# GENERALIZATION AND ENFORCEMENT OF ROLE-BASED ACCESS CONTROL USING A NOVEL EVENT-BASED APPROACH

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

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2006

To my father, mother, and sister's and brother's family.

## ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Sharma Chakravarthy for constantly motivating and encouraging me and for providing me great guidance and support during the course of this research work. I would like to thank him for being an excellent advisor and I am also looking forward to work with him outside of UTA. I would like to thank Professors Alp Aslandogan, Leonidas Fegaras, Mohan Kumar and David C. Kung for their interest in my research, for taking time to serve in my dissertation committee and for their comments, suggestions, guidance and help at the time of need.

I would like to thank J Carter M Tiernan for all her support, encouragement and guidance during the years I have been in UTA as a graduate teaching assistant and an instructor. I would also like to thank Medhat Saleh for all his help. I wish to thank all my colleagues (past and present) at ITLab for their support and encouragement and for making the stay at ITLab over the last five years more enjoyable. I am also grateful for my friends for their interest in my research and for their helpful discussions and invaluable comments. I would also like to thank all my friends who were with me in all walks of my life. I am grateful to all the professors and mentors who have helped me throughout my career, both in India and United States.

Finally, I am also extremely grateful to my parents Adaikkalavan Meyyappan and Soundaram Adaikkalavan, sister R. Meenakshi, brother-in-law MC. Ramanathan, brother A. Kasi Meyyappan, sister-in-law K. Anitha and niece R. Indhumeena and other family members for their endless love and support. Without their encouragement and endurance, this work would not have been possible.

# ABSTRACT

GENERALIZATION AND ENFORCEMENT OF ROLE-BASED ACCESS
CONTROL USING A NOVEL EVENT-BASED APPROACH

Publication No. _____

RAMAN ADAIKKALAVAN, Ph.D.

The University of Texas at Arlington, 2006

Supervising Professor: Sharma Chakravarthy

Protecting information against unauthorized access is a key issue in information
system security. Advanced access control models and mechanisms have now become nec-
essary for applications and systems due to emerging acts, such as the Health Insurance
Portability and Accountability Act (HIPAA) and the Sarbanes-Oxley Act. Role-Based
Access Control (RBAC) is a viable alternative to traditional discretionary and manda-
tory access control. RBAC has been shown to be cost effective and is being employed
in various application domains on account of its characteristics: rich specification, policy
neutrality, separation of duty relations, principle of least privilege, and ease of manage-
ment. Existing RBAC approaches support time-, content- and purpose-based, as well
as context-aware and other forms of access control policies that are useful for develop-
ing secure systems. Although considerable amount of effort has been spent on policy
specification aspects, relatively much less attention has been paid towards flexible en-
forcement of various aspects of RBAC approaches. Furthermore, current approaches are

inadequate, as many applications and systems require the more dynamic and expressive event pattern constraints.

In this thesis, we have focused on several aspects of RBAC, including generalization and enforcement of RBAC, by exploiting and extending a well-established event-based framework that has a solid theoretical foundation. Specifically, we have addressed the following problems and made the following contributions:

- Enforcement of existing RBAC Approaches: Security mechanisms are required for enforcing security policies. We have provided a flexible event-based technique for enforcing the RBAC standard and other current extensions in a uniform manner using an event framework. We have extended the event specification and detection with interval-based semantics for event operators and alternative actions for active rules.

- Generalization of RBAC and Snoop: We have generalized RBAC policies with expressive event pattern constraints. We have shown how to model diverse constraints, such as precedence, dependency, non-occurrence, and their combinations, using event patterns that are not available in existing RBAC approaches. Event patterns are event expressions that have simple and complex events as constituent events and they control the state change. Snoop, an event specification language, provides the basis for extensions needed to support the generalized RBAC. The generalization of RBAC using constraints based on event patterns can be accomplished by the extended Snoop.

- Enforcement of Generalized RBAC: We have shown the modeling and enforcement of generalized RBAC policies using the extended local event detector (LED). We have introduced *event registrar graphs* for capturing simple and complex event occurrences and keeping track of event patterns. We have also shown how RBAC with expressive event pattern constraints can be enforced using event registrar graphs.

When compared to other mechanisms, the proposed event-based enforcement mechanism has the advantage of using the same framework for both policy specification and enforcement. We have briefly explored identification and handling of policy conflicts.

- Usability in RBAC: We have enhanced the usability of RBAC by adding an intelligent module for discovering roles and guiding (or prompting) the user to acquire appropriate roles for performing operations on objects. This approach relieves the user from the details of role-permission assignment and allows concentrating on their task. We have developed several algorithms for discovering roles, and analyzed their complexity and effectiveness.

- Novel Applications: We have developed various applications for demonstrating the applicability of the results obtained in this thesis. i) We have shown how role-based security policies can be supported in web gateways using a smart push-pull approach. ii) We have shown how event operators based on interval-based semantics can be utilized for information filtering. iii) We provided an integrated model for advanced data stream applications that supports not only stream processing but also complicated event and rule processing. We have also shown how the integrated model can be utilized for a network fault management system.

This thesis is a first step in the direction of bridging the gap that currently exists between policy specification and enforcement. By mapping RBAC policies using a framework (event-based in our case) that can be incorporated with the underlying system in various ways (integrated, layered, wrapper-based, and distributed), we have not only extended RBAC to make it more useful, but also shown how the extended specifications can be mapped and enforced. This combination of specification and enforcement using a common framework forms the core contribution of the thesis.

**TABLE OF CONTENTS**

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

Protecting information against unauthorized access is a key issue in information system security. Unabated growth in the number and types of information repositories and the availability of improved access to these repositories makes it difficult to protect information from being accessed without appropriate permission. The problem of access control is more complicated in enterprises with large number of users and shared repositories. Furthermore, advanced access control and authorization models and mechanisms have now become a necessary requirement in applications and systems due to emerging acts, such as the Health Insurance Portability and Accountability Act (HIPAA) and the Sarbanes-Oxley Act. Earlier access control mechanisms [1, 2, 3, 4, 5, 6, 7] used by operating systems (e.g., Discretionary Access Control) or defense establishments (e.g., Mandatory Access Control) are no longer adequate for the complex information access that need to be managed today [8]. Role-Based Access Control (RBAC) [9, 10, 11, 12, 13] allows users to access information systems based on their current job functions (or roles). Current RBAC extensions support time-, content- and purpose-based, context-aware as well as other constraints [14, 15, 16, 17, 18, 19, 20, 21, 22, 23] that are useful for developing secure systems.

We introduce Role-Based Access Control and a well-established event framework in Sections 1.1 and 1.2, respectively. We motivate the need for flexible techniques for enforcing RBAC, generalizing RBAC and coupling both RBAC and the event framework, in Section 1.3, which are the main focuses of this thesis. In Section 1.4, we discuss and

1

list the specific problems addressed in this thesis providing the summary of contributions. We outline the thesis organization in Section 1.5.

## 1.1 Role-Based Access Control

Information assurance and security ensures the *confidentiality, integrity, authentication, availability*, and *non-repudiation* of information systems. Confidentiality (privacy, secrecy) prevents unauthorized users from *reading* and *learning* sensitive information. Integrity prevents unauthorized users from *modifying* objects or data items. Authentication *verifies* user's or subject's identity. Availability prevents *denial of service* or unauthorized withholding of information or resources. Non-repudiation provides *proof* of the origin or delivery of data.

"*Information security*, refers to security measures that implement and assure security services in communication systems (or communication security) and computer systems (or computer security) [24]." "*Communication security*, is a mechanism by which a person or process can communicate directly with a cryptographic module and that can only be activated by the person, process, or module, and cannot be imitated by untrusted software within the module [25]." Communication security [24] is also defined as "the measures that implement and assure security services in a communication system, particularly those that provide data confidentiality and data integrity and that authenticate communicating entities, and is usually understood to include cryptographic algorithms and key management methods and processes, devices that implement them, and the life cycle management of keying material and devices."

On the other hand, "*Computer security*, is a mechanism by which a computer system user can communicate directly and reliably with the trusted computing base (TCB) and that can only be activated by the user or the TCB and cannot be imitated by untrusted software within the computer [26]." Computer security [24] is also defined

as "the measures that implement and assure security services in a computer system, particularly those that assure access control service, and is usually understood to include functions, features, and technical characteristics of computer hardware and software, especially operating systems." Computer security deals with the prevention and detection of unauthorized actions by users of a computer system. Computer systems manipulate data and mediate access control for data items or objects.

*Access control* evaluates all access requests to resources by authenticated users and determines whether the requests must be granted or denied, ensuring both confidentiality and integrity. *Access control policies* correspond to the high-level rules describing the accesses to be authorized by the system and *mechanisms* implementing the policies via low level functions. In other words, "*A security policy is a statement of what is, and what is not, allowed (using Specification language, English)*" and "*A security mechanism is a method, tool, or procedure for enforcing a security policy*" [27]. In this thesis, we concentrate on the specification and enforcement of access control policies and their applications.

**Role-Based Access Control** (RBAC) [9, 10, 11, 12, 13], where object accesses are controlled by roles (or job functions) in an enterprise rather than a user or a group, has proven [28, 29, 30] to be an effective alternative to traditional access control mechanisms, such as Discretionary (DAC) and Mandatory Access Control (MAC) [1, 2, 3, 4, 5, 7]. DAC model (e.g., used in Unix) leaves a certain amount of access control to the discretion of the object's owner or to anyone else who is authorized to control the object's access. MAC or Multi Level Security (MLS) model (e.g., used in defense establishments) does not allow a user to own or change access rights. The need and the relevance of RBAC as an alternative to traditional access control mechanisms for efficiently handling complex authorization management of data in various domains are well-established. Role-Based Access Control has been shown as cost effective [31] and is being employed in various

domains [32] because of various factors: rich specification, policy neutrality, separation of duty relations, principle of least privilege, and ease of management.

**Role-Based Access Control Policies:** *Roles*, *Users*, *Permissions*, and *Sessions* are the basic ingredients of RBAC policies, where role represents job functions, and permissions represent objects and operations. Relationships between these basic element sets form the RBAC standard [12], which consists of four functional components. In the *Core RBAC*, users and permissions are *assigned* to roles. A user is granted access to an object when the user is *active* in a role that has the required permissions. For instance, user `Bob` will be allowed to `read` object `payroll.xls` when he is active in the role `Manager` (i.e., the role `Manger` has the permissions for object `payroll.xls`). In order for `Bob` to *activate* the role `Manager` he has to be *assigned* to that role. The *hierarchical RBAC* adds role hierarchies to the Core RBAC. With role hierarchies, roles (junior) inherit other roles (senior), where users assigned to senior roles are *authorized* to access objects of junior roles. Third and fourth functional components deal with providing separation of duty (SoD) constraints (static or dynamic) on Core or Hierarchical RBAC. A *Static SoD* constraint does not allow users to be *assigned* to mutually conflicting roles. A *Dynamic SoD* constraint does not allow users to be *assigned* to mutually conflicting roles in the *same session*. In RBAC, operations such as role-assignments (de-assignments), role-enabling (disabling) and role-activations (deactivations) are constrained for supporting fine-grained access policies. Most of the research have widely *explored* and *extended* RBAC for supporting various constraints, such as temporal, context-aware, control flow dependency, and so forth. For instance, user-role assignments can be constrained for a particular time period (e.g., allow user `Bob` to be active in role `Manager` only during weekdays between 8 a.m. and 6 p.m.). Enterprises in different domains [32, 33, 34] have different access control requirements. Constraints [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 35, 36, 37, 38] are critical in applying RBAC in diverse domains as they provide flexibility

and allow enterprises to specify fine-grained RBAC policies. Furthermore, RBAC with constraints support accurate modeling of object behavior in the real-world.

## 1.2 Event Framework

Initially, the active capability was proposed to meet some of the critical requirements of non-traditional (database) applications. The active capability extends the normal functionality of the underlying systems with support for monitoring user-defined situations and reacting to them without user or application intervention. These systems continuously monitor situations to initiate appropriate actions in response to the occurrences of interest (or events). Rules, also referred to as triggers, along with events provide active functionality. There is consensus in the database community on the Event-Condition-Action (or ECA) rules [39, 40] as being one of the most general formats for expressing event-based rules in an active database management system.

As the event component was the least understood part of the ECA paradigm (conditions correspond to queries, and actions correspond to transactions), there has been a large body of work on languages for event specification. Snoop [41, 42] was developed as the event specification component of the ECA formalism used as a part of the Sentinel project [43] for active Relational or object-oriented DBMSs. *Snoop provides well-defined event semantics for simple and complex events over various event consumption modes (or parameter contexts). The Local Event Detector (LED) provides the event detection mechanism for Snoop event specification using event detection graphs. Sentinel is an event framework that incorporates Snoop and LED for supporting event specification and detection and is used for situation-monitoring.* ECA rules can be defined either at application or system level, and are used to process event sequences and to make the underlying system active for applications, such as situation monitoring and change detection. In addition to active databases, these rules provide active capability for *applications* in sev-

eral other *domains* (e.g., XML [44, 45], RDF [46], semantic web [47], sensor databases [48], ubiquitous computing [49], web page monitoring [50], P2P database systems [51], information filtering [52], information retrieval, active security, and spatial data mining).

A number of event processing systems using the ECA paradigm have been proposed and implemented in the literature: ACOOD [53], ADAM [54], Alert [55], Ariel [56, 57], COMPOSE [58], Hipac [59], ODE [60, 61], REACH [62], Rock & Roll [63], SAMOS [64, 65], Sentinel [43], SEQ [66], UBILAB [67], and [68, 69]. A comprehensive introduction and description about most of these systems can be found in [39, 40].

An *event* was initially defined to be an instantaneous, atomic (that happens completely or not at all) occurrence of interest. An event, which is an indicator of happening can be either primitive (e.g., depositing cash in a bank) or composite (e.g., depositing cash, followed by withdrawal of cash). *Primitive or Simple* events occur at a point in time (i.e., the time of depositing). *Composite or Complex* events occur over an interval (i.e., the interval starts at the time cash is deposited and ends when cash is withdrawn). Composite events are defined using primitive events and event operators. An event expression specifies a composite event. The time of occurrence of the last event in an event expression is used as the time of occurrence for the entire event expression. Thus, primitive events are detected at a point in time, whereas the composite events can be detected either at the end of the interval (i.e., *detection/point-based semantics*) or can be detected over the interval (i.e., *occurrence/interval-based semantics*).

Snoop provides a large number of event operators: `AND`, `OR`, `NOT`, `Sequence`, `Plus`, `Periodic`, `Aperiodic`, `Cumulative Periodic` and `Cumulative Aperiodic`. Snoop uses the *detection/point-based semantics* for all these event operators. User-defined situations to be monitored are event expressions containing primitive events and composite event operators. LED uses an event graph or an event detection graph (EDG) for representing an event expression expressed in Snoop. EDG is used as opposed to other approaches

such as Petri nets used by Samos [64, 65] and an extended finite state automata used by Compose [58]. By combining event trees on common sub-expressions, an event graph is obtained. A data flow architecture is used for the propagation of primitive events to detect composite events. All leaf nodes in an event tree are primitive events and internal nodes represent composite events. By using event graphs, the need to detect 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 a substantial amount of storage space (for storing partial event occurrences and their parameters), thus leading to an efficient approach for detecting events. Event occurrences flow in a bottom-up fashion. When a primitive event occurs and is raised, it is sent to its leaf node, which propagates it to one or more parent nodes (as needed) for detecting one or more composite events.

Event consumption modes (or parameter contexts) [41, 42, 70] are needed for detecting events, since in an unrestricted context (where none of the event occurrences are discarded after participating in event detection) not all the detected events are meaningful for an application. Consumption modes essentially delimit the events detected and parameters computed, and accommodate a wide range of application requirements. The choice of a consumption mode also suggests the complexity of event detection and storage requirements for a given application. Snoop provides detection/point-based semantics for all event operators in four consumption modes: Recent, Chronicle, Continuous and Cumulative.

## 1.3  Research Motivations and Problems

The primary goal of this work is to explore various issues related to the generalization and enforcement of RBAC by exploiting the approach taken in Sentinel – by

incorporating access control policies as event-based rules that can be enforced using the event detection and rule execution framework.

- **Enforcement of RBAC and its Extensions:** Enterprises can model access control policies using either the RBAC standard or any of its extensions. Both the *specification* and *enforcement* are critical in employing these policies in real-world systems. Most of the research has *explored* and *extended* Role-Based Access Control with constraints, such as time-, content-, purpose-based, context-aware, and so forth for supporting authorization management of data in diverse domains. For example, the health care domain requires extensive temporal (e.g., day doctor, night doctor) and context-aware (e.g., emergency room, intensive care) constraints. On the other hand, most of these systems have concentrated mostly on policy specifications and very little on the ease of their enforcement.

  Access control policies are specified using various languages, such as rule specification languages [71], XML policy specification languages [72], pseudo-natural languages [73], rule definition languages, and so forth. Security mechanisms with existing systems or models [21, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81] are confined to a single policy or a particular form of constraints. For example, some systems do not support role hierarchies but they support separation of duty relations, whereas some other systems support subsets of temporal constraints. Furthermore, in most of the systems, policy specification and enforcement are decoupled. For instance, the system proposed in [73, 77, 78] converts policies specified in pseudo natural language into Java classes for enforcement via a sequence of steps. There is no clear separation between the system code and the RBAC policy enforcement code. Changes or alterations in policies are difficult to accommodate. Thus, in general, existing enforcement mechanisms do not provide a general and flexible suite of

techniques for enforcing the RBAC standard and other extensions in a uniform manner.

Sentinel – a well established framework for supporting event paradigm in an underlying system, integrates (either loosely or tightly) Snoop and the Local Event Detector. Active (or ECA) rules are widely used as a framework to make a system - a server, a database, an operating system or combinations thereof - active. It has been shown that active rules can be integrated with a system, or can be executed using a middle-ware (or an agent), or can be used with legacy systems using wrappers. Furthermore, they can be defined either at the system level or at the application level, and can be executed directly without further translations. Active rules not only have a well-defined semantics, but they can be added to existing systems and executed to *enforce access control policies*, if the policies can be mapped to active rules. Thus, the first challenge is to show the adequacy of active rules for enforcing RBAC approaches. Once access control policy specifications are translated to active rules, they can be directly executed for enforcing the policies they correspond to. In other words, policy specification and enforcement are not orthogonal.

- **Interval-based Semantics:** In all the existing event specification and detection systems, including Snoop and LED, composite events are considered as "instantaneous", although they occur over an "interval". Because of this, all the proposed event specification languages detect a composite event at the end of an interval over which it occurs (i.e., detection/point-based semantics). When events are detected using the point-based semantics, where event occurrence and event detection are not differentiated, it leads to non-intuitive detection of events [82, 83], when certain event operators, such as a sequence, are composed more than once. *Interval-based semantics* is proposed in this work to overcome the problems caused by the point-

based semantics. Both point- and interval-based semantics are needed as one is not sufficient for all applications. Furthermore, since event consumption modes are used in most applications, the interval-based semantics needs to be extended to event consumption modes as well. Even though there has been some work in providing interval-based semantics for Unrestricted [82] and Recent consumption mode [83], it needs to be extended to other modes [41] such as Continuous and Cumulative. Events that are detected using event consumption modes are subsets of events detected using the unrestricted context (except for the cumulative context). Thus, in order to use active rules for the enforcement of RBAC approaches, Snoop event operators have to be formalized in interval-based semantics in all consumption modes for *correct event detection.*

- **Generalization of RBAC and Event Specification:** Roles in Generalized Temporal RBAC [23] assume three different states: enabled, disabled and active. Role operations such as enabling, disabling, activation, and deactivation are *constrained* in RBAC as they control the role state changes. In addition, various other operations, such as assignments, de-assignments and user access requests, are also constrained. Currently, constraints can be specified in many ways: parameterized roles, predicates, simple role-dependent events (events based on role operations), simple role-independent events (events based on external factors such as time), environmental roles (e.g., location) and so forth. In many cases, the aforementioned constraints are inadequate for capturing real-world scenarios. Constraints based on *simple events* (occurrence of interest) and *event patterns* allow capturing of complex yet meaningful real-world polices and control the role and system state changes. As events are omnipresent, event-driven policies, verifications, and enforcements [20, 23, 38, 84, 85, 86], are gaining importance in the RBAC domain.

Simple events can be defined based on role operations (i.e., role-dependent events) or based on the underlying system or application or can be even external to the system (i.e., role-independent events). Constraints on simple role-dependent events are inadequate in many situations as access needs to be controlled based on event patterns (or composition of simple role-dependent or role-independent events). On the other hand, merely capturing the temporal history information over which the simple role-based events occur is not sufficient as the temporal dimension across different occurrence of events (complex events) is required. In other words, the temporal history over which the combinations of role-dependent, role-independent, role-interdependent events are spread across needs to be captured. Thus, a particular state (role or system) change $S_j$ in the system can be controlled by the occurrence of multiple events $E_1, E_2, \ldots, E_i$ following a particular pattern $E_p$ within a time interval $\Delta t$. $E_p$ represents the event pattern, and events $E_1, E_2, \ldots, E_i$ can be simple or complex events. Ordering of events that occur over a period of time $T$ can be specified using event patterns, which combine various constituent simple or complex events using logical, temporal, and other relationships. Thus, all operations in a RBAC system can be constrained by complex yet meaningful event patterns. Temporal RBAC [20] introduced simple *role events* providing periodical enabling and disabling roles using triggers. Generalized Temporal RBAC [23] extended this in the following ways: modeled three different states (disabled, enabled, active) for roles, duration constraints, time based semantics for role hierarchies and separation of duty relations. Nevertheless, as discussed above, constraints based on simple role events alone are insufficient.

Generalization of RBAC with constraints based on event patterns can be accomplished by combining the key features of the current RBAC approaches with Sentinel. For facilitating the generalization of RBAC, current event specification ap-

proaches including point- and interval-based event specification, have to be generalized due to various limitations. One of the major problems with the current event specification is the lack of support for attribute-based semantics, which is critical in information security. For example, an event pattern can be used to track the same user (i.e., an attribute) entering different wards in an hospital. Without the support for attribute-based semantics this cannot be modeled using event patterns. Another major problem with the current event specification is that they support only coarse-grained simple events. For instance, activating a user in a specific role cannot be supported. Thus, in order for generalizing RBAC with constraints based on event patterns, current event specification approaches, including Snoop and LED, have to be generalized as well.

- **Enforcement of the Generalized RBAC:** As mentioned previously, policy specifications need to be complemented with flexible enforcement mechanisms. Even though we have already discussed the motivation of policy enforcement was to mainly develop a flexible mechanism to enforce the existing RBAC approaches, with the generalization of RBAC using event pattern constraints, enforcement techniques also have to be generalized. Furthermore, these techniques must still be able to support current RBAC approaches and should not be disjoint from the policy specification. Sentinel cannot be used as is for the enforcement as the LED (event detection graph) is not designed to handle attribute-based simple and complex events. As simple and complex events have to be tracked for granting access based on event patterns, current event detection graphs have to be generalized. Thus, LED had to be generalize in a way that allows the enforcement of existing RBAC approaches and the generalized RBAC.

- **Usability in RBAC Approaches:** Computing systems have intertwined themselves into daily activities of humans and play a vital role. Well-known security

violations have occurred largely due to human errors and not because of system weakness according to a CompTIA's study ("Committing to Security Benchmark Study: A CompTIA Analysis of IT Security and the Workforce" [87]). In order to reduce security violations, there is an indispensable need to design secure yet usable systems. Traditionally, usability and human computer interaction [88] were not considered as critical design issues. Secure system design should include usability as an inherent design requirement (instead of an afterthought) as systems that are easy to use prevent human errors. On the other hand, we have to avoid building systems that are usable but not secure. With various constraints, RBAC models have the ability to provide fine-grained access control. Even though these policies allow security administrators to manage shared resources and information repositories in an efficient manner, users face some problems while using RBAC systems.

In RBAC, users and objects are *assigned* to one or more roles. Currently, there are some efforts [89, 90] that provide rule-based approaches for assigning users to roles in an effective way. On the other hand, users should be *active* in the role that has the required permissions before access is granted. Thus, they have to know what roles are required to perform operations on objects. In general, with respect to role activations, current systems follow the *human-active, system-passive* model. Users often get swamped with *role activations* due to numerous factors that include increase in the number of objects, multiple role assignments, and shifting roles often, and lean toward activating *all* the assigned roles violating the *principle of least privilege*. Thus, systems that enforce RBAC must be designed with usability to alleviate some of problems faced by users and security administrators.

- **Novel Applications:** Below we discuss several novel applications where we demonstrate the applicability of the results obtained in this thesis.

**RBAC Applications:** Efficient and effective web gateways or proxy servers are important to control the access privileges of users and protect private networks that are connected to the Internet, thus providing a productive and safe web environment. Access control in the form of complex access rules based on users or user sets (groups) has been studied extensively. The objective of this work is to provide role-based security for web gateways utilizing the RBAC. Role-Based security reduces the administrative burden, provides fine grained access control and supports various constraints, such as context-aware and temporal, seamlessly.

**Interval-Based Semantics Applications:** Information filtering includes monitoring text streams to detect patterns that are more complex than those handled by current search engines. Text stream monitoring and pattern detection have far reaching applications such as tracking information flow among terrorist outfits, web parental control, and business intelligence. Pattern characterization requirements of applications entail an expressive language for specifying complex patterns than what is currently provided by Information Retrieval Query Languages and current information filtering systems. In addition to complex pattern specification, we have to design effective pattern detection techniques for filtering information.

**Attribute-Based Semantics Applications:** Several event specification languages and processing models have been developed, analyzed, and implemented. More recently, data stream processing has been receiving a lot of attention to deal with applications that generate large amounts of data in real-time at varying input rates and to compute functions over multiple streams that satisfy quality of service requirements. A few systems based on the data stream processing model have been proposed to deal with change detection and situation monitoring. However, current data stream processing models lack the notion of composite event specification and computation, and they cannot be readily combined with event detection and rule
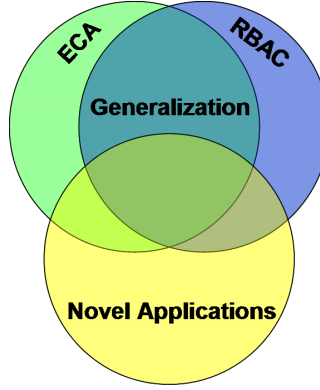
Figure 1.1. Research Contributions Overview.

specification, which are necessary and important for many applications such as the network fault management.

## 1.4   Summary of Contributions

In this thesis, we have generalized the RBAC using a novel event-based approach that combines the key features of the RBAC model with a powerful event framework. In short (see Figure 1.1), during the course of this research, we have extended Snoop operators with interval-based semantics to avert incorrect event detection, we have shown how ECA rules can be used to enforce RBAC approaches, we have generalized Snoop event specification and exploited it for generalizing RBAC with event pattern constraints, we have extended LED event detection mechanism for enforcing generalized RBAC policies and developed several novel applications for both generalized Snoop and RBAC. Below, we summarize our contributions along the lines of the motivations provided above.

- **Interval-Based Semantics:** We have formally defined Snoop event operators using interval-based semantics in continuous and cumulative contexts using event histories. These formal definitions include constraints that are based on the conditions over initiators, detectors, and terminators that should be satisfied for a particular context. Then, we have shown how online events are detected in interval-based

semantics using event detection graphs. The interval-based semantics have been implemented using partial-event histories or event graphs providing procedural semantics. Algorithms for all the operators have been developed for all the contexts including the unrestricted context. We have also shown that the events that are detected by these contexts using interval-based semantics are subsets of the unrestricted context. Most of the above mentioned works have been published in [91, 92, 93].

- **Enforcement of RBAC and its Extensions:** We have shown how active authorization rules or *extended* ECA rules are used to enforce RBAC, and its extensions such as temporal, and control flow dependency constraints in a uniform way. We have also shown how active security is provided to take timely actions to prevent malicious activities. The generated rules have different granularities and classifications based on their functionality. Mapping of the RBAC standard and its extensions using extended ECA rules provides a practically applicable view of RBAC. Event expressions and rules used to realize all of the Role-Based models can be easily enforced in any underlying system (that provide some hooks) to support Role-Based models. Most of the above mentioned works have been published in [85, 94].

- **Generalization of RBAC and Event Specification:** We have motivated the need for the RBAC generalization with constraints using event patterns. We have identified the advantages and limitations of Sentinel – Snoop and LED. We have generalized the current simple event definition and have identified the space of simple events that are required for constraint specification in RBAC. We have also generalized the current complex event operator definitions with both implicit and explicit condition expressions. We have introduced complete, uncomplete, and failed events and rules that are associated with each event type. We have

generalized RBAC with a comprehensive set of constraints based on event patterns. Event patterns with simple and complex events as constituent events have been shown to model constraints, such as precedence, dependency, non-occurrence, and their combinations, that are not available with existing RBAC approaches. Even though we have introduced various simple events and complex event operators that are useful in constraint specification, new operators can be plugged in seamlessly in our framework.

- **Enforcement of the Generalized RBAC:** We provide a flexible and generalized security mechanism for enforcing existing and generalized RBAC policies. We have identified the limitations of LED and event detection graphs. We have introduced *complete*, *uncomplete* and *failed* events and rules, and have shown how rules are used to check basic RBAC policy constraints. We have introduced *event registrar graphs* for capturing simple and complex event occurrences and keeping track of event patterns. In other words, these graphs maintain the temporal history over which the constituent events occur and detect those event patterns that are satisfied. Event registrar graphs follow a bottom-up data flow paradigm and are efficient as they allow the sharing of simple and complex events and event patterns. We have also shown how RBAC with expressive event pattern constraints can be enforced using event registrar graphs and active authorization rules. When compared to other mechanisms, our event-based enforcement mechanism is not disjoint from policy specification. In other words, generalized RBAC policies with event pattern constraints can be readily converted into event registrar graphs and executed. Finally, we have shown how policy conflicts can be identified and resolved, though they require further investigation.

- **Usability in RBAC Approaches:** We have made RBAC more usable for users by allowing them to use RBAC systems without the knowledge of what roles are

required to perform operations on objects. We have developed a number of algorithms for discovering roles and analyzed them. When the users get an access DENIAL message from our algorithms, it actually means that there are no roles that can be activated to make this request happen. This is a much *stronger* denial and is *more useful* than what the current systems provide. Our algorithms are general-purpose and can be used in any system that enforce RBAC approaches. Notifications provided by our approach allow users to concentrate on what data needs to be accessed rather than the roles that are required for access, and thus preserving principle of least privilege. Roles are disclosed to the user without any information leak. Although role discovery has its associated overhead with respect to system response time, it reduces user response time, increases user satisfaction or usability and supersedes other algorithms that provide binary replies and follow the *human-active, system-passive* model. We have published this approach and most of the algorithms in [95].

- **RBAC Applications:** We have provided a smart push-pull approach for supporting role-based security in web gateways. By leveraging RBAC, the number of access rules and their complexity is greatly reduced, thus reducing the administrative burden. Moreover, there are additional advantages, such as seamless constraint specification (e.g., time of the day, quotas based on bandwidth or time, IP address, location, etc.), and fine grained access control decisions. Providing role-based security by leveraging RBAC increases the level of security and productivity, creates a more secure perimeter around enterprise networks, and makes web a safer environment. Our approach enables the proxy server to act smarter, rather than just allow or deny access based on access rules, while preserving the principle of least privileges. We have enhanced the access control decisions (i.e., allow, deny, ask

for) from traditional web gateways or proxy servers that provide just binary access decision (i.e., either allow or deny). This work has been published in [96].

- **Interval-Based Semantics Applications:** We have developed a content-based system for filtering text streams. We support expressive user patterns using PSL (Pattern Specification Language) and provide filtering on streams and notification. With expressive patterns and well-defined semantics, PSL overcomes the limitations of the current information filtering systems used for specifying expressive user patterns. We have also developed PDG (Pattern Detection Graph), a bottom-up data flow paradigm for detecting the patterns specified using PSL. We have published this work in [52, 97].

- **Attribute-Based Semantics Applications:** We have integrated both event and stream processing systems *synergistically* to provide an end-to-end system for many advanced applications. We have provided an integrated model for advanced stream applications that supports not only stream processing, but also complicated event and rule processing. We analyzed the similarities and differences between the stream processing and the event processing models to identify a number of enhancements needed for both the models. We have shown how events can be generated from the stream processing system. We have also shown why current event processing models are inadequate for handling event streams and why attribute-based semantics for event operators are required. We have also extended the rule processing model for handling event streams. On the other hand, we have also proposed a network fault management system where the above mentioned integrated model is utilized. Most of these works have been published in [98, 99].

## 1.5   Thesis Organization

The rest of this thesis is organized as follows: We discuss the related work along the lines of our motivations and contributions in Chapter 2. We explain the interval-based semantics for event operators in Chapter 3. We show how extended ECA rules along with interval-based semantics can be used for the enforcement of existing RBAC approaches in Chapter 4. We discuss how we have generalized and combined both RBAC and Sentinel and how the generalized RBAC policies are enforced in Chapters 5 and 6, respectively. We show how we have made RBAC more usable in Chapter 7. We discuss various applications that we have developed to show the applicability of the results obtained from this thesis in Chapter 8. We conclude and provide future work in Chapter 9.

## CHAPTER 2

## RELATED WORK

### 2.1 Role-Based Access Control

Role-Based Access Control (RBAC) [9, 10, 11], where object accesses are controlled by roles (or job functions) in an enterprise rather than a user or group, has proven to be a positive *alternative* to traditional discretionary and mandatory access control mechanisms. RBAC allows organizations to form access control policies based on roles (or job functions) rather than users or groups. Users and permissions are assigned to roles, which act as the semantic center. RBAC models are policy neutral and they support principle of least privilege. RBAC does not provide a complete solution for all access control issues, but with its rich specification it has proven to be *cost effective* [31] by reducing the complexity in authorization management of data.

Lately, RBAC has been standardized (NIST RBAC [12]) and is defined in terms of four model components; core RBAC, hierarchical RBAC, and static and dynamic separation of duty relations. RBAC Reference Model defines sets of basic RBAC elements; users, roles, permissions, operations and objects. It also defines the relations between these basic elements as types and functions. NIST RBAC Standard [12] is defined in terms of four model components and their restricted combinations:

1) **Core RBAC:** defines relationships between three basic elements (i.e., users, roles, permissions). Permissions consist of objects and associated operations that can be performed on those objects. Core RBAC is shown in Figure 2.1.

2) **Hierarchical RBAC:** defines hierarchies between roles. "Mathematically, a hierarchy is a partial order defining a seniority relation between roles, whereby senior roles
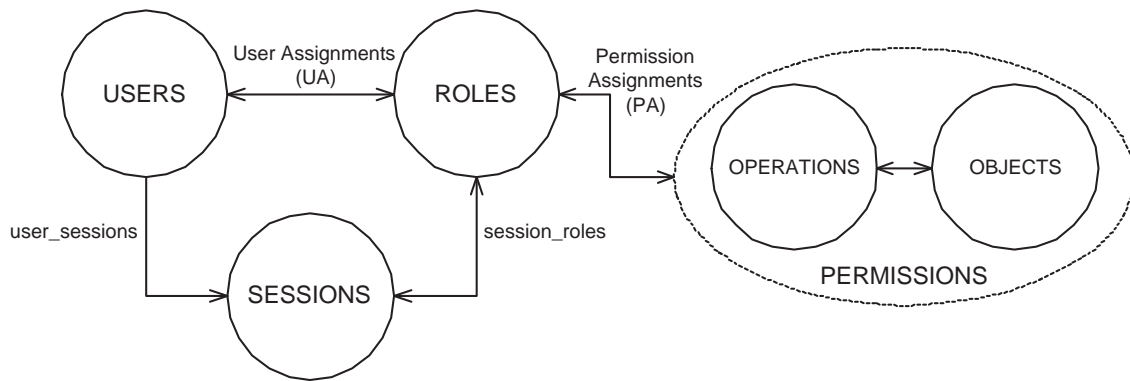
Figure 2.1. Core RBAC.

acquire the permissions of their juniors, and junior roles acquire the user membership of their seniors" [12]. Hierarchical RBAC is shown in Figure 2.2.
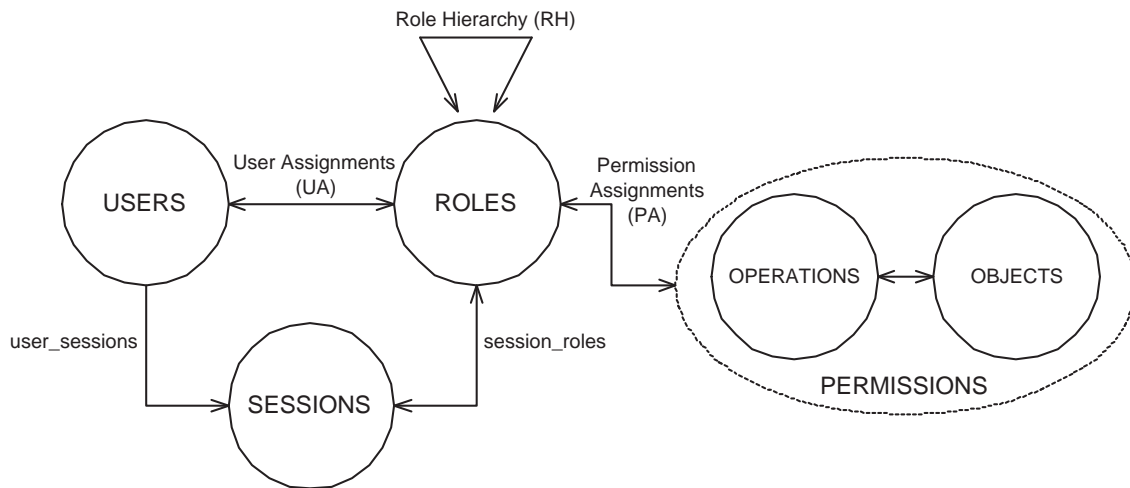


Figure 2.2. Hierarchical RBAC.

3) **Static Separation of Duty (SoD) Relations:** used to enforce conflicts of interest policies which may arise as a result of user gaining permissions to conflicting roles. Static SoD relations prevent these conflicts between roles by placing constraints on the assignment of users to roles. Static SoD relations model component defines relations

in both the presence and absence of role hierarchies. Static SoD relations with the presence of role hierarchies is shown in Figure 2.3.
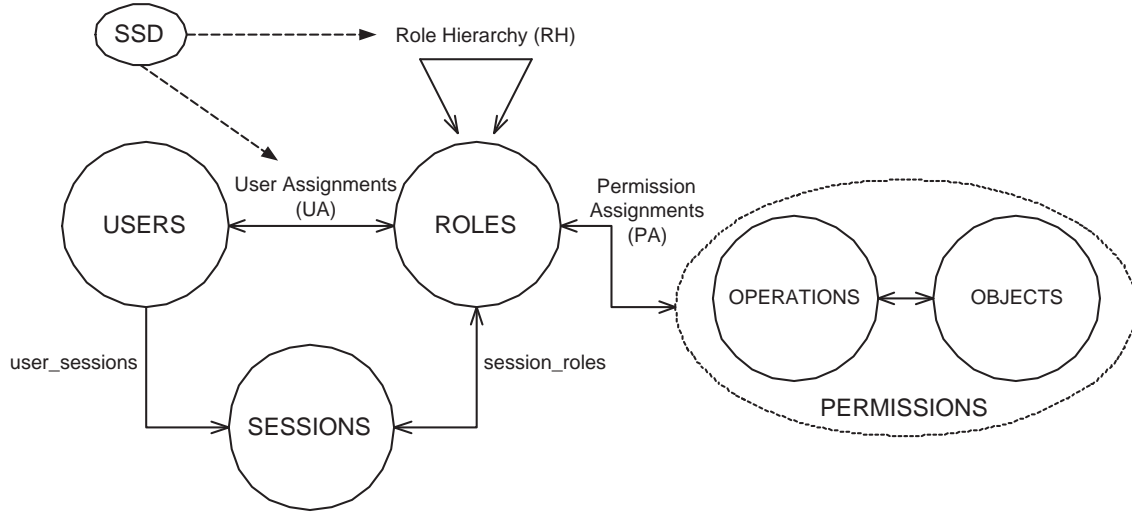


Figure 2.3. Static SoD Relations with Role Hierarchies.

