When to Trigger Active Rules?

Raman Adaikkalavan

CIS Department
Indiana University South Bend
raman@cs.iusb.edu

Sharma Chakravarthy

CSE Department
University of Texas at Arlington
sharma@cse.uta.edu

Abstract

Active rules model and enforce enterprise requirements such as situation monitoring using events, conditions and actions. These rules are termed active rules as they make the underlying application or system such as database management system active capable (i.e., react to changes actively using the push paradigm). Events play a critical role in active rules as they define and detect an *occurrence of interest* in the real world and trigger the associated rules. Currently active rules are triggered only when events *occur* completely. Though this allows active rules to model enterprise policies they are not sufficient in modeling situations warranted by applications such as information security. In this paper, we motivate the need for extending events and active rules. We introduce and discuss event extensions and rule generalization and show how these extensions allow the modeling of situations warranted by emerging applications. Finally, we discuss algorithms and implementation of the extensions using the Sentinel Local Event Detector system.

1 Introduction

A number of event processing systems using Active or Event-Condition-Action rules have been proposed and implemented [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. Traditionally, active rules have been used for various applications in diverse domains such as situation monitoring, workflow modeling, and relational and object oriented database management systems. Lately, there have been a lot of work in Complex Event Processing [12] and event-based systems for solving various other problems [13, 14] such as change detection in Web, information security, Peer-to-Peer computing, information filtering, sensor databases, RFID event processing, multimedia events, etc. There have also been some work on semantic events [15], events over uncertain data [16], and probabilistic events [17].

An active rule is composed of an event and a set of conditions and actions. Large body of work [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] have been carried out on defining simple events and on the language for complex event specification and detection, as event was the least understood component. Simple (or primitive) events are domain based. Complex (or composite) events are constructed from one or more simple or complex events using event operators. Existing event specification languages and detection mechanisms are based on the well-defined point-based [5, 9, 10, 11] and interval-based [1, 2, 3, 4, 7] temporal semantics. Although both event semantics are extensively used in event processing, they were shown to be inadequate for supporting many newer application domains [18].

15th International Conference on Management of Data
COMAD 2009, Mysore, India, December 9–12, 2009

The main shortcoming of the complex event specification and detection using point-based, interval-based or generalized semantics is that an event is detected only when all the required constituent events of that event occur. In all other cases, occurrences of constituent events are *ignored* (i.e., deleted). This is a critical issue since active rules associated with an event is triggered *iff* that event is detected. In other words, an event *decides* when to trigger the associated active rules. Due to the critical role played by events in active rules, this method of event detection does not allow active rules to model the policies required by applications such as access control in the information security domain.

Let us take a policy and model it using existing complex event specification and detection. *"Take actions when the first occurrence of interest precedes the second occurrence of interest."* For example, in an university campus, between 00:00 hrs and 06:00 hrs the building door must be opened before any office room door is opened. This policy can be modeled using active rules and the SEQUENCE complex event operator [1]. SEQUENCE operator is detected whenever the first event precedes the second event. After event detection, rules associated with the event are triggered, conditions are checked and actions are taken. On the other hand when the second event occurs *without* the occurrence of the first event, that occurrence is *ignored*. In our example, *what happens if the office door is opened without opening the building door?*. Current event specification and detection just *ignores* the occurrence. In real life situations this cannot be ignored as it might indicate a break-in or a security violation, and a set of actions must be taken such as notifying the campus security office. We discuss other scenarios in the paper where current event specification and detection falls short.

In this paper, we propose extensions to the existing complex event and active rule processing. A brief discussion of our previous work was discussed in the extended abstract [19]. Existing complex event and rule specification and detection is discussed in Section 2. The need for extensions using real world policies is discussed in Section 3. Event extensions are discussed in Section 4 and rule extensions are discussed in Section 5. Event detection graphs are discussed in Section 6 and event detection algorithms are discussed in Section 7. Section 8 has conclusions and future work.

2 Background

In this section we discuss events and rules.

2.1 Events

Snoop [1, 9, 18, 20, 21] event specification language which is a part of the Sentinel [22] Local Event Detector (LED) system is used in this paper. Main motivation to use Snoop is that it supports expressive event specification using *point*-based [9, 20], *interval*-based [1, 21] and *generalized* semantics [18] in various *event consumption modes* or contexts [1, 20, 21]. LED uses *event detection graphs* to detect events using point-, interval-based and generalized semantics in various event consumption modes.

An *event* is "an occurrence of interest" in the real world that can be either *simple* (e.g., depositing cash) or *complex* (e.g., depositing cash, followed by withdrawal of cash). Simple events occur at a point in time (i.e., time of depositing), and complex events occur over an interval (i.e., starts at the time cash is deposited and ends when cash is withdrawn). Simple events are detected at a point in time, whereas the complex events can be detected either at the *end* of the interval (i.e., detection- or point-based semantics) [5, 9, 10, 11] or can be detected *over* the interval (i.e., occurrence- or interval-based semantics) [1, 2, 3, 4, 7].

Each event has a well-defined set of *attributes* based on the *implicit* and *explicit* parameters [18]. These attributes provide all the information about that event. *Implicit* parameters contain system and user defined attributes such as event name and time of occurrence. *Explicit* parameters are collected from the event itself e.g., stock price and stock value.

Below we discuss simple and complex event specification [1] from [1, 9, 18, 20].

2.1.1 Simple Events

Simple events are the basic building blocks in an event processing system and are derived from various application domains. For example: data manipulation language and data definition language statements in a DBMS, function call invocation in an Object Oriented systems, alarm clock, increase in stock price, change in a web page, pattern match over a text stream, and sensor readings.

Definition 1 (Point-based) *A simple event* E *occurs atomically at a point* $[t]$ *on the time line. It is detected at* $[t]$ *and is defined as*
$\mathcal{P}(E, [t]) \triangleq \exists t \; (E, [t]);$

Definition 2 (Interval-based) *A simple event* E *occurs atomically at a point* $[t]$ *on the time line. It is detected over an interval* $[t, t']$*, where* $[t]$ *is the start time,* $[t']$ *is the end time and* $(t = t')$*. It is defined as*
$\mathcal{I}(E, [t, t']) \triangleq \exists t = t' \; (E, [t, t']);$

Definition 3 (Generalized) *A generalized simple event* $G(E)$ *can be a point-* (G_P) *or interval-based* (G_I) *simple event with conditional expressions based on* $\mathcal{I}_{expr}$ *and* $\mathcal{E}_{expr}$*. They are formally defined as*
$G_P(E, [t]) \triangleq \exists t \; (\mathcal{P}(E, [t]) \wedge (\mathcal{I}_{expr} \wedge \mathcal{E}_{expr}));$
$G_I(E, [t, t']) \triangleq \exists t = t' \; (\mathcal{I}(E, [t, t']) \wedge (\mathcal{I}_{expr} \wedge \mathcal{E}_{expr}));$

