E_1[H]$ between this pair. Formalization of the “$\neg$” operator in the recent context is explained below using the example shown in the Figure 6.

$$E_1[H] = \{(3, 5), (4, 6), (8, 9)\}$$
$$E_2[H] = \{(1, 2), (7, 10), (11, 12)\}$$
$$E_3[H] = \{(5, 5)\}$$

**Not operator in Recent Context:** The “$\neg$” operator in recent context is formally defined as follows:

$$O (\neg (E_3)[E_1, E_2], [t_{s_1}, t_{e_1}]) \triangleq \forall e_1 \in E_1[H] \land \forall e_2 \in E_2[H] \land \forall e_3 \in E_3[H] \land \left\{ (O (e_1, [t_{s_1}, t_{e_1}]) \land O (e_2, [t_{s_2}, t_{e_2}]) \land (t_{s_1} \leq t_{e_1} < t_{s_2} \leq t_{e_2}) \land \neg O_{in} (e_3, [t_{s_1}, t_{e_2}]) \right\} \left\{ \not\exists e_1'[t_{start}, t_{end}] | (t_{s_1} < t_{end} < t_{e_2}) \land e_1' \in E_1[H] \right\}$$

Formally, for an event pair $E_1$ and $E_2$ to be in the recent context, there cannot be an occurrence of any other instance of event $E_1$ from the event history $E_1[H]$ between this pair. Formalization of the “$\neg$” operator in the recent context is explained below using the example shown in the Figure 6.

$$E_1[H] = \{(3, 5), (4, 6), (8, 9)\}$$
$$E_2[H] = \{(1, 2), (7, 10), (11, 12)\}$$
$$E_3[H] = \{(5, 5)\}$$

**OR event:** The semantics of “$\vee$” does not change with the context as each occurrence is detected individually.
{O (E2, [ts1, te1]) ∧ ∃t < ts1 (O (E1↓, t) ∧ ∼Oin (E3, [t+1, te1]))} ∧ (∃ e1′ [tstart, tend] | (t < tend < ts1) ∧ e1′ ∈ E1 [H])

Aperiodic event occurs whenever an event E2 occurs in the interval formed by events E1 and E3. This is formally defined as the non-occurrence of the event E3 as ∼Oin (E3, [t +1, ts1]). In order to extend this to hold for recent context this condition (∃ e1′ [tstart, tend] | (t < tend < ts1)) is added. This condition specifies that, there should not be any occurrence of the event E1 from the event history E1 [H] in the interval (t, te1).

This formal definition is explained using the example shown in the Figure 7. When event e2 occurs there are two events in the event history E1 [H]. Event e2 cannot pair with event e1 since there is an occurrence of the event e1 in the interval formed by these two events. Event e2 is paired with event e1 since there is no other event from E1 [H] has occurred in this interval. Similarly event e2 is paired with the event e1. Event e1 terminates the “A” event initiated by the event e1.

Events in recent context: {(e1, e2, e3) [4, 9], (e1, e2, e3) [4, 10]}

![Figure 7 Examples for A and A* Operators](image)

**Cumulative Aperiodic operator in Unrestricted Context:** In general, Cumulative Aperiodic event is the cumulative version of the aperiodic operator where all the events occurred in the interval formed by event E1 and E3 are accumulated. Below we illustrate how event histories can be used for the detection of (A* (E1, E2, E3), [ts1, te1]) in the unrestricted context defined in Section 3.3 using the example shown in the figure 7.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>e2</td>
<td>e1</td>
<td>e2</td>
<td>e1</td>
<td>e2</td>
<td>e1</td>
<td>e2</td>
<td>e1</td>
<td>e2</td>
<td>e1</td>
<td>e2</td>
</tr>
</tbody>
</table>

E1 [H] = {(3, 5), (4, 6)}
E2 [H] = {(1, 2), (8, 9), (7, 10), (11, 12)}

E3 [H] = {(11, 11)}

In the unrestricted context, above events generate the following pairs of events {(e1, e2, e3, e4)} [8, 10]. In this context all the events are accumulated in the interval formed by events e1 and e2.

**Cumulative Aperiodic operator in Recent Context:** The “A*” operator in recent context is formally defined as follows:

O (A (E1, E2, E3), [tsf, tel]) \(\triangleq\)