4) **Dynamic SoD Relations:** these are similar to the static SoD that limits user permissions, but they differ by the context in which the constraints are placed. A user can be assigned to "m" (i.e., two or more) mutually exclusive roles, but cannot be active in "n" or more mutually exclusive roles at the the same time, where n $\geq$ 2 and n $\leq$ m. Dynamic SoD relations model component defines relations in both the presence and absence of role hierarchies. Dynamic SoD relations without the presence of role hierarchies is shown in Figure 2.4.

**RBAC Extensions:**

Standard RBAC alone does not suffice to handle various constraints that are required in diverse domains. For instance, hospitals and pervasive spaces require temporal and context-aware constraints. Furthermore hospitals might require to track the usage of patient records, thus, requiring usage-based or purpose-based access control. RBAC is being extended extensively, for supporting diverse domains in authorization management
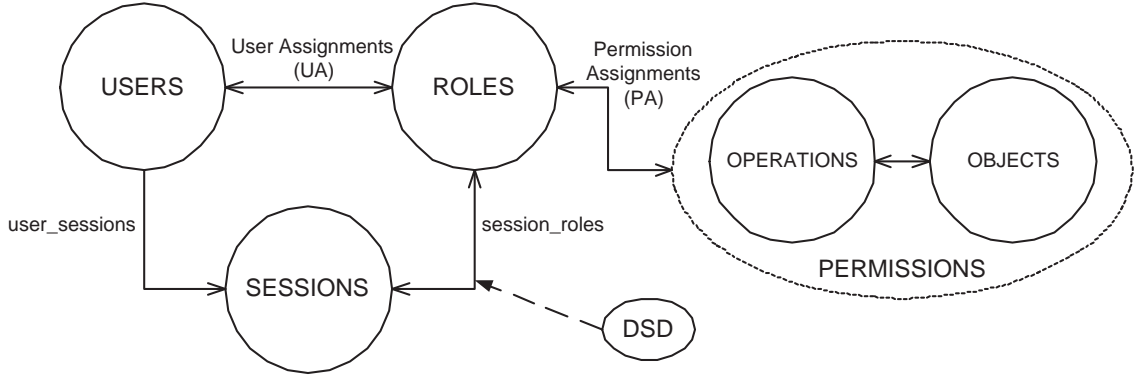
Figure 2.4. Dynamic SoD Relations.

of data, with various constraints such as temporal, context-aware constraints, privacy-aware and so forth. Generalizing RBAC, by supporting diverse constraints in a uniform manner, will make it applicable to a larger class of real life applications.

Constraints are crucial in an access control model [14, 17]. In particular, in Role-Based models, they are critical while managing the user-role and role-permission assignments, with or without the presence of role hierarchies or separation of duty relations. The importance of constraints has been shown in various domains with diverse applications [17]. RBAC has been extended extensively with various constraints for supporting diverse domains in authorization management of data. All the extensions are critical, as enterprises in different domains have different requirements. Below we have briefly explained some of the constraints:

- **Temporal [20, 23]:** Generalized Temporal RBAC extends the Temporal RBAC and provides an exhaustive set of temporal constraints. There have been lot of work in providing temporal constraints for RBAC. These constraints are critical in many domains, for example, health care domain requires extensive temporal constraints (i.e., controlling access to objects based on day-doctor and night-doctor).

- **Context-aware [19, 73]:** allows organizations to include context-aware constraints in RBAC. These constraints play a major role in many domains, for exam-

ple, pervasive computing environments require dynamic context-aware constraints (i.e., access to objects is allowed only in a secured network).

- **Privacy-aware [100]:** allows organizations to include purpose based access control constraints in RBAC.

- EVENT: Controlling the access to objects is based on any arbitrary event. Those events can be temporal (time clock/ system time), location-based (from a sensor), system predefined (invoking of any application or process), database triggers (insert, delete or update of tuples) and so forth.

### 2.1.1 Policy Enforcement Mechanisms

Current systems enforcing RBAC are custom-implemented, domain-specific and are confined to particular form of constraints. None of the existing systems, to the best of our knowledge, support the complete RBAC standard and its extensions in a seamless way. Thus, we explain some of the current systems below along with the features they support.

OASIS [73, 78, 77] contains two types of rules and they are 1) activation rules, and 2) authorization rules. It supports dynamic role deactivations by use of rules, and it does not support role hierarchies explicitly and cardinality constraints. With the extended model, OASIS supports some temporal constraints, and context dependent constraints in the form of environmental predicates. OASIS requires administrators to specify enterprise policies using pseudo-natural language (i.e., Restricted English [78]) for authentication and authorization. These predicates are translated into a series of forms such as higher-order logic, first-order predicate calculus, horn clauses and finally converted to Java classes. The generated Java classes change as the authorization rules change. Even though OASIS uses pseudo-natural language that is transparent, it is

cumbersome and a cognitive-burden for administrators. The implementation of OASIS is not clearly discussed except the fact that it takes a middle-ware approach.

Adage [71, 101], a rule-based authorization system for distributed applications, supports separation of duty by using history based constraints. The system does not support important RBAC features such as role hierarchies and cardinality constraints. It requires the administrators to specify the authorization rules manually. X-GTRBAC [72] is an XML based policy specification framework and architecture for enterprise wide access control. The framework supports GTRBAC specifications [23] and enforces a set of GTRBAC constraints. The model does not support time based SOD. It requires administrators to specify policies using X-GTRBAC specification language based on a BNF-like grammar, called X-Grammar provided by the framework.

Temporal RBAC [20] "supports periodic role enabling and disabling, and temporal dependencies among such actions." Role triggers are used to express these temporal dependencies and to enable and disable roles either immediately or after a specified amount of time. On the other hand, in this work we show how active rules are used to support RBAC, its extensions and active security, seamlessly. We have also shown how the Generalized Temporal RBAC [23, 37] can be enforced, using examples. [102, 103] show how multipolicy access control is supported but they do not explicitly consider the extended RBAC with various constraints and active security. [80] shows how context constraints are enforced in an RBAC environment.

Attribute-Based RBAC or AB-RBAC [89, 90] is a rule-based model that was developed to assign users to roles automatically, based on the authorization rules defined by enterprise administrators. Rule-based language defined in AR-RBAC supports minimal temporal aspects in the form of range constraints, but is not expressive enough to support various other constraints. In addition, this model also requires enterprise administrators

to manually specify rules using the RB-RBAC language. Role hierarchies are induced from the seniority among authorization rule attributes when all the assumptions hold.

In addition to these systems, several commercial and open source systems [74, 81, 104, 105, 106, 107, 108] support some form of RBAC.

### 2.1.2 Constraint Specifications

Role-based access control has been extended extensively with various constraints [14, 15, 16, 17, 18, 19, 20, 21, 22, 23]. All these constraints are critical in providing fine-grained access control and for making RBAC usable over diverse domains. Some of the current models or systems [20, 23, 38, 37] support event-based constraints, minimally. With these models or systems only simple role events and their restricted boolean combinations are allowed. We have extended RBAC with expressive event-based constraints, which allow the modeling of event ordering with non-occurrence, precedence, dependency, control-flow and many other constraints. Event-based constraints can be specified over various granularities.

Bertino et. al, provided event driven RBAC policies in TRBAC [20]. TRBAC supports periodic enabling and disabling of roles and handle their temporal dependencies with role triggers. It allows runtime requests by administrators for enabling and disabling of roles. Role events were associated with priorities to handle conflicts. It also provided a specification language, its formal semantics and a polynomial safeness check to reject inconsistent specifications. Even though extensions provided by TRBAC were necessary in many situations they were not sufficient.

Joshi et. al. extended TRBAC as Generalized TRBAC [23]. extends the TRBAC model with temporal constraints for user-role and permission-role assignments, modeling of three states for roles (active, enabled, disabled), duration and role activation constraints, and time-based semantics of role hierarchies and separation of duty constraints

in the presence of temporal constraints. In general, it provides extensive temporal constraints and overcomes the problems present in TRBAC. Even though [37] extends trigger capability with generalized triggers it assumes only conjunction and disjunction of role events. Shafiq et al. proposed a Petri-Net based verification framework [38] for event-driven RBAC polices, based on roles with three states. These policies are modeled using the extended set of consistency rules [109]. These extended set of rules model cardinality, inheritance, separation of duty, precedence and dependency constraints.

In the above strands of work, simple role-dependent events such as assignments/de-assignments, activations/deactivations, and enabling/disabling are used for the specification of constraints. But these simple events act as used in the specification of simple dependency and precedence constraints. On the other hand, with expressive event constraints both simple and complex events can act as sub-patterns in event patterns, allowing the modeling of real-world situations. In addition, we have introduced new constraints such as non-occurrence and have allowed the specification of constraints based on role-dependent and role-independent events.

## 2.2   Interval-Based Semantics

Although there is a considerable amount of work on event specification languages and operators (ACOOD [53], ADAM [54], Alert [55], Ariel [56, 57], COMPOSE [58], Hipac [59], ODE [60, 61], REACH [62], SAMOS [64, 65], and Snoop [41, 42]), all previous proposals have used detection- or point-based semantics. All these languages detect composite events using different approaches. For example, Sentinel uses event graphs to detect a composite event, whereas Samos uses Petri nets to detect the composite events, but all of them use point-based semantics.

Liebig, C., et al., [110] provide algorithms for event composition and consumption using accurate interval based time stamping that guarantees the property of time

consistent order. They also illustrate the same with a window mechanism to deal with varying transmission delays when composing events from different sources. Though the paper claims that event consumption modes like recent and chronicle can be unambiguously defined by using an accurate interval-based time stamping, it uses the point-based semantics for the composite event detection and has the same drawbacks.

The need for interval-based semantics for event detection is explained with concrete examples by Rönn, P., in [111], using Snoop operators, but does not deal with formal semantics, algorithms and implementation for any of the context in Snoop. Roncancio, C.L. mentioned about detecting events using the duration-based (i.e., interval-based) semantics in [112]. The need for duration-based semantics, operators that are supported, and the formal semantics and implementation of those operators are not explained.

## 2.3   Usability in RBAC

None of the existing systems that enforce RBAC and its extensions [21, 71, 72, 73, 76, 80, 102], to the best of our knowledge, provide discovery-based role activations.

Similar to role activations, assigning users to roles is a critical problem. [89] provides attribute-based RBAC, a rule-based model for assigning users to roles automatically, based on the authorization rules defined by enterprise administrators. It discusses user role assignments, but not the roles that have to be activated by the users. There has been some work that deal with the disclosure or release of policies [113], automatic trust negotiations [114], reasoning services in autonomic communications [115] and so forth. On the other hand, these systems and do not provide discovery-based role activations, and do not deal with complete RBAC and its extensions.

## 2.4   Role-based Security for Web Gateways

Web filtering plays a significant role in providing a more productive and safe web environment. Proxy servers or web gateways do not replace web servers and firewalls,

but complement them by providing multi-layer security for enterprise networks. Many tools/systems have been developed to support access control based on users and groups, and to the best of our knowledge none of the current systems support Role-Based (RB) security. Thus, some of the commercial and research products/models that are related to *SmartGate* are explained below.

Comparisons of the commercial systems are available in [116] and all these systems support various features, but access control is based only on users or user sets.

**BlueCoat's Advanced Web Proxy** [117] supports various granular web access policies based on users, groups, time of the day, location, network, address, user agents and other application domain based attributes. However, it does not support RB security while providing access control. **SurfControl's Web Filter** [118] incorporates quality content understanding, adaptive reasoning technology and supports sophisticated filtering rules based on users and groups, precise bandwidth control, prioritized bandwidth, time based application of rules, time/ volume thresholds. Even though Web Filter provides tools required to control access, these are not based on roles, and thus does not have the advantages of RB security. **Websense's Enterprise** [119] supports customized Internet access policies based on user, group, department, workstation or network. Again it does not support RB security. Similarly, **Microsoft's ISA 2004** [120] and **Novell's Border Manager** [121] do not support RB security.

In addition to the above systems, RB security has been provided for web servers in [122] and is used for providing access control for inbound traffic. Even though it provides RB security, it still follows the access/deny model as opposed to the ternary (which can be expanded further) model followed in *SmartGate*. Roles corresponding to users are provided using the secure cookies or smart certificates for accessing a web page from a web server. Users are provided with the roles in the form of cookies or certificates as they can be connected to the Internet from anywhere. Once authenticated, users are provided

with the set of assigned roles, which does not satisfy the principle of least privilege. On the other hand, cookies or certificates are not required since access control in web gateways are for network users who are authenticated and their roles are activated by the underlying system. *SmartGate* provides RB security for web gateways, essentially for the outbound traffic from an enterprise network to provide a safe web environment. Contrary to [122], *SmartGate* satisfies the principle of least privilege as not all the assigned roles need to be active all the time.

## 2.5 Advanced Information Filtering

Commercially and freely available information filtering systems were developed to provide solutions to assist users in extracting relevant information. SIFT [123, 124] designed at Stanford University, is a content-based filtering system. Boolean queries and Vector Space Model [125] are used to construct user profiles, allowing users to specify keywords that are to be included and those that are to be excluded, when filtering USENET articles. However, SIFT does not consider structural information while filtering documents. It makes no distinction between positions of words in a structured text, such as those appearing in a title or body. It does not support proximity, regular expressions, frequency, structural, and sequence operators. GLIMPSE (Global Implicit Search) [126], another content-based filtering system utilizes indexing and query schema for retrieving files. It supports Boolean queries and approximate matching such as regular expressions. However, it does not support proximity, frequency, structural, sequence, and compositions of queries. Igrep [127] is an approximate matching tool for large data collections. It accepts words, phrases, and set of characters such as wild cards, ranges, etc. Regular expressions are supported to provide approximate matching. Again, it does not support frequency, structural, sequence, and not all the compositions of queries. Inquery [128], is a ranked retrieval system, and Lemur [129], a toolkit based on Inquery is used for

Language Modeling and Information Retrieval. Both the above are typical information retrieval (IR) systems and are not information filtering systems, that can filter patterns over text streams. Even though Inquery's language supports structural queries it does not have the capability to search within a paragraph, or within a range of words, which is possible using InfoFilter's WITHIN operator. It also does not support frequency.

## 2.6    Event Streams and Network Fault Management

Our work on *Event Stream Processing* is directly related to a set of papers [130, 131, 132, 133, 134, 135], which mainly focus on the system architecture, Continuous Query (CQ) execution (i.e., scheduling and various non-blocking join algorithms), and QoS delivery mechanisms for stream processing. The main computations over stream data are limited to the computation of relational operators over high-speed streaming data, and the event and rule processing and the extensions to CQs to enhance their expressive power and computation efficiency are rarely discussed. A number of sensor database projects, Cougar [136, 137], TinyDB [138, 139] have also tried to integrate the event processing with query processing under a sensor database environment. However, the event-driven queries proposed in TinyDB is used to activate queries based on events from underlying operating systems. Our focus in this work is to process large number of high volume and highly dynamic event streams from CQ processing stage for the applications that needs complex event processing and CPU-intensive computation (i.e., CQs) for generating events.

Our work in *Network Fault Management* is related to a set of papers that have tried to provide various solutions for network fault management. [140, 141, 142] discuss a distributed architecture for network management system and the proposed systems are domain specific, rather than an inter-domain system as proposed in this paper. However, most of the telecommunication service providers employ a central fault management

system because of the high demands of various experts and efficiency of their collaborations to fix malfunctions. [143] proposed an architecture for an multi-layered network through various interfaces to exchange information between domain-managers and an inter-domain manager. It is also short on flexibility like other network fault management systems. The architecture proposed in this paper takes a totally different approach and greatly simplifies the complexities of the system and increases its flexibility through CQs.

## CHAPTER 3

## INTERVAL-BASED EVENTS AND THEIR SEMANTICS

Point- or detection-based event semantics does not differentiate between event detection and occurrence and has been used for detecting events in most of the systems that support Event-Condition-Action rules. However, this is a limitation for many applications that require interval- or occurrence-based event semantics. In this chapter, Snoop (an event specification language) event operators have been formalized using interval-based semantics (termed SnoopIB) in various event consumption modes. Local event detector (LED) for detecting simple and complex event detection using event detection graphs is discussed. Representative algorithms for detecting SnoopIB operators and their implementation in the context of Sentinel (a system incorporating Snoop and LED) is also discussed.

### 3.1  Introduction

There is consensus in the database community on Event-Condition-Action (or ECA) rules as being one of the most general formats for expressing rules in an active database management system [39, 40]. Furthermore, it has been shown that an underlying system can be made active capable using ECA rules. 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 languages for event specification and detection. Snoop [41, 42] was developed as the event specification component of the ECA rule formalism used as a part of the Sentinel project [43] on active object-oriented DBMS. On the other hand, local event detector (LED) provides the event detection mechanisms in Sentinel.

An event was initially defined to be an instantaneous, atomic occurrence of interest and the time of occurrence of the last event in an event expression was used as the time of occurrence for an entire event expression (point-based semantics), rather than the interval over which an event expression occurs (interval-based semantics). This introduces semantic discrepancy for some operators when they are composed more than once [82, 83]. In all event specification languages used in active DBMSs (ACOOD [53], ADAM [54], Alert [55], Ariel [56, 57], COMPOSE [58], Hipac [59], ODE [60, 61], REACH [62], SAMOS [64, 65], and Snoop [41, 42]) even composite events are considered as "instantaneous", although they occur over an "interval". Because of this, all the proposed event specification languages detect a composite event at the end of an interval over which it occurs (i.e., point-based semantics). Interval-based semantics is needed to overcome the problems that are caused by the point-based semantics.

Snoop event operators were formally defined in the recent context (i.e., for applications where the events are happening at a fast rate and multiple occurrences of the same type of event only refine the previous data value, e.g., sensor applications.) using interval-based semantics in [83, 144]. Interval-based semantics has substantial differences when compared to the point-based semantics. Below, we explain event detection with a detailed example using both point-based and interval-based semantics, and highlight why the distinction between point-based and interval-based semantics is critical.

### 3.1.1 Event Detection

We will first identify the conditions that are needed to detect an event. Primitive events are predefined in the system and are detected at the time of occurrence. Composite event detection on the other hand involves two steps:

1. checking the detection condition based on the operator semantics
2. determining the time of detection.

These two steps are *handled differently* in point-based semantics and interval-based semantics as explained below. Consider an example where a stock trading agent wants to take some action when "Dow Jones Industrial (DJI) average increases by 5 percent, *followed by* a 5 percent price increase in Sun Microsystem stocks *and* a 2 percent price increase in IBM stocks". The stock trading agent uses Snoop event operators to express this requirement.

"If $(DJIA \gg (Sun \, \Delta \, IBM))$ then take the appropriate action".

- DJIA, Sun, IBM are primitive events that correspond to DJI average increases by 5%, 5% price increase in Sun stocks, and 2% price increase in IBM stocks, respectively.

- $(Sun \, \Delta \, IBM)$ and $(DJIA \gg (Sun \, \Delta \, IBM))$ are composite events, where "$\gg$" (snoop "sequence" event operator) represents a sequence condition and "$\Delta$" (snoop "AND" event operator) represents the AND condition.

- "$\gg$" detects a sequence of two events whenever the first event happens before the second event. "$\Delta$" detects an "AND" event whenever both the events happen. Semantics for these operators are different for point-based semantics and interval-based semantics and are illustrated below.

### 3.1.1.1 Point-Based Semantics

*Case 1:* Assume that primitive events $DJIA$, $Sun$, and $IBM$ occur at 10.30 a.m., 10 a.m., and 11 a.m. respectively. Figure 3.1 depicts events that are detected along with the times of occurrence and detection.

1. Primitive events are detected at the time of occurrence. Thus, events DJIA, Sun, and IBM are detected at 10.30 a.m., 10 a.m., and 11 a.m., respectively.

2. Composite event $(Sun \, \Delta \, IBM)$ detection involves two steps, as previously mentioned:
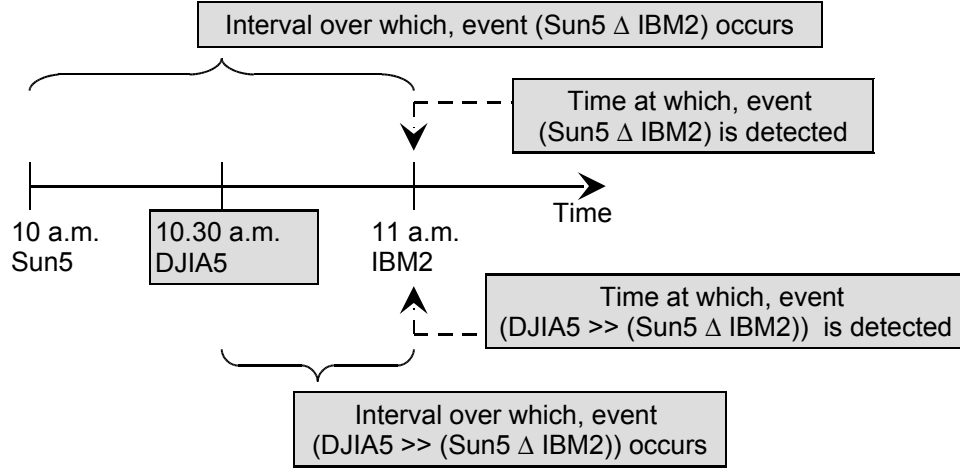
Figure 3.1. Point-Based Semantics.

(a) As both the events Sun and IBM have occurred the "AND" condition is satisfied.

(b) Sun starts the composite event $(Sun \; \Delta \; IBM)$ detection at 10.00 a.m. IBM terminates the composite event $(Sun \; \Delta \; IBM)$ at 11 a.m. Since events are considered as "instantaneous" and are detected at the end of the interval, the composite event $(Sun \; \Delta \; IBM)$ is detected at 11 a.m. (though it occurred over an interval from 10 a.m. to 11 a.m.).

3. Composite event $(DJIA \gg (Sun \; \Delta \; IBM))$ detection involves two steps:

(a) The "Followed by" condition is only satisfied when the event DJIA happens before the event $(Sun \; \Delta \; IBM)$. In our example, this condition is satisfied, since event DJIA is detected at 10.30 a.m. (step 1) and event $(Sun \; \Delta \; IBM)$ is detected at 11 a.m. (step 2b). Thus, the event DJIA happens before the event $(Sun \; \Delta \; IBM)$ (i.e., 10.30 a.m. < 11 a.m.).

(b) The Composite event $(DJIA \gg (Sun \; \Delta \; IBM))$ is detected at 11 a.m. (even though it occurs over an interval from 10.30 a.m. to 11 a.m.)

Even though the event $Sun$ had occurred well before the event $DJIA$ (i.e., 10.00 a.m. < 10.30 a.m.), the composite event $(DJIA \gg (Sun \; \Delta \; IBM))$ is detected, which is incorrect. This is because of the condition checking in step 3a fails to capture the correct semantics, since it does not consider the start of the interval. From this, it is evident that point-based semantics does not detect events in the correct way under certain patterns of event occurrences. The point-based semantics typically used by all the aforementioned event specification languages used in active DBMSs does not differentiate between event occurrence and event detection and have similar problems.

### 3.1.1.2   Interval-Based Semantics

The same examples are used to explain interval-based semantics in this section, where both times (start and end) are considered. This brings out the need for intervals for real-world events and the re-examination of the instantaneous occurrence assumption.

*Case 2:* Let us assume that the primitive events $DJIA$, $Sun$, and $IBM$ occur at 10.30 a.m., 10 a.m., and 11 a.m. respectively. Figure 3.2 depicts events that are detected, along with the time of occurrence and detection.

1. Primitive events are detected at the time of occurrence, thus the events DJIA, Sun, and IBM are detected at [10.30 a.m., 10.30 a.m.], [10 a.m., 10 a.m.], and [11 a.m., 11 a.m.], respectively. As shown above, the primitive events have the same start and end times.

2. Composite event $(Sun \; \Delta \; IBM)$ detection involves two steps, as previously mentioned:

   (a) The "AND" condition is satisfied, since both the events have occurred.

   (b) With the interval-based semantics, the start time of an event is considered and is detected over an interval. As explained in Case 1, the event Sun starts
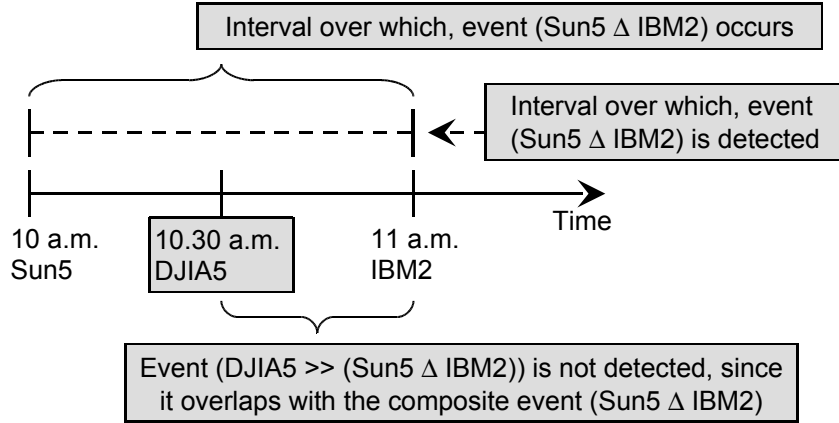
Figure 3.2. Interval-Based Semantics.

and the event IBM terminates the composite event $(Sun\ \Delta\ IBM)$, but the composite event is detected over the interval [10 a.m., 11 a.m.]

3. Composite event $(DJIA \gg (Sun\ \Delta\ IBM))$ detection involves two steps:

(a) The "Followed by" condition is only satisfied when the event DJIA happens before the event $(Sun\ \Delta\ IBM)$. In our example, this condition is not satisfied, since the event DJIA is detected at 10.30 a.m. in step 1, and the composite event $(Sun\ \Delta\ IBM)$ is detected over the interval [10.00 a.m., 11 a.m.] in step 2b. Thus, the event DJIA does not happen before the event $(Sun\ \Delta\ IBM)$ (i.e., 10.30 a.m. $\not<$ 10 a.m.).

(b) The composite event $(DJIA \gg (Sun\ \Delta\ IBM))$ is not detected.

Both the cases 1 and 2 detect the composite event $(DJIA \gg (Sun\ \Delta\ IBM))$ using the same set of primitive events. Case 1 uses point-based semantics and the composite event is detected, which is not correct as the event Sun had occurred well before the event DJIA. Case 2 detects events using interval-based semantics, where the composite event is correctly not detected. Thus, event detection using interval-based semantics is needed
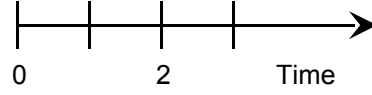
Figure 3.3. Time Line.

for detecting events correctly. The rest of the chapter elaborates on event operators using the interval-based semantics.

Although the above example makes a case for the need for alternative semantics, whether an application/domain can live with non-interval-based semantics is a design decision. It is clear that both detection/point-based and interval-based semantics are needed as one is not sufficient for all applications. Furthermore, since event consumption modes are used in most applications, the interval-based semantics needs to be extended to event consumption models as well. Events that are detected using event consumption modes are subsets of events detected using the unrestricted context. Snoop supports four event consumption modes (i.e., Recent, Chronicle, Continuous, and Cumulative) and a number of event operators. First, we formally define the Snoop event operators in both Continuous and Cumulative contexts using event histories (i.e., a log). The Snoop event operators were formalized in Recent context in [83, 144]. On the other hand, it is nearly impossible to formalize operators in Chronicle context. Second, we show how Snoop event operators detect events occurring online (i.e., one event at a time) using event detection graphs, and comment on the implementation of event operators in Sentinel. Finally, we present a few representative event operator algorithms for event detection.

## 3.2    Interval-Based Semantics of Snoop

We assume an equidistant discrete time domain having "0" as the origin and each time point represented by a non-negative integer, as shown in Figure 3.3. The granularity
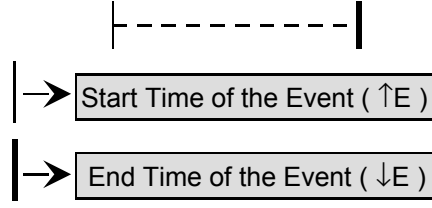
Figure 3.4. Event Notations.

of the domain is assumed to be specific to the domain of interest. In security domain, events can be generated based on user file access. 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 occurs over a time interval $[t_1, t_2]$ and is denoted by $O(E, [t_1, t_2])$, where E is the event, $t_1$ is the start time of the event denoted by $\uparrow E$, $t_2$ is the end time of the event denoted by $E\downarrow$ and "O" represents the interval-based Snoop semantics. Start and end times of an event are shown in Figure 3.4 and are formally defined as:

Start time: $O(\uparrow E, t) \triangleq \exists t \leq t'(O(E, [t, t']))$

End time: $O(E \downarrow, t') \triangleq \exists t \leq t'(O(E, [t, t']))$

### 3.2.1  Primitive Events

Primitive events are finite set of events that are predefined in the (application) domain of interest. A primitive event is detected atomically at a point on the time line shown in Figure 3.3. Primitive events are distinguished as domain specific, temporal and explicit events (for more details refer to [70]). For example, method execution by

an object in an object-oriented system 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).

**Definition 1.** *A primitive event E occurs over an interval [t, t′], where t is the start time and t′ is the end time of an event, and is expressed as O (E, [t, t′]). For event E, t is the same as t′, as a primitive event is detected atomically at a point on the time line. A primitive event is defined as:*

$$O(E, [t, t']) \triangleq \exists t = t'(O(E, [t, t']))$$

### 3.2.2 Event Expressions

For many real-world applications, supporting only primitive events is not adequate as 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. Complex patterns of events 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 [41, 42] is an event specification language that is more comprehensive and subsumes other languages in terms of event operators. It is used to specify combinations of events using operators, such as `AND`, `OR`, `NOT`, `Sequence`, `Plus`, `Periodic`, `Aperiodic`, `Cumulative Periodic` and `Cumulative Aperiodic`. These operators have been chosen based on various classes of applications, such as network monitoring, process control, trigger needs in databases, and so forth [70].

Events, event expressions, conditions, and actions play different semantic roles in the ECA paradigm. Conditions are very different from composite events as conditions

check on the resulting state after events have occurred and detected, whereas events are composed to detect a complex occurrence. If complex events were to be detected as part of a condition then a condition has to implement the semantics of composition of each operator; in other words, the entire local event detector needs to be implemented as part of a condition. In contrast, event detectors receive a large number of events and only combine those that correspond to the operator semantics and event consumption modes. All the details of buffering events and handling them are transparent to the user. This is not dissimilar from providing abstract data types and their composability rather than each programmer building their own version of a data type. On the other hand, conditions check whether to take an action or not based on the parameters passed from the occurrence of events.

### 3.2.3 Composite Events

Composite events are constructed using primitive events and event operators over composite events. A composite event consists of a number of primitive events and composite event operators: The set of 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 [82, 83].

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, a *detector* of a composite event is the constituent event whose occurrence detects the composite event, and a *terminator* of a composite event is the

constituent event that is responsible for terminating the composite event. Note that initiators, detectors, and terminators need not be distinct nor unique for a given event expression.

Snoop operators can be divided into two groups - *binary* operators and *ternary* operators based on the number of events associated with an operator. Binary snoop operators are of the form "$(E1\ op\ E2, [t_1, t_2])$" and ternary snoop operators are of the form "$op\ (E1, E2, E3), [t_1, t_2]$". "$E1\ op\ E2$" and "$op\ (E1, E2, E3)$" are both composite events that occur over the time interval $t_1$ and $t_2$, $t_1$ being the start time of the initiator as well as the composite event, and $t_2$ being the end time of the detector/terminator as well as the composite event.

### 3.2.4 Event Operators

Below, we define Snoop operators intuitively in the unrestricted (or general) context. This means events, once they occur, cannot be discarded at all. For a "$\gg$" (Snoop sequence operator) event, all event occurrences that occur after a particular event will get paired with that event as per the unrestricted context semantics. The formal definitions for these snoop operators in unrestricted context can be found in [82, 83]. Below, "$O$" represents the interval-based Snoop semantics.

**AND ($\Delta$):** $O(E1\ \Delta\ E2, [t_1, t_2])$ formally represents the conjunction of two events. The Conjunction of two events E1 and E2, denoted by "$E1\ \Delta\ E2$", occurs when both E1 and E2 occur, irrespective of their order of occurrence. Either E1 or E2 can act as initiator or terminator.

**SEQUENCE ($\gg$):** The sequential composition of two events is represented as $O(E1\ \gg E2, [t_1, t_2])$. The Sequence of two events E1 and E2, denoted by $E1\ \gg E2$, occurs when E2 occurs, provided E1 has already occurred. This implies that the end time of occurrence

of E1 is guaranteed to be less than the start time of occurrence of E2. E1 is the initiator and E2 is the terminator of the sequence event.

**OR ($\nabla$):** $O(E1 \nabla E2, [t_1, t_2])$ formally represents the disjunction of two events. The Disjunction of two events, denoted by $E1 \nabla E2$, occurs when E1 occurs or E2 occurs. E1 and E2 act as both initiators and terminators.

**NOT ($\neg$):** Determines the time of non-occurrence of an event in the context determined by two other events, and is represented as $O(\neg(E3)[E1 \gg E2], [t_1, t_2])$. It detects the nonoccurrence of event E3 in the closed interval formed by the end time of E1 (i.e., $t_1$ or E1 ) and start time of E2 (i.e., $t_2$ or E2). As it is a non-occurrence of E3 in a predefined interval, the occurrence time of a NOT event is just that interval (i.e., $[t_1, t_2]$).

**Aperiodic (A, A*):** An Aperiodic event E is represented by $O(A(E1, E2, E3), [t_1, t_2])$, where E1, E2 and E3 are event expressions, E1 is the initiator, E2 is the detector and E3 is the terminator. E is signaled each time E2 occurs during the interval defined by the occurrences of E1 and E3 and the occurrence time of E is the occurrence time of E2. In other words, there must be no occurrence of E3 wholly within the interval between the occurrences of E1 and E2. This operator is useful when the occurrence of an event has to be monitored in the context determined by two other events. For example, an application that requires any change in the temperature of an object to be signaled from the beginning of the experiment to the end of that experiment can be modeled by this operator.

On the other hand, there are situations when a given event is signaled more than once during a given interval (e.g., transaction) and, rather than firing the rule every time the event is signaled, we want the rule to be fired only once. The Cumulative Aperiodic operator is provided to meet this requirement and is expressed as $O(A^*(E1, E2, E3), [t_{s2}, t_{e2}])$. This event is similar to the non-cumulative version except that it accumulates the occurrences of E2 within the interval formed by E1 and E3 and is

detected only once when E3 occurs. Thus, occurrence time of an $A^*$ event is the interval formed by the start time of the first occurrence of E2 and the end time of the last occurrence of E2 within the interval formed by E1 and E3.

**Periodic (P, P\*):** The Periodic event is defined as an event $P$ that repeats itself within a constant and finite amount of time after the occurrence of the initiator event so long as the terminator does not occur. The Periodic event is expressed as $O(P(E1, E2, E3), [t_2, t_2])$, where E1 is the initiator, E2 is the detector, and E3 is the terminator. While E1 and E3 can be any type of events, E2 should be a time string (temporal event). The periodic event occurs after time period $[t]$ specified by E2 within the time interval started by E1 and ended by E3. The Periodic event expresses that a certain period of time specified by E2 has elapsed a whole number of times since the initiator has occurred. Hence, the occurrence time of the periodic event is the same as the detection time event E2 (i.e., $[t_2, t_2]$). The Periodic event has many applications, for example, when a bank database is required to print the summary of all transactions of each customer at the end of the month.

Similar to the Aperiodic event, the Periodic event has a cumulative variant $P^*$ expressed as $O(P^* (E1, E2, E3), [t_{s2}, t_{e2}])$. $P^*$ occurs only once when the event E3 occurs and it accumulates the event E2 occurrences at the end of each period and is made available when $P^*$ occurs. Occurrence time of a $P^*$ event is the interval formed by the first occurrence of E2 and the last occurrence of E2 with the interval formed by E1 and E3.

**Plus (PLUS):** It is a relative temporal event that is started by the occurrence of an event and is signaled after the specified time period. The PLUS event is represented as $O(PLUS(E1, E2), [t_2, t_2])$, where E1 is an event expression and E2 is a time string $[t]$. Event E1 is the initiator, and E2 is the terminator. Similar to the Periodic event, the
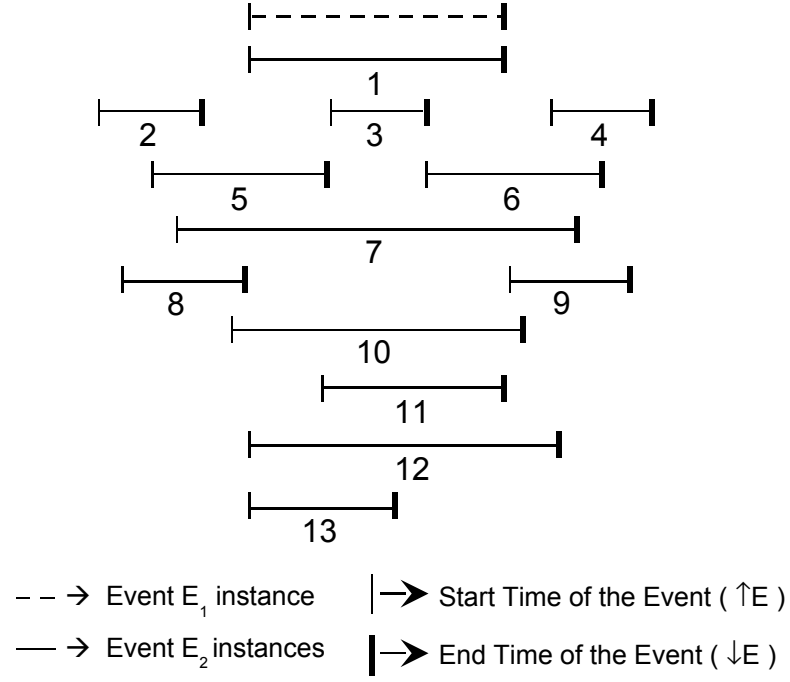
Figure 3.5. Overlapping Event Combinations.

occurrence time of the PLUS event is the time of occurrence of event E2 (i.e., $[t_2, t_2]$). The PLUS event occurs only once after time $[t]$, after the event E1 occurs.

### 3.2.5   Event Combinations

Another aspect of interval-based event occurrences is that they can be either over-lapping or disjoint.

**Overlapping Event Combinations:** When events are allowed to overlap, all combinations in which two events can occur [145, 146] are shown in Figure 3.5. In this work, event operators are formally defined using overlapping event combinations.