Below we show an event from an Object Oriented system where all function invocations are treated as events. E_1 is the event name and $\mathcal{F}$ is function name. Function attributes are the explicit parameters. Time of the function call and the object that invoked the function are the implicit parameters. Implicit and explicit parameter expressions are represented as $\mathcal{I}_{expr}$ and $\mathcal{E}_{expr}$, respectively. They are discussed in Definition 6.

$$E_1 = (\mathcal{F}(), (\mathcal{I}_{expr} \wedge \mathcal{E}_{expr}));$$

2.1.2 Complex Events

Simple events are often not adequate for modeling real-world scenarios. Complex events are defined by composing more than one simple or complex event using *event operators*. A number of event operators have been proposed in the literature based on several application domains. Using *composition conditions*, an event operator defines how a complex event needs to be composed and detected.

Definition 4 (Point-based) *A complex event* $\mathcal{P}(E)$ *occurs over an interval* $[t_s, t_e]$ *and is detected at a time point* $[t_e]$*, where* t_s *is the start time of* initiating *event, and* t_e *is the end time of* detecting *event. Eop is the event operator.*
$\mathcal{P}(Eop \; (E_1, \ldots E_n), [t_e]);$

Definition 5 (Interval-based) *A complex event* $\mathcal{I}(E)$ *occurs and is detected over an interval* $[t_s, t_e]$*. It is defined as*
$\mathcal{I}(Eop \; (E_1, \ldots E_n), [t_s, t_e]);$

Definition 6 (Generalized) *A generalized complex event* $G(E)$ *can be a point-* (G_P) *or interval-based* (G_I) *complex event with conditional expressions based on* $\mathcal{I}_{expr}$ *and* $\mathcal{E}_{expr}$*. They are formally defined as*

$$G_P(Eop \; (E_1, \ldots E_n), (\mathcal{I}_{expr} \; \wedge \; \mathcal{E}_{expr}), [t_e]);$$
$$G_I(Eop \; (E_1, \ldots E_n), (\mathcal{I}_{expr} \; \wedge \; \mathcal{E}_{expr}), [t_s, t_e]);$$

- *Eop* represents a n-ary event operator (And, Or, SEQUENCE, NOT, Plus, Periodic, Aperiodic, Periodic*, and Aperiodic*).
- $(E_1, \ldots E_n)$ are the constituent events. For example, E_1 can be simple or complex.

[1]In this paper, we use $\mathcal{P}$ to represent point-based and $\mathcal{I}$ to represent interval-based semantics.

- A complex event occurrence is based on the initiator, detector and terminator events. *Initiator* is the constituent event whose occurrence starts the complex event. *Detector* is the constituent event whose occurrence detects and raises the complex event. *Terminator* is the constituent event that is responsible for terminating the complex event i.e., no more occurrence of a complex event with the same initiator event is possible. All operators have initiator and detector but not terminator. In addition, detector and terminator can be the same constituent event.
- Implicit parameter expression $\mathcal{I}_{expr}$ subsumes existing point- and interval-based semantics. For instance, a binary event operator with events E_1 and E_2 can have $\mathcal{I}_{expr} = t\_occ(E_1)\ \theta\ t\_occ(E_2)$, where t_occ represents the timestamp of event occurrence, and θ can be any operator $<, >, \leq, \geq, =, \neq, \in, \ldots$
- $\mathcal{E}_{expr}$ has conditions based on the explicit parameters. For instance, a binary event operator with the events E_1 and E_2 can have $\mathcal{E}_{expr} = E_1(A_{xi})\ \theta\ E_2(A_{xj})$, where attributes $E_1(A_{xi})$ and $E_2(A_{xj})$ have values from the same domain.
- Complex event is detected *iff* the required constituent events occur, and both $\mathcal{I}_{expr}$ and $\mathcal{E}_{expr}$ return TRUE. We assume that both expressions are not empty at the same time, otherwise it will detect the complex event always.

Event Consumption Modes: In order to avoid the unnecessary event detection, *event consumption modes* [1, 20, 21] or contexts such as Recent, Continuous, Chronicle, and Cumulative were defined based on the application domains.

Below, SEQUENCE operator is formally defined in the unrestricted mode (i.e., none of events occurrences are removed from the system), and the NOT operator is defined intuitively.

SEQUENCE EVENT $(E_1 \gg E_2)$:

This event is raised when event E_1 occurs before event E_2. It is *detected* when E_2 occurs. E_1 is the *initiator* event and E_2 is the *detector* event. It is formally defined in point and interval semantics as:

$$\mathcal{P}(E_1 \gg E_2, [t_2]) \triangleq \exists t_1, t_2(\mathcal{P}(E_1, [t_1]) \wedge \mathcal{P}(E_2, [t_2]) \wedge (t_1 < t_2));$$

$$\mathcal{I}(E_1 \gg E_2, [t_s, t_e]) \triangleq \exists t_s, t_e, t, t'(\mathcal{I}(E_1, [t_s, t]) \wedge \mathcal{I}(E_2, [t', t_e]) \wedge (t_s \leq t < t' \leq t_e));$$

With point-based semantics, E_1 is detected at $[t_1]$ and E_2 is detected at $[t_2]$. Sequence event is detected if $(t_1 < t_2)$. With interval-based, sequence is detected if $(t_e(E_1) < t_s(E_2))$. Below is the generalized Sequence operator incorporating point-based semantics via $\mathcal{I}_{expr}$.

$$G_P(E_1 \gg E_2, [t_2]) \triangleq \exists t_1, t_2(G_P(E_1, [t_1]) \wedge G_P(E_2, [t_2]) \wedge (\mathcal{I}_{expr} \wedge \mathcal{E}_{expr}));$$
$$\mathcal{I}_{expr} = (t_1 < t_2);$$

NOT EVENT $I(\neg(E_2)(E_1 \gg E_3), [t_1, t_2])$:

NOT event operator detects the non-occurrence of event E_2 between events E_1 and E_3. Event E_1 is the *initiator* event, and E_3 is both the *detector* and *terminator*. E_3's occurrence detects the NOT event when there is an occurrence of E_1 and no occurrence of E_2.