\{ ∀Ei ∈ E3 [H]
\{O (E3, [tsa, tsa]) ∧ (∃ E1' [t, t] | (t < tsa) ∧ E1' ∈ E3 [H]) \∧ \{ ∀ E1 ∈ E1 [H] \∧ ∀ E2 ∈ E2 [H] \∧ O (E2, [tsa, tef]) \∧ (∃ E2' [t, t'] | (t < tsa) ∧ E2' ∈ E2 [H]) \∧ O (E2, [tsa, tef]) \∧ (∃ E2'' [t, t''] | (t < tsa) ∧ E2'' ∈ E2 [H]) \∧ ∃ t < tel (O (E1↓, t) \∧ (∃ E1↓' [t, t'] | (t < t' < te1))))\} \∧ ∃t ∈ E1 [H]
\{O (E1, [tstart, tend]) \∧ (∃E2' [t, t] | (t < tend) ∧ E2' ∈ E2 [H]) \∧ (∃E2' [t, t] | (t < tend) \∧ E2' ∈ E2 [H]) \∧ (∃E2'' [t, t] | (t < tend) \∧ E2'' ∈ E2 [H]) \∧ ∃t < tef (O (E1↓, t) \∧ (∃ E1↓' [t, t'] | (t < t' < te1))))\} \∧ ∃t ∈ E1 [H]

The above formal definition has two cases, one to handle the case for the first occurrence of the terminator in the history (as it groups all constituent events up to that point) and the second to handle a terminator when there are other previous terminators in the history. The above definition produces the set of event occurrences in the recent context given any two histories. We will explain the formulation of the definition using the same example.

When the event e3 occurs, event histories of events E1, E2 and E3 from Figure 7 are as follows:

E1 [H] = {(3, 5), (4, 6)}
E2 [H] = {(1, 2), (8, 9), (7, 10)}
E3 [H] = {(11, 11)}

Event occurrence e1 does not have any effect on the event detection since there is no initiator. Since there are no other events in E3 [H], satisfying the condition (∃ E1' [t, t] | (t < tend) and falls into the first case of the definition. Now the condition is that accumulating all E2 events in the interval formed by events E1 and E3. But this depends on the initiator, whether e1 or e2 is the initiator. First, event e1 [3,5] is taken as the
initiator. But event $e_1^2$ [4, 6] has occurred in the interval formed by $e_1^1$ [3,5] and $e_2^2$ [8,9] and thus fails to satisfy the condition ($\exists (E_1', t') \, (t < t' < t_{sf})$). As the second option event $e_1^2$ [4, 6] is taken. This satisfies the above condition and thus acts as the initiator for this “A*” event. Thus the events in the interval [6, 11] formed by events $e_1^2$ and $e_3^1$ are accumulated and a cumulative aperiodic event is detected with events $(e_1^1, e_2^2, e_3^1, e_3^1[8,10])$.

Terms used in the formal definition are explained and their values are specified in () for this example:

$\text{tsf}$ – Start time of First $E_2$ (8)
$\text{tef}$ – End time of First $E_2$ (9)
$\text{tsl}$ – Start time of Last $E_2$ (7)
$\text{tel}$ – End time of Last $E_2$ (10)
$\text{tta}$ – Start time of $E_3$ which is after Last $E_2$ (11)
$\text{tea}$ – End time of $E_3$ which is after Last $E_2$ (11)

Below given terms are specified only in the second case:

$\text{tsb}$ – Start time of $E_3$ which is before First $E_2$
$\text{teb}$ – End time of $E_3$ which is before First $E_2$

5 Composite Event Detection Using Event Graphs

Sentinel uses an event graph for representing an event expression in contrast to other approaches such as Petri nets used by Samos and an extended finite state automata used by Compose. By combining event trees on common sub expressions, an event graph is obtained. Data flow architecture is used for the propagation of primitive events to detect composite events. All the leaf nodes in an event tree are primitive events and the internal nodes are composite events. By using event graphs, the need for detecting the same event multiple times is avoided since the event node can be shared by many events. In addition to reducing the number of detections, this approach saves substantial amount of storage space (for storing event occurrences and their parameters), thus leading to an efficient approach to detecting events.

Event occurrences flow in a bottom-up fashion. When a primitive event occurs and is detected, it is sent to its leaf node, which forwards it to the parent node (if necessary) for detecting a composite event. As we described in the previous section, introduction of parameter context makes the event detection more meaningful for many applications. In this section, we will illustrate how a composite event is detected in all parameter contexts with an illustrative example using the same set of primitive events occurring over a time line.

The same event graph is used for detecting events in all contexts on a need basis. With each node, there are 4 counters indicating whether that event should be detected in that particular context. The counter is also used to keep track of number of composite events an event participates in. When this counter reaches zero, there is no need to detect that event in that context, as there are no events dependent on that event. Consider the following occurrences of primitive events:

\[
\begin{array}{cccccccccccc}
& & & & & & & & & & & \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
e_1^1 & e_1^2 & e_2^1 & e_2^2 & e_3^1 & e_3^2 & e_1^4 & e_4^2 & e_3^3 \\
\end{array}
\]

\text{Time}

Figure 8 Event occurrences on the time line

In Figure 8, the numbers 1,2,3,4,5,...11 represent time points on the time line at which primitive events occur. If we take the primitive event $e_1^2$, it is said to occur in the time interval [2,2], and event $e_3^1$, is said to occur in the time interval [4,4]. The composite events that combine these two events occur over a time interval [2,4] where [2] is the start time and [4] are the end time of the composite event.