**Disjoint Event Combinations:** 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 when only one of the
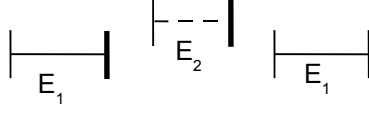
Figure 3.6. Disjoint Event Combinations.

overlapping events is of interest. The possible disjoint event combinations are shown in Figure 3.6.

### 3.2.6  Event Consumption Modes

Events are detected in the unrestricted (or general) context. In the absence of any mechanism for restricting event usage (or consumption), events need to be detected and 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 (with unrestricted context) can be large and not all event occurrences may be meaningful for an application. With diverse 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 expressions to their domain needs. In addition, the detection of these events has substantial computation and space overhead, which may become a problem for situation monitoring applications.

In order to provide more meaningful event occurrences to match application needs, Snoop introduced several parameter contexts (or 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. It is also the case that each context defined generates a subset of events generated by the unrestricted context. The ideal situation is to allow users to roll their own context as needed.

**Recent Context:** 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. We have formalized Snoop operators over recent context in our previous work [83].

**Continuous Context (Sliding Window Events):** In applications where event detection along a moving time window is needed, continuous context can be used. This context is especially useful for tracking trends of interest on a sliding time point governed by the initiator event. For example, computing change of more than 20% in Dow Jones average in any 2-hour period requires each change to initiate a new occurrence of an event, and can be expressed as an Aperiodic event as A (E1, E, PLUS (E1, 2 hours)), where the event E1 starts the 2 hour period, the event PLUS (E1, 2 hours) detects and terminates the 2 hour period, E can be any arbitrary event, and the change in average can be computed in the condition part once the event is detected. 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. In other words, each initiator starts a new window, and the events are detected until (or when) a terminator occurs. For binary Snoop operators, all the constituent events (initiator, detector and/or terminator) are deleted once the event is detected. For ternary Snoop operators, detectors are different from terminators. Detectors detect the event occurrence (e.g., Aperiodic) and are deleted once detected. A terminator terminates the event (e.g., Aperiodic*) and deletes the corresponding initiator and terminator pair along with the constituent events that cannot be used in future events. Future events are the events that are initiated by the initiators that are not paired with this terminator.

**Cumulative Context (Semantic Window Events):** Applications use this context when multiple occurrences of constituent events need to be grouped (or accumulated) and used in a meaningful way when the event occurs (e.g., banking application). In this context, all occurrences of an event type are accumulated as instances of that event until the event is terminated (i.e., forming a semantic window between the earliest initiator that was not terminated and a terminator). An event occurrence does not participate in two distinct occurrences of the same composite event. In both the binary and ternary operators, detector and terminator are the same, and once detected and terminated, all constituent event occurrences that were part of the detection are deleted. Other events that can act as a constituent event for some future event are preserved.

We will use the start and end of an event defined earlier for formally defining the event operators. In order to express this more concisely, the predicate Oin (occurrence in an interval) [82], is used and is formally defined as:

$$Oin(E[t_1, t_2]) \triangleq \exists t_1', t_2'(t_1 \le t_1' \le t_2' \le t_2 \land O(E, [t_1', t_2']))$$

### 3.2.7 Event Histories

The Snoop operators explained in Section 3.2.4 are based on the event occurrences over a time line. In Sections 3.3 and 3.4, using the notion of event histories, we formalize these definitions to take the event consumption modes into account. An event history maintains a history of event occurrences up to a given point in time. Suppose e1 is an event instance of type E1, then E1 [H] represents the event history that stores all the instances of the event E1 (namely $e_1{}^i$). Events in an event history are ordered by their end time.

- **Ei [H]** - Event history for event Ei.
- $[t_{si}, t_{ei}]$ - Indicates the Start time ($t_{si}$) and End time ($t_{ei}$) of an event instance $e_i{}^j$ of the event type Ei.
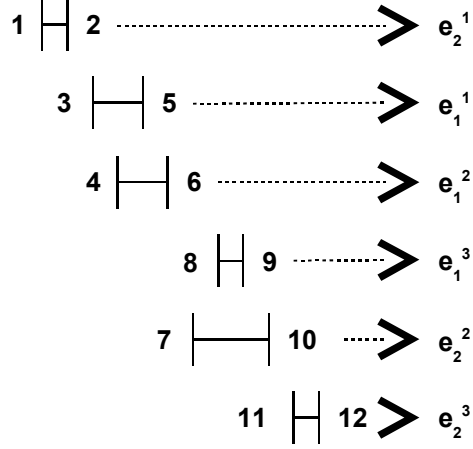
Figure 3.7. Examples of the Sequence Operator.

## 3.3 Interval-Based Event Operator Formalization in Continuous Context

In this section, we define formally in continuous context some of the operators defined in the unrestricted context [82]. Event operators are already defined intuitively in unrestricted context in Section 3.2.4.

In the following subsections, we will use the following way for defining the SnoopIB operators: 1) An example showing the events that are detected in unrestricted context using interval-based semantics, 2) Formal definition for the operators in continuous context, and 3) An example showing the events that are detected in continuous context.

**SEQUENCE($\gg$):**

The event histories shown below are based on the event occurrences shown in Figure 3.7 and are used to explain the detection of sequence event (E1 $\gg$ E2) in both recent and unrestricted contexts.

E1 [H] = $\{e_1{}^1$ [3, 5], $e_1{}^2$ [4, 6], $e_1{}^3$ [8, 9]$\}$

E2 [H] = $\{e_2{}^1$ [1, 2], $e_2{}^2$ [7, 10], $e_2{}^3$ [11, 12]$\}$

**Events detected in Unrestricted Context:** The "≫" event defined in Section 3.2.4 generates the following pairs of events from the above event histories in the unrestricted context.

$\{(e_1{}^1, e_2{}^2)\ [3, 10], (e_1{}^2, e_2{}^2)\ [4, 10], (e_1{}^1, e_2{}^3)\ [3, 12], (e_1{}^2, e_2{}^3)\ [4, 12],$

$(e_1{}^3, e_2{}^3)\ [8, 12]\}$

**Formal Definition in Continuous Context:**

$$O(E1 \gg E2, [t_{s1}, t_{e2}]) \triangleq \forall E2 \in E2[H] \wedge \forall E1 \in E1[H]$$

$$\{O(E2, [t_{s2}, t_{e2}]) \wedge (\nexists(O(E2'[t_s, t_e]))|(t_e < t_{e2}) \wedge E2' \in E2[H])$$

$$\wedge\ (O(E1, [t_{s1}, t_{e1}]) \wedge (t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2}))\}$$

$$\vee\ \forall E2 \in E2[H] \wedge \forall E1 \in E1[H]$$

$$\{O(E2, [t_{s2}, t_{e2}]) \wedge ((\exists(O(E2'[t_s, t_e]))|(t_e < t_{e2}) \wedge E2' \in E2[H])$$

$$\wedge\ (\nexists(O(E2''[t'_s, t'_e]))|(t'_s > t_{e1}) \wedge (t'_e < t_{e2}) \wedge E2'' \in E2[H]))$$

$$\wedge\ (O(E1, [t_{s1}, t_{e1}]) \wedge (t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2}) \wedge (t_{s1} > t_e))\}$$

In this context, a detector or terminator can detect or terminate more than one initiator and produce as many events as the number of initiators. Thus, in order for a set of initiators to form the sequential composition with a terminator (i.e., E2 ∈ E2 [H]), there should be no occurrence of any other instance of event E2 (i.e., E2′ ∈ E2 [H]) in the interval formed by the set of initiators and terminator. This is handled in two cases as shown above; the first case deals with the first terminator instance and the second case deals with other terminator instances. In the first case, $(\nexists(O(E2'[t_s, t_e])))$ defines the first instance of the terminator, and $(t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2})$ composes the set of initiators sequentially with this terminator. In the second case, two instances of the terminator are considered (i.e., E2′ and E2), where E2′ happens before E2. The set of initiators that occur within the time interval formed by these terminators is composed sequentially with
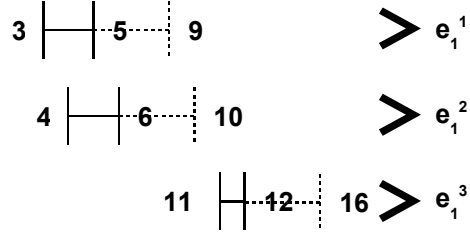
Figure 3.8. Examples of the PLUS Operator.

E2, as it will give the set of initiators terminated by E2. The condition that E2′ happens before E2 is stipulated using $(\nexists(O(E2''[t'_s, t'_e]))|(t'_s > t_{e1}) \wedge (t'_e < t_{e2}) \wedge E2'' \in E2[H])$.

**Events detected in Continuous Context:** When event $e_2^1$ occurs, there is no event in the event history of E1 that satisfies the "$\gg$" operator condition. When the event $e_2^2$ occurs, it detects the continuous event with the event pairs $(e_1^1, e_2^2)$ [3, 10] and $(e_1^2, e_2^2)$ [4, 10]. On the other hand, it does not pair with the event $e_1^3$ as the sequence condition fails. According to the continuous context definition, the events $e_1^1$, $e_1^2$, and $e_1^3$ have already participated in event detection and cannot act as constituent events for event $e_2^3$. When the event $e_2^3$ occurs it does not detect any event as there are no initiators. The event pairs generated by the "$\gg$" operator in continuous context are: $\{(e_1^1, e_2^2)$ [3, 10] and $(e_1^2, e_2^2)$ [4, 10]$\}$

**Plus (PLUS):**

The PLUS event occurs only once after the time interval specified by E2 after the occurrence of event E1 and is denoted by (PLUS (E1, E2), $[t_2, t_2]$), where E1 is an event expression and E2 is a time string. By definition, start and end times of the PLUS event are the same. For example, the event (PLUS (E1, 4)) is detected 4 units after the occurrence of event E1, and is used to explain the event detection in the unrestricted context. Event histories shown below are based on the event occurrences shown in Figure 3.8 and are used to explain the detection of PLUS events in the unrestricted context.
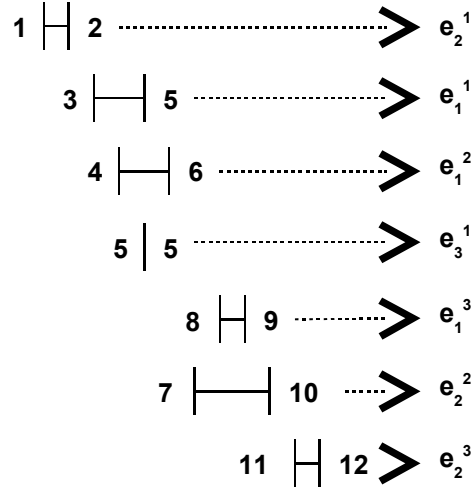
Figure 3.9. Examples of the NOT Operator.

E1 [H] = $\{e_1{}^1$ [3, 5], $e_1{}^2$ [4, 6], $e_1{}^3$ [11, 12]$\}$

**Events detected in Unrestricted Context:** For the events in the event history, the "PLUS" event defined in Section 3.2.4 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]$\}$

The PLUS operator's unrestricted context definition *holds* for the continuous context, since the PLUS operator is detected only once after the occurrence of the event E1.

**NOT ($\neg$):**

The event histories shown below are based on event occurrences shown in Figure 3.9 and are used to explain the detection of NOT event in the unrestricted context. The event histories for events E1 and E2 are the same as in the "$\gg$" operator:

E3 [H] = $\{e_3{}^1$ [5, 5]$\}$

**Events detected in Unrestricted Context:** The "¬" operator generates the following events in unrestricted context: $\{(e_1{}^2, e_2{}^2)\ [4, 10], (e_1{}^2, e_2{}^3)\ [4, 12], (e_1{}^3, e_2{}^3)\ [8, 12]\}$

**Formal Definition in Continuous Context:**

$$O(\neg(E3)[E1, E2], [t_{s1}, t_{e2}]) \triangleq \forall E2 \in E2[H] \wedge \forall E1 \in E1[H] \wedge \forall E3 \in E3[H]$$

$$\{O(E2, [t_{s2}, t_{e2}]) \wedge (\nexists(O(E2'[t_s, t_e]))|(t_e < t_{e2})$$

$$\wedge\ E2' \in E2[H]) \wedge \{(O(E1, [t_{s1}, t_{e1}]) \wedge (t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2})$$

$$\wedge\ \neg Oin(E3, [t_{e1}, t_{s2}]))\}\}$$

$$\vee\ \forall E2 \in E2[H] \wedge \forall E1 \in E1[H] \wedge \forall E3 \in E3[H]$$

$$\{O(E2, [t_{s2}, t_{e2}]) \wedge ((\exists(O(E2'[t_s, t_e]))|(t_e < t_{e2})$$

$$\wedge\ E2' \in E2[H])$$

$$\wedge\ (\nexists(O(E2''[t_s', t_e']))|(t_s' > t_e) \wedge (t_e' < t_{e2}) \wedge E2'' \in E2[H]))$$

$$\wedge\ \{(O(E1, [t_{s1}, t_{e1}]) \wedge (ts1 \leq t_{e1} < t_{s2} \leq t_{e2}) \wedge (t_{s1} > t_e)$$

$$\wedge\ \neg Oin(E3, [t_{e1}, t_{s2}]))\}\}$$

The "¬" operator formal definition has two cases and is similar to the "≫" operator formal definition. The "¬" operator determines the time of non-occurrence of an event (i.e., E3) in the context determined by two other events (i.e., E1, E2). It can be expressed as the sequential composition of (i.e., "≫") of events E1 and E2, where there is no occurrence of event E3 in the interval formed by these events. The sequential composition of two events in the continuous context is explained earlier. Non-occurrence of event E3 in the interval formed by E1 and E2 is specified using the condition $\neg Oin(E3, [t_{e1}, t_{s2}])$.

**Events detected in Continuous Context:** In the above example, when the event $e_2{}^1$ occurs, there is no event in the event history of E1 that satisfies the "¬" operator condition. When event $e_2{}^2$ occurs, only the event $e_1{}^2$ can combine with $e_2{}^2$, as
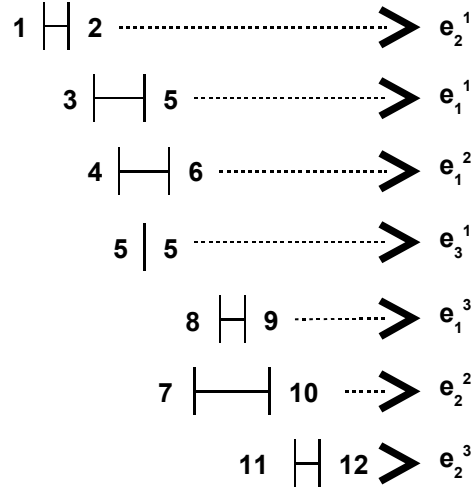
Figure 3.10. Examples of the A and A* Operators.

the event $e_1{}^1$ had already been terminated by the occurrence of $e_3{}^1$, and the event $e_1{}^3$ does not fulfill the sequence condition. The event $e_2{}^2$ detects the "¬" event with the event pairs $(e_1{}^2, e_2{}^2)$ [4, 10]. When the event $e_2{}^3$ occurs, it does not detect any event, as there are no initiators. The event pairs generated in continuous context are: $\{(e_1{}^2, e_2{}^2)$ [4, 10]\}

**OR ($\nabla$):**

The semantics of the "$\nabla$" (OR) operator does not change with the context, as each occurrence is detected individually.

**Aperiodic (A):**

The event histories shown below are based on event occurrences shown in Figure 3.10 and are used to explain the detection of "A" event in both recent and unrestricted context:

E1 [H] = $\{e_1{}^1$ [3, 5], $e_1{}^2$ [4, 6]\}

E2 [H] = $\{e_2{}^1$ [1, 2], $e_2{}^2$ [8, 9], $e_2{}^3$ [7, 10], e24 [11, 12]\}

E3 [H] = $\{e_3{}^1$ [11, 11]\}

**Events detected in Unrestricted Context:** $(A(E1, E2, E3), [t_{s1}, t_{e1}])$ detects the following pairs of events using event histories:

$\{(e_1{}^1, e_2{}^2)\ [8, 9], (e_1{}^2, e_2{}^2)\ [8, 9], (e_1{}^1, e_2{}^3)\ [7, 10], (e_1{}^2, e_2{}^3)\ [7, 10]\}$

In the case of the Aperiodic operator, formal definition given for unrestricted context holds for continuous context. The Aperiodic operator is a ternary operator, where events those occurred before the terminator are deleted, as they cannot take place in future event detection. Thus, events detected by both unrestricted and continuous contexts are the same.

**Events detected in Continuous Context:** By definition, the occurrence time of "A" is the occurrence time for E2. With the event histories shown in Figure 3.10, aperiodic operator detection in continuous context is explained. When the event $e_2{}^2$ occurs, initiators that can be paired with this event are $e_1{}^1$ and $e_1{}^2$. In this case, the event $e_2{}^2$ is just a detector, so that the detection initiators $e_1{}^1$ and $e_1{}^2$ can take part in future event detection till the terminator occurs. So the event $e_2{}^3$ can be paired with the same initiators. But when the event $e_2{}^4$ occurs, there are no initiators that are available for detection, since terminator $e_3{}^1$ has terminated all the initiators. Thus the events generated by this operator in continuous context are:

$\{(e_1{}^1, e_2{}^2)\ [8, 9], (e_1{}^2, e_2{}^2)\ [8, 9], (e_1{}^1, e_2{}^3)\ [7, 10], (e_1{}^2, e_2{}^3)\ [7, 10]\}$

## 3.4 Interval-Based Event Operator Formalization in Cumulative Context

**SEQUENCE ($\gg$):**

**Events detected in Cumulative Context:** In this context, a detector or terminator produces only one event. The event histories are used for the detection of the "$\gg$" operator defined above. The event histories corresponding to the event occurrences shown in Figure 3.7 are given below, where E1 [H] is the initiator event history and E2 [H] is the terminator event history.

E1 [H] = $\{e_1{}^1$ [3, 5], $e_1{}^2$ [4, 6], $e_1{}^3$ [8, 9]$\}$

E2 [H] = $\{e_2{}^1$ [1, 2], $e_2{}^2$ [7, 10], $e_2{}^3$ [11, 12]$\}$

When the terminator event $e_2{}^1$ occurs, there is no initiator event in E1 [H] that satisfies the "$\gg$" operator condition. The event $e_1{}^1$ occurrence initiates a sequence event. The event $e_1{}^2$ occurrence is accumulated. When the event $e_2{}^2$ occurs, E1 [H] has events $\{e_1{}^1$ [3, 5], $e_1{}^2$ [4, 6], $e_1{}^3$ [8, 9]$\}$. Thus, $e_2{}^2$ detects the event initiated by the event $e_1{}^1$ generating the following event $(e_1{}^1, e_1{}^2, e_2{}^2)$ [3, 10], since it satisfies the sequence condition $(t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2})$ (i.e., $(3 \leq 5 < 7 \leq 10)$ for pair $(e_1{}^1, e_2{}^2)$). As shown, all the events in between the pair $(e_1{}^1, e_2{}^2)$, in this case $e_1{}^2$, are accumulated. Even though $e_1{}^3$ occurred before $e_2{}^2$, it is not detected since it does not satisfy the condition $(9 < 7)$. According to the cumulative context definition, the events $e_1{}^1$, $e_1{}^2$ and $e_2{}^2$ are deleted as they have already participated in event detection and cannot act as constituent events for future detections. In addition, the event $e_1{}^3$ is also deleted as it has occurred before the start time of $e_2{}^2$ and does not satisfy the sequence condition. As there are no events after end time of $e_2{}^2$, the event $e_2{}^3$ does not detect any event. The event pairs detected by sequence operator in continuous context are:

$(e_1{}^1, e_1{}^2, e_2{}^2)$ [3, 10]

**Formal Definition in Cumulative Context:**

$O(E1 \gg E2, [t_{s1}, t_{e2}]) \triangleq$

$\qquad \forall E2 \in E2[H]$

$\qquad \{O(E2, [t_{s2}, te2]) \wedge (\nexists E2'[t_s, t_e] | (t_e < te2) \wedge E2' \in E2[H])$

$\qquad \wedge \{\forall E1 \in E1[H](O(E1, [t_{s1}, t_{e1}]) \wedge (t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2}))\}$

$\qquad \}$

$\qquad \vee$

$\qquad \forall E2 \in E2[H]$

$\qquad \{O(E2, [t_{s2}, te2]) \wedge ((\exists E2'[t_s, t_e] | (t_e < te2) \wedge E2' \in E2[H])$

$\qquad \wedge (\nexists E2''[t'_s, t'_e] | (t'_e > t_e) \wedge (t'_e < te2) \wedge E2'' \in E2[H]))$

$\qquad \{\forall E1 \in E1[H](O(E1, [t_{s1}, t_{e1}]) \wedge (t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2})$

$\qquad \wedge (t_{s1} > t_e) \wedge (\nexists E1'[t'_{s1}, t'_{e1}] | (t'_{s1} > t_e) \wedge (t'_{s1} < t_{s1})$

$\qquad \wedge E1' \in E1[H]))$

$\qquad \}$

$\qquad \}$

Two events e1 ∈ E1 [H] and e2 ∈ E2 [H] are said to occur in sequence in the cumulative context only when there is no occurrence of e2′ ∈ E2 [H] before the occurrence of e2 and all the other occurrences of e1′ ∈ E1 [H] that occur in between the pair e1 and e2 are accumulated. There are two cases to formally define the operator (refer the formal definition above). The first case applies when there is no other terminator available in the terminator history (i.e., it is the first occurrence of the terminator). In other words, there should be no occurrence of other terminators before this terminator and this terminator should be in sequence with all initiators till that point. In this case, all the event occurrences of the initiator are accumulated, and the cumulative event is
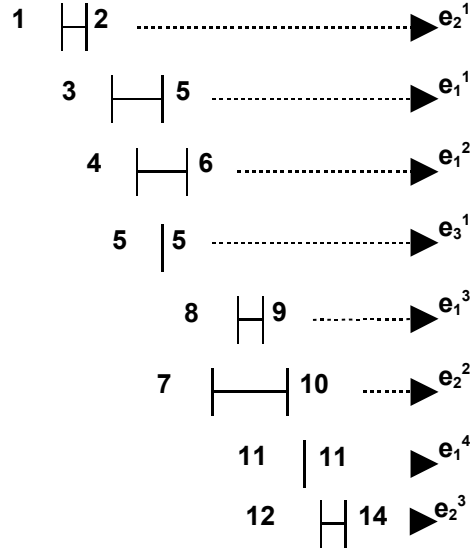
Figure 3.11. Examples of the NOT Operator (Cumulative Context).

detected. The second case applies when there is more than one terminator present in the history. For this case, there should be no occurrence of other terminators in between the start of the initiator and the end of the terminator or a terminator can occur only if its end time is less than the start time of the initiator. In other words, an initiator starts an event occurrence and a terminator terminates and detects the "$\gg$" event, with the events in between taken as constituent events, and there should be no other instance of the terminator.

**OR ($\nabla$):**

The semantics of "$\nabla$" does not change with cumulative context as each occurrence is detected individually.

**Plus (PLUS):**

The PLUS operator's unrestricted context definition holds for the cumulative context, since it is detected only once after the occurrence of the event E1 and there is only one terminator for an initiator.

**NOT ($\neg$):**

**Events detected in Cumulative Context:** The "$\neg$" Operator can be expressed as the sequence of E1 and E2 where there is no occurrence of the event E3 in the interval formed by these events. The event histories corresponding to the event occurrences shown in Figure 3.11 are given below, where E1 [H] is the event e1 history, E2 [H] is the event e2 history, and E3 [H] is the event e3 history.

E1 [H] = $\{e_1{}^1$ [3, 5], $e_1{}^2$ [4, 6], $e_1{}^3$ [8, 9], $e_1{}^4$ [11, 11]$\}$

E2 [H] = $\{e_2{}^1$ [1, 2], $e_2{}^2$ [7, 10], $e_2{}^3$ [12, 14]$\}$

E3 [H] = $\{e_3{}^1$ [5, 5]$\}$

When the terminator event $e_2{}^1$ occurs, there is no initiator event in E1 [H] that can pair with $e_2{}^1$. The event $e_1{}^1$ occurrence initiates a sequence event. The event $e_1{}^2$ occurrence is accumulated. When the event $e_2{}^2$ occurs, E1 [H] has events $\{e_1{}^1$ [3, 5], $e_1{}^2$ [4, 6], $e_1{}^3$ [8, 9]$\}$. But the event $e_1{}^1$ [3, 5] cannot combine with the event $e_2{}^2$ [7, 10] since there is an occurrence of $e_3{}^1$ [5, 5] in between $e_1{}^1$ and $e_2{}^2$ (i.e., $5 \leq 5 \leq 7$), thus a NOT event is not detected. The event $e_1{}^4$ [11, 11] initiates the next NOT event. When the event $e_2{}^3$ [12, 14] occurs, it pairs with the event $e_1{}^4$ detecting $(e_1{}^4, e_2{}^3)$ [11, 14], as there is no occurrence of an event e3 in the interval [11, 12]. The event pair generated by the NOT operator in cumulative context is:

$\{(e_1{}^4, e_2{}^3)$ [11, 14]$\}$

**Formal Definition in Cumulative Context:**

$O(\neg(E3)[E1, E2], [t_{s1}, t_{e2}]) \triangleq$

$\quad\quad\quad \forall E2 \in E2[H]$

$\quad\quad\quad \{O(E2, [t_{s2}, te2]) \wedge (\nexists E2'[t_s, t_e] | (t_e < te2) \wedge E2' \in E2[H])$

$\quad\quad\quad \{\forall E1 \in E1[H] \wedge E3 \in E3[H]$

$\quad\quad\quad (O(E1, [t_{s1}, t_{e1}]) \wedge (t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2})$

$\quad\quad\quad \neg Oin(E3, [t_{e1}, t_{s2}]))\}$

$\quad\quad\quad \}$

$\quad\quad\quad \vee$

$\quad\quad\quad \forall E2 \in E2[H]$

$\quad\quad\quad \{O(E2, [t_{s2}, te2]) \wedge ((\exists E2'[t_s, t_e] | (t_e < te2) \wedge E2' \in E2[H])$

$\quad\quad\quad (\nexists E2''[t'_s, t'_e] | (t'_e > t_e) \wedge (t'_e < te2) \wedge E2'' \in E2[H]))$

$\quad\quad\quad \{\forall E1 \in E1[H] \wedge E3 \in E3[H]$

$\quad\quad\quad (O(E1, [t_{s1}, t_{e1}]) \wedge (t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2}) \wedge (t_{s1} > t_e)$

$\quad\quad\quad \wedge (\nexists E1'[t'_{s1}, t'_{e1}] | (t'_{s1} > t_e) \wedge (t'_{s1} < t_{s1})$

$\quad\quad\quad \wedge E1' \in E1[H])$

$\quad\quad\quad \wedge \neg Oin(E3, [t_{e1}, t_{s2}]))$

$\quad\quad\quad \}$

$\quad\quad\quad \}$

The formal definition above has two cases similar to the sequence operator formal defini-
tion. The non-occurrence of the event e3 ∈ E3 [H] between two events e1 ∈ E1 [H] and
e2 ∈ E2 [H] is said to occur in the cumulative context only when there is no occurrence
of e2′ ∈ E2 [H] before the occurrence of e2 and all the other occurrences of e1′ ∈ E1 [H]

that occur in between the pair e1 and e2 are accumulated. The first case applies when there is no other terminator available in the terminator history (i.e., first occurrence of the terminator). In other words, there should be no occurrence of other terminators before this terminator and this terminator should be in sequence with all initiators till that point and there should not be any occurrence of an event e3 in between the initiator and terminator as specified by the condition $(\neg Oin(E3, [t_{e1}, t_{s2}]))$. In this case, all the event occurrences of the initiator are accumulated and the cumulative event is detected. The second case applies when there is more than one terminator present in the history. For this case, there should be no occurrence of other terminators in between the start of the initiator and the end of the terminator or a terminator can occur only if its end time is less than the start time of the initiator. In addition, there should not be any occurrence of the event e3 in between the initiator of the composite event and the terminator as specified by the condition $(\neg Oin(E3, [t_{e1}, t_{s2}]))$.

## 3.5   Composite Event Detection

### 3.5.1   Composite Event Detection Using Event Graphs

In Sections 3.3 and 3.4, definitions using event histories were given for operators in continuous and cumulative contexts. They are appropriate for applications whose event histories are collected and processed for event occurrences. In many real-world monitoring applications, events are streaming in as they occur and composite events need to be detected on the fly (as they occur) and cause appropriate actions. In this section, we will explain how composite events are detected from the implementation perspective, when events occurs online, and show that the events detected in either way are the same.

Sentinel uses an event graph or event detection graph (EDG) for representing an event expression, in contrast to other approaches, such as Petri nets used by Samos
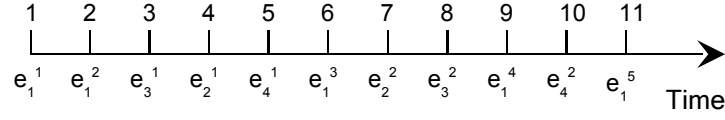
Figure 3.12. Event occurrences on the time line.

and extended finite state automata used by Compose. By combining event trees on common sub-expressions, an event graph is obtained. A data flow architecture is used for the propagation of primitive events to detect composite events. All leaf nodes in an event tree are primitive events and internal nodes represent composite events. By using event graphs, the need to detect 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 a substantial amount of storage space (for storing partial event occurrences and their parameters), thus leading to an efficient approach for detecting events. Event occurrences flow in a bottom-up fashion. When a primitive event occurs and is detected, it is sent to its leaf node, which forwards it to one or more parent nodes (as needed) for detecting one or more composite events.

As described in previous sections, the introduction of contexts makes event detection more meaningful and computationally less expensive for many applications. Below, we illustrate how composite events are detected using interval semantics proposed with an example that uses 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 an event needs to be detected in a particular context. The counter is also used to keep track of the number of composite events an event participates in. When this counter reaches zero, there is no need to detect that event any more in that context, as there are no events dependent on that event.
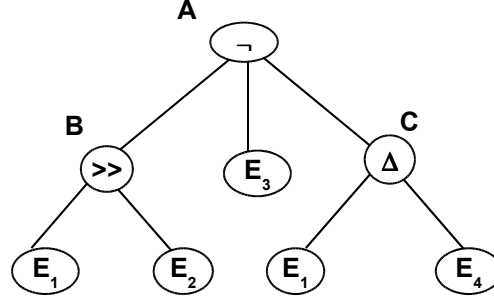
Figure 3.13. An Event Graph.

In Figure 3.12, the numbers 1, 2, 3, 4, 5, 11 represent time points on the time line at which a primitive events occur. If we take the primitive event $e_1{}^2$, it is said to occur in the time interval [2, 2], and the event $e_2{}^1$ 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] is the end time of the composite event. In Figure 3.14 we represent the events in terms of their occurrence intervals in square brackets (e.g., [2, 2] represents event $e_1{}^2$) for simplicity. The composition is shown using multiple events within a bracket (e.g., [[1, 4], [2, 7]] represents [[$e_1{}^1$, $e_2{}^1$], [$e_1{}^2$, $e_2{}^2$]]).

Figure 3.13 represents the composite event $(\neg E3)((E1 \gg E2), (E1\ \Delta\ E4))$. The Leaf nodes, E1, E2, E3, and E4, represent the primitive events and the nodes A, B and C represent the composite events. The NOT event is a composite event that contains AND, SEQUENCE as its constituent events. When any two events are paired in either node B or C, they are passed to node A where the "$\neg$" event is detected. We will present the detection of events in the continuous context. Figure 3.14 shows the snapshot of the event states in continuous context in the event graph at the time of the event $e_4{}^2$ occurrence.

In the continuous context (refer Figure 3.14), events $e_1{}^1$ and $e_1{}^2$ are paired with the event $e_2{}^1$, since one terminator may detect one or more initiators. When the event $e_2{}^2$
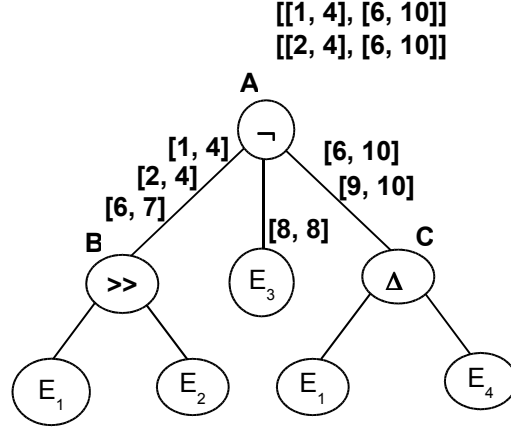
Figure 3.14. An Event Graph in Continuous Context.

occurs, it pairs with the event $e_1{}^3$. The occurrence of the event $e_4{}^2$ terminates the events $e_1{}^3$ and $e_1{}^4$ in the node C. When these events are sent to the node A, the events $(e_1{}^1, e_2{}^1, e_1{}^3, e_4{}^2)$ and $(e_1{}^2, e_2{}^1, e_1{}^3, e_4{}^2)$ are detected. But $(e_1{}^4, e_4{}^2)$ is not paired with the events $(e_1{}^3, e_2{}^2)$ since there is an occurrence of the middle event $e_3{}^2$ between these events. This initiator pair cannot start any more event detection, because of the occurrence of the middle event $e_3{}^2$ and all the events are removed. Figure 3.14 shows all the events that are available in the nodes A, B, C when event $e_4{}^2$ occurs. Thus, events detected in this context are:

$$\{(e_1{}^1, e_2{}^1, e_1{}^3, e_4{}^2) \ [[1, 4], [6, 10]], (e_1{}^2, e_2{}^1, e_1{}^3, e_4{}^2) \ [[2, 4], [6, 10]]\}$$

### 3.5.2  Algorithms and Implementation

The Snoop event operator algorithms in LED are based on the point-based seman-
tics. With current point-based event operator algorithms the processing cost involves
$\alpha$ comparisons of timestamp (i.e., comparing only on the end of the interval). With
interval-based algorithms it will involve at most $2 * \alpha$ comparisons as it has to compare
the start as well as the end timestamp. In other words, the asymptotic upper bound

(i.e., $O$) on the running time of the event operator algorithms are the same for both point-based and interval-based event operator algorithms.

In this section, we will provide algorithms that detect events according to the interval-based semantics. In the manner in which 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. The algorithms presented in the following subsections detect events according to interval semantics, although they do not see the complete history at any given point in time. How the start interval is handled is shown in the algorithm. The algorithms defined in the following subsections are implemented in Sentinel. The formal definitions and algorithms have been designed for all contexts. The notations used in the algorithms are shown below:

**Notations used in the algorithms:**

- ei (e.g., e1, e2) – A Primitive or Composite event instance or occurrence
- Ei (e.g., E1, E2) – An event list that maintains the partial history of the occurrences of event ei
- t_s – Start time of the event (Start Interval)
- t_e – Ending time of the event (End Interval)

### 3.5.2.1 The algorithm for the Sequence operator in Cumulative Context

/* ei can be recognized as coming from the left or right branch of the operator tree and parameter_list represents event properties */

```
PROCEDURE seq_cumulative (ei, parameter_list):
1  If ei is the left event
2     Append e1 to E1
```

```
3  If ei is the right event

4    If E1 is not empty

5      For every e1 in E1 and if (t_s (e2) > t_e (e1))

6        Append e1 to tempE1

7      If tempE1 is not empty

8        Pass <tempE1, e2> to parent with t_s

              (tempE1's EarliestStartTime) and t_e (e2)

9        Remove all event occurrences from tempE1

10     Remove all event occurrences from E1
```

*Explanation of the algorithm:*

1. If the event is from the left child (i.e., initiator of this operator) then continue

2. Accumulate event e1 occurrences in list E1

3. If the event is from the right child (i.e., terminator of this operator) then continue

4. When there is an initiator in the list, then continue

5. Check whether each event occurrence of e1 has preceded the e2 occurrence

6. if above step is true, then add the event e1 to a list tempE1

7. if there is at least one initiator then perform

8. Pass the accumulated event occurrences of e1 and e2 along with the time of occurrence. The start time of the composite event is the start time of the first occurrence of e1 (initiator) and the end time for the composite event is the end time of the terminator.

10. A Terminator has occurred and all the event occurrences in the left child have to be removed.

### 3.5.2.2   The algorithm for the Sequence operator in Continuous Context

/* ei can be recognized as coming from the left or right branch of the operator tree, and parameter_list represents event properties */

```
PROCEDURE seq_continuous (ei, parameter_list):
1  If ei is the left event
2    Append e1 to E1
3  If ei is the right event
4    If E1 is not empty
5      For every e1 in E1 and if (t_s (e2) > t_e (e1))
6        Pass <e1,e2> to parent with t_s (e1), t_e (e2)
7      Remove all event occurrences from E1
```

*Explanation of the algorithm:*

1. – 4. are same as the algorithm for cumulative context

5. For each event occurrence in E1, check whether it has preceded the e2 occurrence

6. if the above step is true, then pass the event occurrences of e1 and e2 along with the time of occurrence. The start time of the composite event is the start time of e1 and the end time for the composite event is the end time of the terminator.

7. A terminator has occurred and all the event occurrences in the left child have to be removed.

### 3.5.2.3   The algorithm for the NOT operator in Cumulative context

The NOT operator detects the nonoccurrence of the event E2 in the closed interval formed by E1 and E3.

```
PROCEDURE not_cumulative (ei, parameter_list)
1.a   If ei is the left event
1.b     Append e1 to E1
```

```
2      If ei is the middle event

3         If E1 is not empty and t_e (E1's EarliestEndTime) t_s (e2)

4         Append e2 to E2


5      If ei is the right event

6         If (E1 is not empty and (t_e (E1's EarliestEndTime) < t_s (e3))

7            If E2 is not empty

8.a            For every e1 in E1

8.b               If (t_e (e1) < t_s (e3))

8.c                  For all e2's in E2

8.d                     If (t_e (e2) > t_s (e3) or t_s (e2) < t_s (e1))

8.e                        Append e1 to tempE1

8.f                     Delete e1 from E1

9.a            If tempE1 is not empty

                  Pass <tempE1, e3> to the parent with t_s

9.b               (tempE1's EarliestStartTime) and t_e (e3)

10.a           For every e2 in E2

10.b              If (t_e (E1's EarliestEndTime) > t_s (e2))

10.c                 Delete e2 from E2

11          Else

11.a           For every e1 in E1

11.b              If (t_e (e1) < t_s (e3))

11.c                 Append e1 to tempE1

11.d                 Delete e1 from E1

                  Pass <tempE1, e3> to the parent with t_s
```

```
11.e            (tempE1's EarliestStartTime) and t_e (e3)
```

*Explanation of the algorithm:*

`1.` If the event is from the left child (i.e., initiator of this operator), then append it to the list E1

`2.` If the event is from the middle child (i.e., event E2 in our case), then continue

`3.` `and 4.` If the list E1 is not empty and the end time of the first occurrence of event e1 is less than or equal to the start time of the this event, then append this event to the list E2

`5.` If the event is from the right child (i.e., event E3 in our case), then continue

`6.` When there is an initiator in the list and the end time of the first occurrence of event e1 is less than to the start time of the this event, then continue

`7. - 10.` Check whether all the event occurrences of e1 have preceded the e3 occurrence and there is no occurrence of event e2 in between them. If there is any event pair, then detect the NOT event. Remove all the event e2 occurrences that satisfy the condition in 10.b

`11.` if there is no occurrence of event e2, detect a NOT event with all the event e1 occurrences and the event e3

## 3.6 Summary

Interval-based semantics is required by applications where the event detection and event occurrence need to be differentiated. In this chapter, we have formally defined the Snoop event operators using interval-based semantics in both continuous and cumulative contexts using event histories. These formal definitions include constraints that are based on the conditions over initiators, detectors, and terminators that should be satisfied for a particular context. Then, we have shown how online events are detected in interval-based semantics using event detection graphs. The interval-based semantics has been

implemented using partial-event histories or event graphs providing procedural semantics. Algorithms for all the operators have been developed for all the contexts, including the unrestricted context. We have also shown that the events that are detected by these contexts are subsets of the unrestricted context. We have published the formalization operators and algorithms using continuous and cumulative context in [91, 92, 93].