2.2 Active Rules

An event can be associated with multiple rules, but a rule can be associated with only one event. Condition and action procedures can be associated/shared between different rules. Rules are executed on an event occurrence and their management involves event detection and rule execution. Rule scheduling involves ordering of rules for execution when several rules are triggered at the same time. *Coupling modes* define when the rules have to be executed i.e., immediate, deferred, or detached. Rules can also be nested i.e., occurrence of an event triggers a rule which in turn detects/raises another event. If an event is not part of a complex event or does not have any rule associated, then that event need not be detected for efficiency. This is possible as rules can be in active or deactive states.

Major components of active rules are discussed below.

$$R_i = (E_j, (C_1 \ldots C_k), (A_1 \ldots A_n));$$

- Rule R_i – denotes the name of the rule. No two rules can have the same name.
- Event E_j – associates event E_j with this rule. This can be a simple or complex event. The occurrence of this event *triggers* the rule.
- Condition $C_1 \ldots C_k$ – defines the set of conditions to be evaluated. The set of conditions are checked once the rule is triggered by the event E_j. Unless the rule is triggered, conditions cannot be evaluated.
- Action $A_1 \ldots A_n$ – defines the set of actions to be triggered when conditions evaluate to TRUE.

```
RULE [   R1
         EVENT       E1
         CONDITION   /* Conditions */
         ACTION      /* Actions */        ]
```

Rule R_1 shown above is triggered when event E_1 is detected. Extended ECA rules [23] rules shown below have an additional component for triggering alternative actions when conditions evaluate to FALSE.

$$R_i = (E_j, (C_1 \ldots C_k), (A_1 \ldots A_n), (AA_1 \ldots AA_p));$$

```
RULE [   R1
         EVENT       E1
         CONDITION   /* Conditions */
         ACTION      /* Actions */
         ALT ACTION  /* Alternative Actions */    ]
```

3 Need for Extending Events and Rules

Events are raised when the occurrence of interest happens and are detected only when all the constraints associated with the event are satisfied. For example, when a simple event is defined along with an $\mathcal{E}_{expr}$ constraint, it is detected when the event occurs and the $\mathcal{E}_{expr}$ returns TRUE. Similarly, complex events that combine events are detected only when all the required constituent events occur and all the constraints are satisfied. Rules associated to an event can only be triggered *iff* that event is detected completely. Though occurrence of all required constituents events is necessary in many situations it is not required in others for event detection.

Protecting information against unauthorized access is a key issue in information system security. Access control evaluates all access requests to resources by authenticated users, and determines whether the requests must be granted or denied, ensuring confidentiality and integrity. There are several access control models [24]: discretionary, mandatory, and role-based.

Below we discuss two policies from access control domain and RFID-based retail stores, respectively. We model these policies using active rules, and discuss the limitations of active rules.

Policy 1 *Between 00:00 hrs and 06:00 hrs, only those who have entered through the building's external doors can enter an office room in that building.*

The above policy can be made more complex by including situations wherein two or more persons enter the building together without using their cards individually. Without the loss of generality, we just discuss the basic policy defined above and assume that each person registers his/her entry individually.

Event E_{ExtReq} (external door request) is detected when there is a request to open any external door between 00:00 hrs and 06:00 hrs. This triggers rule R_{ExtReq}. The condition part authenticates the user, and the action part allows the door to be opened. Access is denied, otherwise.

$$\begin{aligned} E_{ExtReq} = (&doorOpen(bldgId, doorId, doorType, userId), \\ &(((t_{occ} > 00:00hrs) \wedge (t_{occ} < 06:00hrs)) \\ &\wedge(doorType = \text{“}external\text{”}))); \end{aligned}$$

```
RULE [   R_ExtReq
         EVENT        E_ExtReq
         CONDITION    /* Authenticate User */
         ACTION       /* Open door */
         ALT ACTION   /* Deny Access */           ]
```

The external door open request is handled by event E_{ExtReq} but the actual opening of the door is handled in the action part. Thus, another event $E_{ExtOpen}$ needs to be raised by rule R_{ExtReq} in the action part to indicate that the door was opened. The parameters of the event $E_{ExtOpen}$ is same as E_{ExtReq}. The modified rule is shown below.

```
RULE [   R_ExtReq
         EVENT        E_ExtReq
         CONDITION    /* Authenticate User */
         ACTION       /* Open door, Raise E_ExtOpen */
         ALT ACTION   /* Deny Access */           ]
```

Event E_{OffReq} is raised when someone tries to open an office door.

$$\begin{aligned} E_{OffReq} = (&doorOpen(bldgId, doorId, doorType, userId), \\ &(((t_{occ} > 00:00hrs) \wedge (t_{occ} < 06:00hrs)) \\ &\wedge(doorType = \text{“}office\text{”}))); \end{aligned}$$

Similar to rule R_{ExtReq} another rule can be created to allow someone to open an office door. But the policy requirement will not be met i.e., user should have opened the building door first. Thus, a complex event is required to model this requirement i.e., if E_{OffReq} happens after $E_{ExtOpen}$, trigger the rule to check for access. A generalized SEQUENCE event $E_{OffReq2}$ and rule $R_{OffReq2}$ are created as shown below. $\mathcal{I}_{expr}$ can be used to detect the event in either point-based or interval-based semantics. $\mathcal{E}_{expr}$ tracks each user separately.

$$\begin{aligned} E_{OffReq2} =& (\gg (E_{ExtOpen}, E_{OffReq})) \\ &\wedge (\mathcal{I}_{expr} \ \wedge \ \mathcal{E}_{expr})); \end{aligned}$$

$$\begin{aligned} \mathcal{I}_{expr} =& (Point/Interval\ Semantics); \\ \mathcal{E}_{expr} =& ((E_{ExtOpen}.userId = E_{OffReq}.userId \\ &\wedge E_{ExtOpen}.bldgId = E_{OffReq}.bldgId)); \end{aligned}$$

```
RULE [   R_OffReq2
         EVENT        E_OffReq2
         CONDITION    /* Authenticate User */
         ACTION       /* Open door */
         ALT ACTION   /* Deny Access */           ]
```

Existing systems detect events and triggers rules when event $E_{ExtOpen}$ happens before E_{OffReq}. Possible constituent event occurrences (*cases*) of event $E_{OffReq2}$ are as follows:

1. $E_{ExtOpen}$ and E_{OffReq} occur: $E_{ExtOpen}$ event is raised when the external door is opened. When the same person requests for opening an office door, event E_{OffReq} is detected. Since this is the detector event for the complex event $E_{OffReq2}$, the complex event occurrence is completed and the rule $R_{OffReq2}$ is triggered. If the person has proper authentication office door is opened.

2. $E_{ExtOpen}$ occurs alone: This event is raised when the external door is opened. This event is the initiator and starts the complex event $E_{OffReq2}$. The detector event is raised only when the same person tries to open an office door. In case the detector does not happen, a timeout event can be triggered based on the enterprise policy.

