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

Raman Adaikkalavan

MS Thesis CSE-2002-4
August, 2002
The members of the Committee approve the masters thesis of Raman Adaikkalavan

Sharma Chakravarthy
Supervising Professor

Ramez Elmasri

Ałp Aslandogan
SNOOP EVENT SPECIFICATION: FORMALIZATION
ALGORITHMS, AND IMPLEMENTATION USING
INTERVAL-BASED SEMANTICS

by

RAMAN ADAIKKALAVAN

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2002
ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest sincere gratitude to my advisor, Prof. Sharma Chakravarthy, for giving me an opportunity to work on such an interesting and challenging Snoop Event Specification Language and providing me great guidance and support through the course of this research work. I would also like to thank Prof. Ramez Elmasri and Dr. Alp Aslandogan for serving on my committee.

I am also grateful to Pratyush Mishra for maintaining a well-administered research environment and his commitment to work. Sincere appreciation is due to Sreekant Thirunagiri, Ganesh Gopalakrishnan, and all friends in ITLAB for their invaluable help and advice during the design of semantics. I would also like to thank all my friends for their support and encouragement.

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

Last, but not the least, I thank my parents, brother and sister’s family for their endless love and support. Without their encouragement and endurance, this work would not have been possible.

July 11, 2002
ABSTRACT

SNOOP EVENT SPECIFICATION: FORMALIZATION
ALGORITHMS, AND IMPLEMENTATION USING
INTERVAL-BASED SEMANTICS

Publication No.__________
Raman Adaikkalavan, M.S.
The University of Texas at Arlington, 2002

Supervising Professor: Sharma Chakravarthy

Snoop is an event specification language developed for expressing primitive and composite events in Event-Condition-Action rules. 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. 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 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 thesis, we formalize the occurrence of Snoop event operators and expressions using interval-based semantics for the recent context. We discuss the changes that are made to the parameter contexts that are needed for detection of Snoop operators in interval-based semantics. We present algorithms to detect all Snoop operators in the recent context and unrestricted context conforming to the interval-based semantics.
TABLE OF CONTENTS

ACKNOWLEDGMENTS ................................................................................................................. iii
ABSTRACT ....................................................................................................................................... iv
LIST OF FIGURES ........................................................................................................................ viii
LIST OF TABLES .......................................................................................................................... x

Chapter

1. INTRODUCTION ...................................................................................................................... 1

2. OPERATOR SEMANTICS IN UNRESTRICTED CONTEXT ............................................................ 4
   2.1. PRIMITIVE EVENTS ........................................................................................................ 4
   2.2. EVENT EXPRESSIONS ................................................................................................... 5
   2.3. COMPOSITE EVENTS ................................................................................................... 5
       2.3.1. Event Combinations ............................................................................................. 6
       2.3.2. Primitive events O (E [t, t']) ............................................................................... 9
       2.3.3. AND event E = O (E₁ ? E₂, [t₁, t₂]) .................................................................. 9
       2.3.4. Sequence Event E = O (E₁; E₂, [t₁, t₂]) ......................................................... 10
       2.3.5. Or Event E = O (E₁ ∨ E₂, [t₁, t₂]) ...................................................................... 11
       2.3.6. Not Event E = O (¬ (E₃) ∶ [E₁; E₂], [t₁, t₂]) .................................................. 12
       2.3.7. Aperiodic Event Operators (A, A*) ....................................................................... 13
       2.3.8. Periodic Event Operators (P, P*) ........................................................................... 15
       2.3.9. Plus Event E = O (Plus (E₁, n) [t, t]) ................................................................. 16
3. PARAMETER CONTEXTS AND OPERATOR SEMANTICS
FOR RECENT CONTEXT ................................................................. 17

3.1. PARAMETER CONTEXTS ....................................................... 17

3.2. OPERATOR SEMANTICS IN RECENT CONTEXT ....................... 19

3.3. EVENT HISTORIES ............................................................... 19

3.4. OCCURRENCE SEMANTICS IN RECENT CONTEXT .................. 20

  3.4.1. Sequence operator in Unrestricted Context ...................... 20
  3.4.2. Sequence operator in Recent Context .............................. 21
  3.4.3. Plus operator in Unrestricted Context ............................ 22
  3.4.4. Plus operator in Recent Context .................................... 23
  3.4.5. Not operator in Unrestricted Context .............................. 24
  3.4.6. Not operator in Recent Context ..................................... 24
  3.4.7. OR event ................................................................. 25
  3.4.8. Aperiodic operator in Unrestricted Context ...................... 25
  3.4.9. Aperiodic operator in Recent Context ............................ 26
  3.4.10. Cumulative Aperiodic operator in Unrestricted Context .... 27
  3.4.11. Cumulative Aperiodic operator in Recent Context .......... 28

4. COMPOSITE EVENT DETECTION USING EVENT GRAPHS .............. 30

5. ALGORITHMS AND DETAILED EXAMPLES ....................................... 37

  5.1. ALGORITHMS ................................................................. 37

    5.1.1. And Operator in Recent Context ................................. 39
    5.1.2. Sequence operator in Recent Context ......................... 40
    5.1.3. Not operator in Recent Context .................................. 40

  5.2. DETAILED EXAMPLES ...................................................... 41

    5.2.1. Sequence operator .................................................. 41
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.</td>
<td>Example events</td>
<td>2</td>
</tr>
<tr>
<td>2.1.</td>
<td>Overlapping event combinations</td>
<td>8</td>
</tr>
<tr>
<td>2.2.</td>
<td>Disjoint event combinations</td>
<td>9</td>
</tr>
<tr>
<td>2.3.</td>
<td>Example for And Operator</td>
<td>9</td>
</tr>
<tr>
<td>2.4.</td>
<td>Example for Sequence Operator</td>
<td>10</td>
</tr>
<tr>
<td>2.5.</td>
<td>Example for OR operator</td>
<td>11</td>
</tr>
<tr>
<td>2.6.</td>
<td>Example for NOT Operator</td>
<td>13</td>
</tr>
<tr>
<td>2.7.</td>
<td>Example for Aperiodic Operator</td>
<td>14</td>
</tr>
<tr>
<td>2.8.</td>
<td>Example for Periodic Operator</td>
<td>15</td>
</tr>
<tr>
<td>2.9.</td>
<td>Example for Plus Operator</td>
<td>16</td>
</tr>
<tr>
<td>3.1.</td>
<td>Examples for Sequence Operator</td>
<td>21</td>
</tr>
<tr>
<td>3.2.</td>
<td>Examples for Plus Operator</td>
<td>22</td>
</tr>
<tr>
<td>3.3.</td>
<td>Examples for Not Operator</td>
<td>23</td>
</tr>
<tr>
<td>3.4.</td>
<td>Examples for A and A* Operators</td>
<td>27</td>
</tr>
<tr>
<td>4.1.</td>
<td>Typical Event Graph</td>
<td>30</td>
</tr>
<tr>
<td>4.2.</td>
<td>Event occurrences on the time line</td>
<td>31</td>
</tr>
<tr>
<td>4.3.</td>
<td>Event Graph</td>
<td>32</td>
</tr>
<tr>
<td>4.4.</td>
<td>Recent Context</td>
<td>34</td>
</tr>
<tr>
<td>4.5.</td>
<td>Chronicle Context</td>
<td>34</td>
</tr>
<tr>
<td>4.6.</td>
<td>Continuous Context</td>
<td>35</td>
</tr>
<tr>
<td>4.7.</td>
<td>Cumulative Context</td>
<td>35</td>
</tr>
</tbody>
</table>
4.8. Detection of the same event in different contexts ................................................. 36
5.1. Recent Context ..................................................................................................... 37
5.2. Primitive event occurrences .................................................................................. 41
5.3. Event occurrences for Not Operator ..................................................................... 44
5.4. Event occurrences for A, A* operators .................................................................. 46
5.5. Event occurrences for Plus operator .................................................................... 49
5.6. Event occurrence of P, P* operators ................................................................... 51
6.1. Petri net example ................................................................................................. 55
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1. Terms used in Algorithms</td>
<td>38</td>
</tr>
</tbody>
</table>
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 operators and language proposed for event specification. Snoop [1, 2] was developed as the event specification component of the ECA rule formalism used as 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 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 for certain operators, such as Sequence.

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 \textit{end time} of an instance of $E_1$ is \textit{less} than the \textit{end time} of an instance of
E<sub>2</sub>. Composite event E₁;E₂ is *detected* at the point when the last constituent event (i.e., E₂) 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 E₁;(E₂;E₃) and E₂;(E₁;E₃). Intuitively, since ';' is sequential composition, we should expect these two to be different as E₁ strictly precedes E₂ in the first case and E₂ strictly precedes E₁ in the latter case. However, since the detection semantics is used, that subtle difference is lost depending upon the intervals over which E₁, E₂, and E₃ occur.

![Figure 1.1. Example events.](image)

Given the occurrences of E₁ [1,2] (E₁ is the event, 1 is the start interval and 2 is the end interval of event E₁), E₂ [3,4], and E₃ [5,6], as shown in the figure 1.1, they satisfy both the event expressions using the detection semantics. Both these event expressions are satisfied because the detection time (end time) of event E₁ [2] is less than the detection time (end time of last constituent event) of E₂;E₃ [6] (i.e., 2 < 6) in the first case and detection time (end time) of event E₂ [4] is less than the detection time of E₁;E₃ [6] (i.e., 4 < 6) in the second case. However, if interval-based definition is used for occurrence, then only the first expression (E₁;(E₂;E₃)) should be correctly detected and not the second one (E₂;(E₁;E₃)).
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., $2 < 3$) and in the second case, detection time (end time) of event E$_2$ [3,4] is not less than the start time of E$_1$;E$_3$ [1,6] (i.e., $4 > 1$). Galton [7] has pointed out this discrepancy between database work where detection-based semantics has been used to define semantic of the operators in contrast to work in AI where occurrence-based semantics has played a dominant role for inference than detection and hence interval semantics has been used [20, 21]. In the rest of the thesis, we present interval-based semantics of event occurrences and discuss algorithms for event detection and their implementation using event graphs.

In this thesis, we present interval-based semantics for Snoop operators for recent context drawing upon the approach presented in [7] for the general or unrestricted context. We also present algorithms and implementation of composite event detection that uses the interval-based semantics using event graphs.

This thesis is organized as follows. Chapter 2 explains Snoop operators and their semantics in the unrestricted context using interval-based semantics. Chapter 3 extends the above to recent context and explains the event consumption modes. Chapter 4 provides an illustrative example of event detection in interval-based semantics in all contexts using event graphs. Chapter 5 discusses some algorithms and implementation issues for all operators in recent context and detailed examples. Chapter 6 refers to related work on event specification without going in to 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. Chapter 7 has conclusions and future work. Algorithms for all the operators in unrestricted and recent contexts are provided in appendix A and B respectively.
2. SNOOP OPERATOR SEMANTICS IN UNRESTRICTED OR GENERAL CONTEXT

We start with a brief description of an event, an event expression, and an event modifier. Here, 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 although they occur over an interval. 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. An event expression defines an interval on the time line.