## CHAPTER 4

## ENFORCING ROLE-BASED ACCESS CONTROL MODELS

Dynamically monitoring the state changes of an underlying system, and detecting and reacting to changes without delay are crucial for the success of any access control *enforcement* mechanism. RBAC has been widely explored and extended and, in spite of its expressive *specification*, there has been not much work on a flexible and generalized framework for its enforcement. Most of the enforcement mechanisms are for the restricted combinations of NIST RBAC components and are customized to a specific system (e.g., OS, databases) and do not address how RBAC extensions can be supported. Currently, there is no framework that can be used to support various extensions in an uniform manner. Although RBAC has been shown to be better-suited for enterprises in diverse domains, the lack of a framework for its support has affected its usage pragmatically. With their inherent nature, ECA rules are prospective candidates to carry out change detection and to provide access control. As ECA rules provide active capability to the underlying system (i.e., for making the passive systems to active systems) they are also termed as *active rules*.

## 4.1 Introduction

With the ever growing impact of computing systems on our daily activities, security and privacy have a greater role to play. Role-Based Access Control [9, 10, 11], where object accesses are controlled by roles (or job functions) in an enterprise rather than a user or group, has shown to be a positive alternative to traditional access control mechanisms. RBAC does not provide a complete solution for all access control issues, but with its rich specification, it has shown to be cost effective [31] by reducing the

complexity in authorization management of data. ANSI RBAC standard [12] is defined in terms of four model components and their combinations; *core* RBAC, *hierarchical* RBAC, and *static* and *dynamic* separation of duty relations. Constraints play a major role in access control models [17] and RBAC is being extended with various constraints for supporting authorization management of data in diverse domains: Temporal [20, 23], context-aware [19, 73], privacy-aware [100], control flow dependency [37], etc. All these extensions, which are critical for access control, warrants for a flexible and generalized security mechanism for enforcing Role-Based models. Even though a lot of work has been carried out in the specification for making RBAC more expressive, there is still lots of work that need to be carried out in enforcing these specifications in a flexible way that is essential for making RBAC better-suited for enterprises in diverse domains.

Enterprises can formalize their access control (or security) policies using RBAC or its extensions that provide additional constraints. Enterprises in different domains have different access control requirements; for example, the health care domain requires extensive temporal and context-aware constraints (e.g., emergency room, intensive care). Current security mechanisms do not provide a generalized approach for enforcing RBAC and its extensions in a uniform manner. For example, some systems just support separation of duty (SoD) relations, but without role hierarchies. Similarly, they do not adapt to policy or role structure *changes* in enterprises, which is indispensable for making RBAC usable. For example, when an enterprise wants to change its working hours of a role, then the low level semantic descriptors[1] have to be modified. In current systems, it is a burden on the administrator to modify and maintain these low level semantic descriptions, such as authorization rules, manually.

---

[1]In this work, we refer to authorization rules, Java classes, and other mechanisms that are used to enforce the access control as low level semantic descriptors.

Taking timely actions based on the state changes of the underlying system over a period of time and alerting the administrator regarding the malicious activities will complement the access control system. For example, all the constraints that are satisfied by a user when activating a role should hold TRUE until the role is deactivated. When any one of the constraints becomes FALSE before deactivation, then that role should be deactivated. Prevention of malicious activities in the system plays a major role in providing security. Enterprises require the detection and prevention of malicious activities from causing damage without human intervention. Furthermore, prevention and detection will ameliorate the security of the underlying system so that enterprises can be more secure. For example, when access requests by unauthorized roles for some files are more than a certain number of times within a duration, an internal security alert is triggered and some critical authorization rules are disabled[2] and the administrators are alerted.

Existing systems (or models) [71, 72, 73, 74, 75, 81, 89, 90, 147] enforcing RBAC are custom-implemented, domain-specific and are confined to a particular form of constraints. All these systems neither enforce complete RBAC standard nor provide a generalized approach for enforcing it over diverse underlying systems. Furthermore, they do not provide an approach for enforcing RBAC extensions in a uniform manner. Thus, systems (or models) that need to enforce RBAC in a generalized manner should be able to provide uniform and transparent handling of the RBAC standard and its extensions, adapt to policy and role structure changes in an enterprise, and support high level specification of enterprise access control policies.

We provide a security mechanism that is flexible, generalized, and portable for enforcing Role-Based Models. In this work we will address the following; *i)* provide and analyze various approaches for enforcing Role-Based models, *ii)* present approaches

---

[2]Actions are predefined by the security administrators.

for generating events in any underlying system, *iii)* introduce event-based active authorization rules, *iv)* synthesize active authorization rules for access control enforcement, *v)* discuss how RBAC standard and its extensions (i.e., Role-Based Models) are enforced in a uniform and transparent manner, *vi)* show how active rules can be used to provide active security, and *vii)* show the creation of an initial set of rules using Sentinel+ based on an enterprise access control policy and show how the rules are regenerated when there is a change in the access control policies. Sentinel+ is an enhanced version of Sentinel [148, 43], an active object-oriented system.

## 4.2  Approaches for Enforcing Role-Based Models

For brevity, we group all the role-based authorization models containing NIST RBAC and its extensions, such as GTRBAC, GRBAC, PRBAC and so forth, as Role-Based Models or `RBMs`. Security mechanisms enforce or implement security rules and models [149]. These mechanisms have to be independent of the security model so that it is possible to check the correctness of security requirements, consistency of the security polices, and so forth. Furthermore, a generalized, flexible, and extensible security enforcement mechanism will allow enterprises to employ various security policies i.e., RBAC, GTRBAC, GRBAC, and so forth. On the other hand, intertwining and hard-wiring of policies with security mechanisms will limit the ability of an enterprise to adapt to dynamic changes in the security policies. In this section, approaches for supporting RBMs are discussed.

Many open source and commercial projects/products support some form of RBAC: Linux [105, 106, 107, 108], Microsoft Windows [104], Sun Solaris [81] and Relational databases [74] such as Oracle, Sybase ASE, Informix and IBM DB2. Given any underlying system, supporting RBMs would require analysis pertaining to the following dimensions.

1. How to support RBMs in systems that do not explicitly support RBMs.

2. How to support RBMs in systems that provide ways to support user customized functionalities.

3. How to support RBMs in systems that support some RBMs or have some hooks that can be exploited to support RBMs. How much can we leverage the existing support provided by the system.

In order to support RBMs to the above mentioned categories, the following factors should be analyzed carefully.

1. How much can we change the underlying system in order to support RBMs. In other words, what needs to be changed in the underlying system.

2. How many types of constraints can be supported with the security mechanism. In other words, can it be flexible to support constraints such as temporal, event-based, context-based, content-based, and so forth.

3. How should the security mechanism implement NIST RBAC. Should it be implemented as a component (i.e., Core, Hierarchical, Static SOD, Dynamic SOD) or should it be based on the role semantics.

4. Can the security mechanisms support single application, multi application/single system or multi-system. In other words, can it be flexible and generalized to support diverse underlying systems, such as operating systems, databases and so forth. Below we provide three approaches based on the above requirements.

### 4.2.1 The Wrapper-based Approach

Legacy systems that do not support any RBMs can do so using the wrapper-based approach. The basic assumption is that the underlying system does not support any kind of hooks and it is either extremely difficult or impossible, in most cases, to modify the source code of the underlying system. Typically, a wrapper is built for each of the existing (or native) application or application process that is available in the underlying

system. Access control capabilities and other functionalities are added to the wrappers, which in turn check for RB policies whenever an native application (or process) is invoked either by a user or by other processes. Once the required constraints are met, wrapped application will in turn invoke the native application (i.e., function call or process).

Figure 4.1 illustrates a typical wrapper-based approach. As shown, a native application or application process invocation will invoke a wrapped application. All the constraints that need to be checked are available in the wrapper. Enterprise access control policy information, such as roles, objects and so forth, are maintained separately and are used by the wrapped application for checking the required constraints. Once the wrapper has finished checking the constraints it can either invoke the native application if all the constraints are met, or can deny if any one of the constraint is not met. For example, whenever the editor `VI` is invoked, the function call is rerouted to `WVI` (i.e., Wrapped `VI`) which checks for RB policies using the database. After constraint checking is performed the native `VI` is invoked or the denial message is returned.



1 - Native Application or Application Process Call
2 - Wrapped Application or Application Process Call
       with Additional Conditions
3 - Native Application or Application Process Call
4 - Denial Message
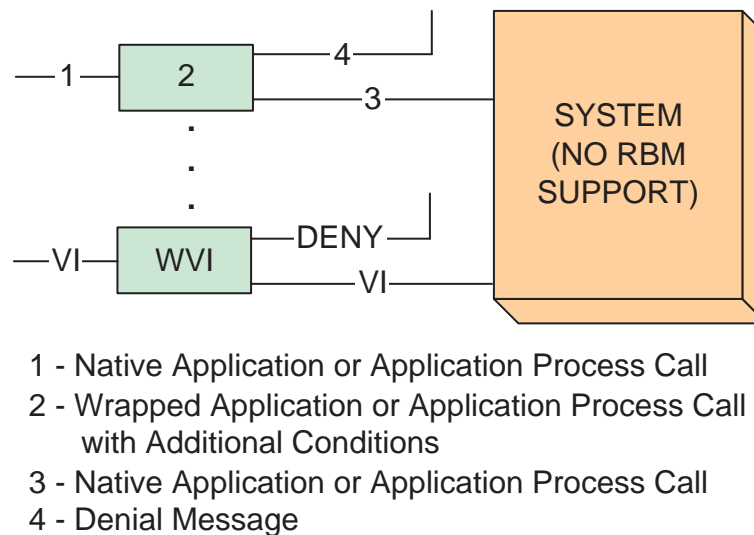
Figure 4.1. The Wrapper-Based Approach.

Wrappers can be either local or global, i.e., there can be a separate wrapper for a single native application or for a set of native applications, but they are system dependent. Enterprise access control policies that are based on RBMs have to be implemented using separate security mechanisms that can be utilized by the wrappers for checking the constraints. In other words, wrappers exploit the available security mechanisms, that enforce RBMs, for providing RBMs to the underlying legacy system. Thus, the extent of support of RBMs, i.e., roles, role hierarchies, separation of duty relations, temporal, location-based, event constraints, and so forth, depends on the security mechanisms that enforce it. In addition, none of the existing systems support event-based constraints.

The main advantage of this approach is that it does not require any modification to the underlying system. In addition, wrappers are transparent to the user or other application processes. On the other hand, this approach needs a good understanding of the underlying system and wrappers have to be developed for each legacy system and application separately. Furthermore, new wrappers have to be written for every new application, thus making the system management and/or maintenance more complex. The wrapper-based approach is preferred only when there is no other alternative to support RBM's in the underlying system as it is not flexible, extensible or portable.

### 4.2.2 The Integrated Approach

The integrated approach requires modification to the underlying system for incorporating RBMs. The basic assumption is that the underlying system provides some way of supporting user customized functionalities and the developers have sufficient understanding of the system to make changes at the lower level. For example, in Linux, `Kernel Modules` can be exploited to add and change functionalities to the kernel on the fly [150]. Similarly `Patches` can be applied to kernels to incorporate user-required functionalities. With `Kernel Modules`, the kernel need not be recompiled as the modules

can be loaded and unloaded on demand. Thus, RBMs can be supported using `Kernel Modules` which does not bloat the kernel and which is easier to debug and develop. On the other hand, there are other systems that do not support any mechanism equivalent of `Kernel Modules` as in the case of Linux. Some projects, such as LIDS [105] have developed a mandatory access control for Linux using kernel patches.

Similar to the wrapper-based approach, security mechanisms are independent of the underlying system, thus limiting the extent of RBMs that can be supported. The primary advantage of the integrated approach is its flexibility to add a minimum amount of code and incorporate many kinds of optimization that result in good performance. However, the implementation of an integrated approach requires access to the internals of the system. As the implementation of the RBMs is by a system-by-system basis, it is very hard and in most cases, impossible to maintain. In addition, it is not clear how event constraints can be supported as none of underlying systems provide a way of specifying and detecting events. In the case of Linux, implementation of RBMs may also depend on the kernel version. Even though an integrated approach is viable in some systems it is not the case always. Analogous to the wrapper-based approach, the integrated approach is not flexible, extensible or portable.

### 4.2.3 The Event-driven Approach

The basic assumption for this approach is that the underlying system provides some *hooks* that can be exploited to incorporate RBMs effectively. Utilizing the hooks provided, additional user-required functionalities, such as the RBMs, can be supported in the underlying system. In this approach, policy enforcement and the incorporation of those policies into the system are separated. This is done using the following steps:

- enterprise access control policies are provided as high level specifications by security administrators.

- event based active rules (please refer Section 4.3) are generated automatically (or semi-automatically) from the high level specifications. In other words, supporting RBMs using event based active rules.

- events are detected from the underlying system whenever any user-identified event occurs. For example, an event can be the invocation of an application or application process (please refer Section 4.3 for other type of events). The detected events trigger appropriate set of active rules, thus enforcing RBMs.

The first step, which involves high level specification of policies, can be done via a graphical user interface and is briefly explained in Section 4.5. In order for generating event-based active rules from the high level specification, first we have to illustrate the adequacy of these active rules for enforcing RBMs. If this is viable, then events can be raised and appropriate conditions can be checked based on roles, and RBMs can be supported in a uniform and transparent manner. We have explained event-based active rules, how they are used in supporting RBMs, how they can be generated, and how active security is provided, in Sections (4.3-4.5).

As this is an event-driven approach, primitive events (i.e., occurrence of interest) have to be identified in a system-by-system basis. Thus, identification and generation of primitive events is crucial in this approach. Primitive events can be based on temporal (system time), location-based (sensors), invocation of applications, processes and library routines in the underlying system, insert, delete and update of tuples in relational databases, and so forth. Composite events (please refer Section 4.3) can be maintained within the system or outside of the system. Primitive events can be generated in various ways and are explained in Section 4.2.3.1. For example, `System Calls` can be intercepted and utilized to generate primitive events in Linux. Similarly, triggers can be used for generating events in relational databases [151]. Once the events are generated or triggered, using event-based active rules (refer Section 4.3), RBMs can be effectively supported.

Event-based active rules support RBMs with constraints, such as temporal, context-aware, and so forth. Furthermore, as this is an event-driven approach, *event* constraints can be supported without any problem. On the other hand, the same set of event-based active rules can be reused in any underlying system for enforcing RBMs, making it portable. Thus, a set of rules can enforce a particular Role-Based Model and the same set of rules can be reused in any underlying system (e.g., Linux, Windows, Databases). As this approach can support RBMs and any arbitrary event, it is flexible and generalized.

### 4.2.3.1 Approaches for Event Generation

In this section, we will elaborate on how events can be generated from an underlying system. As temporal events can be generated from any underlying system and location-based events can be generated from any sensor, we will concentrate on other events, such as invocation of applications, processes, insertion of tuples in relational databases and so forth.

**Linux Systems:** In Linux systems, various approaches can be used for generating primitive events. We have explained two of such approaches below.

- SYSTEM CALLS: These calls cause software interrupts and serve as gates into the kernel when an operating system service is required [150]. Pointers to these calls are maintained in a table called "sys_call_table". Primitive events can be triggered by altering this table, but how the altering is done is totally dependent on the kernel version. The main advantage of this approach is that the generation of primitive events are centralized, as any process in the system has to finally invoke the system call in order to access any service provided by the operating system.

- LIBRARY ROUTINES: "Library functions are linked to programs and are more portable [150]." System calls are wrapped around by library routines for performing certain operations. For example, fopen() (function open) in turn invokes *sys_open()* to actually

open a file. Thus, primitive events can be generated from library routines. The main disadvantage of this approach is the identification of library routines that need to generate primitive events.

**Operating Systems:** The above approaches are particular to Linux-based systems, but there are many generalized approaches for generating primitive events in Linux, Windows, Solaris and other systems. One such approach is the binary instrumentation approach [152, 153].

- BINARY INSTRUMENTATION: In this approach, instrumentation is performed at the binary level [152]. Instrumentation is the ability to control the access to constructs for possible semantic alteration pertaining to any code. Constructs can be either pure, i.e., functions following standard calling conventions, or arbitrary, i.e., code blocks composed of instructions not adhering to standard calling conventions. Similarly, an instrumentation can be local, i.e., for a single process or a thread within the process or global, i.e., for all processes. In addition, instrumentation can also be active or passive. Thus, by utilizing binary instrumentation, primitive events can be triggered whenever a process or thread within a process tries to request a service from the operating system. Once events are triggered, active rules can be used to check for role-based constraints before allowing the process to proceed further.

**Databases:** Similar to the approach presented in [154], events can be generated in object oriented databases whenever a function is invoked by an object. In relational databases [151], triggers can be used to generate primitive events for basic operations, such as insert, delete and update.

**Applications:** Similar to the binary instrumentation discussed above, programming instrumentation can be used to generate events from applications. For example, Bauer et. al. [155] discuss Polymer – a language and system, for defining and composing complex security policies for Java applications at run-time. Polymer enforces security

policies on untrusted Java applications by dynamically monitoring their behavior and modifying them. Although events can be generated from Java applications using this approach, it still falls under the category of the integrated approach discussed above. Thus, it inherits all the problems of the integrated approach, since Java Virtual Machine or Java Language API can change over time.

### 4.2.3.2 Optimizations

It is apparent that in order to support different RBAC components (or RBMs) in the same system, one should not hardwire the semantics of the operations, such as role activate and role assign, precisely as the semantics differ from one component to the other. Let us take an enterprise that formulates policies using hierarchical RBAC. There are $\mathcal{I}$ roles, where $\mathcal{J}$ roles have hierarchies and $\mathcal{K}$ roles don't take part in role hierarchy relationships (i.e., $\mathcal{I} = \mathcal{J} + \mathcal{K}$ and $\mathcal{J} \cap \mathcal{K} = \Theta$). If NIST RBAC is implemented as components, then for all the roles (i.e., $\mathcal{I}$) the conditions pertaining to hierarchies should be checked irrespective of their membership in the hierarchies. On the other hand, if NIST RBAC is implemented following the role semantics, then role hierarchy conditions can be checked only for those $\mathcal{K}$ roles. The implementation of NIST RBAC as components or following the role semantics become even more crucial with SOD relations and other constraints.

With the event-driven approach, it is easier to enforce RBMs based on the role semantics rather than based on the access control model, as one event can be associated with many active rules. The association of various "Condition-Action-Alternative Action" component with "AddActiveRole" operation from [12], is depicted in Figure 4.2. Thus, in the rest of the chapter we will show how event-driven approach is used to enforce RBMs using event-based active rules.

$AddActiveRole$
Condition-Action-Alternative Action  *for core RBAC*
Condition-Action-Alternative Action  *for Hierarchical  RBAC*
Condition-Action-Alternative Action  *for Static SoD  RBAC*
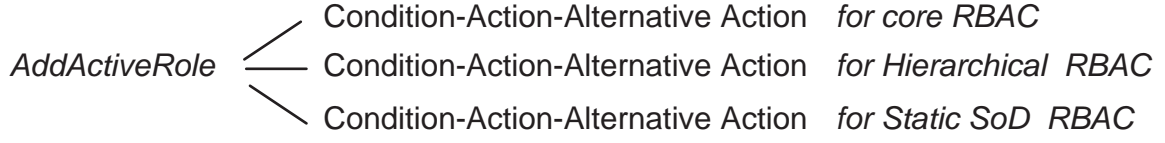
Figure 4.2. Optimizations based on Role Semantics.

## 4.3   Event-Based Active Authorization Rules

We introduce `On-When-Then-Else` authorization rules, which are enhanced `ECA` rules with alternative actions and enhanced operator semantics to support authorization management of data. (*Note: In the rest of the chapter, we will use active authorization rules, active rules, authorization rules, and OWTE rules, interchangeably.*)

Active authorization rules consist of five components and they are

- A Rule name (or $\mathcal{R}_{name}$),

- An occurrence "O" of an event (or an occurrence of interest) $\mathcal{E}_i$ that triggers a set of rules,

- "W" checks the conditions $<\mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_n>$ when an associated event is triggered,

- "T" triggers a set of actions $<\mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_n>$ when the conditions evaluate to TRUE, and

- "E" triggers a set of alternative actions $<\mathcal{AA}_1, \mathcal{AA}_2, \ldots \mathcal{AA}_n>$ when the conditions evaluate to FALSE.

An event occurrence can trigger rules that can be in the form of multiple rules, nested/cascaded rules, prioritized rules, and causality rules. OWTE rules are specified as shown below:

```
RULE [   R_name
```

$\phantom{RULE [ }$ ON $\quad$ Event $< \mathcal{E}_i >$

$\phantom{RULE [ }$ WHEN $\quad < \mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_n >$

$\phantom{RULE [ }$ THEN $\quad < \mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_n >$

$\phantom{RULE [ }$ ELSE $\quad < \mathcal{AA}_1, \mathcal{AA}_2, \ldots \mathcal{AA}_n >$ ]

### 4.3.1 Simple Events

An event is an occurrence of interest in an application or a system. All the events that are predefined in the underlying system (i.e., domain-specific) are known as primitive or simple events (Please refer Chapter 3 for more information on events). *File operations* (i.e., opening, closing, etc.) in operating systems, *method execution* by objects in object oriented systems, *data manipulations*, such as insert, delete and update, in relational database management systems, *system clock* of the underlying system (i.e., absolute or relative temporal events), *external* events (i.e., based on the data from sensors), and so forth, are all `simple` or `primitive` events. For instance, shown below is an event $\mathcal{E}_i$ that is detected when a function $\mathcal{F}$ is invoked by an object $\mathcal{U}$. Parameters $(< \mathcal{PA}_1, \mathcal{PA}_2, \ldots \mathcal{PA}_n >)$ are used by OWTE rules for checking conditions and performing actions.

Event $\mathcal{E}_i = \mathcal{U} \rightarrow \mathcal{F}(< \mathcal{PA}_1, \mathcal{PA}_2, \ldots \mathcal{PA}_n >)$

Some of the above mentioned events are used to enforce various functionalities of RBAC. For example, when *a user moves from one location to another*, external events can trigger some rules that can "activate/deactivate" roles.

### 4.3.2 Conditions

Multiple conditions i.e., $< \mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_n >$ can be associated with an event. These conditions are evaluated when an event occurs. For example, when a user tries to *open*

a protected file in a pervasive computing domain, the system can check whether the network is *secure* or *insecure* and can make decisions accordingly.

### 4.3.3 Actions and Alternative Actions

Once events are detected and all the associated conditions are evaluated to TRUE, the predefined system critical actions (i.e., $< \mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_n >$) are performed. For example, when an internal security is triggered, the system takes the following actions; *i)* generate reports and alert administrators, *ii)* deactivate a set of roles, *iii)* demote certain roles' permissions, and *iv)* block access requests or impose certain access restrictions. On the other hand, current event processing models do not handle when the conditions evaluate to FALSE. In OWTE rules, alternative actions $< \mathcal{A}\mathcal{A}_1, \mathcal{A}\mathcal{A}_2, \ldots \mathcal{A}\mathcal{A}_n >$ are triggered when the condition evaluation returns FALSE. Alternative actions are critical in the security domain. For example, when the user is in an insecure network, then the protected file access should be *denied.*

**Rule 1** (Rule with a Simple Event). *Create a rule that checks for permissions when user Bob tries to open a file "patient.dat" using the command* [3] *"vi (patient.dat)".*

EVENT    $E_1 = Bob \rightarrow vi(patient.dat)$

RULE [    $\mathcal{S}_1$

ON    $E_1$

WHEN  if (checkaccess(Bob, patient.dat) is TRUE)

return TRUE; else return FALSE;

THEN   $< allowopening\ patient.dat >$

ELSE    raise error "insufficient privileges"   ]

---

[3]We have used vi(patient.dat) to indicate that the file is opened using vi editor. This is just for understanding and it is not the case always.

When user Bob opens the file i.e., **O**, permissions are checked using "*checkaccess*" i.e., **W**; if Bob has the permission (i.e., if the condition returns TRUE), **T**hen the file is opened, **E**lse an *error* is raised.

### 4.3.4   Complex Events

In addition to simple events, complex events are often required in many situations. Using complex or composite events additional constraints can be placed on event occurrences while providing access control. Complex events are composed of more than one simple or complex events using event operators [41, 93]. Some of the event operators are AND, OR, NOT, SEQUENCE, Periodic, Aperiodic, and PLUS, and are formally defined in Chapter 3. We explain some of these operators below in the context of RBAC and its extensions, even though all of them are critical in access control.

**Sequence $(E_1, E_2)$:** When two events[4] $E_1$ and $E_2$ occur (i.e., "O"), a SEQUENCE event is detected and the corresponding rules are triggered. With this event operator, $E_1$ should occur before $E_2$. The condition that *a user should be active in role A to activate role B* (i.e., prerequisite roles in RBAC) can be specified using this event operator.

**OR $(E_1, E_2)$:** This event is detected when any one of the two events, $E_1$ or $E_2$, occurs and the corresponding rules are triggered.

**Plus $(E_1, \delta)$:** This event is a relative temporal event. A simple or composite event occurrence starts a PLUS event (i.e., at time "$T$"). After the specified time interval or duration "$\delta$" (i.e., at time "$T + \delta$"), the PLUS event is detected. For example, *a user can be deactivated from a role after a certain duration "$\delta$"* using this operator.

**Aperiodic $(E_1, E_2, E_3)$:** This event is detected whenever the event $E_2$ occurs between two other events $E_1$ and $E_3$. The event $E_1$ starts the Aperiodic event and $E_3$ terminates the same. Event occurrences of $E_2$ cannot detect an Aperiodic event before

---

[4]Events are represented as $E_i$ in all the operators and they can be both simple and complex.

the occurrence of event $E_1$ or after $E_3$. Only when the event $E_2$ occurs within $E_1$ and $E_3$, an Aperiodic event is detected and the corresponding rules are triggered. For instance, using Aperiodic, *a role can be allowed to be enabled only during a transaction.*

**Periodic $(E_1,\ \tau,\ E_3)$:** This event is similar to Aperiodic event except that it is detected at a regular time interval "$\tau$" between two other events $E_1$ and $E_3$. This event operator can be used to *periodically monitor the underlying system and generate reports.*

All the above mentioned operators are critical and are necessary for providing authorization management of data. Apart from the examples provided, all the operators are also used for supporting various other functionalities of RBAC. Even though there are other operators, we have explained only a few in the context of RBAC and its extensions. A rule involving event operator PLUS is shown below.

**Rule 2** (A Rule with a Complex Event). *Create a rule for restricting user Bob from keeping the file "patient.dat" open for more than 2 hours (i.e., $\delta$). In other words, close the file forcefully after 2 hours.*

RULE [   $\mathcal{C}_1$

       ON     PLUS($E_1$, 2 hours)

       WHEN   TRUE

       THEN   $< Close file >$   ]

In the above rule, the PLUS event[5] is started when user Bob opens the file "patient.dat" using the "vi" editor (i.e., event $E_1$ from Rule 1). This event is detected when the duration $\delta$ (i.e., 2 hours) is elapsed, and the file is closed forcefully.

## 4.4    Active Authorization Rules Synthesis for Access Control Enforcement

In this section, we show the mapping between the basic elements in RBAC and its extensions and the OWTE rule specification. In doing so, we are establishing OWTE rules

---

[5]In Rule 2, the PLUS event in the "ON" clause can be a named event as in Rule 1 i.e., "ON $E_2$", where Event $E_2 = \text{PLUS}(E_1, 2 hours)$.

as an enforcement mechanism for the realization of access control policies. In addition, we will demonstrate the following; 1) rules enforcing RBAC, 2) rules enforcing RBAC extensions, and 3) rules supporting active security.

### 4.4.1 Entity Relationship Modeling

RBAC contains three basic element sets namely users (or $\mathcal{U}$), roles (or $\mathcal{R}$), and permissions (or $\mathcal{P}$). In addition to these basic elements, RBAC extensions have additional elements, such as "purpose" and "object-policy" in privacy-aware RBAC [100] and so forth. $\mathcal{U}$ represents humans, user applications and so forth, $\mathcal{R}$ represents a job function in an enterprise, $\mathcal{P}$ represents the operations that can be carried out on objects by $\mathcal{R}$, and "purpose" represents business purposes[6]. All the basic elements are considered as entities as they represent something that has a separate existence or conceptual reality. All the users (i.e., humans, user agents, etc.) and roles (e.g., manager, cashier, etc.) in an enterprise are modeled as *entity instances* of the basic entities $\mathcal{U}$ and $\mathcal{R}$, respectively.

Entities $\mathcal{U}$ and $\mathcal{R}$ have M:N (i.e., many-to-many) relationship. Similarly, entities can be associated with other entities by the means of role-permission assignments, role hierarchies, purpose hierarchies, and so forth. Thus, associations between the entities represents their *relationships* forming an Entity Relationship [156] like model. In this work we will consider only the entities $\mathcal{U}$ and $\mathcal{R}$ and their relationships while discussing our approach. On the other hand, constraints, such as separation of duty relations, temporal, context-aware, active security, and others are placed on the relationships [7] so that only entity instances that satisfy the constraints are allowed to take part in the relationship. For example, constraints that are placed on user-role activation are checked when an instance of entity $\mathcal{U}$ needs to take part in the relationship with an instance of entity $\mathcal{R}$.

---

[6]The purpose for which an operation is executed [100].

[7]These are similar to descriptive attributes [156] in ER model but for different purposes

### 4.4.2 Mapping OWTE and RBAC Elements

Mappings between OWTE rule elements and RBAC elements are provided below.

- All the events are represented as the set $\mathcal{O}$, conditions as $\mathcal{W}$, actions as $\mathcal{T}$, alternative actions as $\mathcal{E}$, and all the OWTE rules as $\mathcal{S}$. External events are represented as $\mathcal{EE}$, system generated or local events as $\mathcal{LE}$, and application generated events as $\mathcal{AP}$. Simple events set $\mathcal{SE} = \bigcup\{\mathcal{AP}, \mathcal{EE}, \mathcal{LE}\}$. Complex events are represented as the set $\mathcal{CE}$ and they contain several constituent events based on the operator semantics. Thus, event set $\mathcal{O} = \bigcup\{\mathcal{CE}, \mathcal{SE}\}$.

- In access control, different requests are initiated by $u \in \mathcal{U}$. For example, when a user needs to activate a role he/she should make a request. When all the requests are considered as a function/method call, then $u \in \mathcal{U}$ will invoke a method (i.e., $u \rightarrow F(\ldots)$) in order to perform any activity in the system. Thus, all the requests, such as role assignments/deactivations/enabling/disabling, accessing objects, etc, are represented as the set $\mathcal{Q}$. All the requests that are made by $u \in \mathcal{U}$ are represented as the set $\mathcal{UQ} \subseteq \mathcal{U} \times \mathcal{Q}$, and all the user requests $\mathcal{UQ}$ form a subset of application events.

- A rule $s \in \mathcal{S}$ is triggered when an event $o \in \mathcal{O}$ is detected. In NIST RBAC, only an element $uq \in \mathcal{UQ}$ can trigger the simple event $se \in \mathcal{SE}$, as it alone can request accesses, assignments, activations, and so forth. On the other hand, in RBAC extensions and active security, temporal events $te \in \mathcal{LE}$, external events $ee \in \mathcal{EE}$, such as locations from sensors, and so forth can trigger simple event $se \in \mathcal{SE}$. For example, when a user $u \in \mathcal{U}$ tries to activate a role $r \in \mathcal{R}$ using a user request $uq \in \mathcal{UQ}$, it will trigger an event $o \in \mathcal{O}$. Similarly, any complex event $ce \in \mathcal{CE}$ is always triggered by simple events, as they compose more than one simple or complex event.

- Constraints are placed while an entity tries to take part in a relationship with another entity (i.e., when requests are made). Diverse constraints are possible in an access control system, such as; i) assignment time constraints, ii) activation time constraints, and iii) temporal constraints. All the constraints can be imposed with $w \subseteq \mathcal{W}, ee \in \mathcal{EE}, le \in \mathcal{LE}$, and $ce \in \mathcal{CE}$. For example, when a user tries to activate a role by triggering an event, conditions will check whether the user has the permission to be active in role (i.e., $(u, r) \in \mathcal{UA}$[8]). On the other hand, constraints such as prerequisite roles, maximum number of active roles for an user, how long a user can be active in a role, etc. can be specified using $ce \in \mathcal{CE}$.

- When a rule $s \in \mathcal{S}$ is triggered $t \subseteq \mathcal{T}$ and $e \subseteq \mathcal{E}$ are performed which in turn allow or deny requests $q \subseteq \mathcal{Q}$.

### 4.4.3 Enforcement using Active Rules

In this section, we demonstrate the use of active rules for enforcing certain functionalities of RBAC and its extensions, and for providing active security. Generation of authorization rules along with their implementation are discussed in Section 4.5.

All the active authorization rules that are generated form a *rule pool*. Three kinds of rules are available in the rule pool; *i)* administrative rules, *ii)* activity control rules, and *iii)* active security rules. Administrative rules are used with high level specification of access control policies, activity control rules are used to control the activities that can be performed by the instances of $\mathcal{U}$, and active security rules are used for monitoring the state changes and taking preventive measures. Rules are generated at different granularities within each classification. *Specialized* rules that are specific to an instance of $\mathcal{U}$ (e.g., Bob), *localized* rules are specific to a particular role and are created based on the role properties, and *globalized* rules are generalized and are not specific to any role.

---

[8]User-role assignments are set $\mathcal{UA} \subseteq \mathcal{U} \text{ X } \mathcal{R}$ (Please refer to [12] for more details).

Consider three simple scenarios; 1) An user Jane should be restricted to a maximum of five active roles at a time, 2) A role Programmer can be activated only by five users at a time, and 3) An user Jim needs to be assigned to a role. Authorization rule corresponding to scenario 1 is a *specialized* rule, as it restricts a particular user Jane from being active in more than five roles. On the other hand, for scenario 2, the rule should be based on the role as there can be many users who can be active in a role. Thus, a *localized* rule that correspond to a particular role is created to limit the number of active users. On the contrary, for scenario 3, user Jim should be assigned to a role and it can be any role. This rule is a *globalized* rule, as it controls all the user-role assignments (i.e., the same rule is invoked with different parameters). Rules corresponding to scenarios 1 and 2 are activity control rules whereas 3 is an administrative rule.

### 4.4.3.1 Rules Illustrating RBAC Enforcement

We demonstrate the enforcement of core, role hierarchies, static SoD, and dynamic SoD RBAC Standard components with the following active authorization rules.

**Rule 3** (Add Active Role). *Assume that a user is assigned to role R1. In order to perform some operations that are allowed for R1, the user has to activate the role R1. The rule activates R1 by adding R1 to the active role set of that user session.*

EVENT $\quad E_2 = user \rightarrow AddActiveRole\text{R1(sessionId)}$

This rule corresponds to the "addActiveRole" from [12]. When the user tries to activate the role R1, the function "AddActiveRoleR1" is invoked with the users' session identifier (or sessionId) as its parameter. This raises the event $E_2$, which in turn triggers the rule that checks whether the user can be activated in the role R1. Below shown are four rules that correspond to different role properties;

RULE  [   $\mathcal{AAR}_1$

ON   $E_2$

WHEN  (user IN userL) && (sessionId IN sessionL) &&

(sessionId IN checkUserSessions(user)) &&

(R1 NOT IN checkSessionRoles(user)) &&

(checkAssignedR1(user) IS TRUE)

THEN   addSessionRoleR1(sessionId)

ELSE   raise error "Access Denied Cannot Activate"  ]

The rule $\mathcal{AAR}_1$ is used when role R1 does not take part in any relationship (i.e., *core RBAC*) such as hierarchies and SoD relations. First it checks whether user is available in list userL[9], then it checks whether the sessionId exists in list sessionL and whether the session is owned by that user. Once verified, it checks whether the user is assigned to role R1 using the function "checkAssignedR1" as a user should be assigned in order to activate any role. It then checks whether the role R1 is not activated in that session using "checkSessionRoles". Once all the above conditions are verified, role R1 is activated in that user session by invoking the function "addSessionRoleR1" and adding it to the *active role set*[10].

---

[9]We assume, users' lists, role lists, and session lists containing user, role and session information, respectively, are already available. We also assume that other functions that are used in the rule are also available.

[10]The set containing all the active roles for an user.

RULE  [  $\mathcal{AAR}_2$

  On $E_2$

  When (user IN userL) && (sessionId IN sessionL) &&

      (sessionId IN checkUserSessions(user)) &&

      (R1 NOT IN checkSessionRoles(user)) &&

      (checkAuthorizationR1(user) IS TRUE)

  Then addSessionRoleR1(sessionId)

  Else raise error "Access Denied Cannot Activate"  ]

The rule $\mathcal{AAR}_2$ is used when the role R1 takes part in the *general role hierarchies.*
All the conditions are same as in rule $\mathcal{AAR}_1$ except one condition, which checks whether
the user is authorized to that role using the function "checkAuthorizationR1" instead of
"checkAssignedR1" . This is carried out, since the user can activate the role R1 if he is
assigned to the role R1 or to any of its senior roles.

RULE  [  $\mathcal{AAR}_3$

  On $E_2$

  When (user IN userL) && (sessionId IN sessionL) &&

      (sessionId IN checkUserSessions(user)) &&

      (R1 NOT IN checkSessionRoles(user)) &&

      (checkAssignedR1(user) IS TRUE) &&

      (checkDynamicSoDSet(user, R1) IS TRUE)

  Then addSessionRoleR1(sessionId)

  Else raise error "Access Denied Cannot Activate"  ]

The rule $\mathcal{AAR}_3$ shown below is used when the role R1 takes part in the *dynamic
SoD relation without hierarchies.* This rule is similar to the rule $\mathcal{AAR}_1$ but with addi-
tional conditions for checking whether the dynamic SoD constraints are satisfied. Func-

tion "checkDynamicSoDSet" checks whether the addition of role R1 to the active role set of the user satisfies the mutual exclusive constraints discussed in [12].

RULE [ $\mathcal{AAR}_4$

   On $E_2$

   When (user IN userL) && (sessionId IN sessionL) &&

      (sessionId IN checkUserSessions(user)) &&

      (R1 NOT IN checkSessionRoles(user)) &&

      (checkAuthorizationR1(user) IS TRUE) &&

      (checkDynamicSoDSet(R1) IS TRUE)

   Then addSessionRoleR1(sessionId)

   Else raise error "Access Denied Cannot Activate" ]

The rule $\mathcal{AAR}_4$ is used when the role R1 takes part in the *dynamic SoD relation with hierarchies*. This rule is similar to $\mathcal{AAR}_2$ but with additional conditions for checking whether dynamic SoD constraints are satisfied, similar to $\mathcal{AAR}_3$.