3. E_{OffReq} occurs alone: Event E_{OffReq} is detected without $E_{ExtOpen}$ i.e., complex event $E_{OffReq2}$ had not been initiated. *What will happen if the detector event happens without any initiator?* With *existing systems*, the detector event is just *ignored* as it does not capture the occurrence of interest. With access control application this should trigger a violation, which is *not* possible with current systems.

Binary Operator Summary: Occurrence of the initiator starts the complex event. A detector/terminator should occur or a timeout event has to be raised in order to complete the event. Detector/terminator event plays an important role in enforcing the policy. Problem araises when the detector occurs *without* the initiator, as it is ignored. Other operators, additional rules or complex conditions/actions will still not be able to model the policy and trigger a violation. This warrants the extension of binary operators.

Policy 2 *Alert security personnel when a shoplifting activity occurs in a RFID-based retail store [25] i.e., items that were picked at a shelf and then taken out of the shop without an entry in the point of sale system.*

This policy requires to alert on a non-occurrence event and can be modeled with the NOT operator as shown below:

$$\begin{aligned}E_{Chk} = (&NOT(E_{Pick}, E_{POS}, E_{Gate})\\ &\wedge (\mathcal{I}_{expr}\ \wedge\ \mathcal{E}_{expr}));\end{aligned}$$

$$\begin{aligned}\mathcal{I}_{expr} =& (Point/Interval\ Semantics);\\ \mathcal{E}_{expr} =& (E_{Pick}.itemId = E_{POS}.itemId\\ &= E_{Gate}.itemId);\end{aligned}$$

We do not show all the constituent event definitions and corresponding rules. Event E_{Pick} represents picking the item from the shelf. Event E_{POS} represents checking out at the point of sale system. Event E_{Gate} represents item leaving the gate. $\mathcal{I}_{expr}$ allows event detection in either point-based or interval-based semantics. $\mathcal{E}_{expr}$ relates all event occurrences with the same item for controlling each item simultaneously. E_{Pick} is the initiator, E_{Gate} is the detector, and middle event E_{POS} is the non-occurrence event (i.e., event that should not occur). Possible constituent event occurrences (*cases*) of event E_{Chk} are discussed below.

1. E_{Pick} and E_{Gate} occur: This detects the NOT event and an alert regarding shop lifting can be sent via the actions part. Current systems handle this correctly.

2. E_{Pick} occurs alone: Item was picked but nothing happened after that. Current systems just wait for a detector to occur. One possible solution would be to raise a timeout event (e.g., shop closing) and take further actions (e.g., re-shelf the item).

3. E_{POS} occurs alone: Someone has checked out an item without picking it from the shelf. This indicates that something is malfunctioning. Since this event is just a *constituent* event it is ignored (i.e., deleted) in the current event systems. But this cannot be ignored since a E_{GATE} event might occur in the future. One possible solution would be store this event, and wait for the E_{GATE} event or raise a timeout event (e.g., shop closing) and take further actions.

4. E_{Pick} and E_{POS} occur: Item has been picked up and checked out. Current systems just wait for the *detector/terminator* event E_{Gate} to occur. One possible solution would be to raise a timeout event (e.g., shop closing) and take further actions (e.g., check gate sensors).

5. E_{POS} and E_{Gate} occur: Item was not picked out, but it was checked out and taken to the gate. Current systems just ignore all these occurrences. This cannot be the case as it might indicate that there is some malfunctioning and it has to be reported.

6. E_{Gate} occurs alone: Item was not picked up or checked out, but has reached the gate. Current systems ignore this *detector/terminator* event and purge it from the system. This is an incorrect action as it might be a shop lifting activity.

7. E_{Pick}, E_{POS} and E_{Gate} occur: All the occurrences are just ignored as the event that should *not* occur has happened. This case indicates that the items were checked out properly. This event occurrence can be used to create a log for inventory maintenance.

Ternary Operator Summary: When the initiator happens alone or when initiator and constituent events occur, but not the detector/terminator event, then raising timeout event is the only solution. When detector/terminator happens without the initiator and constituent events it is a problem that needs immediate attention. Currently, events are simply dropped in all the cases except the first case, which is insufficient. Modeling of the above discussed policy using existing event operators or active rules with complex conditions/actions is not possible.

4 Event Extensions

Utilizing event processing in diverse application domains require an additional capability of current event detection semantics to *infer* that a constituent event of a complex event has been detected, but not other events to complete the detection of that complex event. To identify such occurrences we propose `event detection modes` for complex events. The outcome of simple event detection is the same in all modes. These modes allow for additional actions and alternative actions to be taken when the event is not completed because of the occurrence/non-occurrence of constituent events.

The following are the steps involved in event processing:

1. Define simple and complex events using an event specification language.

2. Define rules.

3. Detect events based on event semantics using an event detection mechanism such as an event graph.

4. Trigger rules when events are detected.

Introducing event detection modes requires changes to all the steps shown above. Modifications to Step 1 are not discussed in this paper, leaving the definition and specification of events discussed in Section 2.1 unchanged. More details about the impact of modes on specification is discussed in Section 6. Modification to other steps are discussed in the following sections. Extensions to part of Step 3 is discussed in this section. Step 2 is discussed in Section 5 and Steps 3 and 4 are discussed in Sections 6 and 7.

Simple events are detected whenever they occur in the system and these extensions do not apply to them. Extensions to both binary and ternary Snoop operators are discussed below.

4.1 Binary Event Operator Semantics

With binary operators, two constituent events are involved and they act as initiator and detector/terminator. When the *detector* event occurs, operator semantics is applied and both $\mathcal{I}_{expr}$ and $\mathcal{E}_{expr}$ are checked. If any of the conditions fail then that event is *not* raised and other constituent events are dropped. Whether an event is complete is checked *only* when the detector is raised. Current systems deal only with complete events. Below we define a complete event:

Definition 7 (Complete Event) *"A complete complex event E occurs when,* i) *initiator occurs, and* ii) *detector occurs and completes that event."*

When the detector event occurs without the initiator, existing event detection semantics has to be modified to trigger active rules. Extending current event detection semantics to handle situations where the detector has occurred *without* the required events to complete the detection will allow the system to take additional actions. We term these events as *partial events* and define them below.

Definition 8 (Partial Event) *"A partial complex event E occurs when* i) *event E is not initiated, and* ii) *detector occurs."*

Three cases that were analyzed in Section 3 under Policy 1 can be handled using these extensions. Specifically, Case 1 is handled by complete events and Case 3 is handled by partial events. Case 2 can be handled by a timeout event which completes the event.