2.1. 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). Primitive events occur over a time interval and are 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 is *detected* at the end of the interval.

### 2.2. 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].

### 2.3. 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).

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

2.3.1. Event Combinations

Another aspect of event occurrences of the constituent events of a composite event is that they can be either overlapping or disjoint. “There is a basic set of mutually exclusive primitive relations that can hold between temporal intervals” [20,21]. When the events are allowed to overlap, there are thirteen possible relationships for their combination and they are shown in figure 2.1. 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 combinations are shown in figure 2.2.

In this thesis, we assume that constituent events can overlap and semantics for all the operators are given for the overlapping case. The number of events that take part in the detection of the composite event depends on the semantics of Snoop operators.

Below, we define Snoop operators intuitively first and then provide a formal definition in the unrestricted context using the interval-based semantics. The formal definitions shown below in the boxes are reproduced from [7] except for Plus. The 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 be 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 definition (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 storage overhead, which may become a problem for situation monitoring applications. Semantics of these event operators are as follows:
Figure 2.1. Overlapping event combinations.

--- → Event E₁ instance

_____ → Event E₂ instances

| → ↑E Start of the interval

| → E↓ End of the interval
2.3.2. Primitive events: \( O(\{E \in [t, t']\}) \)

Primitive events are pre-defined in the subsystem (application), \( E \) is a primitive event that occurs over the interval \([t, t']\), where \( t \) is the start time and \( t' \) is the end time of event \( E \).

\[ t_1 \quad \overline{e_1^i} \quad t \quad \overline{e_2^j} \quad t_2 \]

\[ t_1 \quad \overline{e_1^i \land e_2^j} \quad t_2 \]

Time

Figure 2.2. Disjoint event combinations.

2.3.3. AND event: \( E = O(\{E \land E_2 \} \in [t_1, t_2]) \)

Event \( E \) is the conjunction of two events \( E_1 \) and \( E_2 \), denoted by \( E_1 \land E_2 \), occurs when both \( E_1 \) and \( E_2 \) occur, irrespective of their order of occurrence over the interval \([t_1, t_2]\). Events \( E_1 \) and \( E_2 \) can overlap or they can be disjoint. We can express AND event with the "?" operator and the formal definition is as follows:

\[
O(\{E_1 \land E_2 \} \in [t_1, t_2]) \Leftrightarrow \exists_{t, t'} (t_1 \leq t \leq t_2) \land (t_1 \leq t' \leq t_2) \land \left( (O(\{E_1 \in [t_1, t]\}) \land O(\{E_2 \in [t', t_2]\})) \lor (O(\{E_1 \in [t', t_2]\}) \land O(\{E_2 \in [t_1, t]\})) \right)
\]
Figure 2.3 explains the formal definition of the "?" operator. In figure 2.3, \( e_1^i \) (an instance of the event \( E_1 \)) starts at time point \( t_1 \) and \( e_2^j \) (an instance of the event \( E_2 \)) should start after or at the point \( t_1 \) and end at or after the point \( t \). As shown in figure 2.3, \( t_1 \) is the start time of the first event and \( t_2 \) is the end time of the second event whereas the end time (t) of the event \( E_1 \) and start time (t') of the event \( E_2 \) can overlap or disjoint, but they should be in the closed interval formed by \( t_1 \) and \( t_2 \). Both \( e_1^i \) and \( e_2^j \) can start and end the occurrence (i.e., act as either initiator or terminator) of the "?" event and they can be primitive or composite events. For the event occurrences in figure 2.1, event \( E_1 \) is combined with the following \( E_2 \) occurrences for the "Δ" event to occur: 1,2,4,5,6,8,9,10,11,12,13.

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

Event \( E \) is the sequence of two events \( E_1 \) and \( E_2 \), denoted by \( E_1; E_2 \), occurs when event \( E_2 \) occurs provided event \( E_1 \) has already occurred. This implies that the end time (t) of event \( E_1 \) is guaranteed to be less than the start time (t') of event \( E_2 \). We can express the Sequence event with the "\;" operator and it is formally defined as follows:

\[
O (E_1; E_2, [t_1, t_2]) \ ? \ \exists t, t' (t_1 \leq t < t' \leq t_2 \land O (E_1, [t_1, t]) \land O (E_2, [t', t_2]))
\]

Figure 2.4. Example for Sequence Operator.
Formal definition of the “;” operator is expressed pictorially in figure 2.4. In figure 2.4, $e_1^i$, an instance of event $E_1$, starts at time point $t_1$ and ends at the time point $t$, and $e_2^j$, an instance of event $E_2$, should start and end after the point $t$. “;” event occurs in the time interval $[t_1, t_2]$, where event $e_1^i$ starts and $e_2^j$ ends the occurrence of the “;” event and they can be primitive or composite events. For the event occurrences in figure 2.1, the event $E_1$ is combined with the following occurrence of $E_2$ for the “;” event to occur: 4.

2.3.5. Or Event: $E = O (E_1 \lor E_2, [t_1, t_2])$

Event E is the disjunction of two events $E_1$ and $E_2$, denoted by $E_1 \lor E_2$, occurs when $E_1$ occurs or $E_2$ occurs. We can express the OR event with the “|” operator and it is formally defined as follows:

$$O (E_1 \lor E_2, [t_1, t_2]) \equiv O (E_1, [t_1, t_2]) \lor O (E_2, [t_1, t_2])$$

![Figure 2.5. Example for OR operator.](image-url)
Figure 2.5 explains the formal definition for the “|” that is given above. In figure 2.5, $e_1^i$ (an instance of event $E_1$) detects the “|” event that occurs over the interval $[t_1, t_3]$ and $e_2^j$ (an instance of event $E_2$) detects the event that occurs over the interval $[t_2, t_4]$. Both the events $e_1^i$ and $e_2^j$ can detect the “|” event and they can be primitive or composite events. For the event occurrences in figure 2.1, the following “\(\triangledown\)” events are detected: $1,2,3,4,5,6,7,8,9,10,11,12,13$.

In the following operators, we use the start time and end time of an event defined in the starting of section 2.3. To enable us to express this more concisely the predicate $O_{in}$ is defined as follows [7]:

\[
O_{in} (E [t_1, t_2]) \Rightarrow \exists t_1', t_2' (t_1 \leq t_1' \leq t_2' \leq t_2 \land O (E, [t_1', t_2']))
\]

2.3.6. Not Event: $E = O (\neg (E_3) [E_1; E_2], [t_1, t_2])$

Event $E$ is the not event that detects the non-occurrence of the event $E_3$ in the closed interval formed by end time of event $E_1$ and start time of event $E_2$. Not event is expressed with the “!” operator and is formally defined as follows:

\[
O (\neg(E_2)[E_1, E_3], [t_1, t_2]) \Rightarrow O (E_1 \downarrow, t_1) \land O (\uparrow E_3, t_2) \land \neg O_{in} (E_2, [t_1, t_2])
\]

Event $E_1$ instance $e_1^i$ ends at time point $t_i$, and event $E_3$ instance $e_3^j$ starts after or at the point $e_1^i$ occurred and event $E_2$ should not occur in the closed interval $[t_1, t_2]$ defined by $e_1^i$ and $e_3^j$ as shown in figure 2.6 (Only whole occurrence of $E_2$ is considered). The following figure 2.6 explains the formal definition of the “!” operator.
Figure 2.6. Example for NOT Operator.

For this Snoop event operator, for the event $E_1$, the possible combinations of occurrences of $E_2$ is only 4 (from figure 2.1), since there is no occurrence of $E_3$ in the interval formed by $E_1 \downarrow$ and $E_2 \uparrow$.

2.3.7. Aperiodic Event Operators ($A$, $A^*$)

An aperiodic operator allows one to express the occurrences of an aperiodic event within a closed time interval. There are two variants of this event specification, a cumulative variant and a non-cumulative variant.

The non-cumulative aperiodic event “$E$” is expressed as $E = O \ (A \ (E_1, \ E_2, \ E_3), \ [t_1, \ t_2])$. Event $E$ is an aperiodic event that is signaled each time event $E_2$ occurs within the time interval formed by the end time of event $E_1$ and start time of event $E_3$. An aperiodic event is expressed using the “$A$” operator. On the other hand, the cumulative aperiodic event “$E$” is expressed as $E = O \ (A^* \ (E_1, \ E_2, \ E_3), \ [t_1, \ t_2])$. Event $E$ is an aperiodic cumulative event that accumulates all the occurrences of event $E_2$ within the closed interval formed by $E_1$ and $E_3$ and it occurs when event $E_3$ occurs. A cumulative aperiodic event is expressed using the “$A^*$” operator.
The occurrence time of the event “A” is the occurrence time for event 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 [7]. The formal definition of an “A” operator is as follows:

\[ O(\text{A}(E₁, E₂, E₃), [t₁, t₂]) \Rightarrow \exists t < t₁ (O(E₁↓, t) \land \neg O_{in}(E₃, [t₁, t₂])) \]

In figure 2.7, shown below, we can see that event \( e₁^i \) (an instance of event E₁) ends at time point t and event \( e₂^k \) (an instance of event E₂) should be a sequence to \( e₁^i \) (i.e., start after \( e₁^i \) has happened). No E₃ should start after \( e₁^i \) has occurred, and finish before or at the point \( e₂^i \) finishes (Only whole occurrence of E₃ is taken into consideration).

![Diagram](image)

Figure 2.7. Example for Aperiodic Operator.

An aperiodic event can occur zero or more times (Zero times when E₂ does not occur in the interval or when no interval exists for the definitions of E₁ and E₃).
2.3.8. Periodic Event Operators (P, P*)

A periodic event is a temporal event that occurs periodically. Periodic event also has two variants similar to the aperiodic event.