Similar to all the above scenarios, activating a role that has the *static SoD relation* will use the rule $\mathcal{AAR}_2$ if the role R1 takes part in the role hierarchies or the rule $\mathcal{AAR}_1$ is used if it does not take part.

**Rule 4** (Cardinality Constraints). *Restrict the number of users who can be active in a role at the same time. For instance, there is only one person in the role of a university president. Below shown rule allows only five users to be active in role R1 at a time. Similarly, the number of roles a user can be active at the same time can also be restricted.*

EVENT $E_3 = addSessionRoleR1(sessionId)$

EVENT $E_4 = removeSessionRoleR1(sessionId)$

RULE [  $\mathcal{CC}_1$

    ON    $E_3$

    WHEN  if (CardinalityR1(INCR) IS TRUE) return TRUE

          else return FALSE

    THEN   perform action <add role R1 to session with sessionId>

    ELSE   raise error "Maximum Number of Roles Reached"   ]

RULE [  $\mathcal{CC}_2$

    ON    $E_4$

    WHEN  TRUE

    THEN   CardinalityR1(DECR)   ]

Cardinality constraint for the above mentioned scenario requires to restrict the *role activation* so that no more than five users are activated. When the user tries to activate role R1 it triggers any one rule from $\mathcal{AAR}_1 \ldots \mathcal{AAR}_4$ based on the access control policy and role property, since these rules are used to activate the role R1. When the user satisfies all the conditions then the function "addSessionRoleR1" is invoked in order to add the role R1 to *active role set* of that user session. This function raises event $E_3$ which in turn raises rule $\mathcal{CC}_1$ shown above. It checks whether the maximum limit of 5 users is reached by invoking the function "CardinalityR1" with value INCR indicating the addition of a user. If the maximum number of users for role R1 is not reached, then the role is activated, else the error "Maximum Number of Roles Reached" is raised by rule $\mathcal{CC}_1$. Similarly, when the role R1 is deactivated event $E_4$ is detected and the function "CardinalityR1" is invoked with DECR as the parameter, which reduces the count of the number of users active in role R1 by one, so that new users can be activated.

**Rule 5** (Check Access). *Check whether the subject (i.e., instance of $\mathcal{U}$) of a given session is or is not allowed to perform a given operation (e.g., read, write, etc.) on a given object (e.g., .dat file, etc.)*

EVENT $\quad E_6 = user \rightarrow checkAccess(sessionId, operation, object)$

RULE [ $\quad \mathcal{CA}_1$

$\qquad$ ON $\quad E_6$

$\qquad$ WHEN (sessionId IN sessionL) &&

$\qquad\qquad$ (operation IN opsL) && (object IN objL) &&

$\qquad\qquad$ (For ANY role IN getSessionRoles(sessionId)

$\qquad\qquad\quad$ (IF checkPermissions(operation,object, role) IS TRUE

$\qquad\qquad\qquad$ return TRUE))

$\qquad$ THEN $\;$ < allow Access >

$\qquad$ ELSE $\;$ raise error "Permission Denied" ]

The rule $\mathcal{CA}_1$ is triggered when the user tries to perform any operation on any object (i.e., event $E_6$). The rule allows the user-requested operation only when at least one role from his *active role set* for that session has the required permission. This is carried out by the "For" statement that retrieves all the roles from the *active role set* and checks whether at least one role has the required permission using the "checkPermissions" function. Above shown rule is the same for all the roles that do or do not take part in any type of relationships.

### 4.4.3.2 Rules Illustrating RBAC Extensions Enforcement

Even though RBAC has been extended extensively with various constraints, below we demonstrate the support for temporal and control flow dependency constraints. Generalized Temporal RBAC [23] provides an exhaustive set of temporal constraints.

Control flow dependency constraints often occur in task oriented systems and are stricter forms of dependency constraints [37]. We show how a subset of time based SoD and post-condition constraints from [37] are supported.

**Rule 6** (Disabling Time SoD). *Two roles from a given role set $\mathcal{RS}$ cannot be disabled at the same time in the interval $(I, P)$ (NOTE: $(I, P)$ corresponds to $\langle[begin, end], P\rangle$, where P is a periodic expression denoting an infinite set of periodic time instants, and [begin, end] is a time interval denoting lower and upper bounds that are imposed on instants in P). Role disabling is of main concern where availability is a primary concern [37]. For instance, both "Nurse" and "Doctor" roles cannot be disabled at the same time within the interval ([begin, end], P)[11].*

EVENT     $StartD = event corresponding to date expression$

EVENT     $EndD = event corresponding to date expression$

EVENT     $ET_1 = roleDisableNurse()$

EVENT     $ET_2 = roleDisableDoctor()$

EVENT     $ET_3 = OR(ET_1, ET_2)$

EVENT     $ET_4 = Aperiodic([StartD], ET_5, [EndD])$

EVENT     $ET_5 = Aperiodic([10:00:00/*/*/*], ET_3,$

$$[17:00:00/*/*/*])$$

---

[11] In this example we consider P as 10 a.m. to 5 p.m. every day. 10 a.m. every day is represented as [10:00:00/*/*/*], where the general form is "24h:mi:ss/mm/dd/yyyy". Periodic expression P in GTRBAC can be represented using event operators in active rules.

RULE [  $\mathcal{TSOD}_1$

        ON    $ET_4$

      WHEN ( if roleDisableNurse == TRUE

                ((if checkActiveDoctor() IS TRUE) return TRUE

                    else return FALSE)

              else if roleDisableDoctor == TRUE

                ((if checkActiveNurse() IS TRUE) return TRUE

                    else return FALSE))

      THEN  ( if roleDisableNurse == TRUE Then disableNurse()

             else if roleDisableDoctor == TRUE Then disableDoctor())

      ELSE  ( if roleDisableNurse == TRUE

              raise error "Denied as Doctor Already Disabled"

            else if roleDisableDoctor == TRUE

            raise error "Denied as Nurse Already Disabled"  ]

The rule $\mathcal{TSOD}_1$ provides time based SoD constraints, which does not allow both the roles "Nurse" and "Doctor" to be disabled at the same time in the interval $(I, P)$. We have represented the interval $I$ and $P$ as [StartD, EndD] and ([10:00:00/*/*/*], [17:00:00/*/*/*]), but they can any type of simple or complex event. For example, [StartD] can be the start of the year and [EndD] can be the end of the year. The event $ET_1$ is raised whenever the role "Nurse" needs to be disabled. This will trigger the event $ET_3$ that is an OR event, which propagates the same to both the Aperiodic events $ET_4$ and $ET_5$. The event $ET_4$ triggers the rule $\mathcal{TSOD}_1$ when an user tries to disable the role Nurse within 10 a.m. and 5 p.m. and within [begin, end]. The rule $\mathcal{TSOD}_1$ determines whether the role "Nurse" can be disabled by checking whether the role "Doctor" is active, if so it allows to disable the role "Nurse" else it raises an error. In the similar way,

disabling of the role "Doctor" is addressed. All the internal conditions such as checking with role lists are not shown in the above rule.

**Rule 7** (Deactivating a Role after $\delta$). *Deactivate an activated role after a duration $\delta$. This is similar to limiting car parking to a fixed number of hours at one time [23]. Below shown rules restrict the duration constraints on a per user-role basis, where a role R3 is deactivated after the specified maximum duration in one activation by the user Bob.*

EVENT     $ET_5 = Bob \rightarrow addActiveRoleR3(sessionId)$

EVENT     $ET_6 = startEventET_7(sessionId)$

EVENT     $ET_7 = PLUS(ET_6, \delta)$

RULE [   $\mathcal{AAR}_5$

ON     $ET_5$

WHEN  ...

THEN   (...) startEvent$ET_7$(sessionId)

ELSE   ...   ]

RULE [   $\mathcal{TSOD}_2$

ON     $ET_7$

WHEN  TRUE

THEN   deactivateRoleR3(sessionId)   ]

The event $ET_5$ is raised whenever the user Bob activates the role R3. This triggers the rule $\mathcal{AAR}_5$ which in turn raises the event $ET_6$. We have not shown all the other clauses of the rule $\mathcal{AAR}_5$, intentionally. The event $ET_6$ starts the PLUS event $ET_7$. After the duration $\delta$ the event $ET_7$ is detected and the rule $\mathcal{TSOD}_5$ is raised, which deactivates the role R3. The event $ET_5$ cannot be used to start the PLUS event $ET_7$ as the event $ET_7$ should be started only after the role R3 is activated.

**Rule 8** (Post-condition CFD). *If an event occurs, then the other event must also occur. For instance, if role "SysAdmin" role is enabled then role "SysAudit" must also be enabled, other wise both the roles should not be enabled.*

EVENT $\quad ET_8 \ = \ enableRoleSysAdmin()$

EVENT $\quad ET_9 \ = \ enableRoleSysAudit()$

EVENT $\quad ET_{10} \ = \ disableRoleSysAdmin()$

EVENT $\quad ET_{11} \ = \ disableRoleSysAudit()$

RULE [ $\ \mathcal{CFD}_1$

$\qquad$ ON $\quad E_8$

$\qquad$ WHEN $(\ldots)$

$\qquad$ THEN $\ (\ldots)$ enableRoleSysAudit()

$\qquad$ ELSE $\ $ raise error "Cannot Activate SysAdmin" ]

RULE [ $\ \mathcal{CFD}_2$

$\qquad$ ON $\quad E_9$

$\qquad$ WHEN $(\ldots)$

$\qquad$ THEN $\ (\ldots)$

$\qquad$ ELSE $\ $ disableRoleSysAdmin() &&

$\qquad\qquad$ raise error "Cannot Activate SysAudit" ]

As shown above, when the role "SysAdmin" has to be enabled, it will trigger event $ET_8$ which will trigger rule $\mathcal{CFD}_1$. This rule enables the role "SysAdmin" and tries to enable role "SysAudit" by triggering event $ET_9$. This event triggers rule $\mathcal{CFD}_2$ which in turn enables the role "SysAudit" when all conditions are met. When it cannot enable the role "SysAudit" it disables the role "SysAdmin" by invoking the function "disableRoleSysAdmin". As function "enableRoleSysAudit" is the only function that can enable the role "SysAudit" OWTE rules overcomes the problems faced by [37].

### 4.4.3.3   Rules Illustrating Active Security

Active security is critical for detecting and monitoring the state changes of the underlying system to take timely actions.

**Rule 9** (Transaction Based Activation). *Any junior employee is allowed to activate the role "JuniorEmp" ONLY IF the role "Manager" is activated. On the other hand, if the role "Manager" is deactivated, then role "JuniorEmp" should also be deactivated.*

EVENT $\quad ET_{12} \ = \ user \rightarrow addActiveRoleManager(sessionId)$

EVENT $\quad ET_{13} \ = \ user \rightarrow addActiveRoleJuniorEmp(sessionId)$

EVENT $\quad ET_{14} \ = \ user \rightarrow deactivateRoleManager(sessionId)$

EVENT $\quad ET_{15} \ = \ user \rightarrow deactivateRoleJuniorEmp(sessionId)$

EVENT $\quad ET_{16} \ = \ startEventET_{16}(sessionId)$

EVENT $\quad ET_{17} \ = \ startEventET_{17}(sessionId)$

EVENT $\quad ET_{18} \ = \ Aperiodic(ET_{16}, ET_{13}, ET_{17})$

RULE [ $\ \mathcal{ASEC}_1$

$\qquad$ ON $\quad E_{12}$

$\qquad$ WHEN $(\dots)$

$\qquad$ THEN $\ < activateRoleManager >$

$\qquad\qquad <$startEvent$ET_{16}$(sessionId)$>$

$\qquad$ ELSE $\quad$ raise error "Permission Denied" ]

RULE [ $\mathcal{ASEC}_2$

     ON    $E_{14}$

     WHEN  $(\dots)$

     THEN  $< deactivateRoleManager >$

          if activeJuniorEmp() IS TRUE

             $< deactivateRoleJuniorEmp >$

          startEvent$ET_{17}$(sessionId)

     ELSE   raise error "Permission Denied"  ]

RULE [ $\mathcal{ASEC}_3$

     ON    $E_{18}$

     WHEN  $(\dots)$ return (TRUE|FALSE))

     THEN  $< activateJuniorEmp >$

     ELSE    raise error "Permission Denied"  ]

The event $ET_{12}$ and $ET_{13}$ are raised when roles "Manager" and "JuniorEmp" have to be activated, respectively. Similarly, events $ET_{14}$ and $ET_{15}$ are raised when the roles have to be deactivated. On the other hand, the event $ET_{16}$ is raised after the role "Manager" is activated and the event $ET_{17}$ is raised after the role "Manager" is deactivated. As shown, the event $ET_{12}$ triggers the rule $ASEC_1$ which in turn activates the role and raises the event $ET_{16}$. Similarly, the event $ET_{14}$ triggers the rule $ASEC_2$ which in turn deactivates the role and raises the event $ET_{17}$.

Let us assume that a user is trying to activate role "JuniorEmp". It will raise the event $ET_{13}$, which does not take any action as the Aperiodic event $ET_{18}$ is not yet started. Activating role "Manager" triggers the event $ET_{12}$ which checks for the necessary conditions and activates the role, and raises the event $ET_{16}$ that in turn starts the Aperiodic event $ET_{18}$. Now, when an user tries to activate role "JuniorEmp", it raises

the event $E_{13}$ which in turn raises the Aperiodic event $ET_{18}$ that had been already started. This triggers the rule $\mathcal{ASEC}_3$ which in turn checks whether all the constraints are satisfied in the "W" clause and returns TRUE or FALSE. If it returns TRUE, role "JuniorEmp" is activated using $< activate Junior Emp >$. On the other hand, deactivating the role "Manager" raises the event $ET_{14}$. After deactivation, the role "JuniorEmp" is deactivated and the event $ET_{17}$ is raised, which in turn terminates the Aperiodic event $ET_{18}$. As the event $ET_{17}$ acts as a terminator it stops the Aperiodic event $ET_{18}$ and the future activation of role "JuniorEmp", until the role "Manger" is activated again.

### 4.4.4  Summary and Advantages of OWTE Rules

In this section we have explained the synthesis of active authorization rules for enforcing RBAC and its extensions. We have demonstrated the seamless approach for supporting RBAC, its extensions and active security with various examples and rules. In addition to the above, privacy-aware RBAC [100] can also be enforced using OWTE rules as it also follows the Entity Relationship model described before.

### 4.5  Prototype Implementation

Sentinel [148, 43] is an active object oriented system that supports event based rule capability i.e., Event-Condition-Action model, using a uniform framework. In Sentinel, a reactive object is an object that has traditional object definition plus an event interface and a notifiable object is capable of being informed of the occurrence of some event. The event interface lets the object designate some or all of reactive object methods as primitive event generators. Together, both the kind of objects enable asynchronous communication with the rest of the system. Sentinel includes an event detector that is responsible for processing all the notifications from different objects and eventually signaling to the rules that some event has occurred triggering them. In addition, external monitoring module supports external events such as those from sensors, thus, supporting location/context-

aware events. Sentinel+ is an enhanced version of Sentinel that supports OWTE rules. In our implementation, users are allowed to provide high level specification of enterprise access control policies (ACPs) using RBAC Manager, a graphical tool (i.e., a widget tool kit). In RBAC Manager, ACPs can be specified using various widgets which takes the form of a Entity-Relationship like model as described in Section 4.4.

Enterprise $XYZ$ described below will be used in this section to explain user specification, rule generation and implementation. In enterprise $XYZ$, ACPs are formulated using NIST RBAC with *static SoD with role hierarchies*. It consists of two major departments "purchase" and "approval". Purchase department is authorized to place the "purchase order" for equipments or other materials required by the enterprise. Approval department is the one that can authorize the purchases. Thus, static SoD relations are required, since the same person placing purchase orders cannot authorize it. In enterprise $XYZ$, there are five roles with the following hierarchies *purchase manager* (PM) $\rightarrow$ *purchase clerk* (PC) $\rightarrow$ *clerk*, and *approval manager* (AM) $\rightarrow$ *approval clerk* (AC) $\rightarrow$ *clerk*. For instance, role PM is a senior role to PC. Roles AC and PC have static SoD constraint relation between them. Since role hierarchies are present PM inherits the static SoD constraints from PC. Thus, a user assigned to the role PM cannot be assigned to the role AM or AC. Likewise, an user assigned to the role AM cannot be assigned to the role PM or PC.

High level specification for enterprise $XYZ$ is shown in Figure 4.3. All the nodes represent an instance of entity $\mathcal{R}$ (i.e., roles). First, the role nodes corresponding to roles PM, AM, PC, AC and Clerk are created. Flags corresponding to relationships (i.e., hierarchy, station SoD relations, and active security constraints) are stored in the node. For instance, role nodes PC and AC have the Static SoD flags set once they are connected using the *dashed* line. All the flags are set when the role node to TRUE or FALSE when the policies are specified using a graphical tool RBAC Manager. Parent
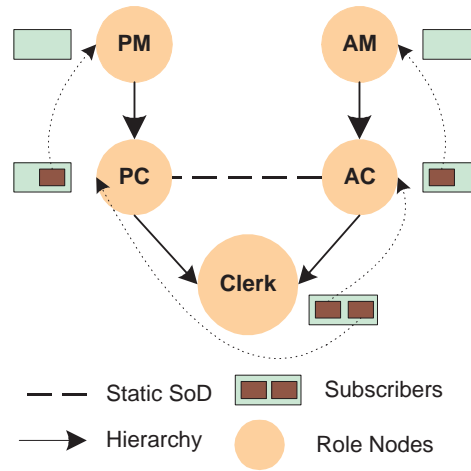
Figure 4.3. Access Control Policy Specification.

nodes are connected to the child nodes when there is a hierarchical relationship and static SoD constraints are represented as a dashed line between two nodes. Each node has an internal *subscriber* list that is used to point to the parent node. This pointer allow the child nodes to identify their parent nodes when the list of authorized users is required. On the other hand, constraints can be propagated in a bottom up manner using the pointers. (Note: Pointers shown in the Figure are not specified by users and they are generated by the system using the flags.)

Once the policies are specified, they are instantiated and the rules are generated. Let us take role PC for the discussion. Role PC has a static SoD and role hierarchy and when the policies are instantiated, rules corresponding to the role PC are generated. For instance, rule corresponding to activating role PC (i.e., "addActiveRolePC") is created. This rule is similar to rule $\mathcal{AAR}_2$ that was explained in Section 4.4 as role PC has static SoD and role hierarchies. Similarly all the other rules corresponding to PC and all the other roles are also created. Once all the rules are created it can enforce the policy specified by the enterprise. Currently, we assume that the policies specified using NIST

RBAC and others do not have inconsistencies, but we are in the process of developing advanced consistency checking mechanisms.

When there is a change in the policy, for example, the shift time of role "day doctor" is changed from (8 a.m. to 4 p.m.) to (9 a.m. to 5 p.m.), it can be easily changed in the high level specification and the corresponding rules can be regenerated. This can be done without burdening the administrator as the rules are created using the access specification graph. With current systems and models it is a cumbersome process as all the low level semantic descriptions have to be changed manually. When there are thousands of rules, it is highly error prone to change them manually.

## 4.6  Summary

In this chapter we have shown how active authorization rules rules are used to enforce RBAC, and its extensions such as temporal, and control flow dependency constraints in a uniform way. We have also shown how active security is provided that can take timely actions and prevent malicious activities. Rules generated have different granularities and classifications based on their functionalities. Expression of RBAC standard and its extensions in terms of OWTE rules provide practically applicable view of RBAC. Same set of rules can be used to realize all of the Role-Based models and the same set of rules can be realized in any underlying system (that provide some hooks) to support Role-Based models. Large enterprises have hundreds of roles, which requires thousands of rules for providing access control, and generating these rules manually is error-prone and a cognitive-burden for non-computer specialists. High level specifications of access control policies eliminate the above problems and are transparent to non-computer specialists. OWTE rules are not created manually by administrators and we have shown briefly how authorization rules are generated from high level specifications of access control policies using Sentinel+.

## CHAPTER 5

## GENERALIZATION OF ROLE-BASED ACCESS CONTROL

Time-, content-, and purpose-based, context-aware as well as other constraints play a critical role in realizing Role-Based Access Control in diverse domains. In this chapter, we extend and generalize RBAC with event pattern constraints. We introduce constraints based on the *expressive event patterns*[1] that allow RBAC security policies to include both simple and complex event patterns that occur over a period of time. These constraints can be specified on, and using, both role-dependent and role-independent operations, facilitating the modeling of complex constraints that are required in many real-world applications. First, we motivate the need for event pattern constraints in RBAC. Second, we generalize the simple and complex event definitions. Finally, we identify simple or basic events in the context of RBAC, and introduce various operators for modeling event pattern constraints.

## 5.1 Introduction

Enterprises in different domains have different access control requirements [32, 33, 34]. Constraints [14, 15, 16, 17, 18, 19, 20, 21, 22, 23] are critical in realizing RBAC over diverse domains, as they provide the flexibility for specifying fine-grained RBAC policies. Users, roles, and permissions are the three basic element sets of RBAC, and they are constrained for supporting fine-grained access policies. There has been a lot of work in adding constraints to RBAC such as time-, event-, content-, purpose-based, context-aware, and so forth.

---

[1]Event patterns are event expressions defined in Chapter 3.

Briefly, roles in RBAC can assume three different states [23]: *enabled, disabled* and *active*. Role operations such as enabling, disabling, activation, and deactivation are mainly constrained as they control the role-state changes. In addition, various other operations such as assignments, de-assignments and user access requests are also constrained. On the other hand, constraints can also be specified based on the users or permissions. Currently, constraints can be specified in many ways; parameterized roles, predicates, simple role-dependent events (events based on role operations), simple role-independent events (events based on external factors such as time), environmental roles (e.g., location) and so forth. Although RBAC has been explored and extended with various constraints, they use some static way of providing constraints (elaborated below). In particular, instead of extending RBAC along various dimensions (e.g., temporal dimension) individually, is it possible generalize constraint specification? Our *premise* is that any occurrence of interest can be considered as an event, and those events can be used to generalize constraint specification. This is detailed with the following example.

### 5.1.1 Motivation Examples

Generalized Temporal RBAC [23] extends the RBAC in the temporal dimension and provides exhaustive set of temporal constraints. Let us take the scenario of a class room lecture from GTRBAC [157]. In this example, GTRBAC places periodicity/duration constraints on the role-permission assignments (i.e., Time duration/period over which the object's permissions can be assigned to roles).

With virtual universities and online degree's awarded, online course management play a critical role. Consider the example of an online course management, shown in Figure 5.1 (reproduced from [157]). The main reason for selecting an online course management application, is their need and utilization of access control policies. Assume that a professor offers an online course on "Computer Security". The professor creates
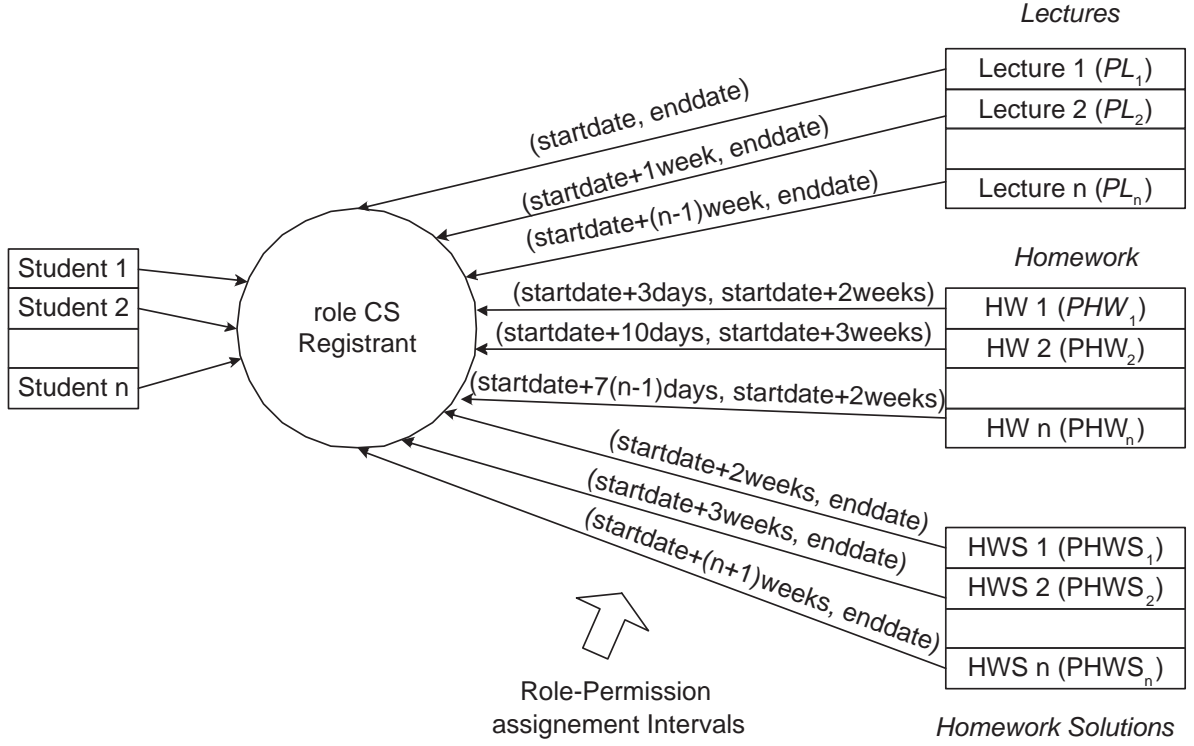
Figure 5.1. GTRBAC Online Course Example (Role-Permission Assignment).

a role "CSRegistrant" as shown in Figure 5.1, and divides the course materials to be posted online into three sections (object categories):

1. Lectures: There are n lectures – Lecture 1, Lecture 2, . . . Lecture n. The permission set related to the $i^{th}$ lecture is represented as $PL_i$.

2. Home works: There are n homework assignments (shown as HW 1, HW 2, . . . HW n) corresponding to n lectures. The permission set associated with the $i^{th}$ homework assignment is $PHW_i$.

3. Homework solutions: There are n homework solutions (shown as HWS 1, HWS 2, . . . HWS n) corresponding to the n homework assignments. The permission set associated with the $i^{th}$ homework solution is $PHWS_i$.

The course starts on the date `startdate` and ends on `enddate`. The course duration is `n+2` weeks – `n` lectures for `n` weeks and `2` remaining weeks for reviews and exams. The professor uses the following three rules to control access to the different sections:

*The rule corresponding to the lectures is:*

**Rule 1:** The $i^{th}$ lecture is made available to students on the start of the $i^{th}$ week of the course (say Monday) and is made accessible till the end of the course.

*For homework assignments, the professor uses the following rules:*

**Rule 2a:** `Three` days after the start of a weekly lecture, the corresponding homework assignment will be made accessible to the students.

**Rule 2b:** `One week` after the end of the $i^{th}$ lecture, the $i^{th}$ homework assignment will be removed (i.e., it is made inaccessible).

*The rule corresponding to the homework solutions is:*

**Rule 3:** when an homework assignment is removed as in the rule *2b* above, then the corresponding solution is posted and it is made available till the end of the course.

**Problems with Current Approach:** Figure 5.1 shows the constraint specification and how rules *1* through *3* are enforced. Let us take `Lecture 1`; as shown, `Lecture 1` is posted on *startdate* and ends in *enddate*. This is a static date assignment and needs to be changed; every semester, every course, every section, etc. Thus, making this assignment dynamic this problem can be averted. For instance, when the date is considered as a *simple event*, which is triggered from a smart calendar every semester, then there is no need of static assignment of dates. Let us take `Lecture 2 to n`; as shown, `Lecture 2` should be posted *exactly* one week from the date `Lecture 1` started and `Lecture n` should be posted exactly one week from the `Lecture n-1`. The above postings do not take into account the real life scenarios where a lecture can *span* more than one week. In this case, it will be appropriate, if the `Lecture i+1` can be started when `Lecture i` is completed.

Thus, completion of the Lecture i can trigger a simple event that can in turn start Lecture i+1.

Consider Homeworks 1 to n; all the homeworks are assigned after 3 days from the start of the weekly lecture, and they are disabled one week after the end of the corresponding lecture (see Rules *2a* and *2b*). These static assignments will also inherit the problems discussed above, and will certainly lead to cascading mix up situations. Thus, modeling the enabling of homework as an relative event (i.e., an event is triggered after a certain time of the associated event occurrence) to a `lecture event`, will allow dynamic modeling without any mix up. Similarly, homework assignments have to be associated with the homework assignments using events to avoid the problems discussed above. With event-based access control, the above enhancements will make the online course management more *effective*, and *convenient* to the professor.

*Summary:* From the above online course management example it is discernible that events are required to model real-world scenarios. The above example brought out the need for simple events (events generated from smart calendar) and complex events (a relative event that is dependent on another event).

In addition to the online course management, we discuss another example where events are required. Health care domain requires both temporal and context-aware constraints, but it also requires event-based access control. Consider a scenario where a Nurse is trying to enter the patient wards. Consider, three wards *virus*, *pregnant*, *hygiene*, where the virus ward has patients with virus, the pregnant ward has patients who are pregnant, and the hygiene ward is a place for the employees to hygienize themselves. A typical requirement would be to control the access to the wards, so that the nurses exiting virus ward do not enter (intentionally or unintentionally) the pregnant ward, without hygienizing. On the other hand, if the nurses come directly, or from the hygiene ward to the pregnant ward, they need to be granted access. In order to model these

requirements, complex events are required, and cannot be modeled and enforced using the current access control models. This typical scenario requires the detection of an non-occurrence event pattern (or NOT event described in Chapter 3), where the event pattern will be *initiated* when the nurse *enters* the virus ward and will be *terminated* when the *same* nurse enters the hygiene ward. When the nurse enters the virus ward, and then try to enter the pregnant ward, non-occurrence of the hygienizing event should be captured, and the access must be denied.

Events defined in the above two examples are based on the users, roles, objects, and operations. There are other scenarios where various event patterns have to be combined, and not just a non-occurrence event pattern. We have shown many other scenarios in the following sections, and have shown how they can be specified using event patterns. In general, current systems do not support expressive event patterns and their unrestricted combinations.

### 5.1.2 Event-Based Generalization

Temporal RBAC [20] introduced simple *role events* providing event-based constraints for periodically enabling and disabling roles using triggers. Generalized Temporal RBAC [35, 36, 37, 158, 23] extended this further: by modeling three different states (disabled, enabled, active) for roles, by adding duration constraints, time based semantics for role hierarchies, and separation of duty relations. As events are ubiquitous, event-driven [20, 23, 84, 94, 85, 38, 86] policies, verifications, and enforcements are gaining importance in RBAC.

Figure 5.2 illustrates a typical system enforcing the ANSI RBAC functional specification. Users represent the subjects and objects represent the system resources. Database shown in the figure represents the users, roles and other mappings that need to be maintained for enforcing the RBAC policies. Function or method *definitions* shown in the
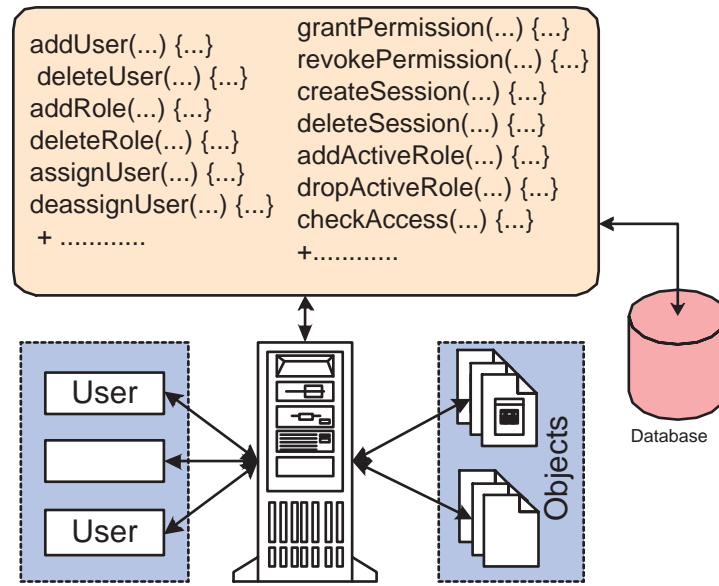
Figure 5.2. Enforcing ANSI RBAC Specification.

figure represents the NIST RBAC standard functions (refer Chapter 2 and [12]). Functions "addUser(...){...}", "addActiveRole(...){...}", and so forth, are invoked based on the user operation. For instance, when a user is trying to activate any role in the system, function "addActiveRole(...)" is invoked. The function definition is shown in Figure 5.3. This function accepts user, role and session as an input. It checks whether the information regarding user, role and session is TRUE. It also checks whether the user is assigned that role. It then adds the role to this session (i.e., activates the role). When the user does not have the permission it denies the access.

How can events be used to generalize the RBAC constraint specification in all the four components of the ANSI RBAC reference model needs to be addressed. One of the main goal is to *preserve* the ANSI RBAC functional specification without dissevering the function definitions. Constraints on events can be based on the context, temporal, or other events. Simple (or primitive) events can be defined based on the RBAC basic elements sets or the operations performed (i.e., role-dependent events) or based on the

```
addActiveRole(user, session, role) { //Core RBAC
    IF (user ∈ U && session ∈ S && role ∈ R &&
        (session ∈ userSessions(user))
        && (role ∈ sessionRoles(session))
        && (user ∈ assignedUsers(user, role)) ) {
            addSessionRole(session, role); //activate role
            return TRUE; }
    ELSE {
            raise error "Access Denied";
            return FALSE; }}
```

Figure 5.3. Add Active Role - Core RBAC.

underlying system or application or external to the system (i.e., role-independent events). Constraints on the simple role-dependent events, proposed by current systems, are inadequate in many situations as the role-dependent events need to be controlled based on the complex event patterns (or the composition of simple role-dependent or role-independent events). In addition, merely capturing the temporal history information over which the simple events occur are not sufficient as the temporal dimension across different occurrence of events (complex events) is required. In other words, temporal history over which the combinations of role-dependent, role-independent, role-interdependent events are spread across needs to be captured.

As event pattern specification assumes a domain-specific set of simple events and the rest of the complex events are built using these primitive events and a number of event operators, it fits well with the needs of applications domains. Event-based approach also allows one to expand the set of primitive events dynamically as the need arises. This approach readily permits the use of event patterns for controlling the roles, as the system states change. For example, occurrence of multiple events $E_1, E_2, \ldots, E_i$ following a particular pattern $E_p$ within a time interval $\Delta t$ can allow a particular state change $S_j$ in the system. $E_p$ represents the event pattern, and events $E_1, E_2, \ldots, E_i$ can be simple or

complex events or other event patterns. Ordering of events that occur over a period of time $T$ can be specified using event patterns, which combine various constituent events using logical, temporal, and other relationships or operators. Thus, all operations in a RBAC system can be constrained by complex yet meaningful event pattern constraints. Extending RBAC with event pattern constraints will provide a dynamic and generalized way of providing constraints. Furthermore, it will allow applications to model real life situations and will cater a larger class of applications.

Invocations of the function definitions shown in Figure 5.4, can be captured as simple events. Thus, the generalization of RBAC with event pattern policies can be achieved by introducing an abstraction between the function invocations and the function definitions, as shown in Figure 5.4. Event pattern policies are specified on these role-dependent operations. Thus, whenever a function is invoked it triggers an event which in turn is checked for event ordering based on the event pattern constraint. Once the event pattern constraint is met original RBAC definitions are invoked using the function definitions "_addUser(...)...", "_addActiveRole(...)...", and so on. Thus, event pattern constraints act on top of the original RBAC constraints. This abstraction also allows the support of all the four components of the ANSI RBAC in a seamless manner, as function definitions from any of the component can be plugged-in as needed. On the other hand, event pattern policies can also contain role-independent operations as constituent events. For example, when an event corresponding to the role activation is triggered, other constraints based on context, time, etc can be checked before invoking the function "_addUser(...)..." to check basic RBAC constraints.

**Primary Goals:** Based on our discussions the primary goals for our work are:

- To incorporate RBAC into the existing event framework.
- To *preserve* the ANSI RBAC functional specification without dissevering the function definitions.
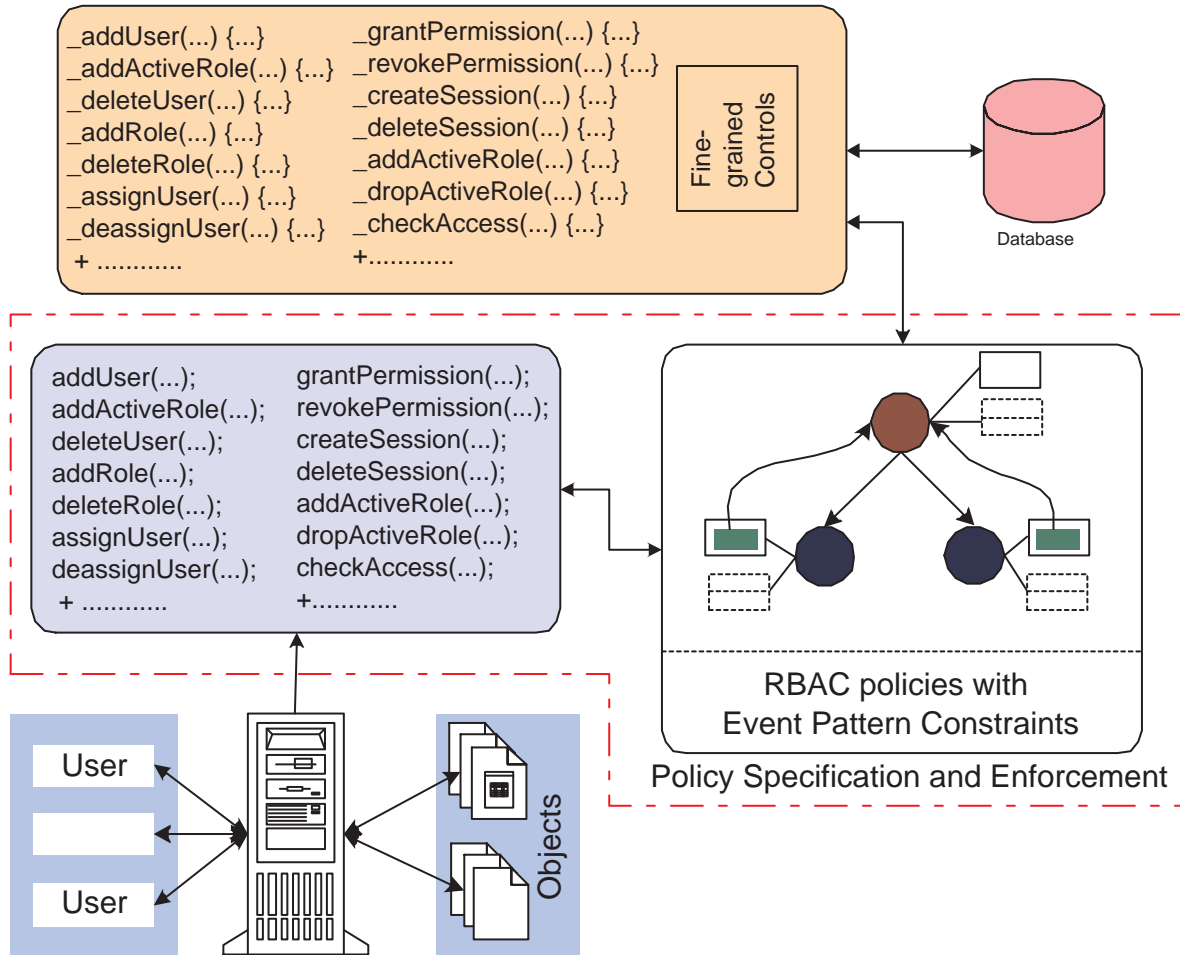
Figure 5.4. RBAC with Event Pattern Constraints Overview.