4.2 Ternary Event Operator Semantics

Similar to binary operators ternary events have initiator and detector/terminator. In addition there is another event that is just a constituent event. Binary complete event definition is further refined as shown below.

Definition 9 (Complete Event) *"A complete complex event E occurs when,* i) *initiator occurs,* ii) *all the required constituent events occur, and* iii) *detector occurs and completes that event."*

Partial binary event definition is further refined as shown below.

Definition 10 (Partial Event) *"A partial complex event E occurs when* i) *event E is not initiated,* ii) *other constituent events can occur, and* iii) *detector occurs."*

In addition to the above events, we define *failed events* as shown below. For example, this event is detected when the non-occurrence has failed for a NOT operator.

Definition 11 (Failed Event) *"A failed complex event E occurs when* i) *initiator occurs,* ii) *other constituent events occur, and* iii) *detector occurs and completes the event, but the event fails because some constituent event that should not have occurred has occurred."*

Seven cases that were analyzed in Section 3 under Policy 2 are handled as shown below. Current systems handle only Case 1. All seven cases can be handled using the proposed extensions. Specifically, Case 1 is handled by complete events, Cases 5 & 6 are handled by partial events, and Case 7 is handled by failed events. Case 3 & 4 are handled by partial events using timeout events. Case 2 can be handled by a timeout event which completes the event.

4.3 Summary

Events can be detected as complete, partial or failed. These three types are termed as event detection modes. In Step 4 shown above active rules are triggered when events are triggered. With the extension proposed, active rules can be triggered in all the three event detection modes. Though the specification has not been changed, detection has to be changed to trigger appropriate rules. In Sections 6 and 7 we discuss event detection in detail.

5 Active Rules Generalization

As explained in Section 2 active rules consist of four components and various other attributes. Active rules are triggered when complete events are detected. With new event detection modes, *when should active rules be triggered?* Partial and failed events should also trigger associated active rules. In this section we discuss the extensions needed to support complete, partial, failed and other future detection modes in a seamless way.

Active rules specification discussed in Section 2 is shown below:

```
RULE [  R_1
        EVENT       E_1
        CONDITION   /* Conditions */
        ACTION      /* Actions */
        ALT ACTION  /* Alternative Actions */    ]
```

Event detection modes are handled by adding an optional DMODE attribute to the existing rule specification.

Currently, the values of the DMODE attribute are: complete, partial, or failed. If a value is not specified, the rule will be triggered when a complete event is detected, by default. This extension allows the Local Event Detector to trigger rules accordingly. Using the DMODE attribute different sets of condition-action-alternative actions are associated to the rules. This is similar to the select-case conditional structure used in programming languages.

```
RULE [  $R_i$
        EVENT  $E_j$

        DMODE:[COMPLETE | PARTIAL | FAILED] {
           CONDITION  /* Conditions */
           ACTION     /* Actions */
           ALT ACTION /* Alternative Actions */ }
     ]
```

This generalization allows the specification of the proposed and future event modes and their associated condition-action-alternative actions. Below we show the specification for the proposed event detection modes.

```
RULE [  $R_1$
        EVENT  $E_1$

        DMODE:COMPLETE {
           CONDITION  /* Conditions */
           ACTION     /* Actions */
           ALT ACTION /* Alternative Actions */ }
        DMODE:PARTIAL {
           CONDITION  /* Conditions */
           ACTION     /* Actions */
           ALT ACTION /* Alternative Actions */ }
        DMODE:FAILED {
           CONDITION  /* Conditions */
           ACTION     /* Actions */
           ALT ACTION /* Alternative Actions */ }
     ]
```

With complete, partial and failed events and rules we can model and capture policies that cannot be captured using existing systems. Below we show the extended rules corresponding to policies discussed in Section 3.

5.1 Rules for Policy 1

Rule R_{ExtReq} need not be changed as it is associated with a simple event. By default the event is triggered as a complete event. Rule $R_{OffReq2}$ that is associated with the SEQUENCE complex event $E_{OffReq2}$ has been modified using the generalized rule specification.

```
RULE [  $R_{OffReq2}$
        EVENT  $E_{OffReq2}$

        DMODE:COMPLETE {
           CONDITION  /* Authenticate User */
           ACTION     /* Open door */
           ALT ACTION /* Deny Access */ }

        DMODE:PARTIAL {
           CONDITION  /* TRUE */
           ACTION     /* Notify Security */
           ALT ACTION /* No Alternative Actions */ }
     ]
```

In rule $R_{OffReq2}$, DMODE:COMPLETE handles authentication when the external door is opened and the office door is opened after that. Specifically, it handles Case 1 in Section 3 under Policy 1. DMODE:PARTIAL handles Case 3 (i.e., when the office door is opened without the external door opening). When triggered it notifies security personnel of a possible break-in.

5.2 Rules for Policy 2

Below we create a rule and associate it with the NOT complex event E_{Chk}. When events E_{Pick} and E_{Gate} occur (Case 1), it detects the non-occurrence of the checkout (E_{POS}) event. This detects the NOT event and triggers the DMODE:COMPLETE part of rule shown below.

```
RULE [  $R_{Chk}$
        EVENT  $E_{Chk}$

        DMODE:COMPLETE {
           CONDITION  /* TRUE */
           ACTION     /* Notify Security */
           ALT ACTION /* No Alternative Actions */ }
        DMODE:PARTIAL {
           CONDITION  /* TRUE */
           ACTION     /* Notify Security */
           ALT ACTION /* No Alternative Actions */ }
        DMODE:FAILED {
           CONDITION  /* TRUE */
           ACTION     /* Update Log */
           ALT ACTION /* No Alternative Actions */ }
     ]
```

When the detector/terminator event occur with other constituent events and no initiator, partial event is detected and the partial rule is triggered. In our example when E_{Gate} occurs alone (Case 6), or when E_{POS} and E_{Gate} occur (Case 5), it indicates some problem and should be notified. In either case the security is notified via the DMODE:PARTIAL part of rule. In addition, Cases 3 & 4 are also handled using timeout events and DMODE:PARTIAL part of the rule.

When all the events E_{Pick}, E_{POS} and E_{Gate} occur (Case 7), failed event is detected. This is because event E_{Chk} is modeling the non-occurrence of E_{POS}, but it has occurred. This triggers the DMODE:FAILED part of the rule.

All other cases where there is no occurrence of a detector/terminator can be handled using a timeout event.

6 Event Detection

Events specified using Snoop are detected using event detection graphs in the Local Event Detector [22]. In this section we will briefly explain event detection graphs that keep track or record of event occurrences for detecting complete, partial and failed events, and triggering appropriate rules.