A periodic event “E” is denoted as \( E = O (P (E_1, [t], E_3), [t, t]) \). While \( E_1 \) and \( E_3 \) can be any type of event, event \( E_2 \) “[t]” should be a time string (temporal event). The Periodic event occurs whenever the time string \([t]\) occur in the time interval formed by the end time of the event \( E_3 \) and start time of the event \( E_1 \) and is denoted by “P”. P has a cumulative variant “P*” expressed as \( E = O (P* (E_1, [t], E_3)) \). 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 it available when P* occurs. Formal definition of a “P” operator is as follows:

\[
O (P (E_1, n, E_3), [t]) \leftarrow t' < t \exists i \in \mathbb{Z}^+ (t = t' + ni \land O (E_1 \downarrow, t') \land \neg O \text{in} (E_3, [t'+1, t]))
\]

The formal definition of a “P” operator is shown pictorially in figure 2.8. An instance of event \( E_1 \), \( e_1^i \) ends at time point \( t \), an instance of event \( E_2 \), \( e_2^j \) should be \((t + n*i)\) (t-> end time for \( E_1 \), \( n*i \) -> time interval) and \( E_3 \) should start after \( e_1^i \) occurred (at time point \( t \)) and finish before or at the time point \([t_2]\) \( e_2^j \) finishes (Only whole occurrence of \( E_3 \) is taken into consideration).

![Figure 2.8. Example for Periodic Operator.](image_url)
2.3.9. Plus Event: $E = O\left(\text{Plus} \left( E_1, n \right), [t, t] \right)$

Plus event is expressed with the ‘Plus’ operator and is used to specify a relative time event [23]. A “Plus” operator combines two events $E_1$ and $E_2$ where $E_1$ can be any type of event and event $E_2$ (‘n’) is the terminator event and it is a time string $[t]$. The Plus event occurs after time $[t]$, after the event $E_1$ occurs. Below, we give the formal definition for the Plus operator for the unrestricted context. In the definition, $E_1$ is the initiator, and “n” is the terminator.

\[
O\left(\text{Plus} \left( E_1, n \right), [t, t] \right) \iff \exists t' < t \left( O \left( E_1 \downarrow, t' \right) \wedge t = t' + n \right)
\]

In figure 2.9, $e_1^i$ starts Plus event specified by the time string $[t]$ and the “Plus” event occurs at end time $(e_1^i)^+ [t]$.

![Figure 2.9. Example for Plus Operator.](image-url)
3. PARAMETER CONTEXTS AND OPERATOR SEMANTICS FOR RECENT CONTEXT

3.1. 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 (event consumption modes): 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.

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 initiator and the corresponding terminator are deleted, and the constituent events (except the initiator and terminator) that can be used in future events are preserved. Future events are those 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.

**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.

### 3.2. 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. Below, “O” represents the occurrence-based Snoop semantics.

### 3.3. 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 history (chronological with respect to the end time) of event occurrences up to a given point in time. Suppose $e_i$ is an event instance of type $E_i$ then $E_i[H]$ represents the event history that stores
all the instances of the event $E_i$ (namely $e_i$). In order to extend these definitions to parameter contexts the following notation is used.

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

$t_{si} –$ Starting time of an event $e_i$

$t_{ei} –$ Ending time of an event $e_i$

Below, we first describe the history-based event occurrences intuitively before defining them formally.

3.4. Occurrence Semantics in Recent Context

3.4.1. 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.

$E_1[H] = \{(3, 5), (4, 6), (8, 9)\}$

$E_2[H] = \{(1, 2), (7, 10), (11, 12)\}$

For the events shown in figure 3.1, $(E_1; E_2)$ generates the following pairs of events in the unrestricted context: $\{(e_1^1, e_2^2) [3,10], (e_1^2, e_2^2) [4, 10], (e_1^1, e_2^3) [3,12], (e_1^2, e_2^3) [4, 12], (e_1^3, e_2^3) [8, 12]\}$. 
3.4.2. Sequence operator in Recent Context

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

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

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 figure 3.1.

\[ E_1 [H] = \{(3, 5), (4, 6), (8, 9)\} \]
\[ E_2 [H] = \{(1, 2), (7, 10), (11, 12)\} \]
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 \( e_2^1 \) \([4,6]\) in the interval formed by the end time of event \( e_1^1 \) \([3,5]\) and end time of event \( e_2^2 \) \([7,10]\), which does not satisfy the condition given above. Event \( e_2^2 \) does not detect the sequence event in the recent context, since the recent initiator is \( e_1^3 \). 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: \{\((e_1^3, e_2^3)\) \([8,12]\)\}.

3.4.3. 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 \((\text{Plus} (E_1, n) \,[t,t])\). By the definition of the Plus event the start time and end time are the same. For example, Plus event \((\text{Plus} (E_1, 4))\) is taken, which is detected after 4 units after the occurrence of event \( E_1 \), to explain the unrestricted and recent context.

For the events shown in figure 3.2, Plus event defined in section 3.3 generates the following pairs of events in the unrestricted context: \{\((e_1^1, 4)\) \([9,9]\), \((e_1^2, 4)\) \([10,10]\), \((e_1^3, 4)\) \([16,16]\)\}.

![Figure 3.2. Examples for Plus Operator.](image-url)
3.4.4. Plus operator in Recent Context

Below, in the formal definition for the Plus event in the recent context, event \( E_1 \) is assumed to end at a time point \([t']\) and Plus event at the time point \([t]\. \) “Plus” event is an absolute time event so that it occurs in the interval \([t, t]\). Plus event occurs in the recent context when ever there is no other instance of \( E_1 \) event from \( E_1 [H] \) occurs in the interval formed by \([t']\) and \([t]\). This is given as the condition \( \exists t' < t (O (E_1 \downarrow, t') \land t = t' + n \land (\forall (E_1 \downarrow, t'') | (t' < t' \leq t)) \).

Formal definition given above is explained with the events shown in figure 3.2. Event \( e_1^1 \) initiates a Plus event at the time point \([5]\). When the event \( e_1^2 \) 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 \( e_1^3 \) detects a Plus event at the time point \([16]\).

Events in recent context: \( \{ (e_1^2, 4) [10,10], (e_1^3, 4) [16,16]\).
3.4.5. Not operator in Unrestricted Context

Not 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 "¬" 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.

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

In the unrestricted context, above events shown in figure 3.3 generate the following pair of events \{(e_1^2, e_2^2) [4, 10], (e_1^2, e_2^3) [4, 12], (e_1^3, e_2^3) [8, 12]\}.

3.4.6. Not operator in Recent Context

Not operator in recent context is formally defined as follows:

\[
O (\neg (E_3)[E_1, E_2], [t_{s1}, t_{e2}]) \equiv
\forall e_1 \in E_1 [H] \land \forall e_2 \in E_2 [H] \land \forall e_3 \in E_3 [H] \land
\{(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 \neg O_{in} (e_3, [t_{c1}, t_{c2}]))
\land (? e_1'[t_{start}, t_{end}] \mid (t_{c1} < t_{end} \leq t_{e2}) \land e_1' \in E_1 [H])
\)

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 "¬" operator in the recent context is explained below using the example shown in figure 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 $e_2^1$ does not detect any event since the event history $E_1 [H]$ is empty. Event $e_1^1$ initiates “$\neg$” event in the recent context, and is terminated by event $e_3^1$. “$\neg$” event in the recent context is initiated by event $e_1^2$. This event is terminated by the occurrence of the event $e_3^3$, which acts as the most recent initiator. Event occurrence $e_3^2$ does not pair with the initiator $e_3^3$ since it does not satisfy the condition $(t_{s_1} < t_{e_1} < t_{s_2} \leq t_{e_2})$, where as the event $e_3^3$ detects a recent event with the initiator $e_3^3$ 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 $(t_{s_1} < t_{e_1} < t_{s_2} \leq t_{e_2})$ and $(? e_1^1 [t_{start}, t_{end}] \cap (t_{e_1} < t_{end} \leq t_{e_2}) \cap e_1^1 \in E_1 [H])$.

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

3.4.7. OR event

The semantics of ‘$\lor$’ does not change with the context as each occurrence is detected individually. Simultaneous occurrences are not considered in this thesis.

3.4.8. 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.

$E_1 [H] = \{(3, 5), (4, 6)\}$

$E_2 [H] = \{(1, 2), (7, 9), (8, 10), (11, 12)\}$

$E_3 [H] = \{(5, 5)\}$

In the unrestricted context, above events shown in figure 3.4 generate the following pair of events $\{(e_1^1, e_2^2) [3, 9], (e_1^2, e_2^2) [4, 9], (e_1^3, e_2^3) [3, 10], (e_1^2, e_2^3) [4, 10]\}$.
3.4.9. Aperiodic operator in Recent Context

The “A” operator in recent context is formally defined as follows:

\[
O (A (E_1, E_2, E_3), [t_{s1}, t_{e1}]) \equiv \\
\{ \forall e_1 \in E_1 [H] \land \forall e_2 \in E_2 [H] \land \forall e_3 \in E_3 [H] \land \\
O (E_2, [t_{s1}, t_{e1}]) \land \exists t < t_{s1} (O (E_1 \downarrow, t) \land \neg O_{in} (E_3, [t + 1, t_{e1}])) \land (? e_1' [t_{start}, t_{end}] \mid (t < t_{end} \leq t_{e1}) \land e_1' \in E_1 [H]) \land \\
\}
\]

Aperiodic event occurs whenever an event $E_2$ occurs in the interval formed by events $E_1$ and $E_3$. This is formally defined as the non-occurrence of the event $E_3$ as $\neg O_{in} (E_3, [t + 1, t_{e1}])$. In order to extend this to hold for recent context, the condition $(? e_1' [t_{start}, t_{end}] \mid (t < t_{end} \leq t_{e1}))$ is added. This condition specifies that, there should not be any occurrence of the event $E_1$ from the event history $E_1 [H]$ in the interval $(t, t_{e1})$.

This formal definition is explained using the example shown in figure 3.4. When event $e_2^2$ occurs there are two events in the event history $E_1 [H]$. Event $e_2^2$ cannot pair with event $e_1^1$ since there is an occurrence of the event $e_1^2$ in the interval formed by these two events. Event $e_2^2$ is paired with event $e_1^2$ since there is no other event from $E_1 [H]$ has occurred in this interval. Similarly, event $e_3^3$ is paired with the event $e_2^2$. Event $e_3^1$ terminates the “A” event initiated by the event $e_1^2$.