- To generalize the existing event framework so that it can be used for a larger class of applications including RBAC.

- To generalize RBAC with the generalization of event framework.

In this chapter we will discuss the generalization and extension of RBAC with event pattern policies. In the rest of the chapter we present expressive event pattern constraints for RBAC. First, we explain simple or basic events, complex events, event attributes, event operators and event types in Snoop. We discuss the limitations of the current event specification and then generalize it. We then discuss events in the context of RBAC and show the adequacy of events for capturing RBAC policies and extend

them with event pattern constraints. We also show how event pattern constraints are specified for all RBAC operations. Specification of constraint alone are *not* sufficient as security mechanisms to enforce them are equally important. The main advantage of our event-based approach is due to the enforcement techniques. With the Local Event Detector (LED), once event pattern policies are specified using Snoop language they can be directly enforced in the underlying system. In other words, in our approach specification and enforcement are not orthogonal, as opposed to the current approaches. We discuss the enforcement of these generalized event pattern policies in Chapter 6.

## 5.2   Event Specification Generalization

Events have been employed successfully in various application domains for situation monitoring. Snoop and SnoopIB discussed in Chapter 3 provide well-defined point-based and interval-based event semantics, respectively. Although both event semantics can be used for event patterns, they are inadequate for supporting many newer application domains including RBAC. First, we discuss why current event patterns are insufficient using different application domains, and then show how we generalize the current event specification. We then show how the generalized event specification is used in generalizing the ANSI RBAC with event-based constraints.

### 5.2.1   Existing Event Definitions

Events[2] are organized into a hierarchy of event classes for providing a common structure and behavior. Each event occurrence of an event class is the instance of that particular event. Events are classified as *simple* and *complex*, where the simple events are domain-specific and complex events are compositions of simple and complex events using predefined event operators. First, we define the set of simple events that act as the basic

---

[2]We give a brief overview of the existing event specification in this section. For a more detailed discussion please refer to Chapter 3.

building block for event patterns. We also explain event attributes, types, properties, definitions, and representations. Second, we discuss the set of event operators that are used for constructing constraints, and introduce the BNF for the constraint specification. Finally, we discuss the semantics of both the simple events and complex event operators.

**Definition 2** (Event). *An event is defined as any* occurrence of interest *in an application, system, or environment. An event $E$ occurs over a time interval $[t_1, t_2]$ and is denoted by $O(E[t_1, t_2])$, where $t_1$ is the start time of the event denoted by $\uparrow E$, $t_2$ is the end time of the event denoted by $E \downarrow$ and "O" represents the interval-based event semantics*[3].

Start and end time of an event is formally defined as:

$$Start\ time : O(\uparrow E, t) \triangleq \exists t \leq t'(O(E, [t, t']))$$

$$End\ time : O(E \downarrow, t') \triangleq \exists t \leq t'(O(E, [t, t']))$$

### 5.2.1.1   Event Attributes, Types and Properties

Each event has a well-defined set of *attributes* that provide the necessary information about that event. These attributes are based on *implicit* and *explicit* parameters. *Implicit* parameters are optional and, contain system and user defined attributes, such as: event name, time of occurrence ($t_{occ}$), the object instance that raised the event if it is an object-oriented model. For example, implicit parameters for any user operation in the system employing RBAC can be user, session id, and $t_{occ}$. On the other hand, *explicit* parameters are collected from the event itself and values for these parameters are assigned when it is raised. For example, explicit parameters for a function invocation are the formal parameters of that function itself. In this work we represent event

---

[3]Even though events occur over an interval they can be detected using either detection-based or interval-based semantics (Chapter 3)

attributes as shown below where $A_{il}$ and $A_{xm}$ represent optional implicit and available explicit parameters, respectively.

$$< [A_{i1}, A_{i2}, \ldots, A_{il}], (A_{x1}, A_{x2}, \ldots, A_{xm}) >$$

For any occurrence interest, events can be defined over different granularities. We term these as *event types* as they act as the event schema addressing the specification of attributes. For instance, in relational databases, event types can be based on:

- insert operation on any relation, and
- insert operation on a particular relation.

In addition, each event has a unique name, which can also act as an event type. An event type can occur more than once, for example, insertion of two tuples into a relation with separate insertion calls will trigger the same event type twice. Thus, event occurrences are assigned an identifier *eid* to identify them uniquely. We use an event type and an event instance interchangeable in our discussions, as the distinction is usually clear from the context.

Events have *structure* and *behavior* and can be either *definite* or *non-definite*. Structure of an event is similar to the structure of an object in object-oriented programming. For example, when a binary event is defined, it requires two constituent events. Similarly, behavior of an event is same as that of an object. For instance, in a binary event, behavior defines when the event will be raised based on its constituent events. Both these properties are required when defining an event. In the context of RBAC, binary events allow the specification of constraints with two simple or complex events. Events that are bound to occur are termed as *definite* events (e.g., time-based) while others are termed as *non-definite* events.

*Occurrence* of an event indicates the happening of the occurrence of interest, initialization, and collection of attribute values. *Detection* of an event indicates the satisfaction

of conditions, event propagation, and rule triggering. For instance, when a binary event is detected it represents the satisfaction of the constraints enforced by the complex event pattern.

### 5.2.1.2 Simple or Primitive Events

Domain-specific, predefined events are known as simple or primitive events. A simple event is detected atomically at a point on the time line. Simple events are distinguished as domain specific, temporal and explicit events. Some examples of simple events are:

1. *File operations* (e.g., file open) in operating systems.

2. *Method invocations* by objects in object oriented systems.

3. *Data manipulation* operations (e.g., insert) in databases.

4. *System clock* of the underlying system (i.e., absolute or relative temporal events).

5. *External* events (i.e., based on the data from sensors).

**Definition 3** (Simple Event). *A simple event **E** occurs over an interval $[t, t']$, where t is the start time and t' is the end time of an event, and is expressed as $O(E[t, t'])$. For event **E**, t is the same as t', as a simple event is detected atomically at a point on the time line. A simple event is defined as:*

$$O(E[t, t']) \triangleq \exists t = t'(O(E, [t, t']))$$

In accordance with the object-oriented paradigm, existing event specifications allow two kind of primitive event definitions, based on the method invocations (or function calls). Simple events defined on the method invocations are shown below:

$$\texttt{Event } E_{S1} \; = \; \mathcal{F}(A_{f1}, \; A_{f2}, \; \ldots, A_{fr});$$

$$\texttt{Event } E_{S2} \; = \; \mathcal{U} \rightarrow \mathcal{F}(A_{f1}, \; A_{f2}, \; \ldots, A_{fr});$$

- Event $E_{S1}$ is detected when the function $\mathcal{F}$ is invoked by any object.
- $E_{S2}$ is detected when a function $\mathcal{F}$ is invoked by an object instance $\mathcal{U}$.

$(A_{f1}, A_{f2}, \ldots, A_{fr})$ are the formal parameters and $\mathcal{U}$ is the object instance that is invoking the function. In the above, the formal parameters are explicit parameters, and the object instance is an implicit parameter.

### 5.2.1.3   Event Patterns

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. Let us consider a scenario, where an activity `d` is allowed after several other activities `a`, `b`, and `c` happen in a certain *order* forming a `pattern`. This can be formulated using a complex event `Eop((Eop(a, b, c)), d)`, where `Eop` represents event operators, and activities `a` through `d` can be simple or complex events. This entire expression can be considered as a event pattern. A number of event operators [93, 39, 40] have been proposed in the literature based on the requirements of several application domains that were considered. Most of these operators are discussed in Chapter 3. Below we provide the Backus Naur Form (BNF) [70] for the event pattern specification. This can be extended whenever a new event operator is added. This includes the event operators and constituent events, but not the composition conditions. In other words, compo-

sition conditions can be based on the point-based semantics, interval-based semantics, or the generalized semantics discussed later in this Chapter. The BNF provided is left associative.

$$
\begin{aligned}
\texttt{E1} ::=&\quad \texttt{E1}\ \Delta\ \texttt{E2}\ |\ \texttt{E2}\\
\texttt{E2} ::=&\quad \texttt{E2}\ \nabla\ \texttt{E3}\ |\ \texttt{E3}\\
\texttt{E3} ::=&\quad \texttt{E3} \gg \texttt{E4}\ |\ \texttt{E4}\\
\texttt{E4} ::=&\quad \text{ANY}(\texttt{E3},\ \texttt{VALUE})\ |\ \texttt{E5}\\
\texttt{E5} ::=&\quad \texttt{E5, E6}\ |\ \texttt{E6}\\
\texttt{E6} ::=&\quad \text{NOT}(\texttt{E1, E1, E1})\\
&\quad |\ \text{A}(\texttt{E1, E1, E1})\\
&\quad |\ \text{A*}(\texttt{E1, E1, E1})\\
&\quad |\ \text{P}(\texttt{E1},\ [\texttt{ts}],\ \texttt{E1})\\
&\quad |\ \text{P}(\texttt{E1},\ [\texttt{ts}]{:}p,\ \texttt{E1})\\
&\quad |\ \text{P*}(\texttt{E1},\ [\texttt{ts}],\ \texttt{E1})\\
&\quad |\ \text{P*}(\texttt{E1},\ [\texttt{ts}]{:}p,\ \texttt{E1})\\
&\quad |\ \text{PLUS}(\texttt{E1},\ [\texttt{ts}])\\
&\quad |\ [\texttt{ts}]\\
&\quad |\ \textsc{External Events}\\
&\quad |\ \text{EN:}(\texttt{E1})\\
&\quad |\ (\texttt{E1})\\
\texttt{VALUE} ::=&\quad \text{integer}\ |\ \infty
\end{aligned}
$$

In the BNF `E1` represents an event and `EN` represent the event name. These names can be used while specifying the constituent events of the complex event. Without event names it is impossible to differentiate an event occurrence corresponding to a constituent

event and for accessing its attributes. `ts` represents the time string (an absolute time point), and `VALUE` represents the number of events that are needed for detecting an ANY event.

### 5.2.2 Advantages and Limitations of Event Specification

Below we discuss both the advantages and limitations of the event specification based on newer application domains.

**Advantages**

- Snoop and SnoopIB event specification languages allow the specification of complex event-based policies. On the other hand, LED converts event specifications into event detection graphs (detailed in Chapter 6) and detects events based on the event specification. In other words, event specification and detection are not disjoint in our framework. Thus, when both are used for RBAC, both policy specification and enforcement can be done in a combined manner.

- Snoop supports various complex event operators that were derived from diverse domains. On the other hand, new event operators have to be added when a new kind of event pattern or complex event needs to be captured. Snoop is extensible so that new operators can be plugged-in as and when needed.

- Current event operator semantics is purely based on timestamps. With newer domains requiring other kind of semantics, current event operator semantics have to be changed. With Snoop and SnoopIB event specification we can plug-in the new semantics, seamlessly.

- Once simple events are identified, complex events are composed from simple events via operators. Similarly, with conditions and actions are separated from the events themselves. This abstraction allows the preservation of the ANSI RBAC functional specification without dissevering the function definitions.

**Limitations**

- With existing event semantics, simple events are captured and detected when an occurrence of interest happens. Conditions associated with the event are checked and the corresponding actions are taken. On the other hand, with security domain, events and conditions have to treated together and not as separate entities. For example, consider the role activation as an occurrence of interest. In this case, event has to be raised when someone is trying to activate the role, conditions have to be checked whether the user can be activated, and then the event has to be detected and the user should be activated.

- As discussed above, in accordance with the object-oriented paradigm, Snoop allows only two kinds of simple events. One event based on the instance invoking the method, and the other based on the class. This is a limitation, as events can be defined based on many other aspects, such as a condition on the attribute. Thus, primitive event definition has to be generalized to include other parameters as well.

- Similar to the above, with previous work, including our work in Chapter 3, event semantics were based solely on $t_{occ}$ (i.e., time-based event semantics), and does not include attributes, predicates, or their combinations. In other words, conditions for compositions in complex events are based on the time of occurrence of constituent events. For example, a sequence event is detected when the first event happens before the second event. With newer domains this is necessary but not sufficient. For example, a sequence event can be detected only when the first event happens before the second event, and the user attribute of both events are same (i.e., tracking of two events by the same user). Thus, current event operators have to generalized, as they cannot be used directly in modeling access control policies.

- In security domain, capturing violations are critical. Currently, only the events that follow a particular order are considered for event detection. For example,

with a Sequence operator there are two constituent events where the first event should occur ahead of the second event. What happens when a second event occurs without the first event's occurrence?. Currently, the second event is dropped and it is a limitation as this occurrence of the second event without the first event can be a violation of a security policy. In other words, partial and failed events have to be captured.

### 5.2.3    Generalized Simple Events

Currently, simple events are captured/raised when the occurrence of interest happens, and are detected. Corresponding rules are executed and complex events are notified of its occurrence. As discussed previously, there can be situations where events have to be detected only when a condition is satisfied. Let us consider two events, ES1 and ES2 defined on the function "setPrice(price);".

Event ES1 = setPrice(price);

Event ES2 = setPrice(stockId = GOOG, price);

Event ES1 is a class-level event and is raised whenever the method "setPrice" is invoked. Event ES2 is an instance-level event and is raised whenever the method "setPrice" is invoked by an object "GOOG". On the other hand, the following occurrence of interests cannot be specified with the existing simple event definitions:

1) `Price > $100` (explicit parameter)

2) `stockId = GOOG` AND `Price > $500` (implicit and explicit parameter)

3) Events that occur only *after* `18.00` hrs everyday (implicit parameter)

Thus, in order to handle these kind of requirements, Snoop simple events have to be generalized. With simple events, event attributes can be categorized into implicit and explicit parameters as discussed in Section 5.2.1.1. With method invocations, all the formal arguments or parameters can be treated as *explicit parameters* of that method.

Similarly, objects instances that invoke the method, time of invocation, and so on can be considered as the *implicit parameters* of that event. In general, only these two type of parameters can be associated with any simple or complex event. On the other hand, when events need to be detected based on some conditions, only the parameters associated with that event can be checked for those conditions. Thus, simple events can be generalized with expressions based on these two kinds of parameters. On the other hand, there can be cases where arbitrary conditions can be checked that are not based on the event parameters, and we do not discuss those in this work. Thus, we have generalized simple events with two kind of expressions; *implicit expressions* ($\mathcal{I}_{expr}$) and *explicit expressions* ($\mathcal{E}_{expr}$) based on the implicit and explicit parameters, respectively.

$$Event\ EventName = (FunctionName(objInstance, A_{x1},\ A_{x2},\ \ldots, A_{xr}), (\mathcal{I}_{expr} \wedge \mathcal{E}_{expr}));$$

In the above, *EventName* corresponds to the name of the event, *FunctionName* corresponds to the name of the function, $(A_{x1}, A_{x2}, \ldots, A_{xr})$ correspond to the formal arguments or explicit parameters, $\mathcal{I}_{expr}$ represents the implicit expression, and $\mathcal{E}_{expr}$ represents the explicit expression. These expressions evaluate to either TRUE or FALSE. As an example, consider

$$\mathcal{E}_{expr} \leftarrow (A_{x1} = ``Nurse")$$

The $\mathcal{E}_{expr}$ expression returns TRUE when the value of attribute $A_{x1}$ is equal to "Nurse".

Below we show four types of events based on both $\mathcal{I}_{expr}$ and $\mathcal{E}_{expr}$ expressions.

```
Event E_P1  =  F(A_x1, A_x2, ..., A_xr); /* I_expr = E_expr = ∅ */
```
$$\texttt{Event } E_{P1}\ =\ \mathcal{F}(A_{x1},\ A_{x2},\ \ldots, A_{xr}); / * \mathcal{I}_{expr} = \mathcal{E}_{expr} = \emptyset * /$$

$$\texttt{Event } E_{P2}\ =\ \mathcal{U} \rightarrow \mathcal{F}(A_{x1},\ A_{x2},\ \ldots, A_{xr}); / * \mathcal{I}_{expr};\ and\ \mathcal{E}_{expr} = \emptyset * /$$

$$\texttt{Event } E_{P3}\ =\ (\mathcal{F}(A_{x1},\ A_{x2},\ \ldots, A_{xr}) : \texttt{E}); / * \mathcal{I}_{expr} = \emptyset\ and\ \mathcal{E}_{expr}; * /$$

$$\texttt{Event } E_{P4}\ =\ (\mathcal{U} \rightarrow \mathcal{F}(A_{x1},\ A_{x2},\ \ldots, A_{xr}) : \texttt{E}); / * \mathcal{I}_{expr};\ and\ \mathcal{E}_{expr}; * /$$

Event $E_{P1}$ and $E_{P2}$ are same as our existing event definition. As shown event $E_{P1}$ is detected when the function $\mathcal{F}$ is invoked by any object with both $\mathcal{I}_{expr} = \emptyset$ and $\mathcal{E}_{expr} = \emptyset$. Similarly event $E_{P2}$ is detected when the function $\mathcal{F}$ is invoked by object $\mathcal{U}$ (i.e., $\mathcal{I}_{expr} = \mathcal{U}$) and $\mathcal{E}_{expr} = \emptyset$. In other words, event $E_{P2}$ is raised when the $\mathcal{I}_{expr}$ expression evaluates to TRUE. Thus, both $E_{P1}$ and $E_{P2}$ are specific cases of our generalization. On the other hand event events $E_{P3}$ and $E_{P4}$ are simple events that cannot be specified with existing event definition. Event $E_{P3}$ is detected when the method is invoked by any object and satisfies the $\mathcal{E}_{expr}$ expression as $\mathcal{I}_{expr} = \emptyset$. The event is raised when the method is invoked and the condition specified by $\mathcal{E}_{expr}$ evaluates to TRUE. Similar to the above, event $E_{P4}$ is raised when both $\mathcal{I}_{expr}$ and $\mathcal{E}_{expr}$ expressions return TRUE.

For the previously discussed occurrence of interests, we define simple events based on the above generalization:

1) Event ES3 = setPrice(price);

2) Event ES4 = (setPrice(price), (price > 100)); /* $\mathcal{E}_{expr}$ */

3) Event ES5 = (setPrice(price), (t_occ > 18.00)); /* $\mathcal{I}_{expr}$ */

4) Event ES6 = (setPrice(price), (stockId = GOOG)); /* $\mathcal{I}_{expr}$ */

5) Event ES7 = (setPrice(price), ((stockId = GOOG) $\wedge$ (price > 500))); /* $\mathcal{I}_{expr}$ and $\mathcal{E}_{expr}$ */

Although the effect of simple event generalization can be achieved by moving both $\mathcal{I}_{expr}$ and $\mathcal{E}_{expr}$ to the `rules` it will be inefficient - due to unnecessary rule processing as events are raised and filtered at the rules, and cannot be used as part of the event patterns.

### 5.2.4 Generalized Event Patterns

**Definition 4** (Event Pattern). *A complex event $E$ occurs over an interval $[t, t']$, where $t$ is the start time of* initiator, *and $t'$ is the end time of* detector.

User activities over a period of time can be characterized as event patterns and is captured using complex events.

Similar to the primitive event definition generalization, we have generalized the current event operator definitions. Typically, an event operator defines how the complex event patterns need to be detected based on incremental composition of other complex or simple event patterns that occur over a period of time. In other words, complex event operators collect all the occurrences of constituent events and detect an event if they follow the required pattern or behavior. In general complex event consists of event operator, constituent events, and *composition conditions*. Existing event operators compose constituent events using timestamps including SnoopIB detailed in Chapter 3. On the other hand, composition conditions can be generalized to be based on the event attributes (i.e., implicit and explicit parameters). Thus, current event operator semantics with just timestamp comparison act as a special case of the generalization with $\mathcal{I}_{expr}$ and $\mathcal{E}_{expr}$ expressions. Below we show the generalized way of specifying complex events, that include event attributes.

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

- *Eop* represents an n-ary event operator (e.g., unary, binary and so forth)
- $(E_1, \ldots E_n)$ represents all the simple constituent events of the complex event. The constituent event that starts the event detection is termed as the `initiator`. The constituent event that detects and raises the complex event is termed as the `detector`. The constituent event that terminates the complex event is termed as `terminator`. Thus, the event that acts as the detector raises the event.
- $[t_s, t_e]$ represents the start and end time of the complex event where $t_s$ is the start time of the first constituent event and $t_e$ is the end time of the last constituent event.

- $\mathcal{I}_{expr}$ represents condition expression based on the implicit parameters of the constituent events and returns a Boolean value. When we represent the current operator semantics, all the timestamp comparison is a part of the $\mathcal{I}_{expr}$. In other words, interval-based semantics (SnoopIB) introduced in Chapter 3 will be $\in \mathcal{I}_{expr}$. Similarly, point-based semantics (Snoop) will also be $\in \mathcal{I}_{expr}$. For example, for a binary event operator with events $E_1$ and $E_2$, $\mathcal{I}_{expr} = t_{occ}(E_1) \; \theta \; t_{occ}(E_2)$, where $t_{occ}$ represents the timestamp of event occurrence, and $\theta$ can be any operator $<, >, \leq, \geq, =, \neq, \in, \ldots$.

- Similar to the simple event, $\mathcal{E}_{expr}$ represents condition expression based on the explicit parameters of the constituent events and returns a Boolean. For example, for a binary event operator with the events $E_1$ and $E_2$, $\mathcal{E}_{expr} = E_1(A_{xi}) \; \theta \; E_2(A_{xj})$, where attributes $E_1(A_{xi})$ and $E_2(A_{xj})$ have values from the same domain.

- complex event $Eop$ is detected iff all the above mentioned expressions return TRUE. These expressions can be empty and return TRUE always. We assume that all these expressions must not be empty at the same time, as it will detect the complex event always.

We will explain the binary event operator *Sequence* ($\gg$), which captures the sequential occurrence of constituent events. In other words, it is raised only when the second constituent events happens after the first constituent event. Sequence event is represented as "$O(E_1 \gg E_2, [t_1, t_2])$" and *occurs* when event $E_1$ occurs[4] before event $E_2$ (i.e., $((E_1 \downarrow) < (\uparrow E_2)))$, and $t_1 = \uparrow E_1$ and $t_2 = E_2 \downarrow$. This event is *detected* when event $E_2$ occurs, which is the detector. In the above, $(\mathcal{I}_{expr} = t_{occ}(\uparrow E_1) < t_{occ}(E_2 \downarrow)) \wedge (\mathcal{E}_{expr} = \emptyset)$.

Below we explain the event operators intuitively except OR and AND where we show how current timestamp based semantics is formulated in the unrestricted context

---

[4]Even though events are represented as $E_i$ in all the operators, they can be either simple or complex.

using the generalization. In all the operators $\mathcal{E}_{expr}$ is empty. We will explain the expressive event pattern constraint specification using these operators in the following sections.

- **OR** $O(E_1 \nabla E_2, [t_1, t_2])$**:** It is a disjunction operator and is detected when *any one* of the two events $E_1$ and $E_2$ occur, and $\mathcal{I}_{expr} = \emptyset$.

- **AND** $O(E_1 \Delta E_2, [t_1, t_2])$**:** It is detected when *both* events $E_1$ and $E_2$ occur. Order of occurrence between $E_1$ and $E_2$ is not considered, and ($\mathcal{I}_{expr} = (t_{occ}(E_2 \downarrow) < t_{occ}(\uparrow E_1)) \vee (t_{occ}(E_1 \downarrow) < t_{occ}(\uparrow E_2)))$.

- **NOT** $O(\neg(E_2)(E_1 \gg E_3), [t_1, t_2])$**:** Non-occurrence of an event $E_2$ in between two other events $E_1$ and $E_3$ triggers the NOT event. $\mathcal{I}_{expr}$ includes the interval- and point-based semantics.

- **Plus** $O(Plus(E_1, \Delta), [t_2, t_2])$**:** It is a relative temporal event. A simple or composite event occurrence starts the Plus event (i.e., at time "$T$"). After the specified time interval or duration "$\Delta$" (i.e., at time "$T + \Delta$") the Plus event is detected.

- **Aperiodic** $O(A(E_1, E_2, E_3), [t_1, t_2])$**:** It is detected whenever event $E_2$ occurs between two other events $E_1$ and $E_3$. Event $E_1$ starts the Aperiodic event and $E_3$ terminates the same. Event occurrences of $E_2$ cannot detect an Aperiodic event before the occurrence of event $E_1$ or after $E_3$. Only when event $E_2$ occurs within $E_1$ and $E_3$, an Aperiodic event is detected and the corresponding rules are triggered.

- **Any(**$m$**, (**$E_1$**, ..., **$E_n$**))** It is detected when $m$ number of events are detected, where $0 < m \leq n$ and $n$ corresponds to the number of distinct events specified. In this event operator order of occurrence between events is not considered.

- **Periodic** $O(P(E_1, \tau, E_3), [t_2, t_2])$**:** This event is similar to Aperiodic event except that it is detected at a regular time interval "$\tau$" between two other events $E_1$ and $E_3$.

- **Cumulative Aperiodic** $O(A*(E_1, E_2, E_3), [t_1, t_2])$: Event $E_1$ starts and event $E_3$ terminates and detects the A* event. Event occurrences of $E_2$ between $E_1$ and $E_3$ are accumulated.

- **Cumulative Periodic** $O(P*(E_1, \tau, E_3), [t_2, t_2])$: This event is similar to A* event except that event $E_2$ is a time string specifying periodicity.

As opposed to the alternate simple event generalization, the effect of complex event generalization cannot be achieved by moving both $\mathcal{I}_{expr}$ and $\mathcal{E}_{expr}$ to rules. This is because composite events will be detected incorrectly if $\mathcal{I}_{expr}$ or $\mathcal{E}_{expr}$ are moved to rules. In other words, expressions associated with composite events cannot be moved to rules.

### 5.2.5 Complete, Uncomplete and Failed Events

Complex events occur over an interval and are detected over an interval. Start of the interval is the start time of the first event (i.e., initiator) and end of the interval is the end time of the last (i.e., detector) constituent event. For the event $E_{SEQ}$ defined before, start time $t$ is the time of detection of a and end time $t'$ is the time of detection of b as both these events were assumed as simple events. On the other hand, when we consider a as a complex event then $t$ will be $\uparrow$ a. Thus, the time of *occurrence* and *detection* for a complex event is an interval formed by $[t, t']$.

As complex events combine more than one event, they are detected only when the event `completes`. We categorize these events as *complete* events. Actions corresponding to an event can be performed iff that event is complete. In other words, current event detection semantics are based on the *If-Then* paradigm. Even though event completion is necessary in many situations it is not required in all the domains including the access control domain.

Let us consider a policy in RBAC, where role operation `b` has to be allowed after operation `a` has been performed. This particular policy can be formulated as a event pattern shown below using the $\gg$ operator;

$$\text{Event } E_{SEQ} = (\gg (\text{a}, \text{b}));$$

In the above we assume that `a` and `b` are simple events and are already defined. Thus, when `b` occurs after `a`, the $\gg$ event is raised and the corresponding action is taken (i.e., allow operation `b`). Consider the event $E_{SEQ}$; when event `b` occurs (request for performing the activity) and event `a` has not occurred (activity has not been performed), access request operation `b` should be declined and user should be notified that he cannot perform that operation. This requires an additional capability of current event detection semantics paradigm to infer that a composite event (always the detector) of a complex event has been detected, but not other events to complete the detection of the complex event. To identify such occurrences the *If-Then-Else* mechanism is proposed (because the *If-Then* mechanism only detects complete events and ignores the others). Thus, the extension allows for additional actions to be taken when the detector occurs and the event is not completed because of the non-occurrence of other constituent events. We term these events as `uncomplete` events. Similar to complex events, simple events can also be complete or uncomplete. For instance, when the predicate value does not satisfy $\mathcal{E}_{expr}$ expression it is considered as an uncomplete event.

In addition to complete and uncomplete events there can be other type of events. Consider a NOT event operator. When the second event does not occur between the first and third event, an NOT event is detected. When the first event and third event occur without the middle event it is a complete event. When the second event and third event occur without the first event or when the third event occurs without the first event then they are considered as uncomplete events. On the other hand, when the first event,

second event and third event occur, a NOT event is not detected and is categorized as *failed* event.

In general, complete, uncomplete, and failed events are defined as:

"A complete event E occurs when, i) an initiator initiates that event, ii) all the constituent events occur, and iii) a detector occurs and completes that event."

"An uncomplete event E occurs when i) event E is not initiated, ii) other constituent events can occur, and iii) a detector occurs."

"A failed event E occurs when i) an initiator initiates that event, ii) other constituent events occur, and iii) a detector occurs and completes that event, but the event fails because some constituent event has occurred."

### 5.2.6 Complete, Uncomplete and Failed Rules

An event (simple or complex) occurrence can trigger multiple rules, nested/cascaded rules, prioritized rules, and causality rules. In this section we have introduced rules that are associated with *complete*, *uncomplete* as well as *failed* events. Event occurrences can raise complete, uncomplete, and failed events and thereby rules associated with them. With complex events, uncomplete rules are raised when the *detector* event occurs without the occurrence of the *initiator* event. On the other hand, all the three kinds of rules are specified in the same way. Active authorization rules consist of five components (refer Chapter 4 for more details) and they are:

- $\mathcal{R}_{name} : Type$ – Name of the rule and Type corresponds to Complete, Uncomplete and Failed.

- "ON" an event $\mathcal{E}_i$ that triggers this rule.

- "WHEN" checks the conditions $< \mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_n >$ when an associated event is triggered.

- "THEN" triggers a set of actions $< \mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_n>$ when the conditions evaluate to TRUE.

- "ELSE" triggers a set of alternative actions $< \mathcal{A}\mathcal{A}_1, \mathcal{A}\mathcal{A}_2, \ldots \mathcal{A}\mathcal{A}_n>$ when the conditions evaluate to FALSE.

Consider complete rule `R:C` and uncomplete rule `R':U` that are associated with previously defined event $E_{SEQ}$ for checking required policies and taking appropriate actions. When event `a` occurs followed by the occurrence of event `b`, then complete event $E_{SEQ}$ is detected and rule `R:C` is triggered. Once the rule is fired it checks the required policies (encoded as part of conditions and actions) and it either allows the user to perform the operation or denies the same. In the case of uncomplete event $E_{SEQ}$, rule `R':U` is triggered and appropriate actions are triggered.

## 5.3   Simple Events in RBAC

The objectives of identifying simple events in the context of RBAC are:

1. to show the adequacy of events to represent RBAC occurrence of interests and to model RBAC policies.

2. to enhance and extend RBAC policies with expressive event patter constraints.

3. to demonstrate the versatility of the event-based approach as both a specification and enforcement mechanism.

In our approach, any user operation can be designated as an event. However, in the context of RBAC, a subset of them can be designated as events if that is meaningful. Event types in RBAC are based on: i) subject or user; ii) role; and iii) system. These elements form various event types that are based on both role and non-role operations. For example, event types can be based on: 1) specific operation performed on a role, or 2) specific operation requested by a user.

Some examples of simple events in the context of RBAC are:

1. *User operations* (e.g., role activation) in RBAC.

2. *Role operations* (e.g., role enabling) in RBAC.

3. *System clock* of the underlying system (i.e., absolute or relative temporal events).

4. *External* events (i.e., based on the data from sensors).

With RBAC, operations are performed by users in the system by invoking functions (internally). As simple events are the basic building blocks of any event-driven system, it should be able to represent all the occurrence of interest associated with the application domain (e.g., RBAC). In any RBAC system a *subject* is allowed to perform an *operation* on an *object* when it has the required *permission*. Typically, every subject has a set of attributes that uniquely identifies the subject at various instances. Permissions are granted to subjects only when they possess the *required* values for these set of attributes. Apart from the attribute values possessed by subjects, the access control system also uses additional values from the underlying system or environment that are related to the subject for access checking, if defined as part of an access control policy. Thus, considering $\mathcal{R_A}$ as the set of attribute conditions that are required to be satisfied for granting permission, $\mathcal{R_{DA}}$ as the set of attribute values possessed by the subject at the time of access request, and $\mathcal{R_{IA}}$ as the set of attribute values from the environment related to the subject, we can define $\mathcal{R_A}$ as:

$$\mathcal{R_A} \triangleq \mathcal{R_{DA}} \bigcup \mathcal{R_{IA}}$$

Occurrence of interest in RBAC can be either based on the *user or role operations* (e.g., enabling, disabling, assignments, de-assignments, activations, and deactivations) or based on the underlying *system* (e.g., system clock) or *environment* (e.g., location). Both these can be mapped and defined as simple event types, where the former occurrence of interests that are based on role operations are defined as role-dependent events ($\mathcal{E_{RD}}$) and the latter as role-independent events ($\mathcal{E_{RI}}$). For example, whenever a subject *initiates*

a role operation, a simple $\mathcal{E}_{\mathcal{RD}}$ is raised. Similarly, whenever a predefined occurrence of interest based on the system clock happens, a simple $\mathcal{E}_{\mathcal{RI}}$ is raised. As explained before, implicit and explicit parameter vales $A_{il}$ and $A_{xm}$ associated with these events $\mathcal{E}_{\mathcal{RD}}$ and $\mathcal{E}_{\mathcal{RI}}$ should supply the required values for $\mathcal{R}_{\mathcal{A}}$.

In the context of RBAC, simple events ($\mathcal{E}_{\mathcal{RD}}$) can be divided into four different categories or types based on the set of users (subjects) $\mathcal{U}$ and roles $\mathcal{R}$. All these events are specialized cases of the generalized simple event definition.

1. $\forall u \in \mathcal{U}$ and $\forall r \in \mathcal{R}$ /* $\mathcal{I}_{expr} = \mathcal{E}_{expr} = \emptyset$ */

2. $\exists u \in \mathcal{U}$ and $\forall r \in \mathcal{R}$ /* $\mathcal{I}_{expr}$; and $\mathcal{E}_{expr} = \emptyset$ */

3. $\forall u \in \mathcal{U}$ and $\exists r \in \mathcal{R}$ /* $\mathcal{I}_{expr} = \emptyset$; and $\mathcal{E}_{expr}$; */

4. $\exists u \in \mathcal{U}$ and $\exists r \in \mathcal{R}$ /* $\mathcal{I}_{expr}$; and $\mathcal{E}_{expr}$; */

The above event types can be effectively used for specifying common and tailor-made constraints and other operations. With the first type, events are raised irrespective of the *user* or *role* (i.e., $\mathcal{I}_{expr}$ and $\mathcal{E}_{expr}$ expressions are empty). Let us consider the role activation operation for defining a simple event. Whenever a user is trying to activate a *role* in a *session*, the system invokes the function *addActiveRole*(). Formal parameters of the function are: *user*, *session* and *role*. As event $E_{AAR}$ defined below is based on the function that will be invoked irrespective of the user, it is raised when any user tries to activate any role. This event is similar to event $E_{S1}$.

`Event` $E_{AAR} =$ addActiveRole(user, session, role);

In the second category, subject-level events are defined on operations that are performed by a *specific* subject irrespective of his/her role. When a subject performs some operation in the system these events are triggered allowing the specification of user-based constraints. For example, an event can be defined for user `Tom` when he performs role activation. Thus, specific constraints along with common constraints can

be placed when `Tom` activates a role. The event corresponding to this is defined below. This event is similar to $E_{S2}$.

`Event` $E_{TomAAR}$ = (addActiveRole(user, session, role),(userId = "Tom"));

Simple events based on the final two categories are *specific* to a role. Role-level events are based on a specific role and are triggered when *any* or a *specific* subject associated with that role performs the operation. Events in this category allows the specification of role-based constraints to all users or to a specific user. Below, we define two events $E_{AARM}$ and $E_{TomAARM}$ corresponding to users trying to activate role "Manager" and for `Tom`'s activation of role `Manager`. These events correspond to the event types $E_{S3}$ and $E_{S4}$ described earlier.

`Event` $E_{AARM}$ = (addActiveRole(user, session, role), (role = "Manager"));

`Event` $E_{TomAARM}$ = (addActiveRole(user, session, role),

$$((\text{userId} = \text{"Tom"}) \wedge (\text{role} = \text{"Manager"})));$$

addUser(...);          grantPermission(...);
addActiveRole(...);    revokePermission(...);
deleteUser(...);       createSession(...);
addRole(...);          deleteSession(...);
deleteRole(...);       addActiveRole(...);
assignUser(...);       dropActiveRole(...);
deassignUser(...);     checkAccess(...);
 + ............          +............

Figure 5.5. RBAC Operations as Simple Events.

In general, all ANSI RBAC operations [12] shown in Figure 5.5 can be considered as simple events.

## 5.4 Constraints on Simple Events using Rules

Users are allowed to carry out an operation iff they satisfy the access control policy. Even though user operations are captured as events, their operations will be granted only

if they satisfy the standard RBAC or tailor-made access control policies. Active authorization rules are the security mechanisms that act as low level semantic descriptors for either allowing or denying the requested operations. RBAC operations such as assigning users to roles, enabling roles, disabling roles, activating roles for user sessions, checking access, adding users and roles, and many others are enforced via rules. Simple events are triggered based on user requests and complex events are detected based on the history of simplex events that are triggered and both simple and complex events are associated with active rules, as appropriate, to enforce the access control policy checking. Separation of events for capturing user activities and rules for policy checking facilitate the reuse of the same rule for several events. For example, for all the four event types there can be a single rule that checks condition when roles have to be activated. Rules can invoke other rules allowing rule abstractions. Chapter 4 discusses how extended ECA rules are used as security mechanism for the enforcement of RBAC and extended RBAC policies.

We will explain two rules for handling role activations where the one is tailor-made for user Tom and the other is for *all* other users.

"*Allow any user to activate any role.*" This will be handled by the event $E_{AAR}$ defined in Section 5.3 and Rule 1.

**Rule 1.** *Rule for handling role activation for* all *the users in a system that has RBAC policies.*

$$\text{RULE} \quad [ \quad R_{AAR} : CR$$
$$\text{ON} \quad E_{AAR}$$
$$\text{WHEN TRUE} \quad \backslash\text{* specific constraints *}\backslash$$
$$\text{THEN} < call > \_addActiveRole(user, session, role)$$
$$\text{ELSE} \quad \text{raise error "Access Denied Cannot Activate" }]$$

"*Allow user Tom to activate any role.*" Event $E_{TomAAR}$ defined in Section 5.3 and Rule 2 will handle this requirement.

**Rule 2.** *Rule for handling role activation for user* Tom *in a system that has RBAC policies.*

$$\text{RULE} \quad [ \quad R_{TomAAR} : CR$$

$$\text{ON} \quad E_{TomAAR}$$

WHEN TRUE \* specific constraints *\

$$\text{THEN} < call > \_addActiveRole(user, session, role)$$

ELSE raise error "Access Denied Cannot Activate" ]

Rule 1 is based on the event $E_{AAR}$ defined before. Whenever a user tries to activate a role in a session, event $E_{AAR}$ is raised. If there are constraints associated with the event, they are checked. Event attributes, in this case, user, session and role are passed on to the the rule. When all the constraints are satisfied, "T" part in the rule $R_{AAR}$ is executed. In the above rule, there are no constraints associated, thus "C" part returns TRUE. Similarly, Rule 2 is triggered when user Tom activates a role (i.e., event $E_{TomAAR}$). As both events $E_{AAR}$ and $E_{TomAAR}$ are for role activations, they invoke $\_addActiveRole(user, session, role)$, a function for checking common constraints for role activations [12].