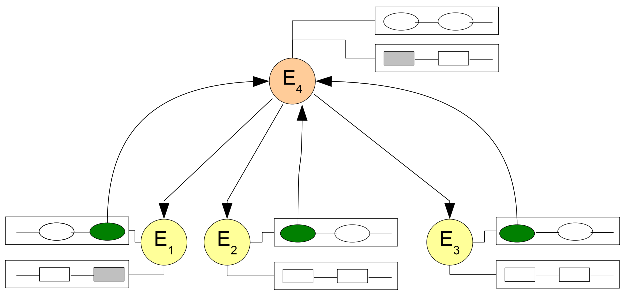

Figure 1: Event Detection Graph

Event Detection Graph:

Event graphs record event occurrences as and when they occur and keep track of the constituent event occurrences over the time interval they occur. They are acyclic graphs, where each complex event is a *connected tree*. In addition, events that appear in more than one complex event are shared.

Event graph shown in Figure 1 has three leaf nodes representing simple events E_1, E_2, and E_3. Internal node E_4 represents the complex event operator. The graph as a whole models a policy. In Figure 1, the complex event is a ternary event operator e.g., NOT. Although the child events are simple events in the Figure 1 it can be complex events. Each simple and complex event node check implicit and explicit condition expressions. In order to facilitate the propagation of events as and when they occur, each node in the graph has two lists; *event subscriber list* shown as connected ovals and *rule subscriber list* shown as connected rectangles. Colored shapes represent the presence of events and rules whereas empty shapes are just holders. Event subscriber list contains all the events that require this event (node) to propagate once this event is detected. Rule subscriber list contains all the rules that need to be triggered when the event represented by the node (leaf or internal) is detected.

Event Graph Construction:

Leaf nodes for simple events (E_1, E_2, and E_3) are constructed. Internal node (E_4) corresponding to the event operator is constructed. Parent (internal) node places its pointer in the event subscriber lists of all the child (leaf) nodes. Child nodes are linked with the parent nodes. Rules are created and linked with the appropriate nodes using the rule subscriber lists. The graph represents a policy modeled using a ternary operator e.g., Policy 2 from Section 3.

Event Propagation and Detection:

Event E_1 occurrence signals the left node. Using the event subscriber list the occurrence is propagated to the internal node E_4. Rules associated with the event are also triggered. Similarly events E_2 and E_3 are handled. Occurrence of the detector event E_3 invokes the event detection procedure in an event consumption mode (e.g., seq_recent() for SEQUENCE operator in recent consumption mode). The procedure evaluates both the implicit and explicit expressions. Once evaluated to TRUE, rule lists associated with the internal node are probed and rules are triggered. If there are any parent nodes that subscribed to E_4, then the occurrence is propagated. In Figure 1 there are no parent nodes for event E_4. Event detection algorithms are discussed in the next Section.

6.1 Extended Event Detection Graphs

Figure 2 represents Policy 2 discussed in Sections 3, 4 and 5. Event detection graph discussed above has been extended to handle event detection modes by creating and associating additional rule subscriber lists. Extended event detection graphs have four lists; *event*, *complete rule*, *partial rule* and *failed rule* subscriber lists. Complete rule subscriber list (shown as slanting line rectangles) contain all the complete rules that need to be triggered when the event represented by the node is detected. Partial rule subscriber list (horizontal line rectangles) and failed rule subscriber lists (vertical line rectangles) contain all partial and failed rules, respectively. Another approach would be to keep all rules in one list. But this would increase the computation needed to traverse and trigger appropriate rules.

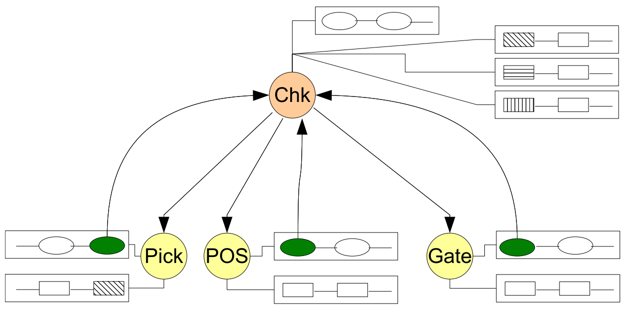

Figure 2: Event Detection Graph with Extensions

Event E_{Chk} defined in Section 3 is represented in Figure 2 as the internal node Chk. Events E_{Pick}, E_{POS} and E_{Gate} are represented as leaf nodes. As shown in the figure all the events in the leaf node are constituent events of the NOT event node. Events E_{Pick} and E_{Chk} have rules associated with them. In the figure, rule subscriber lists attached to event E_{Chk} have complete, partial and failed rules.

Event Propagation and Detection:

When event E_{Pick} occurs, left node is signaled and is propagated to the internal node. Rules associated with the event are triggered. When event E_{POS} occurs middle node is signaled and is propagated to the internal node. When event E_{Gate} occurs, right node is signaled and is propagated to the internal node. Occurrence of detector event E_{Gate} starts the event detection procedure. These procedures are discussed in the next section and they detect events, trigger rules and propagate events.

Impact of Event Detection Modes:

As mentioned earlier complex events can act as constituent events in other complex events. New event detection modes proposed in this paper affects how events are propagated after detection and how rules are triggered. We discussed in detail the impact of modes on rules and generalized the rule specification. On the other hand, we did not discuss impact of modes on the propagation of events to parents.

For example, consider the event NOT(A, B, SEQUENCE(C, D)), where sequence event acts as the detector/terminator event. Currently, the SEQUENCE(C,D) event is propagated to the NOT event when it is complete. Though event A can start the NOT event, only a complete sequence event can detect it. With the introduction of partial and failed events *propagation* of events to parent events have to be studied and is outside the scope of this paper. In addition to propagation, *specification* of complex events using modes needs to be studied as well. For example, can modes be included in the sequence event specification (e.g., SEQUENCE(C, D [Complete, Failed])) and be used to detect and propagate event occurrences.

In summary, with the current extensions: i) events are defined using existing specification, ii) rules are triggered using all three event detection modes, and iii) only events detected in complete event detection mode are propagated to parent nodes.

7 Event Detection Algorithms

Current event operator algorithms are based on point-based or interval-based semantics, and implementing event detection modes does not change the complexity of the algorithms. In other words, the asymptotic upper bound (i.e., $\mathcal{O}$) on the running time of the event operator algorithms does not change due to these extensions.

In the manner in which active rules are used for monitoring situations, events occur over a time line and are sent to the event detector (or propagated in the event graph). 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.