Events in recent context: \{(e_1^2, e_2^2) [4, 9], (e_1^2, e_2^3) [4, 10]\}
3.4.10. 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 \( E_1 \) and \( E_3 \) are accumulated. Below we illustrate how event histories can be used for the detection of \((A^*(E_1, E_2, E_3), [t_{s1}, t_{e1}])\) in the unrestricted context defined in section 3.3 using the example shown in figure 3.4.

\[ E_1 [H] = \{(3, 5), (4, 6)\} \]

\[ E_2 [H] = \{(1, 2), (8, 9), (7, 10), (11, 12)\} \]

\[ E_3 [H] = \{(11, 11)\} \]

In the unrestricted context, above events generate the following pairs of events \((e_1^1, e_1^2, e_2^2, e_2^3, e_3^1) [8, 10]\). In this context all the events are accumulated in the interval formed by events \( e_1^1 \) and \( e_3^1 \).
3.4.11. Cumulative Aperiodic operator in Recent Context

The “A*” operator in recent context is formally defined as follows:

\[ O (A (E_1, E_2, E_3), [t_{sf}, t_{el}]) \]

\[
\forall E_3 \in E_3 [H] \\
\{ O (E_3, [t_{sa}, t_{ca}]) \land (\exists E_3' [t_s, t_e] \land (t_e < t_{ca}) \land E_3' \in E_3 [H]) \land (\forall E_1 \in E_1 [H] \land \forall E_2 \in E_2 [H] \land (O (E_2, [t_{sf}, t_{el}]) \land (\exists E_2' [t_s', t_e'] \land (t_e' < t_{sf}) \land E_2' \in E_2 [H]) \land (O (E_2, [t_{sf}, t_{el}]) \land (\exists E_2'' [t_s'', t_e''] \land (t_{el} < t_{sf})) \land E_2'' \in E_2 [H]) \land \exists t < t_{sf} (O (E_1 \downarrow, t) \land (\exists E_1' \downarrow, t') \land (t < t' \leq t_{el})))) \} \\
\lor \forall E_3 \in E_3 [H] \\
\{ O (E_3, [t_{sa}, t_{ca}]) \land (\exists E_3' [t_{sb}, t_{eb}] \land (t_{eb} < t_{ca}) \land E_3' \in E_3 [H]) \land (\exists E_3'' [t_{s3'}, t_{e3'}] \land (t_{e3'} > t_{eb}) \land (t_{e3'} < t_{ca}) \land E_3'' \in E_3 [H])) \land (O (E_2, [t_{sf}, t_{el}]) \land (t_{el} < t_{sf}) \land (\exists E_2' [t_{s'}, t_{e'}] \land (t_{e'} \leq t_{el}) \land E_2' \in E_2 [H]) \land (O (E_2, [t_{sf}, t_{el}]) \land (t_{el} < t_{sf}) \land (\exists E_2'' [t_{s''}, t_{e''}] \land (t_{el} < t_{s''}) \land E_2'' \in E_2 [H]) \land \exists t_{sb} = t < t_{sf} (O (E_1 \downarrow, t) \land (\exists E_1' \downarrow, t') \land (t < t' \leq t_{el})))) \} 
\]

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 \( e_3^1 \) occurs, event histories of events \( E_1, E_2 \) and \( E_3 \) from figure 3.4 are as follows:

\[ E_1 [H] = \{(3, 5), (4, 6)\} \]
\[ E_2 [H] = \{(1, 2), (8, 9), (7, 10)\} \]
E₃ [H] = {(11, 11)}

Event occurrence e₂¹ does not have any effect on the event detection since there is no initiator. Since there are no other events in E₂ [H], satisfying the condition (? E₂' [tₛ₁, tₑ] | (tₑ < tₑa)) and falls into the first case of the definition. Now the condition is that accumulating all E events in the interval formed by events E₁ and E₂. But this depends on the initiator, whether e₁¹ or e₁² is the initiator. First, event e₁¹ [3, 5] is taken as the initiator. But event e₁² [4, 6] has occurred in the interval formed by e₁¹ [3, 5] and e₂² [8, 9] and thus fails to satisfy the condition (? (E₁', t') | (t < t' ≤ tₑ)). As the second option event e₁² [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 e₁² and e₃¹ are accumulated and a cumulative aperiodic event is detected with events (e₁¹, e₂², e₂³, e₃¹ [8, 10]).

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

- tₛ₂ – Start time of First E₂ (8)
- tₑ₂ – End time of First E₂ (9)
- tₛ₃ – Start time of Last E₂ (7)
- tₑ₃ – End time of Last E₂ (10)
- tₛ₃ – Start time of E₃ which is after Last E₂ (11)
- tₑ₃ – End time of E₃ which is after Last E₂ (11)

Below given terms are specified only in the second case:

- tₛ₃ – Start time of E₃ which is before First E₂
- tₑ₃ – End time of E₃ which is before First E₂
4. COMPOSITE EVENT DETECTION USING EVENT GRAPHS

Sentinel uses an event graph (figure 4.1) 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 detect events.

Figure 4.1. Typical Event Graph.

Event occurrences flow in a bottom-up fashion. In figure 4.1, leaf nodes E1 and E2 represents primitive events and the internal node represent the “^” composite event. 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 chapter, 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:

\[
\begin{align*}
1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 & \quad 9 & \quad 10 & \quad 11 \\
 e_1^1 & \quad e_1^2 & \quad e_3^1 & \quad e_2^1 & \quad e_4^1 & \quad e_1^3 & \quad e_2^2 & \quad e_3^2 & \quad e_1^4 & \quad e_4^2 & \quad e_1^5 \\
\end{align*}
\]

**Figure 4.2.** Event occurrences on the time line.

In figure 4.2, 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 $e_2^2$, it is said to occur in the time interval [2,2], and event $e_1^1$, 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.
In figures 4.3–4.7, we represent the events in terms of their occurrence times in brackets (e.g., [2,2] represents event e\_1^2) for simplicity. Composition is shown using multiple events with in a bracket (e.g., [[1,4], [2,7]] represents events e\_1, e\_2, e\_1^2, and e\_2^2). Figures 4.3–4.7 represent the composite event (¬E3) ((E1; E2), (E1 ? 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 “¬” event is detected. We will present the detection of events in the order of Recent, Chronicle, Continuous and Cumulative contexts. Figures 4.4–4.7 show the snapshot of the event states in the event graph at the time of event e\_4^2 occurrence.

![Event Graph](image)

**Figure 4.3. Event Graph.**

In the recent context (refer figure 4.4), events e\_1^2 (recent initiator) and e\_2^1 are combined in the node B (Sequence event) and are sent to their parent node A, which is a “¬” event. When e\_1 occurs, “?” event is detected and sent up to the node A; however, they do not satisfy the “¬”. When the event e\_3 occurs it combines with the event e\_4^1 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 E3, these events e1^2, e2^1, e4^1 and e1^3 are combined and are detected as the composite event.

In the Chronicle context (refer figure 4.5), events e1^1 and e2^1 are combined in the node B and are passed to the node A. The pair e1^1, e2^1 is said to be the oldest pair where e2^1 pairs with e1^1, the oldest initiator that is already present in the node B. In the same way, the events e1^2 and e2^2 are combined and sent to node A. Nevertheless, all the events that occur after that do not make any pair in the node C in order to detect a “¬” event in the node A.

In the Continuous context (refer figure 4.6), events e1^1 and e1^2 is paired with the event e2^1 since one terminator may detect one or more initiators. When the event e2^2 occurs, it pairs with the event e3^2. The occurrence of event e2^2 terminates the events e1^3 and e1^4 in the node C. When these are sent to node A, the events (e1^1, e2^1, e1^3, e2^2) and (e1^2, e2^1, e1^3, e4^2) are detected. But (e1^4, e4^2) it is not paired with the events (e1^3, e2^2) since there is an occurrence of the middle event e3^2 in between these events. The initiator pair cannot start anymore event detection, because of the occurrence of middle event e3^2 and all the events are removed.

In the Cumulative context (refer figure 4.7), events (e1^1, e1^2, e2^1) are paired together and accumulated as a single event in the node B in contrast to the two events that are detected in the continuous context and passed to the node A. In the node C, when the event e1^2 occurs it is paired with the events (e1^3, e1^4) and it is accumulated as a single and sent to the node A where this event is paired with the event (e1^1, e2^1, e1^4) that is already present. But this event (e1^3, e1^4, e2^1) is not paired with events (e1^3, e2^2) since there is an occurrence of the middle event e3^2 in between these events. The initiator pair cannot start anymore event detection, because of the occurrence of middle event e3^2 and all the events are removed.

Figure 4.8, explains the different combinations of events in different parameter contexts with the event occurrences shown over the time line. It explains the initiator,
terminator for the combinations of events shown in the event graph. Figure 4.8 summarizes the explanations of the examples given in figures 4.4–4.7.

Figure 4.4. Recent Context.

Figure 4.5. Chronicle Context.
Figure 4.6. Continuous Context.

Figure 4.7. Cumulative Context.
Figure 4.8. Detection of the same event in different contexts.
5. ALGORITHMS AND DETAILED EXAMPLES

5.1. Algorithms

Semantics of the event operators in recent context are defined using the event history in chapter 4. In this chapter, we will describe an implementation that detects events according to the interval-based semantics. In the way ECA rules are used for monitoring situations, events occur over a time line and are sent to the event detector. All events in the form of an event history are not submitted to the event detector. In fact, as part of event detection, the event detector at any point sees only a partial history in time. Algorithms are presented below to detect the events according to the interval semantics although they do not see the complete history at any given point in time.

Figure 5.1. Recent Context.
Table 5.1. Terms used in Algorithms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_i$</td>
<td>Primitive or Composite event instance</td>
</tr>
<tr>
<td>$E_i$</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>
<tr>
<td>$t_e$</td>
<td>Ending time of the event (End Interval)</td>
</tr>
<tr>
<td>HEAD</td>
<td>Head of the Event Occurrence List $E_i$</td>
</tr>
<tr>
<td>TAIL</td>
<td>Tail of the Event Occurrence List $E_i$</td>
</tr>
<tr>
<td>EarliestStartTime of $E_i$</td>
<td>The event which has the earliest start time in $E_i$</td>
</tr>
<tr>
<td>EarliestEndTime of $E_i$</td>
<td>The event which has the earliest end time in $E_i$</td>
</tr>
<tr>
<td></td>
<td>($Head$ of the $List$ always)</td>
</tr>
<tr>
<td>LatestStartTime of $E_i$</td>
<td>The event which has the latest start time in $E_i$</td>
</tr>
<tr>
<td>LatestEndTime of $E_i$</td>
<td>The event which has the latest end time in $E_i$ ($Tail$ of the $List$ always)</td>
</tr>
<tr>
<td>EventID</td>
<td>ID associated with an event in the case of temporal events</td>
</tr>
<tr>
<td>TimeString</td>
<td>The time interval specified in the temporal events</td>
</tr>
</tbody>
</table>
5.1.1. And Operator in Recent Context

We will explain the and_recent algorithm using figure 5.1 and the primitive events occurrences from figure 5.2.