The function $\_addActiveRole(user, session, role)$ shown below in Figure 5.6 is similar to $addActiveRole(user, session, role)$ in Figure 5.3 except that this function is for RBAC with role hierarchies. Thus, all the four components of RBAC can be supported seamlessly with our approach.

First it checks whether the *user* is available in set $\mathcal{U}$ that contains the list of users[5]. Next, it checks the *role*, *session* and whether the session is owned by that user. Once verified, it checks whether the user is authorized to that role using the function

---

[5]We assume that users lists $\mathcal{U}$, role lists $\mathcal{R}$, session lists $\mathcal{S}$ that contain user, role and session information, respectively, are already available. In addition we also assume that other functions that are used in the rule are also available.

```
_addActiveRole(user, session, role) { //Hierarchical RBAC
    IF (user ∈ U && session ∈ S && role ∈ R &&
        (session ∈ _userSessions(user))
        && (role ∈ _sessionRoles(session))
        && (user ∈ _authorizedUsers(user, role)) ) {
            _addSessionRole(session, role); //activate role
            return TRUE; }
    ELSE {
            raise error "Access Denied";
            return FALSE; }}
```

Figure 5.6. Add Active Role - Hierarchical RBAC.

$\_authorizedUsers()$ as a user should be assigned/authorized in order to activate any role. This is carried out as the user can activate role if he is assigned to that role or to any of its senior role. It then checks whether the role is not activated in that session using $\_sessionRoles()$. Once all the above conditions are verified, role is activated in that user session by invoking the function $\_addSessionRole()$ by adding it to the *active role set* (the set containing all the active roles for an user).

We defined two events $E_{AAR}$ and $E_{TomAAR}$, and rules corresponding to them. Function $\_addActiveRole(user, session, role)$ is predefined based on the RBAC functional components. Rules corresponding to events are triggered when users try to activate any role. Shared constraints for any role activation is provided in the function $\_addActiveRole(user, session, role)$. This abstraction or granularity has two advantages: 1. tailor made constraints can be specified in rules $R_{TomAAR}$ and $R_{AAR}$, 2. functional components can be just plugged in as needed. For example, the above function $\_addActiveRole(user, session, role)$ checks for role activations in the presence of role hierarchies, but can be changed to static SoD constraint checking, seamlessly.

A context-based constraint has to be specified for $\forall u \in \mathcal{U}$ and $\forall r \in \mathcal{R}$, while activating a role. This policy can be enforced by placing context constraints on the

previously defined event $E_{AAR}$. Rule 3 is the modified version of Rule 1. It checks whether the location is "Room B" using a function *checkLocation* whose return type is Boolean. Thus, users are allowed to activate roles only from "Room B". Once the context constraint is satisfied, function *_addActiveRole(user, session, role)* is invoked as in the case of Rule 1.

**Rule 3.** *Allow all users to activate any role only if they are in location "Room B"*

> RULE [ $R_{AAR} : CR$
>
> > ON $E_{AAR}$
> >
> > WHEN checkLocation("Room B"); /* Modified */
> >
> > THEN $< call > \_addActiveRole(user, session, role)$
> >
> > ELSE raise error "Access Denied Cannot Activate"
>
> ]

This policy requires to check a context constraint for $\exists u \in \mathcal{U}$ and $\forall r \in \mathcal{R}$. Event $E_{TomAAR}$ handles the role activation for user `Tom` and placing a context constraint on this event will enforce the policy. Rule 4 is the modified version of Rule 2 and it checks whether the location is "Room C" using a function *checkLocation* whose return type is Boolean.

**Rule 4.** *Allow user `Tom` to activate any role only if he is in location "Room C"*

> RULE [ $R_{TomAAR} : CR$
>
> > ON $E_{TomAAR}$
> >
> > WHEN checkLocation("Room C"); /* Modified */
> >
> > THEN $< call > \_addActiveRole(user, session, role)$
> >
> > ELSE raise error "Access Denied Cannot Activate"
>
> ]

Modified rules 3 and 4 enforce the required policies. Only condition part of these rules are modified to meet the requirements. Thus, constraints over simple events can

be specified, and corresponding rules can be modified automatically to enforce those constraints. On the other hand, simple events can also be used in the policy specification.

**Policy 1.** *Role* `Nurse` *should be enabled between 10 a.m. to 5 p.m. everyday.*

This policy requires a repeating absolute temporal event that should occur every day at two time points. We can model this policy with two events:

$$\text{Event } E_{P1a} = (10{:}00{:}00/*/*/*);$$

$$\text{Event } E_{P1b} = (17{:}00{:}00/*/*/*);$$

Everyday at 10 a.m. event $E_{P1a}$ is raised that triggers rule $R_{P1a}$. Once triggered, role `Nurse` is enabled by invoking the function "enableRole()". Similarly, at 5 p.m. event $E_{P1b}$ is raised and the rule $R_{P1b}$ is triggered, which disables the role.

RULE [   $R_{P1a} : CR$

       ON    $E_{P1a}$

       WHEN TRUE \* Policy Conditions *\

       THEN $< call > enableRole(\text{``}Nurse\text{''})$ ]

RULE [   $R_{P1b} : CR$

       ON    $E_{P1b}$

       WHEN TRUE \* Policy Conditions *\

       THEN $< call > disableRole(\text{``}Nurse\text{''})$ ]

In addition to the above absolute temporal events, Snoop supports various other relative temporal operators which are discussed in the next section.

## 5.5   Event Pattern Constraint Specification

Roles can assume any of the three states [23]– disabled, enabled or active. When a role is disabled it cannot be assumed in any user session. When a user assumes a enabled role it makes the role active. In our model, simple and complex event pattern constraints

control the states of the role. Constraints can be of any type involving the event attributes or system predicates. Operations such as role enabling, disabling, assignments, de-assignments, activations, and deactivations are controlled by events. Some of these operations trigger events which in turn changes the role state based on the satisfaction of constraints.

Even though specification of constraints over simple events are necessary, they are not sufficient in many situations. *Events can depend on other events, thus acting as constraints.* These constraints that are enforced by events on other events are *event pattern constraints.* In other words, with event patterns (or expressions), constituent events depend on other constituent events based on the event operator. For example, there can be a policy that will allow user `Tom` to activate any role *iff* user `Jane` is active in role `Manager`.

Below we define three event pattern policies EPP1, EPP2, and EPP3, where one event pattern is dependent (or controlled by) on another event pattern. As shown, in policy EPP1 event E2 is granted access only if it follows event (sequence operator) E1. If E2 occurs without the occurrence of event E1 then it is considered as an uncomplete event and the associated uncomplete rule is executed. Consider event pattern EPP2, which is a conjunction of event pattern EPP1 and event E3. The event pattern EPP3 with events E4 and E5 depend on the occurrence of the event pattern EPP2. Similar to EPP1, EPP2, and EPP3, other event patterns can be specified using the BNF discussed previously. Thus, event patterns can be simple or complex and it purely depends on the access control policies of enterprises.

`Event` EPP1 $= \gg$(E1, E2);

`Event` EPP2 $= \Delta$(EPP1, E3);

`Event` EPP3 $= NOT$(EPP2, E4, E5);

Both role-dependent $\mathcal{E}_{\mathcal{RD}}$ and role-independent $\mathcal{E}_{\mathcal{RI}}$ events can be combined separately using complex event operators, for detecting complex event patterns that occur over a period of time. In addition, both classes of events can be combined together allowing the detection of role-interdependent event ($\mathcal{E}_{\mathcal{RID}}$) patterns. Previously we have shown how constraints can be specified over simple events. On the other hand, requested operations can be allowed when an *event pattern* that includes various events occur. Although in Section 5.2.4 we have explained pattern operators that allow event pattern specification, new pattern operators can be easily plugged-in in our framework.

### 5.5.1 Sample Event Pattern Policies

Below, we show some sample policies and show how pattern operators are used in extending RBAC with event pattern constraints.

**Policy 2** (Relative Temporal Policy)**.** *Deactivate user* `Tom` *from any active role after two hours.*

Event $E_{TomDAR} = \texttt{PLUS}(E_{TomAAR}, 2 \text{ hours})$;

This policy is based on duration based constraint. As it requires to deactivate user `Tom` from any active role after two hours, we need to associate this with the previously defined role activation event $E_{TomAAR}$. There are two steps involved; first step requires to know when two hours is elapsed, and second is deactivating the role itself. `PLUS` operator triggers an event after $\Delta T$ time of its constituent event's occurrence. Thus, using the `PLUS` event with $E_{TomAAR}$ and two hours, we can capture the first requirement.

```
RULE[   R_{TomDAR} : CR

        ON    E_{TomDAR}

        WHEN TRUE \* Policy Conditions *\

        THEN < call > _dropActiveRole(user, session, role);

     ]
```

Once the event for two hours of elapsing is captured, the second requirement of deactivating the role can be carried out using the above complete rule. Role is dropped from the Tom's active role set by invoking the function _dropActiveRole(user, session, role).

PLUS event operator is a complete event as the detector of this event is a time string (i.e., a definite event). Thus, there is no uncomplete rule associated with this event. Similar to the above, this policy "Enable Role TrainingNurse fifteen minutes after role Nurse is enabled." can also be specified using the Plus event.

In the above Plus event pattern, event consumption modes can be used to address various instances of Tom's activation. In other words, when Tom activates the role from various sessions, he be dropped based on the first occurrence, last occurrence, or each occurrence using event consumption modes.

**Policy 3** (Sequential Pattern Policy). *Allow user* Tom *to activate any role only* after Jane *has activated role* Nurse.

This policy does not require the deactivation of Tom's roles or stop Tom from activating other roles, even after Jane has deactivated the role Nurse.

Previously defined event $E_{TomAAR}$ handles role activation for user Tom. Similar to this event, we have defined $E_{JaneAAR}$ below for handling user Jane's role activation. As the above policy requires to place constraints on the role activation of user Tom, we will utilize the event $E_{TomAAR}$, but not rule $R_{TomAAR}$. Thus, invocation of function "addActiveRole()" by user Tom will invoke $E_{TomAAR}$ and will detect the event $E_{P3}$ defined below.

Event $E_{JaneAAR}$ = (addActiveRole(user, session, role),(userId = "Jane"));

Event $E_{P3}$ = $\gg$(($E_{JaneAAR}$, $E_{TomAAR}$), ($E_{JaneAAR}$.role = "Nurse"));

The event $E_{P3}$ places constraints on the role activation. Let us assume that Jane has activated role Nurse. This will trigger the event $E_{JaneAAR}$ that will in turn initiate the event $E_{P3}$. Now when Tom tries to activate a role then it will detect the

event $E_P3$ initiated by the event $E_{JaneAAR}$. As the role activation constraints are satisfied and the event $E_P3$ is detected, rule $R_P3$ will be triggered. This rule invokes $\_addActiveRole(user, session, role)$, which allows the role activation, if `Tom` has the required permissions.

RULE [   $R_P3 : CR$

      ON   $E_P3$

      WHEN TRUE \\* Policy Conditions *\\

      THEN $< call > \_addActiveRole(user, session, role)$;

  ]

On the other hand, let us assume that user `Tom` tries to activate a role, but `Jane` has not activated the role `Nurse`. In other words, the policy constraint is not satisfied. In this case, the detector occurs albeit the initiator has not initiated event $E_P3$. This is an uncomplete event occurrence and it triggers rule $\_U\_R_P3$, which in turn returns a "Denied" message to `Tom`.

RULE [   $\_U\_R_P3 : UR$

      ON   $E_P3$

      WHEN TRUE \\* Policy Conditions *\\

      THEN $< raise\ action >$ "$Denied$" ]

Similar to the above policy, pre-requisite role constraints can also be modeled using the $\gg$ operator.

**Policy 4** (Together Pattern Policy). *Enable role* `Training Nurse` *when users* `Tom` *and* `Jane` *have activated role* `Nurse`.

The above policy requires two users (Tom and Jane) to be active in role "Nurse" before allowing other users to assume role "Training Nurse". This is a constraint on *role enabling*. It is a constraint that requires occurrence of two $\mathcal{E}_{\mathcal{RD}}$ events and their order of

occurrence is not relevant. This is modeled using and AND event operator[6]. Both these events are non-definite, as they are based on *users*. Previously defined events $E_{TomAAR}$ and $E_{P3}$ handle user Tom's role activation. Similarly, event $E_{JaneAAR}$ handles user Jane's role activation.

Event $E_{P4}$ specifies and enforces the Policy 4. This event is formulated using the $\Delta$ operator. When both the events $E_{TomAAR}$ and $E_{JaneAAR}$ occur, this event is detected. As the order of occurrence is not considered both these events act as *detector* as well as *initiator*. For example, when Tom activates a role at first it acts as the initiator. Once the event is initiated, when ever user Jane activates it raises this event. Similarly, user Jane can activate a role at first and then user Tom can activate, which will also raise event $E_{P4}$. Thus, $\Delta$ event falls into the category of *complete* event.

$$\text{Event } E_{P4} = \quad \Delta((E_{JaneAAR}, E_{P3}),$$
$$((E_{JaneAAR}.\text{role} = \text{``Nurse''}) \wedge$$
$$(E_{P3}.\text{role} = \text{``Nurse''})));$$

In the event $E_{P4}$, we have specified attribute based composition condition. As events $E_{P3}$ and $E_{JaneAAR}$ are triggered for any role activation it cannot be used directly in the complex constraint specification. Thus, we have specified that attribute role in both the events should have a value "Nurse". The event $E_{P3}$ is used instead of the event $E_{TomAAR}$ as the former controls Tom's role activation. If we use the event $E_{TomAAR}$ in this policy, then it will lead to a policy conflict. On the other hand, if there is a conflict it will be detected by the enforcement mechanism.

Above shown is one way of specifying the composition condition and it is similar to the WHERE clause of a SQL statement. Thus, successful activation of role "Nurse" by both the users raise event $E_{P4}$ triggering Rule $R_{P4}$. Once the rule is triggered, it checks

---

[6]If the order of events are important then the sequence operator can be used.

for policy conditions (if any) and then call the function "enableRole()", thus enabling role "Training Nurse".

RULE[  $R_{P4} : CR$

ON    $E_{P4}$

WHEN TRUE \* Policy Conditions *\

THEN $< call > enableRole(\text{``}TrainingNurse\text{''});$

]

Some of the below shown policies control the role activation of Tom, similar to the policy $P3$. On the other hand, all these policies cannot coexist in the system at the same time as it will lead to policy conflicts.

**Policy 5.** *Allow user `Tom` to activate any role `iff` `Jane` is active in role `Nurse`.*

In addition to Policy 3, this policy does not allow user `Tom` to activate other roles once `Jane` has deactivated. On the other hand, it does not require to deactivate `Tom` from all the active roles. This policy is specified using the Aperiodic operator of Snoop. With Aperiodic operator, first event is the initiator, second event is the detector and third event is the terminator. Thus, whatever event that has to be detected is constrained by the initiator and terminator. We define event $E_{JaneDAR}$ for handling deactivation of user user `Jane`.

Event $E_{JaneDAR}$ = (deactiveRole(user, session, role), (userId = "Jane"));

In the Aperiodic event $E_{P5}$, event $E_{JaneAAR}$ that handles role activation for `Jane` is the initiator, event $E_{JaneDAR}$ that handles the role deactivation is the terminator and event $E_{TomAAR}$ that handles role activation for `Tom` is the detector.

Event $E_{P5}$ =   $A((E_{JaneAAR}, E_{TomAAR}, E_{JaneDAR}),$

$((E_{JaneAAR}.\text{role} = \text{``Nurse''}) \wedge$

$(E_{JaneDAR}.\text{role} = \text{``Nurse''})));$

When event $E_{JaneAAR}$ occurs, Aperiodic event $E_{P5}$ is initiated. Now, when event $E_{TomAAR}$ occurs, event $E_{P5}$ is raised and rule $R_{P5}$ is triggered allowing the role activation for Tom. When event $E_{JaneDAR}$ occurs, Aperiodic event is terminated. Once terminated, role activation requests of $E_{TomAAR}$ are denied by triggering the rule $\_U\_R_{P5}$, as it will be an uncomplete event.

```
RULE [   R_P5 : CR

        ON     E_P5

        WHEN TRUE \* Policy Conditions *\

        THEN < call > _addActiveRole(user, session, role);

    ]


RULE [   _U_R_P5 : UR

        ON     E_P5

        WHEN TRUE \* Policy Conditions *\

        THEN < raise action > "Denied"  ]
```

**Policy 6.** *Allow user* `Tom to be active` *in any role iff* `Jane` *is active in role* `Nurse`*.*

In addition to Polices 3 and 5 this requires to deactivate Tom from all the active roles once Jane has deactivated role Nurse. This policy can be specified with the Aperiodic operator as in Policy 5. The additional action of deactivating Tom's roles can be carried out in the role deactivation event $E_{JaneDAR}$.

**Policy 7.** *Allow user* `Tom` *to activate any role iff user* `Jane` *has activated role* `Nurse` *and user* `Jim` *has not activated role* `TrainingNurse`*.*

In contrast to the above defined policies, this policy allows Tom to activate any role iff user Jim has not activated the role TrainingNurse. First, we define the role activation event for user Jim below.

`Event` $E_{JimAAR}$ = (addActiveRole(user, session, role), (userId = "Jim"));

Previously defined events $E_{JaneAAR}$ and $E_{TomAAR}$ are used for defining Policy 7. In this policy, non-occurrence of user `Jim`'s role activation is required for allowing user `Tom` to activate any role. Thus, we model this policy using the NOT event operator as shown below.

Event $E_{P7} = \quad NOT((E_{JaneAAR}, E_{JimAAR}, E_{TomAAR}),$

$$(E_{JaneAAR}.\text{role} = \text{``Nurse''} \land$$

$$E_{JimAAR}.\text{role} = \text{``Training Nurse''}));$$

With the NOT operator, first event is the initiator $(E_{JaneAAR})$, last event is the detector $(E_{TomAAR})$ and middle event is the non-occurrence event $(E_{JimAAR})$. When user `Jane` activates the role `Nurse`, event $E_{P7}$ is initiated. Let us assume that `Tom` tries to activate a role thus raising event $E_{TomAAR}$. Once raised, the NOT event is detected as the middle event $E_{JimAAR}$ has not occurred. This triggers the rule $R_{P7}$, activating the role for user `Tom`.

RULE [   $R_{P7} : CR$

      ON     $E_{P7}$

      WHEN TRUE \* Policy Conditions *\

      THEN $< call > \_addActiveRole(user, session, role)$;

      ELSE    raise error "Access Denied Cannot Activate"

   ]

Let us analyze two more cases: First, let us assume that user `Jane` has not activated the role `Nurse`, similar to the one dealt with Policy 3. In this case, when user `Tom` tries for role activation, it raises an uncomplete event triggering Rule $\_U\_R_{P7}$. This will send a denial message to the user.

```
RULE [ _U_R_P7 : UR
```

$$\text{ON} \quad E_{P7}$$

WHEN TRUE

THEN $< raise\ action >$ "*Denied*" ]

With the other case, assume user `Jane` and user `Jim` have activated their roles. When user `Tom` tries to activate the role, the NOT event $E_{P7}$ will not be detected as there is an occurrence of `Jim`'s role activation. Thus, this will be a failed event triggering the rule $_F\_R_{P7}$ with a similar denial action.

```
RULE [ _F_R_P7 : FR
```

$$\text{ON} \quad E_{P7}$$

WHEN TRUE

THEN $< raise\ action >$ "*Denied*" ]

In addition to the above NOT event, the motivation example discussed in Section 5.1 can be modeled using a NOT event. Below we show how to model the motivation example.

**Policy 8.** *Allow users active in role Nurse TO enter Pregnancy Ward FROM Virus Ward IFF the user has made a Hygiene Stop*

Below we define three events EVW, EHS, EPW corresponding to the access request to virus ward, pregnancy ward, and hygiene stop, respectively. We then define event $E_{P8}$ corresponding to the policy P8.

`Event` EVW = (checkAccess(session, operation, object), (object=Virus Ward));

`Event` EHS = (checkAccess(session, operation, object), (object=Hygiene Stop));

`Event` EPW = (checkAccess(session, operation, object), (object=Pregnancy Ward));

`Event` $E_{P8}$ = (NOT (EVW, EHS, EPW), (EVW.userId = EHS.userId = EPW.userId));

When user enter from virus ward to the pregnancy ward, the complete rule $R\_P8 : CR$ is triggered. When the user enter from virus ward to hygiene stop and then to

pregnancy ward, the failed rule $\_F\_R\_P8 : FR$ is triggered. When the users enter from hygiene stop to pregnancy ward or directly to the pregnancy ward, the uncomplete rule $\_U\_R\_P8 : UR$ is triggered.

```
RULE [   R_P8 : CR \* VWard TO PWard *\
```

   ON   $E_{P8}$

   WHEN TRUE

   THEN $< raise\ action > $ "*Denied*"  ]

```
RULE [   _F_R_P8 : FR \* VWard TO HStop TO PWard *\
```

   ON   $E_{P8}$

   WHEN TRUE \* Policy Conditions *\

   THEN $\_checkAccess(session, operation, object);$  ]

```
RULE [   _U_R_P8 : UR \* HStop TO PWard or directly TO PWard *\
```

   ON   $E_{P8}$

   WHEN TRUE

   THEN $\_checkAccess(session, operation, object);$  ]

**Policy 9.** *Enable role* X *when any two roles of* A, B *and* C *are enabled.*

In this policy, a role is enabled when two other roles from a set of three roles are enabled. This policy can be modeled as shown below. Let us assume that three events $E_{EnableA}, E_{EnableB}$ and $E_{EnableC}$ are defined for enabling of roles $A$, $B$ and $C$, respectively. Event $E_{P9}$ models the policy using ANY operator. Whenever two of the events occur, the ANY event is detected and rule $R_{P9}$ is triggered.

```
Event E_P9 = ANY(2, (E_EnableA, E_EnableB, E_EnableC));
```

```
RULE [  R_P9
        ON    E_P9
        WHEN TRUE \* Policy Conditions *\
        THEN < call > __enableRole(role)      ]
```

All the above policies are based on complex constraints and are modeled using complex events. In addition, all the constituent events of the complex events are simple events based on user operations, time- or context-based. According to complex event specification BNF, complex event pattern constraints can be specified, where constituent events of the complex events are in turn complex events. For example, we can have a constraint using two sequences $(\gg (\gg (A, B), C))$. Below we illustrate them using some examples.

**Policy 10.** *Allow user* Jack *to activate role* Training Nurse *only when both users* Tom *and* Jane *have activated role* Nurse

This policy places a precedence constraint on role activation, but the constraints depend on other users. Role activations corresponding to users `Tom` and `Jane` can be utilized from Policy 4. Event $E_{P10}$ captures the sequence between $E_{P4}$ and $E_{JackAAR}$. Thus, event $E_{JackAAR}$ is allowed only when event $E_{P4}$ has occurred. Once, event $E_{P10}$ is detected, corresponding rules can be raised and user `Jack` can be allowed to activate.

$$\text{Event } E_{P10} = \gg((E_{P4}, E_{JackAAR}),$$
$$(E_{JackAAR}.\text{role} = \text{``Nurse''}));$$

**Policy 11.** *Enable role* Training Nurse *thirty minutes after both users* Tom *and* Jane *have activated role* Nurse

This policy is based on Policy 4. Thirty minutes after event $E_{P4}$ defined in Policy 4 is detected, enable role *Training Nurse*. Thus, event $E_{P11}$ is defined below using event operator PLUS capturing the policy. This event is detected thirty minutes after the $\Delta$ event is detected, and corresponding rules are triggered for enabling role *Training Nurse*.

Event $E_{P}11 = PLUS((E_{P4}, 00:30:00));$

**Policy 12.** *Allow any user to activate role* D *after users have activated two of three roles* A, B *and* C

The above policy is based on role activations i.e., $\forall u \in \mathcal{U}$ *and* $\exists r \in \mathcal{R}$. On the other hand, the roles activation constraints are not based on the same user. We assume that there are three events $E_{A\_AAR}$, $E_{B\_AAR}$, $E_{C\_AAR}$ and $E_{D\_AAR}$ corresponding to role activations A, B, C and D, respectively.

Event $E_{P}12\_1 = ANY(2, (E_{A\_AAR}, E_{B\_AAR}, E_{C\_AAR}));$

Event $E_{P}12\_2 = \gg(E_{P}12\_1, E_{D\_AAR});$

Event $E_{P}12\_1$ is detected when any of the two roles A, B or C are raised. This will initiate the sequence event $E_{P}12\_2$. Similar to the explanation of Policy 3, when any user is trying to activate role D, it triggers the uncomplete rule. On the other hand, when users try to activate after event $E_{P}12\_1$ has initiated event $E_{P}12\_2$, it triggers the complete rule allowing them to activate role D.

**Variations to Policy 12:** We discuss various other additional constraints for Policy 12 and how they can be easily accommodated with the same events used in the policy. Let us consider four cases (1) any user can activate role A, B, C and D; (2) specific users activating roles A, B, C and D; (3) same user should activate roles A, B, C, and role D can be activated by some other user; (4) user who activates role D should have activated roles A, B and C.

In these four cases, first case is handled by Policy 12. Second case can be handled when user level events are used with events $E_{P}12\_1$ and $E_{P}12\_2$. Third case can be handled by modeling event $E_{P}12\_1$ as an attribute-based event. Similar to the third case, fourth case can be modeled using the attribute-based for both the events $E_{P}12\_1$ and $E_{P}12\_2$.

**Audit Policies:** Provision of auditing with access control system is an advantage as auditing complements access control. Below we show the modeling of audit policies using event operators.

**Policy 13.** *Keep track of the number of activations for a role when it is enabled*

In the above policy all the role activations have to be tracked, from the start of the enabling of the role till it is disabled. Thus, the enabling should start the tracking and disabling should end the tracking. We can model this policy using the cumulative aperiodic operator A*. In this operator, the first event will be the role *enabling* event, second event will be the role *activation* event and the third event will be the role *disabling* event. Again, this policy can have variations similar to Policy 12.

**Policy 14.** *Periodically monitor the underlying system and generate reports every one hour between 8 a.m. and 8 p.m.*

According to the above policy, monitoring of certain activities should begin everyday at 8 a.m. and end at 8 p.m. and it should monitor and generate reports event one hour. This policy can be modeled using a periodic operator P. In the P operator the first event will be 8 a.m. everyday, second event will be the time interval one hour, and third event will be 8 p.m. when the event should end.

**Policy 15.** *Periodically monitor the underlying system every one hour between 8 a.m. and 8 p.m. and generate reports every day at 8 p.m.*

This policy has a subtle difference from Policy 14 as it requires to monitor the system every one hour but the reports are generated at the end of the day. This policy can be modeled using the cumulative periodic operator. First event is a time based event at 8 a.m., second event is the time interval for checking and taking some action, and third event is 8 p.m. when the report has to be generated. This operator allows the accumulation of the event occurrences and is detected when the third event occurs.

### 5.5.2 Constraints Summary

Events allow the modeling of various constraints that cannot be modeled using current access control models. We have shown diverse policies involving various types of constraints and how they are modeled using event patterns. Constraints shown above include both simple and complex constraints based on time, precedence, non-occurrence, dependency and others, and their combinations. Using event pattern constraints, we have shown how various occurrences of role-dependent and role-independent operations and their combinations occurring over an interval can be modeled. Our approach allows the modeling of current constraints in role-based access control models and extends them using expressive constraint patterns.

## 5.6 ANSI RBAC Generalization Summary

Figure 5.7 illustrates the generalization of the ANSI RBAC using event-based rules. In the previous sections we have motivated the generalization, identified the primary goals, generalized event specification, and then generalized RBAC with event pattern constraints. As shown in the Figure 5.7, RBAC functional specifications are *preserved* without dissevering the function definitions. Functions such as the addActiveRole() have been abstracted as addActiveRole() and _addActiveRole(), where the former is modeled as a simple event and the latter is the exact functional specification provided in [12]. On the other hand, simple events act as a part of the event pattern, which allows the specification of event-based constraints. Both simple events and event patterns are associated with rules. Thus, when functions are invoked, simple events are raised which in turn triggers rules or notifies event patterns. Based on the conditions, actions or alternative actions are triggered. Abstracted functions such as the _addActiveRole() are invoked from the actions or alternative actions.

Because of the abstraction, notification of a simple event occurrence to an event pattern is not straightforward, but is transparent to the user. Consider the sequential pattern policy discussed previously. When the second event is controlled by the first event occurrence, and the constraint checking corresponding to the first event is in the action or alternative action part of the rule, then the sequential operator should be notified only based on the result of that action. For instance, consider the role activation of Tom is controlled by the role activation of Jane. Thus, when Jane tries to activate the role, the simple event is raised. But, Jane can be either activated or denied in the action part of the rule based on the constraints in the _addActiveRole() function. Thus, the sequential operator should be notified and Tom should be allowed to activate the role iff the function _addActiveRole() returns TRUE. This is illustrated by the dashed line between the action/alternative action part and the event pattern, in the Figure 5.7.

Thus, based on our primary goals, the existing event framework has been generalized allowing it to be used for a larger class of applications including RBAC, and RBAC has been generalized and incorporated with the generalized event framework.

## 5.7   Summary

Constraints play a vital role in realizing role-based access control over diverse domains. First, we motivated the need for generalizing RBAC based on event pattern constraints with some critical examples. We identified several advantages and limitations of Snoop, and proposed several generalizations to overcome those limitations. In particular, we have generalized the traditional simple and complex event definitions. We then identified the simple or domain events that are required for constraint specification in RBAC. We illustrated how policy checking is carried out via authorization rules. We have shown how constraints can be placed on simple events using authorization rules. We then generalized RBAC with event pattern constraints. Event patterns with com-

plex events and simple events as constituent events were used to model constraints such as temporal, context, precedence, dependency, non-occurrence, and their combinations. Even though we have discussed various pattern operators that are useful in constraint specification, new operators can be plugged in seamlessly in on our framework.
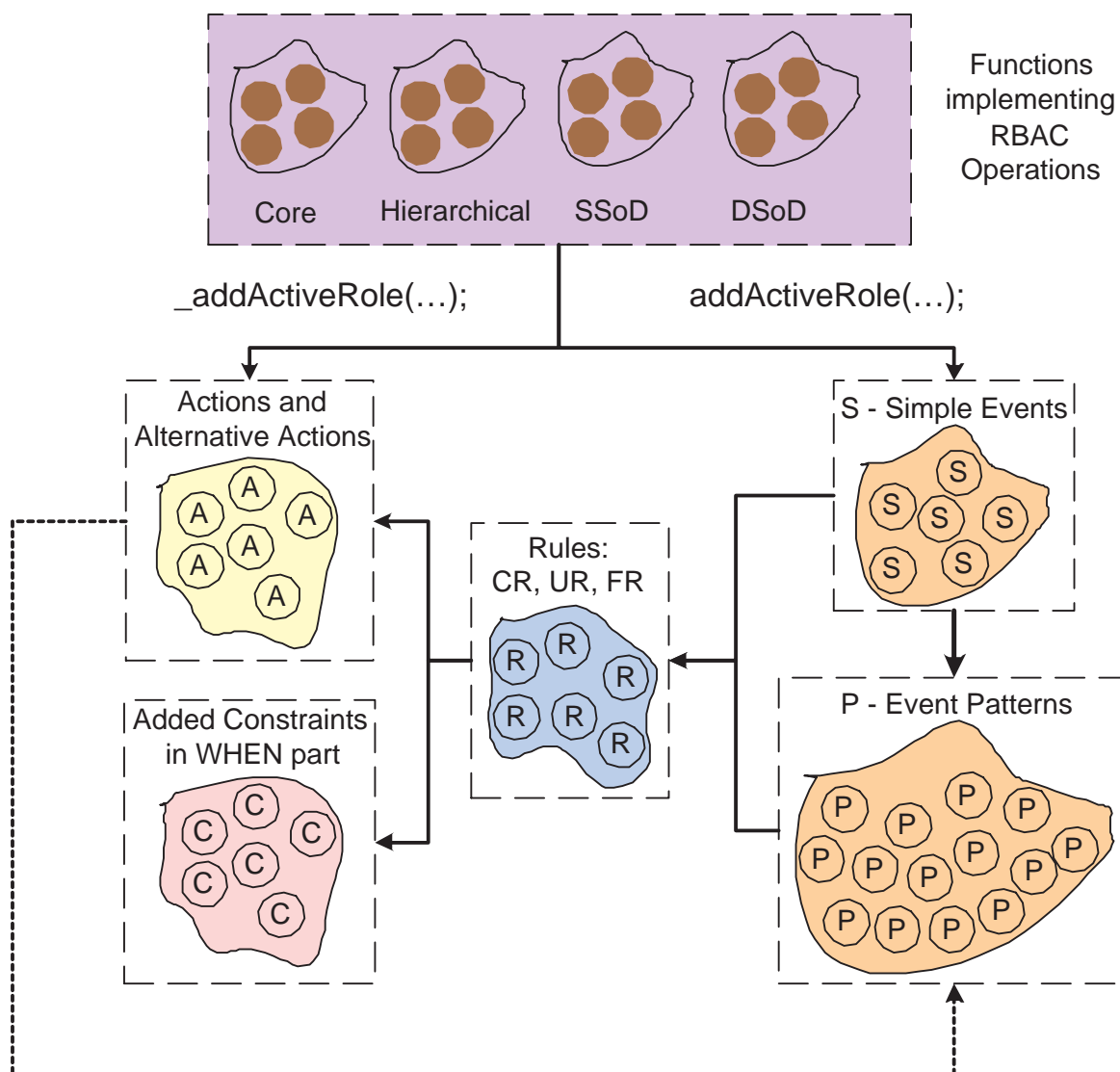
Figure 5.7. ANSI RBAC Generalization Summary.

## CHAPTER 6

## GENERALIZED ROLE-BASED ACCESS CONTROL ENFORCEMENT

Role-based access control generalization with *expressive* event pattern constraints was discussed in Chapter 5. In this chapter, we discuss the enforcement of those RBAC policies with event pattern constraints as enforcement mechanisms are equally important in order to employ them in real-world applications. First, we explain the existing event detection graph in Local Event Detector (LED) and show why it is inadequate to enforce event patterns with implicit and explicit conditions expressions. We then introduce the notion of an *event registrar graph* – a generalized event detection graph, for keeping track of constituents events, that are a part of an event pattern, which typically occur over a period of time. We show how expressive event pattern constraints are enforced. Although we have extended RBAC with event pattern constraints, constraints required by the ANSI RBAC [12] have to be satisfied in order to grant user requests. Thus, we show how user- and role-tailored policies and standard RBAC constraints are enforced using the rules associated with the event patterns.

### 6.1 Event Detection Graphs

With point-based semantics, LED [43, 148] uses an event detection graph (EDG) (see Figure 6.1) for representing an event expression specified using Snoop [41]. LED uses EDG in contrast to other approaches such as Petri nets used by Samos [64, 65] or an extended finite state automata used by Compose [58]. By combining event trees on common sub-expressions, an event graph is obtained. By using event graphs, the need to detect 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

a substantial amount of storage space (for storing partial event occurrences and their parameters), thus leading to an efficient approach for detecting events. All leaf nodes in an event tree are primitive events and internal nodes represent complex events. Event occurrences flow in a bottom-up fashion. When a primitive event occurs and is detected, it is sent to its leaf node, which forwards it to one or more parent nodes (as needed) for detecting one or more composite events. When both simple and composite events are detected, associated rules are triggered.

Rules can be defined both on primitive and composite events. A rule can be specified with a coupling mode, a triggering mode and a priority. Coupling modes as described in HiPAC [159], specify when a rule is to be executed relative to the event firing the rule. They were initially proposed for a transaction based execution environment such as a DBMS. In a transaction based execution environment, all the events occur within some transaction. The coupling mode of the rule indicates when a rule should be executed relative to the event occurring in the triggering transaction. HiPAC defines three coupling modes, namely the immediate, deferred and the detached modes. In Sentinel, event and rule definitions can be placed anywhere within the application. It should be noted that only named events can be used in rule definitions. Intermediate event expressions that are not named cannot be associated with a rule. The definition of an event that is used in a rule definition precedes the definition of the rule. As a result, it is possible that a rule gets triggered by event occurrences that temporally precede the rule definition time itself. As this might not be desirable in all situations, there is an option (the rule trigger-mode) for specifying the time from which event occurrences to be considered for the rule. Two options, NOW (start detecting all component events starting from this time instant) and PREVIOUS (all component events since the event was detected last are acceptable) are supported as rule triggered modes, with NOW being the default. It should also be mentioned that an event is detected only if there are rules defined on that

event. In addition to the parameter context, coupling mode and trigger mode associated with a rule, there is also a priority assigned to each rule. The default priority of a rule is a priority of 1. The priorities increase with the increase in numerical values i.e., 2 is a higher priority than 1, 3 is a higher priority than 2 and so on. Rules of the same priority are executed concurrently and rules of a higher priority are always executed before rules of a lower priority. It is possible that a rule raises events that in turn could fire more rules and so on. This results in a cascaded rule execution. Furthermore, rules can be specified either in the immediate coupling mode or the deferred coupling mode. Both the priority and coupling mode of a rule have to be taken into account for scheduling the rule for execution. (Please refer [148] for more details.)

Figure 6.1 (from [148]) illustrates the event detection graph. As shown leaf nodes represent primitive or simple events and internal nodes represent the complex or composite event. Two lists are associated with each node. *Event subscriber list* maintain the child to parent relationship. Whenever a particular node is notified of an event occurrence, it will sweep the event subscriber list and propagate its occurrence. When this list is empty then there are no other events that are interested in this event node. *Rule subscriber list* associates all the rules that need to be executed or triggered when there is an event occurrence. Thus, whenever an event occurs it first notifies its event subscriber list and then sweeps the rule subscriber list executing the rules. As mentioned above rules are executed based on their priority and coupling mode.

As discussed in previous chapters, any method invocation can be considered as an event. These events can be classified as instance level and class level. Instance level events are notified based on a particular instance invoking an object. Class level events are notified when any instance invokes an object. Consider a rule (rule:A) associated with a class level event (class:USER) and another rule (rule:B) associated with an instance level event (USER:Tom). Let us assume that two users (Tom, Jim) are invoking the

same method. When Jim invokes, rule:A is executed and when Tom invokes both rules A and B are executed. Rule A is referred as class level rule and rule B is referred as instance level rule. On the other hand, LED allows the creation of an instance level rule without an instance level event. In this case, an object instance is specified when the rule is created.

In order to accommodate the different type of instance level rules, LED follows two approaches. When instance level rules are specified without instance level events, it stores all the instance level rules in the same primitive node to reduce the overhead of creating multiple events. In order to facilitate this, it uses a special data structure as shown in Figure 6.2. As shown, all the class level rules are associated with the NULL instance and all other instance level rules are associated with the instances. On the other hand, when users define instance level events and instance level rules on these events then separate primitive event nodes are created and rules are associated.