In this section, we will provide algorithms and the extensions that are required for SEQUENCE and NOT event detection in the *recent* event consumption mode [1, 20]. This mode is used by applications where recent event occurrences are of interest. In this mode whenever a new initiator occurs, it replaces the old occurrence of the initiator. Based on the discussion from Section 2, algorithms discussed below use *interval-based semantics* [1] for I_{Expr} and E_{Expr} is empty. Notations used in the algorithms are shown in Table 1.

Table 1: Notations used in Algorithms

e_i (e.g., e_1, e_2)	Simple or complex event instance or occurrence
E_i (e.g., E_1, E_2)	An event list that maintains the partial history of the occurrences of event e_i
t_s	Start time of the event occurrence
t_e	End time of the event occurrence

7.1 SEQUENCE Event Operator

Procedure *seq_recent()* shown in Algorithm 1 detects a SEQUENCE event in recent consumption mode using interval-based semantics. Each consumption mode has a corresponding procedure for event detection.

Algorithm 1: SEQUENCE Event Detection

```
/* $e_i$ can be recognized as coming
   from the left or right branch of
   the operator tree                    */
PROCEDURE seq_recent ($e_i$, parameter_list)
if $e_i$ is the left event then
    Replace $e_1$ in $E_1$
end
if $e_i$ is the right event then
    if $E_1$ is not empty and ($t_e(e_1) < t_s(e_2)$) then
        Pass $< (e_1, e_2), [t_s(e_1), t_e(e_2)] >$ to parent
        // Trigger Rules from the list
    end
end
```

Since SEQUENCE is a binary event operator there are two child nodes that can propagate. The left child node is the initiator and the right child is the detector/terminator. As shown in the algorithm whenever the left child node propagates an event, the first if statement is executed. This replaces the current initiator occurrence with the new one. Since the algorithm is for the recent mode, the initiator is not removed unless a new initiator occurs.

When the right or detector event occurs the time stamp of the occurrence is compared with the initiator's timestamp. If the detector follows the initiator, a new SEQUENCE event is constructed with the new combined timestamp and propagated. Rules from the rule subscriber list are also triggered.

Based on the proposed extensions, Algorithm 1 has to be modified to detect complete and partial events. Algorithm 2 is the extended version of Algorithm 1. As shown the handling of left child in not modified. Proposed extension have to be handled when the right child propagates the event occurrence. When a detector occurs, the list E_1 is checked. If there is an initiator that has occurred before this e_2, then a complete event is created and propagated and complete

Algorithm 2: SEQUENCE Event Detection Extension

```
PROCEDURE seq_recent ($e_i$, parameter_list)
if $e_i$ is the left event then
    Replace $e_1$ in $E_1$
end
if $e_i$ is the right event then
    if $E_1$ is not empty and ($t_e(e_1) < t_s(e_2)$) then
        Pass $< (e_1, e_2), [t_s(e_1), t_e(e_2)] >$ to parent
        // Trigger Complete Rules
    else
        // Trigger Partial Rules
    end
end
```

rules are triggered. If there is no matching initiator, then partial rule list is probed and rules are triggered in the *else* part. Instead of an else statement, an elseif can also be used which checks whether the initiator list is empty and then trigger partial rules. But that will not handle events that enter the system out of sequence. Also, partial events are not created and are not propagated to parent nodes currently, as discussed in Section 6.

Algorithm 3: NOT Event Detection

```
1  PROCEDURE not_recent ($e_i$, parameter_list)
2  if $e_i$ is the left event then
3      Replace $e_1$ in $E_1$
4      Flush $E_2$
5  end
6  if $e_i$ is the middle event then
7      if $E_1$ is not empty and ($t_e(e_1) < t_s(e_2)$) then
8          Append $e_2$ to $E_2$
9      end
10 end
11 if $e_i$ is the right event then
12     if $E_1$ is not empty and ($t_e(e_1) < t_s(e_3)$) then
13         if $E_2$ is empty then
14             Pass $< (e_1, e_3), [t_s(e_1), t_e(e_3)] >$ to the
               parent
               // Trigger Rules
15         else
16             if For all $e_2$ in $E_2$ and ($t_e(e_2) > t_s(e_3)$ or
               $t_s(e_2) < t_s(e_1)$) then
17                 Pass $< (e_1, e_3), [t_s(e_1), t_e(e_3)] >$ to
                   parent
18                 Flush $E_2$
                   // Trigger Rules
19             end
20         end
21         Flush $E_1$
22     end
23 end
```

7.2 NOT Event Operator

Procedure *not_recent()* shown in Algorithm 3 detects a NOT event in recent consumption mode using interval-based semantics. Similar to SEQUENCE operator discussion we assume E_{Expr} to be empty.

When the left child event node propagates an occurrence, current initiator is replaced by the new occurrence. Since this initiates a new NOT event, all occurrences of constituent event E_2 is removed. When the middle child event node propagates an occurrence, it is just appended to the list E_2 if it happens after the initiator. It is appended since the NOT event is detected when the detector event e_3 occurs.

When the detector/terminator event is propagated from the right child multiple conditions are checked. Lines 12 to 22 are executed if E_1 list in not empty and e_3 is a sequence of e_1. If E_2 is empty, then the NOT event is detected and propagated in line 14, and rules are triggered. If it is not empty, check for non occurrence of e_2 in the interval formed by e_1 and e_3 is carried out according to interval-based semantics. If there are no occurrences then a NOT event is detected and propagated in line 17, and rules are triggered. Event list E_1 is flushed (line 21) since e_3 acts as the terminator event.

Algorithm 4: NOT Event Detection Extension

```
1  PROCEDURE not_recent ($e_i$, parameter_list)
2  if $e_i$ is the left event then
3      Replace $e_1$ in $E_1$
4  end
5  if $e_i$ is the middle event then
6      Append $e_2$ to $E_2$
7  end
8  if $e_i$ is the right event then
9      if $E_1$ is not empty and ($t_e(e_1) < t_s(e_3)$) then
10         if $E_2$ is empty then
11             Pass $< (e_1, e_3), [t_s(e_1), t_e(e_3)] >$ to the
               parent
               // Trigger Complete Rules
12             Flush $E_1$
13         else
14             if For all $e_2$ in $E_2$ and ( $t_e(e_2) > t_s(e_3)$
               or $t_s(e_2) < t_s(e_1)$) then
15                 Pass $< (e_1, e_3), [t_s(e_1), t_e(e_3)] >$ to
                   parent
16                 Flush $E_1$
                   // Trigger Complete Rules
17             else
                   // Trigger Failed Rules
18                 Flush $E_2$
19                 Flush $E_1$
20             end
21         end
22     else
           // Trigger Partial Rules
23         Flush $E_2$
24     end
25 end
```