PROCEDURE and_recent (ei, parameterlist)
/* ei can be recognized as coming from the left or right branch of the operator tree */
/* E1 and E2 have at most 2 event instances in them*/
If ei is the left event
  If (E2 is not empty and (t_s(e2) ≤ t_s(e1)) and (t_e(e2) ≤ t_e(e1)))
    Pass <e2, e1> to the parent with t_s(e2) and t_e(e1)
    Replace e1 in E1 with ei
If ei is the right event
  If (E1 is not empty and (t_s(e1) ≤ t_s(e2)) and (t_e(e1) ≤ t_e(e2)))
    Pass <e1, e2> to the parent with t_s(e1) and t_e(e2)
    Replace e2 in E2 with ei

When the first event e_1 occurs at the time interval [1,1] it is stored in the E1 list since there is no event in the E2 list. When the second event e_2 occurs at the time interval [2,2] it replaces the event e_1 in the E1 list since there is no event in E2 list and this is the recent event compared to the e_1. When the event e_4 occurs at the time interval [5,5] it is checked with the events in the E1 list since it is not empty. The condition that is being checked is (t_s(e1) ≤ t_s(e2)) and (t_e(e1) ≤ t_e(e2)). When we substitute the values which we got above it will be (t_s(2) ≤ t_s(5)) and (t_e(2) ≤ t_e(5)). Since the condition turns out to be true the next statement Pass <e1, e2> to the parent with t_s(e1) and t_e(e2) is executed.

Thus, we will Pass <e_1^2, e_1^1> to the parent with t_s(2) and t_e(5) which implies the AND event is detected in a recent context with the composite event <e_1^2, e_1^1> over the time interval [2,5]. Once the event detection is done, this event e_2 is replaced with the event that is already present in the E2 list, since this will be the recent initiator for the incoming e_1 events.
5.1.2. Sequence operator in Recent Context

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

PROCEDURE seq_recent (ei, parameter_list)
   If ei is the left event
      Replace e1 in the E1 with ei //most recent initiator
   If ei is the right event
      If (E1 is not empty and (t_s(e2) > t_e(e1)))/when there is an initiator in the list
         Pass <e1, e2> to parent with t_s(e1) and t_e(e2) //time of occurrence of the sequence event

5.1.3. Not operator in Recent Context

Whenever the right event e3 is signaled then it acts as the detector for his composite event only when there is no e2 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 not_recent (ei, parameter_list)
   If ei is the left event
      Replace e1 in E1 //most recent initiator
      Delete E2 // all e2’s in E2 should have occurred before this event e1
   If ei is the middle event
      If (E1 is not empty and (t_e (e1) ≤ t_s (e2)))
         Append e2 to E2 // since the “not” event is detected at the time of event e3 occurrence
   If ei is the right event
      If (E1 is not empty and (t_e (e1) < t_s (e3))) //if e3 is a sequence of e1
         For all e2’s in E2
            If (t_e (e2) > t_s (e3) or t_s (e2) < t_s (e1))
               // Check for non occurrence of e2 in the interval formed by e1 and e3
               Pass <e1, e3> to parent with t_s (e1) and t_e (e3)
            Delete e2 from E2 // e2’s that have occurred before this recent initiator if any
         Else
            Pass <e1, e3> to the parent with t_s (e1) and t_e (e3) // When there no e2’s in E2
      Delete E1
5.2. Detailed Examples

For explaining the binary operators (Sequence, And, Or), following event occurrences are taken. For easy understanding, we will take all the events, as primitive events so that start and end time are the same. Let the set of event occurrences be \( (e_1^1 [0,0], e_1^2 [1,1], e_2^1 [2,2], e_2^2 [3,3], e_1^3 [4,4]) \).

![Figure 5.2. Primitive event occurrences.](image-url)

5.2.1. Sequence operator

Sequence (";;") is a binary Snoop operator. Based on the event definition, we will explain how the event \((E_1; E_2)\) is detected in all contexts.

**Recent:** In recent context, a recent initiator is used to initiate a sequence event until a new initiator occurs and there is only detector and no terminator. So for the above event occurrences, when event \( e_1^1 [0,0] \) occurs, it acts as the recent initiator. When the next occurrence of the event \( E_1 \) (i.e., \( e_1^2 [1,1] \)) occurs, it acts as the recent initiator. Event \( e_2^1 \) occurs in the interval \([2,2]\) that is a sequence of the event \( e_1^2 \), thus detecting an ";;" event in the interval \([1,2]\) with the events \((e_1^2 \text{ and } e_2^1)\). Since there are no terminators, event \( e_1^1 \) continues to initiate the next event. Event \( e_2^2 \) occurrence in the interval \([3,3]\) detects a ";;" event with the events \((e_1^2 \text{ and } e_2^2)\) over the interval \([1,3]\). Next occurrence of event \( E_1 \) (i.e., \( e_1^3 [4,4] \)) makes it as the new initiator and initiates a ";;" event.
**Chronicle:** In chronicle context, the initiator (oldest) and terminator (oldest) pair should be unique and they are paired in the chronological order. Event $e_1^1$ and $e_1^2$ occurs in the interval [0,0] and [1,1] respectively. When the event $e_2^1$ occurs in the interval [2,2], it acts as the terminator and it pairs with the event $e_1^1$, since these two events form the oldest pair to detect the “;” event over the interval [0,2]. After this detection, event $e_1^2$ acts as the initiator for the next “;” event since this is the oldest event currently in the event $E_1$ list. Event $e_2^2$ detects the “;” event over the interval [1,3] as it pairs with the event $e_1^2$.

**Continuous:** In this context, a terminator can terminate more than one initiator and detect more number of events with respect to the initiator. When the event $e_2^2$ occurs in the interval [2,2], it terminates both the initiators ($e_1^1$ and $e_1^2$) that is already present in the list $E_1$, thus detecting two “;” events (($e_1^1$, $e_2^2$) and ($e_1^2$, $e_2^2$)) over the interval [0,2] and [1,2]. Event $e_2^2$ is not paired with any events, since there are no initiators.

**Cumulative:** In this context, a terminator can terminate more than one initiator and detect only one event that cumulates all the events between the earliest initiator and the terminator. Event $e_2^2$ occurrence in the interval [2,2] terminates both the initiators ($e_1^1$ and $e_1^2$) that is already present in the list $E_1$, thus detecting a “;” event ($e_1^1$, $e_1^2$, $e_2^2$) over the interval [0,2]. In this context, according to the context definition the terminator and the initiator that takes part in the detection are deleted after the detection.

The event pairs are given below as the summary of the above explanation

Recent: ($($e_1^2$, e_2^1$) [1,2], ($e_1^2$, $e_2^2$) [1,3])

Chronicle: ($($e_1^1$, e_2^1$) [0,2], ($e_1^2$, $e_2^2$)[1,3])

Continuous: ($($e_1^1$, e_2^1$) [0,2], ($e_1^2$, e_2^1$) [1,2])

Cumulative: ($($e_1^1$, e_1^2$, e_2^2$) [0,2])
5.2.2. And operator

And ("Δ") is a binary snoop operator and we will explain how this event is detected using the composite event (E₁ ∆ E₂).

Recent: In the recent context, there are only detectors in the case of “Δ” operator. Thus, an initiator will initiate the events until a new initiator occurs. Considering the same event occurrences shown in figure 5.2, when the event e₁ occurs in the interval [0,0] it starts the “Δ” event. Event e₂ occurrence replaces the event e₁ as the initiator. When the event e₂ occurs it detects an “Δ” event occurrence over the interval [1,2]. Now both the lists E₁ and E₂ contain one element each. If the next event occurrence is an instance of event E₂, then the element in the list E₁ acts as the initiator and vice versa. Event e₂ occurrence in the interval [3,3] replaces the event e₂ in the E₂ list as the initiator and detects an “Δ” event over the interval [1,3]. When the event e₃ occurs, it detects a “Δ” event over an interval [3,4] where e₂ is the initiator and it also replaces the event e₁ in the event E₁ list.

Chronicle: In this context, the initiator and the terminator pair is unique and it is paired in the chronological order. At the time of occurrence of event e₁, event e₁ is paired with e₂ and an “Δ” event is detected over an interval [0,2] and both the events are deleted. When the event e₂ occurs, it is paired with the event e₂ over the interval [1,3] to detect an “Δ” event and are deleted. All the events paired are in the chronological order of occurrence.

Continuous: A terminator can terminate more than one initiator and can detect more than one event in the continuous context. When the event e₁ occurs, it terminates the events e₁ and e₂ and detects two “Δ” events over the interval [0,2] and [1,2] with the event pairs (e₁, e₂) and (e₁, e₂) respectively. When the event e₂ occurs it is kept in the E₂ list. When the event e₃ occurs, it is paired with the event e₂, which acts as the initiator for the “Δ” event detected over the interval [3,4].
Cumulative: This is same as the continuous context, except that there is only one “Δ” event detected when the event \( e_2^1 \) occurs which contains events \( (e_1^1, e_1^2, e_2^1) \) as the constituent events and is detected over the interval \([0,2]\). When the event \( e_1^3 \) occurs, it is paired with the event \( e_2^2 \) that has occurred in the interval \([3,3]\) and detects a “Δ” event in the interval \([3,4]\).

The event pairs are given below as the summary of the above explanation

Recent: \((e_1^2, e_2^1) [1,2], (e_1^2, e_2^2) [1,3], (e_2^2, e_1^3) [1,3]\))

Chronicle: \((e_1^1, e_2^1) [0,2], (e_1^2, e_2^2) [1,3]\))

Continuous: \((e_1^1, e_2^1) [0,2], (e_1^2, e_2^1) [1,2], (e_2^2, e_1^3) [3,4]\))

Cumulative: \((e_1^1, e_1^2, e_2^1) [0,2], (e_2^2, e_1^3) [3,4]\))

5.2.3. Or operator

Binary snoop operator “∇” is detected whenever an event in the event expression occurs. If we take the above event occurrences then the composite event \((E_1 \lor E_2)\) is detected whenever any one of these events occur. So for the above event occurrences the “∇” event is detected as \((e_1^1 [0,0], e_2^2 [1,1], e_1^2 [2,2], e_2^2 [3,3], e_1^3 [4,4])\). This implicitly means that all the contexts produce the same events in case of the “∇” event.

![Figure 5.3. Event occurrences for Not Operator.](image-url)
5.2.4. Not operator

Not ("¬") operator is a ternary operator, but it behaves as a binary Snoop operator, since it detects the non-occurrence of the second event. We will explain the "¬" event (! (E₁, E₂, E₃)), using the event occurrences shown in figure 5.3.

Recent: Recent initiators (E₁) detect the non-occurrence of the event E₂ before the event E₃ occurs. Since the "¬" operator behaves like the binary Snoop operator, this contains only detectors. Occurrence of the event e₁², replaces the event e₁¹ as the initiator. When the event e₃¹ occurs it detects the non-occurrence of the event E₂ ("¬" event) in the interval formed by the events E₁ and E₃ (i.e., [1,2]). Event e₁³ occurrence replaces the event e₁² as the initiator and it initiates the next interval. Event e₃² occurrence does not produce any event since there are two occurrences of the event E₂. So the event e₃¹ deletes the events e₁², e₂² and e₁³, since e₁³ cannot detect any non-occurrence of E₂.