Figure 9 Event Graph

Figure 10 Recent Context

In Figure 9, we represent the events in terms of their occurrence times in brackets (e.g., [2,2] represents
event $e_1$ for simplicity. Composition is shown using multiple events with in a bracket (e.g., $[[1,4], [2,7]]$ represents events $e_1$, $e_1$, $e_1$ and $e_2$). Figure 9 represents the composite event ($\neg E3$) ($(E1; E2), (E1 \Delta E4)$). Leaf nodes, $E1$, $E2$, $E3$, and $E4$, represent the primitive events. NOT event is a composite event that contains AND, SEQ as its constituent events. When any two events are paired in either node $B$ or $C$, they are passed to node $A$ where the “$\neg$” event is detected. We will present the detection of events in the order of Recent, Chronicle, Continuous and Cumulative contexts. Figures 10, 11, 12 and 13 show the snapshot of the event states in the event graph at the time of event $e_4$ occurrence.

In the recent context (refer Figure 10), events $e_1$ (recent initiator) and $e_2$ are combined in the node $B$ (Sequence event) and are sent to their parent node $A$, which is a “$\neg$” event. When $e_4$ occurs, “$\Delta$” event is detected and sent up to the $A$ node; however, they do not satisfy the “$\neg$”. When the event $e_3$ occurs it combines with the event $e_1$ which is the recent initiator in the node $C$ (AND event) and is propagated to the node $A$. In node $A$, since there is an initiator waiting and there is no occurrence of the middle event $e_2$, these events $e_1$, $e_2$, $e_3$, and $e_4$ are combined and are detected as the composite event.

### Table 1 Notations used in Algorithms

<table>
<thead>
<tr>
<th>$ei$ (e.g. $e_1$, $e_2$)</th>
<th>Primitive or Composite event instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_i$ (e.g. $E_1$, $E_2$)</td>
<td>An event List that maintains the history in the chronological order of the occurrences of event $e_i$</td>
</tr>
<tr>
<td>$T_s$</td>
<td>Starting time of the event (Start Interval)</td>
</tr>
</tbody>
</table>

### 6.1 Algorithms for RECENT Context:

#### Sequence operator in Recent Context:

/* $ei$ can be recognized as coming from the left or right branch of the operator tree */

PROCEDURE $\text{seq}_{\text{recent}} (ei, \text{parameter\_list})$

- If $ei$ is the left event
  - Replace $e_1$ in the $E_1$ //most recent initiator
- If $ei$ is the right event
  - If ($E_1$ is not empty and ($t_s(e_2) > t_e(e_1)$)) //when there is an initiator in the list
    - Pass $<e_1, e_2>$ to parent with $t_s(e_1)$ and $t_e(e_2)$ //time of occurrence of the sequence event

#### Not operator in Recent Context: Whenever the right event $e_3$ is signaled then it acts as the detector for this composite event only when there is no $e_2$ has occurred between the end interval of the left event and start interval of the right event.

/* $ei$ can be recognized as coming from the left or right branch of the operator tree */

PROCEDURE $\text{not}_{\text{recent}} (ei, \text{parameter\_list})$

- If $ei$ is the left event
  - Replace $e_1$ in $E_1$ //most recent initiator
  - Delete $E_2$ // all $e_2$’s in $E_2$ should have occurred before this event $e_1$
- If $ei$ is the middle event
  - If ($E_1$ is not empty and ($t_e(e_1) \leq t_s(e_2)$))
    - Append $e_2$ to $E_2$ // since the “not” event is detected at the time of event $e_3$ occurrence
- If $ei$ is the right event
  - If ($E_1$ is not empty and ($t_e(e_1) < t_s(e_3)$)) //if $e_3$ is a sequence of $e_1$
    - If $E_2$ is not empty
      - // When there are some $e_2$’s present in $E_2$
        - For all $e_2$’s in $E_2$
          - If ($t_e(e_2) > t_s(e_3)$ or $t_s(e_2) < t_s(e_1)$) // check for non occurrence of $e_2$ in the interval formed by $e_1$ and $e_3$
            - Pass $<e_1, e_3>$ to parent with $t_s(e_1)$ and $t_e(e_3)$
          - For every $e_2$ in $E_2$
            - If ($t_e(e_1) > t_s(e_2)$)
7 Conclusions and Future Work

In this paper, we have extended the formal definitions of occurrence-based semantics to recent context. These definitions add constraints that are based on the conditions over initiators, detectors, and terminators that should be satisfied for a particular context. The semantics have been implemented using event histories providing procedural semantics. All operators have been formally defined and implemented for all the contexts (recent, continuous, and cumulative) including the unrestricted context [24]. We are in the process of extending the semantics using the disjoint characterization of composite events. This work needs to be extended to the distributed event occurrence and detection as well.

8 Acknowledgment

We would like to acknowledge the support of AF/Spawar grant (AF 26-0201-13) and the NSF grant (IIS-0112914) for this work.

9 References