In order to handle the proposed extensions the procedure *not_recent()* shown in Algorithm 3 is extended in Algorithm 4. When the left event is propagated it replaces the existing initiator (lines 2 to 4). When the middle event occurs it is just appended to the list (lines 5 to 7). NOT event is detected in lines 10 to 12 if middle event E_2 list is empty, and complete rules are triggered. Since the NOT event is detected E_1 list is flushed. If E_2 is not empty then all occurrences in the list are checked to see if they do not occur after e_1 and before e_3. If none of the events e_2 in E_2 satisfy this condition, then a NOT event is detected and complete rules are triggered (lines 14 to 16). If event e_2 has occurred, then the non-occurrence has failed and failed rules are triggered (lines 17 to 19). If the initiator is empty, then partial rules are triggered (lines 22 to 24). Also, partial and failed events are not created and are not propagated to parent nodes currently, as discussed in Section 6.

8 Conclusions and Future Work

Active rules have been used for various applications in diverse domains. Though active rules are powerful in modeling enterprise policies they still lack some capabilities. Need for extending active rules were discussed using examples that are critical to the information security domain and RFID-based retail stores. Detailed analysis was performed on extending active rules along with event processing without modifying the event specification. New *event detection modes* such as complete, partial and failed were introduced. Though event specification has not been modified, rule specification has been generalized to handle these three new modes and future modes. Event detection graphs were extended and detection algorithms were modified to detect new event modes. Though these new modes seem to be simple on the face of it, they are powerful and inevitable as they extend the situations that can be monitored and policies that can be modeled by active rules.

In this paper, we discussed the new *event detection modes* and their impact on rules. On the other hand, the impact of event detection modes on nested complex event *specification* as well as *propagation* of events to parent events have to be explored. In other words, though events are detected in all modes only complete events are propagated to parent event nodes and the impact of propagating partial and failed events need to be analyzed. This would be critical to domains such as information security. In addition the impact of *event detection modes* on *event consumption modes* or contexts have to be explored in detail.

References

[1] R. Adaikkalavan and S. Chakravarthy, "SnoopIB: Interval-Based Event Specification and Detection for Active Databases," *DKE*, vol. 59, no. 1, pp. 139–165, Oct. 2006.

[2] J. Carlson and B. Lisper, "An Interval-based Algebra for Restricted Event Detection," in *International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS)*, ser. Lecture Notes in Computer Science, K. G. Larsen and P. Niebert, Eds., vol. 2791. Springer-Verlag, Sep. 2003, pp. 121 – 133.

[3] J. Mellin and S. F. Adler, "A formalized schema for event composition," in *Proc. of Conf on Real-Time Computing Systems and Applications (RTCSA)*. Tokyo, Japan: IEEE Computer Society, Mar. 2002, pp. 201–210.

[4] A. Galton and J. Augusto, "Two Approaches to Event Definition," in *Proc. of the DEXA*. Springer-Verlag, 2002, pp. 547–556.

[5] D. Zimmer, "On the semantics of complex events in active database management systems," in *Proc. of the ICDE*. Washington, DC, USA: IEEE Computer Society, 1999, p. 392.

[6] N. W. Paton, *Active Rules in Database Systems*. New York: Springer, 1999.

[7] C. Roncancio, "Toward Duration-Based, Constrained and Dynamic Event Types," in *Second International Workshop on Active, Real-Time, and Temporal Database Systems*. LNCS 1553, 1997, pp. 176–193.

[8] J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules*. Morgan Kaufmann Publishers, Inc., 1996.

[9] S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language for Active Databases," *DKE*, vol. 14, no. 10, pp. 1–26, 1994.

[10] S. Gatziu and K. R. Dittrich, "Events in an Object-Oriented Database System," in *Proceedings of Rules in Database Systems*, Sep. 1993.

[11] N. H. Gehani, H. V. Jagadish, and O. Shmueli, "Composite Event Specification in Active Databases: Model & Implementation," in *Proc. of VLDB*, 1992, pp. 327 – 338.

[12] "Complex Event Processing." [Online]. Available: http://www.complexevents.com/

[13] S. Chakravarthy and R. Adaikkalavan, "Provenance and Impact of Complex Event Processing (CEP): A Retrospective View," *Special Issue of Information Technology - Complex Event Processing*, vol. 51, no. 5, pp. 243–249, Sep. 2009.

[14] A. Hinze, K. Sachs, and A. Buchmann, "Keynote address: Event-based applications and enabling technologies," in *Proc. of the DEBS*, July 6–9 2009.

[15] A. Nagargadde, S. Varadarajan, and K. Ramamritham, "Semantic Characterization of Real World Events," in *DASFAA*, 2005, pp. 675–687.

[16] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin, "Complex event processing over uncertain data," in *Proc. of the DEBS*. New York, NY, USA: ACM, 2008, pp. 253–264.

[17] Z. Shen, H. Kawashima, and H. Kitagawa, "Efficient probabilistic event stream processing with lineage and kleene-plus," *International Journal of Communication Networks and Distributed Systems*, vol. 2, no. 4, pp. 355–374, 2009.

[18] R. Adaikkalavan and S. Chakravarthy, "Event Specification and Processing For Advanced Applications: Generalization and Formalization." in *DEXA*. LNCS 4653, Sep. 2007, pp. 369–379.

[19] R. Adaikkalavan and S. Chakravarthy, "Events must be complete in event processing!" in *Proceedings of the ACM symposium on Applied computing*. New York, NY, USA: ACM, 2008, pp. 1038–1039.

[20] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite Events for Active Databases: Semantics, Contexts, and Detection," in *Proc. of the VLDB*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 606–617.

[21] R. Adaikkalavan and S. Chakravarthy, "Formalization and Detection of Events Using Interval-Based Semantics," in *Proc. of the COMAD*, Goa, India, Jan. 2005, pp. 58–69.

[22] S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra, "Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules," *IST*, vol. 36, no. 9, pp. 559–568, 1994.

[23] R. Adaikkalavan and S. Chakravarthy, "Active Authorization Rules for Enforcing Role-Based Access Control and its Extensions," in *Proceedings, IEEE International Conference on Data Engineering (International Workshop on Privacy Data Management)*, Tokyo, Japan, Apr. 2005, p. 1197.

[24] M. Bishop, *Computer Security: Art and Science*. Addison-Wesley Professional, Dec. 2002.

[25] D. Gyllstrom *et al.*, "SASE: Complex Event Processing Over Streams," in *Proc. of the CIDR*, 2007.