Chronicle: At the time when event e₃¹ occurs, there are two events present in the event list E₁ and there are no occurrences of event E₂. Thus, this event e₃¹ detects the non-occurrence in the interval [0,2] and pairs with e₁¹. When the event e₈² occurs there are two events in both E₁ and E₂ list, and there are occurrences of E₂ in the interval formed by (e₁², e₃²) as well as (e₁³, e₃²). Thus, the occurrence of the event e₈² does not detect "¬" event occurrence and it deletes the lists E₁ and E₂ since the events in the list E₃ cannot start any "¬" event.

Continuous: Occurrence of the event e₅¹ detects the non-occurrence of the event E₂ in the interval formed by (e₁¹, e₃¹) and (e₁², e₃¹), thus, it detects two event pairs (e₁¹, e₅¹) [0,2] and (e₁², e₅¹) [1,2] and once the "¬" event is detected the entire constituent event are deleted. Event e₅² occurrence does not detect any non-occurrence of the event E₂, in the interval
formed by $e_3^3$ and $e_2^2$. Thus, both the events in the list $E_2$ and $e_1^3$ is deleted, since $e_1^3$ cannot initiate any “$\rightarrow$” event.

**Cumulative:** Occurrence of the event $e_3^1$ detects the non-occurrence of the event $E_2$ in the interval formed by $(e_1^1, e_3^1)$ and $(e_1^2, e_3^1)$, thus, it detects a “$\rightarrow$” event with events $(e_1^1, e_1^2, e_3^1)$ over the interval $[0,2]$. Event $e_3^2$ occurrence has the same effect as the continuous context.

The event pairs are given below as the summary of the above explanation

**Recent:** $((e_1^2, e_3^1) [1,2])$

**Chronicle:** $((e_1^1, e_3^1) [0,2])$

**Continuous:** $((e_1^1, e_3^1) [0,2], (e_1^2, e_3^1) [1,2])$

**Cumulative:** $((e_1^1, e_1^2, e_3^1) [0,2])$

![Figure 5.4](image-url)  
**Figure 5.4.** Event occurrences for A, A* operators.

### 5.2.5. Aperiodic operator

“A” is an aperiodic event operator and it behaves with respect to the ternary operator context definitions. Aperiodic event $A$ ($E_1$, $E_2$, $E_3$) is detected whenever the event $E_2$ occurs in the interval formed by the events $E_1$ and $E_3$. Aperiodic operator is explained using the primitive event occurrences shown in figure 5.4.

**Recent:** Event $e_2^1$ occurrence replaces the event $e_1^1$ as the initiator for the “A” event. When the event $e_2^1$ (detector) occurs in the time interval $[2,2]$, it detects the “A” event
initiated by the event $e_1^2$ over the interval $[2,2]$. Event $q^3 [3,3]$ initiates the next “A” event. When the event $e_2^2$ occurs in the interval $[4,4]$ it detects the “A” event over the interval $[4,4]$, and since this is just the detector, initiator is not deleted. Similarly, “A” event is detected when the event $e_2^3$ occur. The occurrence of $e_3^1$ terminates the “A” event initiated by the event $e_1^3$.

**Chronicle:** When the event $e_1^2$ occurs, it is appended to the list $E_1$ that already contains $e_1^1$. At this time both the events are in the list, and both $e_1^1$ and $e_1^2$ has initiated the “A” event detection. When the event $e_2^1$ occurs it detects the “A” event detection that was initiated by the event $q^1$ and $e_1^2$ over the interval $[2,2]$. Event $e_2^2$ and $e_2^3$ occurrence detects “A” over the interval $[4,4]$ and $[5,5]$. When the event $e_1^1$ occurs it terminates the “A” event detection that was initiated by the event $e_1^1$.

**Continuous:** “A” event initiated by the events $e_1^1$ and $e_1^2$ are detected by the occurrence of the event $e_2^1$ with event pairs $(e_1^1, e_2^1)$ and $(e_1^2, e_2^1)$ over the interval $[2,2]$ and $[2,2]$ respectively. Each of the events $e_2^2$ and $e_2^3$ detects three “A” events initiated by $e_1^1$, $e_1^2$ and $e_1^3$ with event pairs $(e_1^1, e_2^2) [4,4]$, $(e_1^2, e_2^2) [4,4]$, $(e_1^3, e_2^2) [4,4]$, $(e_1^1, e_2^3) [5,5]$, $(e_1^2, e_2^3) [5,5]$, and $(e_1^3, e_2^3) [5,5]$. Event $e_3^1$ occurrence terminates the “A” event initiated by the events $e_1^1$, $e_1^2$ and $e_1^3$.

**Cumulative:** Events $e_1^1$ initiates the “A” event. Event $e_1^1$ occurrence is accumulated. Event $e_1^3$ initiates an “A” event. Event $e_2^2$ and $e_2^3$ occurrences are also accumulated. Event $e_3^1$ occurrence detects and terminates the “A” event in the cumulative context initiated by the event $e_1^1$ with event pair $(e_1^1, e_1^2, e_1^3, e_2^1, e_2^2, e_2^3, e_3^1) [6,6]$.

The event pairs are given below as the summary of the above explanation

Recent: $((e_1^2, e_2^1) [2,2], (e_1^3, e_2^2) [4,4], (e_1^3, e_2^3) [5,5])$

Chronicle: $((e_1^1, e_2^1) [2,2], (e_1^2, e_2^1) [2,2], (e_1^1, e_2^2) [4,4], (e_1^2, e_2^2) [4,4], (e_1^3, e_2^2) [4,4], (e_1^1, e_2^3) [5,5], (e_2^2) [5,5], (e_1^3, e_2^3) [5,5])$
Continuous: \(((e_1^1, \ e_2^1) \ [2,2], \ (e_1^2, \ e_2^1) \ [2,2], \ (e_1^1, \ e_2^2) \ [4,4], \ (e_1^2, \ e_2^2) \ [4,4], \ (e_1^3, \ e_2^2) \ [4,4], \ (e_1^1, \ e_2^3) \ [5,5], \ (e_1^2, \ e_2^3) \ [5,5], \ \text{and} \ (e_1^3, \ e_2^3) \ [5,5])\)

Cumulative: \(((e_1^1, \ e_1^2, \ e_1^3, \ e_2^1, \ e_2^2, \ e_2^3, \ e_3^1) \ [6,6])\)

5.2.6. Cumulative aperiodic

Cumulative aperiodic operator “A*” is similar to the “A” operator except that the events are detected and terminated only when the event \(E_3\) occurs (i.e., cumulative) until then the event \(E_2\) occurrences are accumulated. We will explain A* event, \(A^* (E_1, \ E_2, \ E_3)\) using the primitive event occurrences shown in figure 5.4.

**Recent:** Event \(e_1^2\) occurrence, initiates “A*” event and terminates the “A*” event initiated by event \(e_1^1\). Event \(e_1^1\) occurrence does not detects an “A*” event and the event is just accumulated. Event \(e_1^3\) occurrence, initiates “A*” event and terminates the “A*” event initiated by event \(e_1^2\) and also removes the event \(e_2^1\) from \(E_2\) buffer. Events \(e_2^2\) and \(e_2^3\) occurrences are just accumulated. Event \(e_3^1\) occurrence detects the event pair \((e_1^3, \ e_2^2, \ e_2^3, \ e_3^1) \ [4,5]\) and terminates the detection of “A*” event initiated by event \(e_1^3\).

**Chronicle:** When the event \(e_1^2\) occurs it is appended to the list \(E_1\) that already contains \(e_1^1\). Event \(e_1^1\) occurrence is accumulated; event \(e_1^3\) occurrence is appended to the \(E_1\) list and the event occurrences \(e_2^2\) and \(e_2^3\) are accumulated. When the event \(e_1^1\) occurs, oldest initiator (i.e., \(e_1^1\)) is paired with this event and an “A*” event is detected with the event pair \((e_1^1, \ e_2^1, \ e_2^2, \ e_2^3, \ e_3^1) \ [2,5]\) and then both the initiator, terminator and all the constituent event that cannot participate in the future event detections are deleted. In this case, events \((e_1^1, \ e_3^1)\) are deleted and events \((e_2^1, \ e_2^2, \ e_2^3)\) are not deleted since they can take part in the future event detections.

**Continuous:** When the event \(e_1^2\) occurs it is appended to the list \(E_1\) that already contains \(e_1^1\). Event \(e_1^1\) occurrence is accumulated; event \(e_1^3\) occurrence is appended to the \(E_1\)
list and the event occurrences $e_2^2$ and $e_2^3$ are accumulated. When the event $e_3^1$ occurs, events $e_1^1$, $e_1^2$, $e_1^3$ are paired with this event and three “A*” events are detected with the event pairs 
$(e_1^1, e_2^1, e_2^2, e_2^3, e_3^1) [2,5], (e_1^2, e_2^1, e_2^2, e_2^3, e_3^1) [2,5], (e_1^3, e_2^2, e_2^3, e_3^1) [4,5]$ and then both
the initiator, terminator and all the constituent event that cannot participate in the future event
detections are deleted.

**Cumulative:** When the event $e_1^2$ occurs it is appended to the list $E_i$ that already contains $e_1^1$. Event $e_1^1$ occurrence is accumulated; event $e_1^3$ occurrence is appended to the $E_i$
list and the event occurrences $e_2^2$ and $e_2^3$ are accumulated. When the event $e_3^1$ occurs, events $e_1^1$, $e_1^2$, $e_1^3$ are paired with this event and one “A*” event is detected with the event pair $(e_1^1, e_1^2, e_1^1, e_1^3, e_2^2, e_2^3, e_3^1) [2,5]$ and then both the initiator, terminator and all the constituent
event that cannot participate in the future event detections are deleted.

The event pairs are given below as the summary of the above explanation

Recent: $((e_1^3, e_2^2, e_2^3, e_3^1) [4,5])$

Chronicle: $((e_1^1, e_2^1, e_2^2, e_2^3, e_3^1) [2,5])$

Continuous: $((e_1^1, e_2^1, e_2^2, e_2^3, e_3^1) [2,5], (e_1^2, e_2^1, e_2^2, e_2^3, e_3^1) [2,5], (e_1^3, e_2^2, e_2^3, e_3^1) [4,5])$

Cumulative: $((e_1^1, e_1^2, e_1^3, e_2^1, e_2^2, e_2^3, e_3^1) [2,5])$

![Figure 5.5. Event occurrences for Plus operator.](image)
5.2.7. Plus operator

“Plus” operator is used to specify a relative time. Plus operator occurs only once whenever the time specified, relative to an event $E_1$ happens. Plus ($E_1 + [10 \text{ mins}]$) occurs only one time after 5 minutes after an occurrence of the event $E_1$. Plus operator is explained using the primitive event occurrences shown in figure 5.5.

Recent: Plus operator treats all contexts the same except the recent context where the recent initiator replaces the immediate recent initiator. So, once replaced the Plus event initiated by the immediate recent initiator won’t occur. In the example, $e_1^1$ occurs at the time $[0,0]$ and the “Plus” event starts at the time 0. At the time point 5, event $e_2^2$ occurs and it starts a “Plus” event and since it is the recent initiator it replaces the event $e_1^1$ and terminates the Plus event started by it. “Plus” event occurs over the interval $[15,15]$ and the event started by $e_2^2$ gets terminated. The event $e_3^3$ starts the next occurrence of the “Plus” event.

Chronicle, Continuous and Cumulative: In all these contexts the “Plus” event will occur after the time specified by the time string after the event $E_1$ occurrence. In our example the “Plus” event occurs three times over the interval $[10,10]$, $[15,15]$, $[35,35]$ with respect to the events $e_1^1$, $e_2^2$ and $e_3^3$.

The event pairs are given below as the summary of the above explanation

Recent: $((e_2^2, [t,t]) [15,15])$

Chronicle, Continuous, Cumulative: $((e_1^1, [t,t]) [10,10], (e_2^2, [t,t]) [15,15], (e_3^3, [t,t]) [35,35])$
5.2.8. Periodic operator

“P” is a periodic event operator and it behaves with respect to the ternary operator context definitions. Periodic event is detected whenever the event $E_2$ (time string) occurs in the interval formed by the events $E_1$ and $E_3$. Let $P(E_1, [5], E_3)$ be the periodic event and the primitive event occurrences are shown in figure 5.6.

**Recent:** “P” event is initiated by the occurrence of an event $e_1^1$. At the time point 5, “P” event occurs over the interval $[5,5]$ and at time point 10, the next “P” event occurs over the interval $[10,10]$ and so on. Event $e_1^2$ initiates the next “P” event as the recent initiator from the time point 15. At time point 20, “P” event initiated by $e_1^2$ occurs. Occurrence of $e_3^1$ terminates the “P” event started by $e_1^2$.

**Chronicle:** Event $e_1^1$ initiates the “P” event. All the event occurs as in the recent context except that the event occurrence of $e_1^2$ does not stops the “P” event initiated by the event $e_1^1$. Occurrence of the event $e_3^1$ terminates the “P” event initiated by the event $e_1^1$, since it is the oldest initiator.

**Continuous:** Event $e_1^1$ initiates the “P” event. All the event occurs as in the recent context except that the event occurrence of $e_1^2$ does not stops the “P” event initiated by the event $e_1^1$. Occurrence of the event $e_3^1$ terminates the “P” events initiated by the event $e_1^1$ and $e_1^2$. 

![Figure 5.6. Event occurrence of P, P* operators.](image-url)
**Cumulative:** In the cumulative context, all the event occurrences are accumulated and are detected when the terminator occurs. Event \( e_1 \) initiates the “P” event. All the events and time string occurrences are accumulated until a terminator occurs. So when the event \( e_3 \) occurs it terminates the “P” event initiated and it occurs over the interval \([5,25]\) with events \( e_1, e_2, e_3 \) and corresponding \( E_2 \)'s as constituent events.

5.2.9. Cumulative periodic operator

“\( P^* \)” detects the occurrence of the events when the terminator occurs. How “\( P^* \)” event detection is different from “P” is explained below with the same event occurrences.

**Recent:** “\( P^* \)” event is initiated by the event \( e_1 \), but at the time point 5 there is no occurrence of the “\( P^* \)” event, and this time string event (\( E_2 \)) is accumulated. But according to this context definition, when the event \( e_2 \) occurs it acts as the recent initiator, so this stops the “\( P^* \)” event started by \( q_1 \). When event \( e_3 \) occurs it detects “\( P^* \)” event with events \( (e_1^2, 20, 25, e_3^1) \) over the interval \([20,25]\) and terminates the event initiated by \( e_1^2 \).

**Chronicle:** Events \( q_1, e_1^2 \) initiates the “\( P^* \)” event. Whenever the event \( E_2 \) occurs with respect to these initiators it is accumulated. When event \( e_1 \) occurs it detects the “\( P^* \)” event with \( (e_1^1, 5, 10, 15, 20, 25, e_3^1) \) over the interval \([5,25]\) that is corresponding to the oldest initiator (i.e., \( e_1^1 \)) and terminates. But event \( E_2 \) occurrences with respect to \( e_1^2 \) are accumulated until a new terminator occurs.

**Continuous:** Events \( e_1^1, e_1^2 \) initiates the “\( P^* \)” event. Whenever the event \( E_2 \) occurs with respect to these initiators it is accumulated. When event \( e_3 \) occurs it detects the “\( P^* \)” events with \( (e_1^1, 5, 10, 15, 20, 25, e_3^1) \) over the interval \([5,25]\), \( (e_1^2, 20, 25, e_3^1) \) over the interval \([20,25]\) and terminates both the “\( P^* \)” events initiated.

**Cumulative:** Events \( e_1^1, e_1^2 \) initiates the “\( P^* \)” event. Whenever the event \( E_2 \) occurs with respect to these initiators it is accumulated. When event \( e_3 \) occurs it detects the “\( P^* \)”
events with \((e_1^1, 5, 10, 15, 20, 25, e_1^2, 20, 25, e_3^1)\) over the interval \([5,25]\) and terminates the “P*” event.
6. RELATED WORK

This chapter summarizes related work on event specification without going in to the details as all of them use detection-based semantics.

6.1. SAMOS

The combination of active and object-oriented characteristics within one, coherent system is the overall goal of SAMOS [10,11]. It addresses rule specification, rule management and rule execution. Even tough there is not much difference between the event specifications between the detection based Sentinel and SAMOS, we will explain briefly the events and event constructors.

Primitive events are the events that are associated with a point in time and they are method events, transaction events, time events and abstract events.

1. Time events are specified at a specific point in time.
2. Method events are the events that are raised by the object invocation and can be related to one class, to a particular object or to multiple classes.
3. Transaction events are defined by the start or termination time of user-defined transactions.
4. Abstract events are not detected by SAMOS and should be notified to the system by explicit operation.

Composite events are built from the primitive events using six event constructors.

1. Disjunction (E1|E2) occurs when either E1 or E2 occurs.
2. Conjunction (E1, E2) occurs when both E1 and E2 have occurred, regardless of order.
3. Sequence (E1; E2) occurs when first E1 and afterwards E2 occurs.

4. The other three operators define how many times the specific event occurred during a predefined interval and they are “*” constructor, history event and negative events. Event patterns are parameterized and are passed to the condition and action parts. Time and Negative events have no parameters.

5. Only chronicle context semantics is used by SAMOS

   Petri nets are used for the detection of composite events. But, because of the occurrence of an event is considered as a point in time, the sequence event faces the same problem in SAMOS as in Sentinel. The problem is explained in detail below.

   Two sequence events be ((E1; E2); E3) and (E2;(E1; E3)). Let the order of event occurrences be E1, E2 and E3. Both the sequence is detected in the Petri nets shown in figure 6.1, since the event occurrence is taken as the point in time.

![Petri net example](image)

Figure 6.1. Petri net example.

6.2. Ode

   Ode an object oriented database developed at AT&T Bell Labs. O++ is an object facility based on the C++ object facility and is called the class [8,9]. In order to provide persistence of objects O++ is an extension of C++. It also has events, which are of basic, logical and composite type and how they are specified.
Events happen at specific points in time. Basic events are the events supported by the system. These can be divided into 4 categories as follows:

1. Object State Events
2. Method Execution Events
3. Time Events
4. Transaction Events

Logical events are the basic events with a mask, which is used to mask or hide the event occurrence. This means that the Event and Condition portion of the ECA rule is combined thus making the EA rules.

Composite events are the combinations of logical events using logical and special event specification operators. Composite event is said to occur at the point in time when the last logical event in the composite event occurs. As seen in Sentinel [1,2] and SAMOS [10,11] this will lead to problems. Ode supports many logical operators (such as relative, prior, sequence, every etc.).

6.3. Towards a General Theory of Action and Time

In Active Databases events are considered as “instantaneous,” whereas real life events have duration. There has been considerable work done in the area of AI where the events are considered to have duration and are based on temporal logic. “Temporal logic is based on the temporal interval rather than points.” Allen [20,21] considers 13 mutually exclusive primitive relations that can hold between temporal intervals. Each of these relations is represented using a predicate in this temporal logic. These 13 relations are shown below, where the first six relations have inverse relation. Based on these temporal relations all the interval based Snoop operators are defined.

1. DURING (t1, t2)
2. STARTS (t1, t2)
3. FINISHES (t1, t2)
4. BEFORE (t1, t2)
5. OVERLAP (t1, t2)
6. MEETS (t1, t2)
7. EQUAL (t1, t2)

From all the systems discussed so far we can conclude that the systems using detection-based semantics does not recognize multiple compositions of some operators in the intended way.
7. CONCLUSIONS AND FUTURE WORK

In this thesis, we have extended the formal definitions of occurrence-based semantics to recent context. These definitions add constraints over the formulation in the unrestricted context in the form of conditions over initiators, detectors, and terminators appropriate for that particular context. The semantics have been implemented using event histories providing procedural semantics. Algorithms for all the operators in the recent and unrestricted context have been developed. All operators have been implemented for recent and unrestricted context.

We are in the process of extending the semantics to other contexts (such as chronicle, continuous and cumulative) and using the disjoint characterization of composite events. This work needs to be extended to the distributed event occurrence and detection as well.
APPENDIX A

ALGORITHMS FOR UNRESTRICTED CONTEXT
1. Algorithm for AND operator

PROCEDURE and_general (ei, parameter_list):

   If ei is the left event
      If E2 is not empty
          For every e2 in E2 and if (t_s(e2) ≤ t_s(e1)) and (t_e(e2) ≤ t_e(e1))
              Pass <e2, e1> to the parent with t_s(e2) and t_e(e1)
          Append e1 to E1

   If ei is the right event
      If E1 is not empty
          For every e1 in E1 and if (t_s(e1) ≤ t_s(e2)) and (t_e(e1) ≤ t_e(e2))
              Pass <e1, e2> to the parent with t_s(e1) and t_e(e2)
          Append e2 to E2

2. Algorithm for OR operator

PROCEDURE or_recent (ei, parameter_list):

   For any event <e> signaled
      Pass <e> to the parent with t_s(e) and t_e(e)

3. Algorithm for SEQUENCE operator

PROCEDURE seq_general (ei, parameter_list):

   If ei is the left event
      Append e1 to E1

   If ei is the right event
      If E1 is not empty
          For every e1 in E1 and if (t_s(e2) > t_e(e1))
              Pass all <e1, e2> to parent with t_s(e1) and t_e(e2)

4. Algorithm for PLUS operator

PROCEDURE plus_general (ei, parameter_list):

   If ei is the left event
      Get the EventID of e1
      If recent, (chronicle/continuous/cumulative) context is not set
Create a time queue item with the EventID and TimeString of E2
   Append e1 to E1

If ei is the right event
   Get the EventID of e2
   If E1 is not empty
      If any one of the e1’s EventID in E1 and that of e2 are same
         Pass <e1, e2> to the parent with t_s(e2) and t_e(e2)

5. Algorithm for APERIODIC operator

PROCEDURE a_general (ei, parameter_list)

   If ei is the left event
      Append e1 to E1

   If ei is the middle event
      If E1 is not empty
         For every e1 in E1 and t_e(e1) < t_s(e2)
            Pass <e1, e2> to the parent with t_s(e2) and t_e(e2)

   If ei is the right event
      If E1 is not empty
         For every e1 in E1 and if (t_e(e1) < t_s(e3))
            Delete e1 from E1

6. Algorithm for CUMULATIVE APERIODIC operator

PROCEDURE astar_general (ei, parameter_list)

   If ei is the left event
      Append e1 to E1

   If ei is the middle event
      If (E1 is not empty and (t_e (E1’s EarliestEndTime) < t_s (e2)))
         Append e2 to E2

   If ei is the right event
      If (E1 is not empty and (t_e (E1’s EarliestEndTime) < t_s (e3)))
         For every e1 in E1
            If (t_e(e1) < t_s(e3))
               Append e1 to tempE1
         If E2 is not null
For every $e_2$ in $E_2$
   If $(t_e(e_2) < t_s(e_3))$ and $(t_e(e_1) < t_s(e_2))$
      Append $e_2$ to tempE2
   If tempE2 is not null
      Pass $<tempE1, tempE2, e3>$ to the parent with $t_s(tempE2$’s EarliestStart Time) and $t_e(tempE2$’s LatestEndTime)
      Delete tempE2
   Delete tempE1
   Delete $e_1$ from $E_1$
If $E_1$ is empty
   Delete $E_2$
Else
   For every $e_2$ in $E_2$
      If $(t_s(e_2) < t_e(E1$’s EarliestEndTime))
         Delete $e_2$ from $E_2$

7. Algorithm for PERIODIC operator

PROCEDURE p_general (ei, parameter_list)

   If $ei$ is the left event
      Get the EventID of $e_1$
      Create a time queue with the EventID of $e_1$ and TimeString of $e_2$
      Append $e_1$ to $E_1$

   If $ei$ is the middle event
      Get the EventID of $e_2$
      If $E_1$ is not empty
         For the $e_1$ in $E_1$ whose EventID and $e_2$ EventID are same
            Create a time queue with the EventID of $e_1$ and TimeString of $e_2$
            Pass $<e_1, e2>$ to the parent with $t_s(e_2)$ and $t_e(e_2)$

   If $ei$ is the right event
      If $E_1$ is not empty
         For every $e_1$ in $E_1$ and if $(t_e(e_1) < t_s(e_3))$
            Delete $e_1$ from $E_1$

8. Algorithm for CUMULATIVE PERIODIC operator

PROCEDURE pstar_general (ei, parameter_list)

   If $ei$ is the left event
      Get the EventID of $e_1$
Create a time queue with the EventID of e1 and TimeString of e2
Append to e1

If ei is the middle event
   If E1 is not empty
      Get the EventID of e2
      If any one of the EventID of e1 matches with EventID of e2
         Create a time queue with the EventID of e1 and TimeString of e2
         Append e2 to E2

If ei is the right event
   If (E1 is not empty and (t_e (E1’s EarliestEndTime) < t_s (e3)))
      For every e1 in E1
         If (t_e (e1) < t_s (e3))
            Get the EventID of e1
            Append to tempE2 all e2’s in E2 whose EventID is as same as e1
            If tempE2 is not null
               Pass <e1, tempE2, e3> to the parent with t_s (tempE2’s EarliestStartTime) and t_e (tempE2’s LatestEndTime)
               Delete tempE2
            Delete all e2’s from E2 those that have matched
         Delete e1 from E1
APPENDIX B

ALGORITHMS FOR RECENT CONTEXT
1. Algorithm for AND operator

PROCEDURE and_recent (ei, parameter_list):

    If ei is the left event
        If (E2 is not empty and (t_s(e2) ≤ t_s(e1)) and (t_e(e2) ≤ t_e(e1)))
            Pass <e2, e1> to the parent with t_s(e2) and t_e(e1)
            Replace e1 in E1 with ei
        
    If ei is the right event
        If (E1 is not empty and (t_s(e1) ≤ t_s(e2)) and (t_e(e1) ≤ t_e(e2)))
            Pass<e1, e2> to the parent with t_s(e1) and t_e(e2)
            Replace e2 in E2 with ei

2. Algorithm for OR operator

PROCEDURE or_recent (ei, parameter_list):

    For any event <e> signaled
        Pass<e> to the parent with t_s(e) and t_e(e)

3. Algorithm for SEQUENCE operator

PROCEDURE seq_recent (ei, parameter_list):

    If ei is the left event
        Replace e1 in the E1 with ei

    If ei is the right event
        If (E1 is not empty and (t_s(e2) > t_e(e1)))
            Pass<e1, e2> to parent with t_s(e1) and t_e(e2)

4. Algorithm for PLUS operator

PROCEDURE plus_recent (ei, parameter_list):

    If ei is the left event
        Get the EventID of e1
        Create a time queue item with the EventID of e1 and TimeString of E2
        Replace e1 in E1 with ei

    If ei is the right event
Get the EventID of e2
If E1 is not empty and EventID of e1 and e2 are the same
   Pass <e1, e2> to the parent with t_s(e2) and t_e(e2)

5. Algorithm for APERIODIC operator

The event is detected whenever the middle event is occurs and it is terminated whenever the
right side event occurs.

PROCEDURE a_recent (ei, parameter_list)

   If ei is the left event
      Replace e1 in E1 with ei

   If ei is the middle event
      If (E1 is not Empty and (t_e (e1) < t_s (e2)))
         Pass <e1, e2> to the parent with t_s (e2) and t_e (e2)

   If ei is the right event
      If E1 is not empty
         If (t_e (e1) < t_s (e3))
            Delete E1

6. Algorithm for CUMULATIVE APERIODIC operator

PROCEDURE astar_recent (ei, parameter_list)

   If ei is the left event
      Replace e1 in E1 with ei
      If E2 is not empty
         Delete E2

   If ei is the middle event
      If (E1 is not empty and (t_e (e1) < t_s (e2)))
         Append e2 to E2

   If ei is the right event
      If (E1 is not empty and (t_e (e1) < t_s (e3)))
         If E2 is not null
            For every e2 in E2
               If (t_e (e2) < t_e (e3))
                  Append e2 to tempE2
         If tempE2 is not null
Pass <e1, tempE2, e3> to the parent with t_s (tempE2’s EarliestStartTime) and
t_e (tempE2’s LatestEndTime)
Delete tempE2
Delete E1 and E2

7. Algorithm for PERIODIC operator

PROCEDURE p_recent (ei, parameter_list)

If ei is the left event
   Get the EventID of e1
   Create a time queue with the EventID of e1 and TimeString of E2
   Replace e1 in E1 with ei

If ei is the middle event
   Get the EventID of e2
   If E1 is not empty and EventID of e1 and e2 are same
      Create a time queue with the EventID of e1 and TimeString of e2
      Pass <e1, e2> to the parent with t_s (e2) and t_e (e2)

If ei is the right event
   If E1 is not empty
      If (t_e (e1) < t_s (e3))
         Delete E1

8. Algorithm for CUMULATIVE PERIODIC operator

PROCEDURE pstar_recent (ei, parameter_list)

If ei is the left event
   Get the EventID of e1
   Create a time queue with the EventID of e1 and TimeString of E2
   Replace e1 in E1 with ei
   If E2 is not empty
      Delete E2

If ei is the middle event
   Get the EventID of e2
   If E1 is not empty and the EventID of e1 and e2 are same
      Create a time queue with the EventID of e1 and TimeString of e2
      Append e2 to E2

If ei is the right event
If \((E1\text{ is not empty and } (t\_e(e1) < t\_s(e3)))\)
   
   Get the EventID of \(e1\)
   
   Append to tempE2 all \(e2\)'s in \(E2\) whose EventID is as same as \(e1\)
   
   Delete \(E1\) and \(E2\)
   
   If tempE2 is not null
      
      Pass \(<e1, \text{tempE2}, e3>\) to the parent with \(t\_s\) (tempE2's EarliestStartTime)
      
      and \(t\_e\) (tempE2's LatestEndTime)
      
      Delete tempE2

9. Algorithm for NOT operator

   Whenever the right event \(e3\) is signaled then it acts as the detector for this composite event only when there is no \(e2\) has occurred between the end interval of the left event and start interval of the right event.

PROCEDURE not_recent (\(ei, \text{parameter\_list}\))

   If \(ei\) is the left event
      
      Replace \(e1\) in \(E1\) with \(ei\)
      
      Delete \(E2\)

   If \(ei\) is the middle event
      
      If \((E1\text{ is not empty and } (t\_e(e1) \not\subseteq t\_s(e2)))\)
         
         Append \(e2\) to \(E2\)

   If \(ei\) is the right event
      
      If \((E1\text{ is not empty and } (t\_e(e1) < t\_s(e3)))\)
         
         If \(E2\) is not empty
            
            For all \(e2\)'s in \(E2\)
               
               If \((t\_e(e2) > t\_s(e3) \text{ and } t\_s(e2) < t\_s(e1))\)
                  
                  Pass \(<e1, e3>\) to parent with \(t\_s(e1)\) and \(t\_e(e3)\)
                  
                  Delete \(e2\) from \(E2\)
               
            Else
               
               Pass \(<e1, e3>\) to the parent with \(t\_s(e1)\) and \(t\_e(e3)\)
               
               Delete \(E1\)
REFERENCES


BIOGRAPHICAL INFORMATION

Raman Adaikkalavan was born July 11, 1978 in Pudukkottai, India. He received his Bachelor of Science degree in Computer Science Engineering from Bharathidasan University, Tamil Nadu, India in May 1999. In the Fall of 2000, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science and Engineering from The University of Texas at Arlington, in August 2002. His research interests include active databases, stream data processing, data mining and data warehousing.