In event detection, notifying a primitive event or a leaf node is an interesting issue as there can be instance level and class level events. In LED, all the named events (user defined events) and their *event handles* are kept in a hash table (eventNamesEventNodes). Event handles for a particular event is obtained when the event is created. Similarly, event signatures and event handles are kept in another hash table (eventSignaturesEventNodes). Event signatures are nothing but the method signatures over which the event is defined. eventSignaturesEventNodes hash table is maintained only for primitive events as they are raised from method invocations. As shown in Figure 6.1, whenever a method is invoked its event signature is used to match the hash table and notify the corresponding primitive event or leaf node. Similarly, complex or composite events are notified using the eventNamesEventNodes hash table.
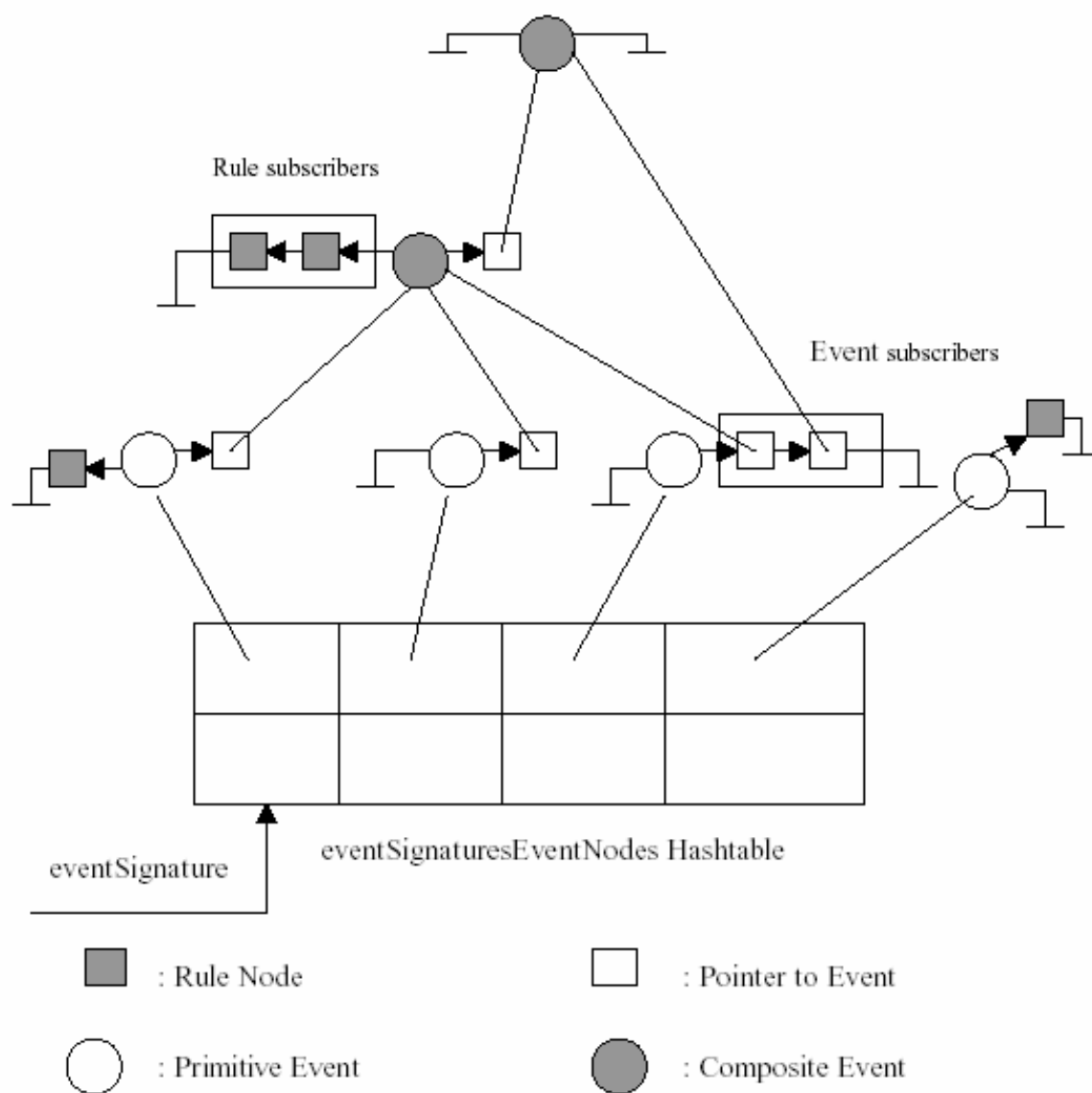
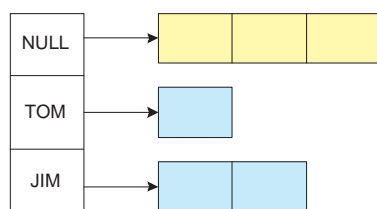Figure 6.1. LED's Event Detection Graph.



Figure 6.2. Instance Rule List.

### 6.1.1 Limitations of LED

With LED, events are detected based on the current primitive and complex event definitions. Limitations of the current event specification, and how those are overcome by the generalization were discussed in Chapter 5. Currently, whenever a primitive node is notified it just propagates its occurrence to the internal node. On the other hand, whenever a composite node is notified it checks for the semantics of the operator associated with that node. If the conditions are satisfied, it will propagate it to its event subscribers. For instance, when there is a Sequence ($\gg$) operator, then it checks whether the timestamp of the left event (initiator) is less than the right side event (detector).

With the generalization of both primitive and complex event definitions current event detection graph cannot be used directly and needs to be generalized. Primitive event generalization involves two condition expressions; implicit expression and explicit expression. Extant primitive event definition can be considered as a special case of the current generalization where the implicit expression represents the instance level events and explicit expression is empty. With the generalization implicit expression can be based on any implicit parameter and explicit expression can be based on any explicit parameter. For example, implicit expression can be still instance level event, can be some condition on the timestamp, and so forth. On the other hand, explicit expression involve condition checking on attributes and current detection does not support this. Thus, current method of primitive event notification is restricted to instances and cannot be utilized for the generalized definition. Similar to the primitive event, complex event definitions involve implicit and explicit condition expression. Current complex event semantics can be considered as a special case of the generalization. For example, the timestamp comparison of the Sequence operator can be considered as a special case of the implicit expression. More specific, current operators are tightly coupled with

the time-based semantics. On the other hand, current complex event operators do not consider explicit expressions. Thus, current EDG cannot be directly used for detecting generalized complex event definitions.

Another major problem with current event detection is that the primitive events are raised as soon as they have been notified. On the other hand, in many applications whenever a primitive event is notified they have to be considered as potential events. Consider a complex event based on a simple file open event. Whenever a user requests for a file open, the corresponding simple event node is notified. Once notified it propagates its occurrence to the complex event. On the other hand, this is incorrect as the complex event should be notified only after the file is opened. In other words, when the user does not have the required permissions to open a file the complex event should not be notified. This problem is due to the abstraction between the event, condition, and action in the ECA rule paradigm. The conditions are checked as part of the rule and actions are triggered from the rule. Thus, current event detection has to be extended to propagate event occurrence only when the corresponding rules have successfully executed. On the other hand, propagation of event occurrence after the rule has been executed is required but not for all applications. Similar to the primitive event, complex event detection also has the same problem.

Currently, with EDGs rules are associated with events and are executed when the event occurs (complete rules). On the other hand, there are many applications where rules have to be executed when a part of the event occurs (uncomplete rules). We have motivated and discussed the complete, uncomplete, and failed rules in Chapter 5. Thus, EDG has to be extended to incorporate all types of rules.
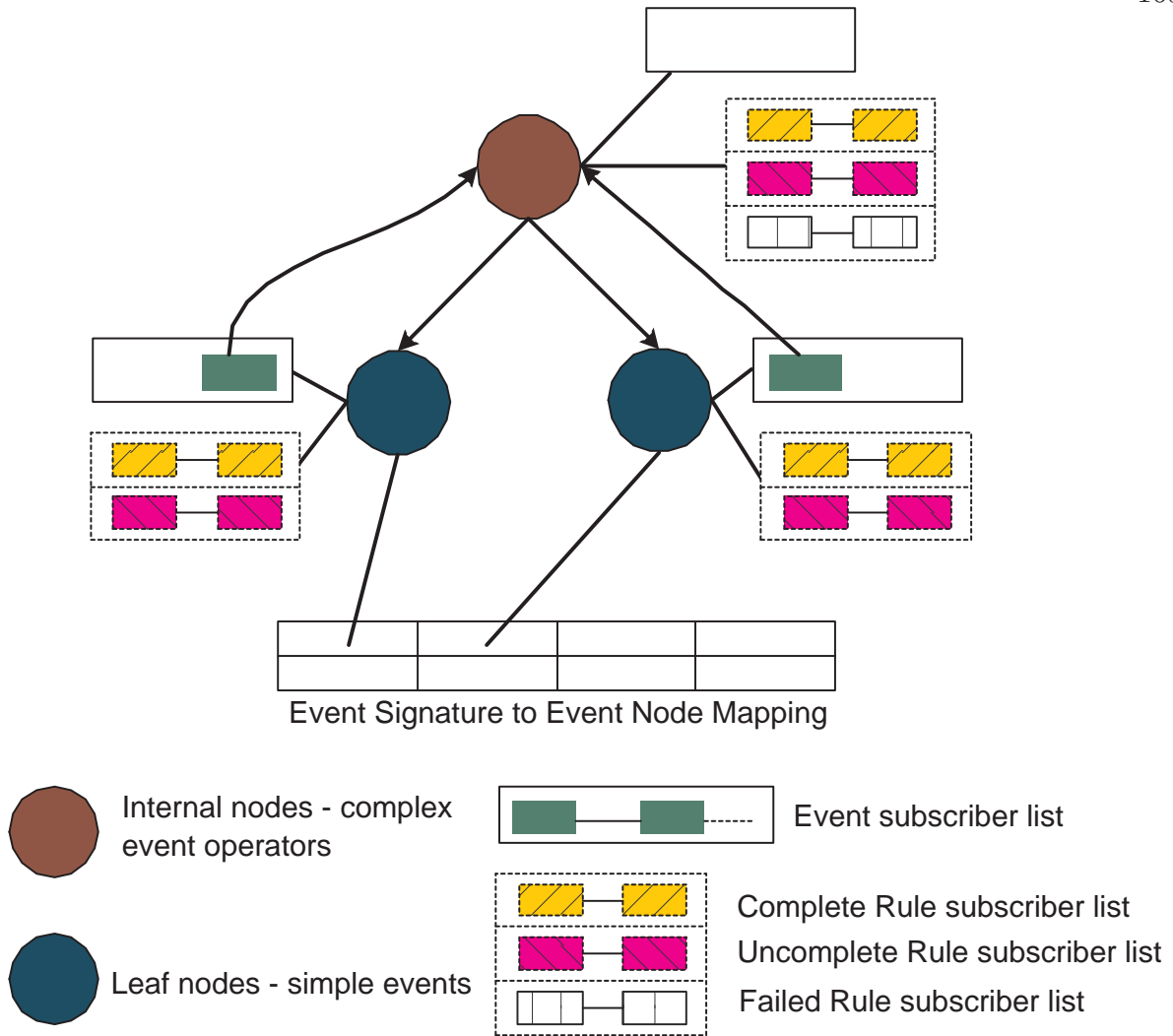
Figure 6.3. Event Registrar Graph.

## 6.2  Event Registrar Graphs

Event registrar[1] graphs (or extended event detection graphs) keep track or record of event occurrences. ERGs record event occurrences as and when they occur and keep track of the constituent event occurrences over the time interval they occur. ERGs are acyclic graphs, where each event pattern is a *connected tree*. In addition, event sub-patterns that appear in more than one event pattern are shared. ERG shown in Figure

---

[1]Merriam-Webster: *an official recorder or keeper of records*

6.3 has two leaf nodes and each of them represent a simple or primitive event. Similarly the internal node represents the complex event. The ERG as a whole represents an event pattern. In Figure 6.3, the complex event is a binary event operator (e.g., AND), thus having two child events. Although the child events are simple events in the Figure 6.3 it can be any other complex event, thus allowing a complex event pattern. Extant simple and complex event nodes from EDG (Figure 6.1) are extended with the computation of implicit and explicit condition expressions.



Event Signature to Event Node Mapping

Figure 6.4. Event Registrar Graph With Shadow Event Node.

In order to facilitate the propagation of events as and when they occur, each node in the ERG (see Figure 6.3) has three lists; *event subscriber list, complete rule subscriber*

*list* and *uncomplete rule subscriber list.* Event subscriber list contains all the events that requires this event (node) to propagate once the event is detected. For ins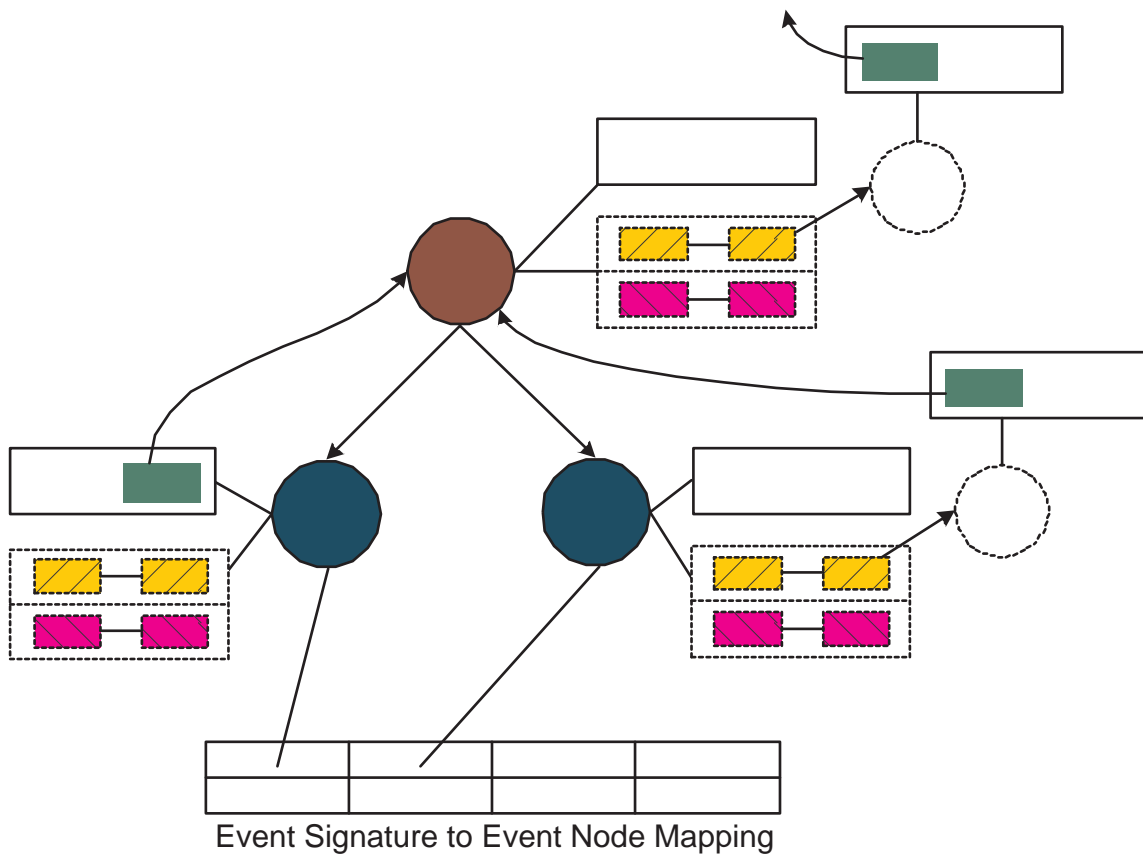tance, consider a conjunction event pattern. First, leaf nodes for both the constituent events are constructed. Second, internal node corresponding to the conjunction event operator is constructed. Third, the parent (internal) node places its pointer in the event subscriber lists of both the child (leaf) nodes. Finally, the child nodes are linked with the parent nodes. Complete rule subscriber lists contain all the complete rules that need to be triggered when the event represented by the node (leaf or internal) is detected. Similarly, uncomplete rule subscriber list contain all the uncomplete rules. Extant EDGs (Figure 6.1) does not have two kind rules, thus with ERG event node semantics have to be changed to handle this.

On a simple event occurrence the corresponding leaf node is notified and is propagated to the internal nodes, if required. In other words, when a simple event occurs, it should be propagated to the internal node if it is a part of that event pattern. Similarly, internal nodes also propagate when they are the sub-patterns of other event patterns. On the other hand, expressions and detection mechanisms have to be handled differently for simple and complex event nodes. Another major difference with ERG is the handling of potential events and actual events. As explained before, current EDG does not differentiate between these two phases of event detection. Figure 6.4 illustrates how the ERG handles the two phases of event detection (i.e., from potential event to an actual event). Consider the binary operator AND with two child events, where the right child event is a file open event. As shown in Figure 6.4, let us assume that there are two rules for the right child event. First rule is to trigger a message, and the second rule checks conditions and opens the file. Thus, the parent event (AND) node should be notified from the second rule. Thus, we create a shadow node for the event-rule combination, which has the event subscribers that need to be notified based on that rule. All application domains

do not require shadow nodes and there are some event types (e.g., temporal event) that do not require them as well. On the other hand, shadow nodes are created transparent to the user and the creation of shadow nodes can be based on the application domain or can be a user-defined parameter at the time of rule creation.

Figures 6.3 and 6.4 illustrates the ERGs corresponding to a binary operator (compose two constituent events). Similar to these ERGs other ERGs are created for all the event patterns. Even though events are detected in the nodes, they follow different semantics. Below we describe both simple and composite event detection.
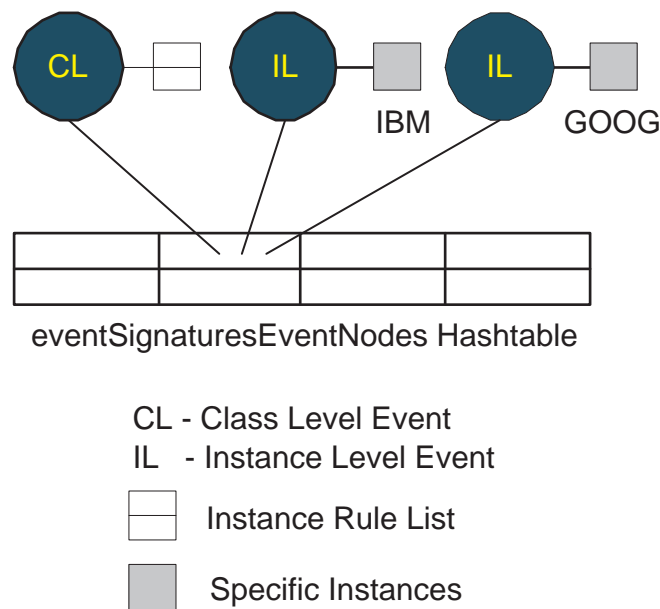


Figure 6.5. Simple Event Detection in Detection Graphs.

## 6.2.1 Simple Event Detection

Figure 6.5 illustrates how primitive event nodes in an EDG are notified when an object instance invokes a method. Primitive events in an EDG are detected using the following steps:

1. Named events are defined by the user on member functions of a class. Users can define class level and instance level events.

2. Method invocation by an object instance is captured as an event.

3. Using the method signature (also called as event signature), eventSignaturesEventNodes (refer Figure 6.5) hash table is traversed.

4. When a method signature matches, `ALL` the events (class and instance level) that were defined using this method are notified. In the Figure 6.5, IL (instance level) nodes have instance checking mechanism, so that when there is a match the IL event is detected and propagated. Similar to the list shown in Figure 6.2, CL (class level) nodes are associated with instance rule list.

On the other hand, primitive events are detected in a slightly different manner in an ERG due to implicit/explicit expressions and potential/actual events. Figure 6.6 illustrates how primitive event nodes in an ERG are notified when an object instance invokes a method. As shown in the figure, each event signature can notify four types of primitive event nodes. For detecting these type of events, *virtual* primitive event nodes are created for each method signature that has a event defined on it. Implicit and explicit condition expressions that were part of the primitive event nodes (refer Figure 6.3) are moved into the virtual node (refer Figure 6.6). Primitive events in an ERG are detected using the following steps:

1. Named events are defined by the user on member functions of a class. Users can define events with or without implicit or explicit condition expressions. Primitive events can be of four types IE (events with implicit and explicit expressions), I (events with implicit expressions), E (events with explicit expressions) or $\emptyset$ (events without expressions). When mapped to the EDG, instance level events can be represented using the implicit expression, and class level events are events without both expressions. Current EDG does not support the other two types of events.

2. Method invocation by an object instance is captured as an event.

3. Using the method signature eventSignaturesEventNodes (refer Figure 6.6) hash table is traversed.

4. When a method signature matches, it notifies the corresponding virtual event node. The virtual event node checks for `ALL` the four type of events i.e., IE, I, E, $\emptyset$. It then notifies according to the expression satisfaction. On the other hand, the *order* in which these expressions have to be checked purely depends on the application domain. Furthermore, some application domains (e.g., RBAC) requires the notification of exactly one event. In other words, when the instance matches, only that event should be notified and not the class level. (In the following sections we will detail this in the context of RBAC.) When compared to the event detection graph, this approach with virtual event node is efficient as NOT all nodes (instance and class level events) corresponding to that method signature are notified.

### 6.2.2 Event Pattern Detection

Primitive events have to notify once they are detected. As discussed previously, events have two phases; capturing of potential events and propagating the actual event. We have introduced shadow nodes in order to facilitate the two phases of event detection. Thus, primitive event nodes notify the complex event in two ways; 1) traversing the event subscriber list, and 2) executing the rules and then notify. Once complex events are notified by all their child events then complex events can be detected.

Event pattern or complex event detection starts with a constituent event initiating the event pattern. When the last constituent event occurs (detector), the event pattern has to be detected. Similar to the primitive event detection, event pattern detection has both implicit and explicit condition expression (refer Figure 6.3). Event pattern detection involves the following steps:
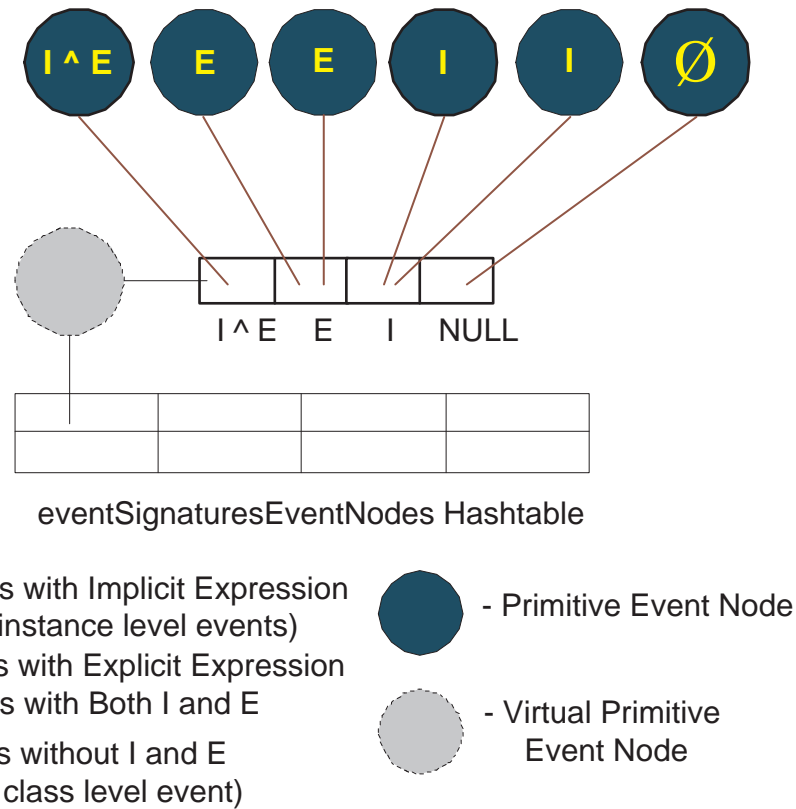
Figure 6.6. Simple Event Detection in Registrar Graphs.

1. Named complex events are defined by the user based on other named events, and an event operator. Users can define events with or without implicit or explicit condition expressions. For instance, the user can define that attribute one of the left child and attribute two of the right child should be equal. On the other hand, the current event operator semantics include timestamp comparison as default. Thus, event patterns can have IE, I, E or $\emptyset$.

2. Whenever all the constituent events occur, the event pattern is ready for the detection.

3. Detection involves the checking of both implicit and explicit expressions. Once the expressions are satisfied, both subscriber lists (event and complete rule) can be notified. On the other hand, the order in which these expressions have to be

checked purely depend on the application domain. Currently, we have included the traditional timestamp semantics (can be considered as a subset of implicit expression) with the event operators, but can be removed based on the application domain.

4. Similar to the event notification by the primitive events, event patterns also use shadow events for notifying its parents when required.

5. When the event pattern is not initiated, but the detector event has occurred then it is treated as the uncomplete event and it should execute the uncomplete rule subscriber list. For instance, with the Sequence operator, whenever the right child notifies of its occurrence and the left child has not notified, then it is an uncomplete event. On the other hand, for some operators (e.g., AND) this cannot happen and uncomplete rules are never executed. Similarly, for the NOT operator, complete, uncomplete, and failed rules are executed. Again this depends the operator semantics by itself and over various consumption modes.

### 6.2.3 Summary

Generalization of event detection graphs as event registrar graphs was discussed above. We have shown how primitive and complex events are notified and detected. In the following sections we will discuss how event registrar graphs can be used to *enforce* the generalized RBAC with event pattern policy constraints.

### 6.3 Event Pattern Policies with ERG

Complex access control policies are modeled using the notion of interesting events that occur based on the actions taken by the individual users in a system using events, event operators and event patterns. The sequence of interesting events (or history) is important, as complex specifications are modeled using event operators and patterns on the sequence of event occurrences (or history). Constraint specification and access control

policy modeling using events are not effective unless there is a generalized mechanism to detect events and trigger corresponding authorization rules. On the other hand, event detection requires preserving the order of event occurrences that are spread across the temporal dimension. In this section we discuss how event registrar graphs (ERGs) are exploited for keeping track of and detecting events and triggering authorization rules.

### 6.3.1  Simple Event Detection

Generalized simple events are classified into four types based on the implicit and explicit expressions; IE, I, E and $\emptyset$, and have to handled in the ERG. As detailed in Chapter 5 events are captured based on the user operations or function invocations. Even though the events are captured, they are detected via different ways based on users and roles in the ERG. Four categories of events that need to be detected are;

1. $\forall u \in \mathcal{U}$ and $\forall r \in \mathcal{R}$ ($\emptyset$ event)

2. $\exists u \in \mathcal{U}$ and $\forall r \in \mathcal{R}$ ($\mathcal{I}_{expr}$ event)

3. $\forall u \in \mathcal{U}$ and $\exists r \in \mathcal{R}$ ($\mathcal{E}_{expr}$ event)

4. $\exists u \in \mathcal{U}$ and $\exists r \in \mathcal{R}$ ($\mathcal{I}_{expr}$, $\mathcal{E}_{expr}$ event)

Events from the first two cases can be thought of either class level or instance level events, where users form the class. First case are *class* level events ($\emptyset$) as these are raised irrespective of *users or roles*. Second case are *instance* level events as these are based on a particular *user* (Implicit parameter). With the third case events are raised only when the when a user in a particular role has requested an operation. Final case have both $\mathcal{I}_{expr}$ and $\mathcal{E}_{expr}$ expression conditions.

Even though all these events are captured based on the user operation they are detected/raised using different semantics. In other words, the order of evaluation of IE, I, E, and $\emptyset$ are specific to RBAC. With the ERG when ever a simple event is captured from a user operation the leaf nodes corresponding to the event are notified. As discussed

previously, with RBAC domain, all the user operations captured might not lead to event detection and it would require *shadow event nodes.* Consider events $E1, \ldots, E4$;

$$E1 = (addActiveRole(user, session, role), (userId = \text{``Tom''})); /*I*/$$

$$E2 = (addActiveRole(user, session, role), (role = \text{``Manager''})); /*E*/$$

$$E3 = (addActiveRole(user, session, role), (userId = \text{``Tom''} \land role = \text{``Manager''})); /*IE*/$$

$$E4 = addActiveRole(user, session, role); /*\emptyset*/$$

Event E1 is based on user `Tom` and should be detected when he is trying to activate any role. Event E2 is detected when any user is trying to activate role "Manager". Event E3 is detected when user `Tom` tries to activate role Manager. Event E4 is detected when any user tries to activate any role.

Let us consider a scenario, where user `Tom` is trying to active role "Nurse". In this case, which events from E1, E3 or E4 should be raised ? Event E3 cannot be raised as it should be raised only when a "Manager" role is activated. Since the occurrence of interest is only the activation of role by `Tom` only that event (E1) is raised. Let us consider a scenario where `Tom` tries to activate role "Manager". In this case, only event (E3) is activated as event E3 is more specific than E1 and E2. Event E4 should be raised *only* when users other than `Tom` try to activate any role except "Manager". Thus, events are raised/detected in the following order (refer Figure 6.7) in the RBAC domain:

1. $\exists u \in \mathcal{U}$ and $\exists r \in \mathcal{R}$ $(\mathcal{I}_{expr}, \mathcal{E}_{expr})$
2. $\exists u \in \mathcal{U}$ and $\forall r \in \mathcal{R}$ $(\mathcal{I}_{expr})$
3. $\forall u \in \mathcal{U}$ and $\exists r \in \mathcal{R}$ $(\mathcal{E}_{expr})$
4. $\forall u \in \mathcal{U}$ and $\forall r \in \mathcal{R}$ $(\emptyset)$

As shown above there can be multiple leaf nodes (primitive events) corresponding to a function invocation. Virtual event nodes are implicit nodes that are created for a function and is notified when that function is invoked. Once notified, based on the
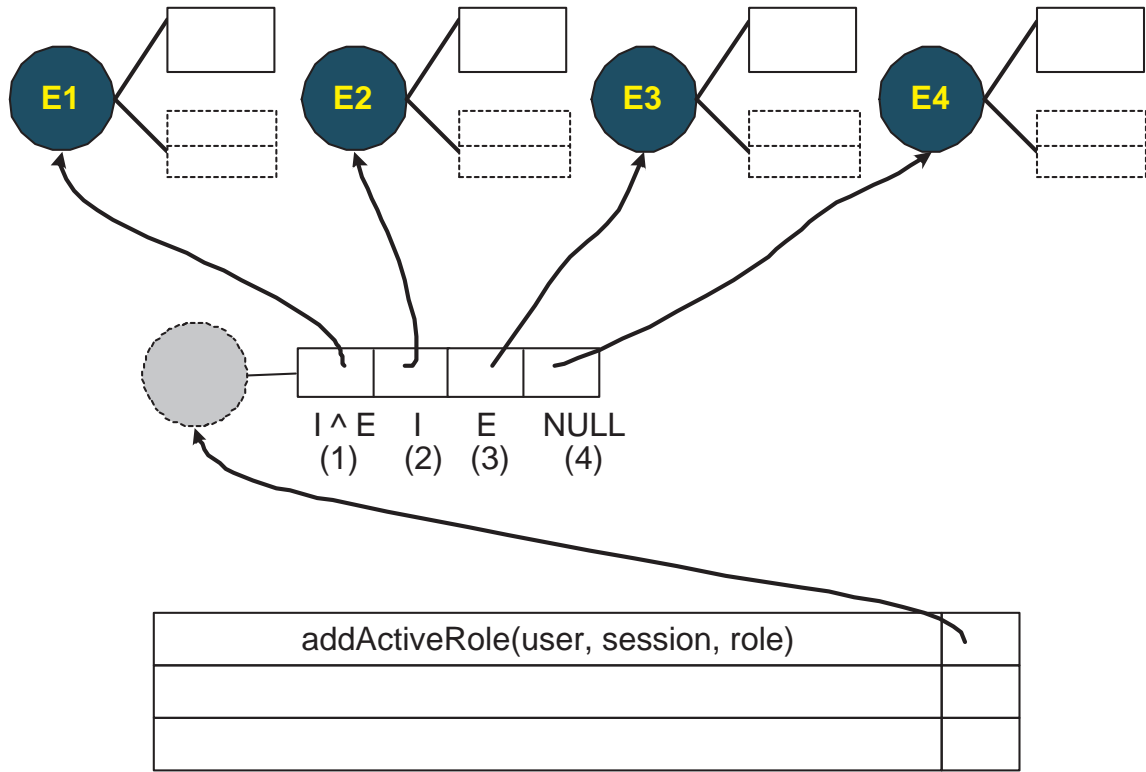
Figure 6.7. Simple Event Detection in RBAC.

event categories they notify the leaf nodes corresponding to the simple event. Figure 6.7 illustrates the event registrar graph for events E1 through E4 explained above. A virtual event node is shown corresponding to the function "addActiveRole(user, session, role)". With the expression evaluation ordering discussed above, the virtual node in Figure 6.7 has four lists. It will traverse the list in the clockwise direction starting from IE (or $\mathcal{I}_{expr}$, $\mathcal{E}_{expr}$) (i.e., 1). Once it finds a match it will notify those primitive events and stop.

Consider the user Tom trying to activate the role "Nurse" using the function "addActiveRole()". Virtual node corresponding to the function "addActiveRole()" is notified and it checks and finds no match in the IE list (i.e., 1). It then traverses to I list (i.e., 2) and finds a match, and notifies the event E1 and stops. On the other hand, when user Jane tries to activate role "Nurse", the same virtual node is notified. It checks and finds no match in IE, I or E (i.e., 1, 2 or 3). Thus, it traverses to $\emptyset$ (i.e., 4) and then notifies

event E4. Thus, using the virtual node, all the leaf nodes corresponding to events from all different categories are notified.

## 6.3.2   Event Pattern Detection

After the primitive event nodes are notified, their occurrence has to propagated to the event pattern in which they take part. Even though simple event nodes notify complex event nodes once they occur in the EDG/ERG, it is not the case in access control domain, as events are detected only after policy constraints associated with the rule is satisfied. In other words, even though user operations are captured as events not all of them lead to the event propagation. Thus, event patterns must be notified from the simple event nodes iff the constraints in the rules associated with the simple events are satisfied.

Consider a complex constraint shown in Figure 6.8, "`Tom` is allowed iff `Jim`" has activated any role. When user `Jim` tries to activate a role, the virtual node corresponding to the function "_addActiveRole()" is notified which in turn notifies the primitive event node. Rules corresponding to that node are triggered. If `Jim` has the required permissions, he is activated in the rule by the "_addActiveRole()" function. Thus, only after `Jim` has been activated in the rule, `Tom` should be allowed to activate. In other words, just requesting for role activation by `Jim` should not allow `Tom` to activate as `Jim` might not have the required policy permissions when checked in the rule. Thus, complex constraint satisfaction depends on the RBAC policy constraints. In other words, complex events should be notified by the constituent events after the rule condition that checks for the policy constraints returns TRUE. Thus, with the RBAC domain, *shadow nodes* are required for all the user initiated events *expect* for events that are controlled (i.e., act as a detector).
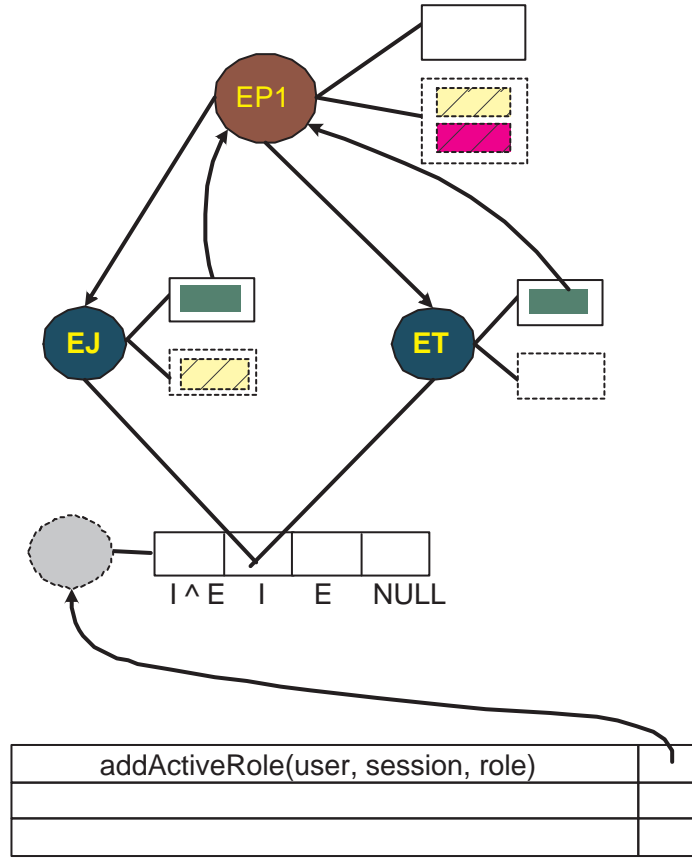
Figure 6.8. Event Pattern Detection in RBAC.

In order to model the complex event detection we define a *shadow event* that can be associated with any event that has a policy constraint to be verified, transparent to the user. The purpose of the shadow event is to propagate the event occurrence only when the policy is satisfied. We have defined events corresponding to the above example.

$$EJ = (addActiveRole(user, session, role), (userId = ``Jim"));$$

$$ET = (addActiveRole(user, session, role), (userId = ``Tom"));$$

$$EP1 = \gg(EJ, ET);$$

Event EJ and ET are simple events with implicit expression. Event EP1, is a sequential event pattern with implicit expression ("time_of_occurrence(EJ) ¡ time_of_occurrence(ET)").
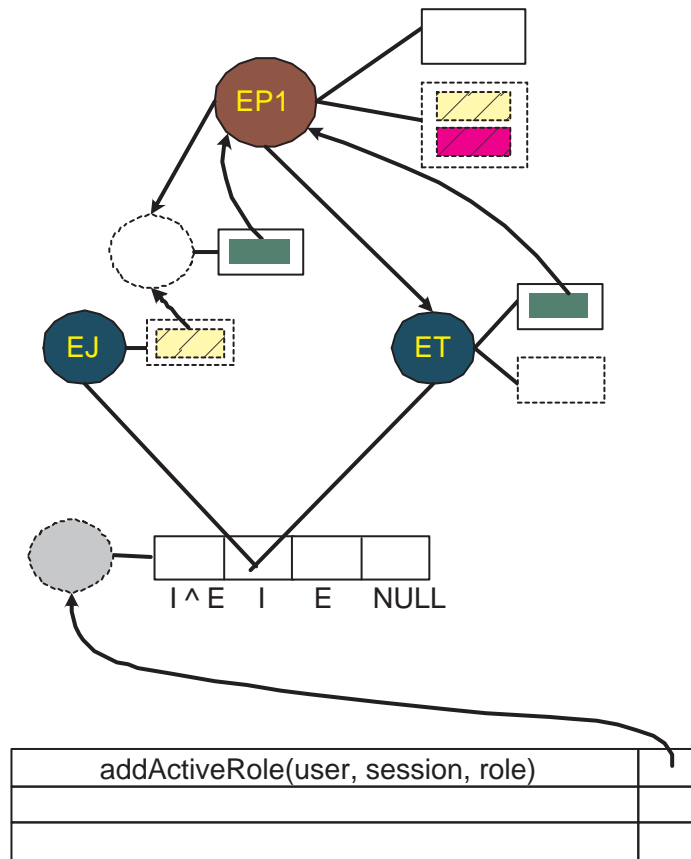
Figure 6.9. Event Pattern Detection in RBAC With Shadow Node.

Figure 6.8 illustrates the complex constraint EP1 without the shadow node whereas Figure 6.9 illustrates with the shadow node. Event node EJ is notified from the virtual node corresponding to the function "addactiveRole()" when user Jim is trying to activate any role. RBAC policies corresponding to the role activation of Jim are associated with the event EJ in the form of a rule. Once the policies are verified and Jim has been activated in the requested role, it is propagated to event node EP1 *through* the shadow node. On the other hand, if Jim does not have the permission, then the event is not propagated to EP1.

With the ECA paradigm, event EP1 has two constituent events EJ and ET, where EJ is the *initiator* of the event pattern and ET is the *detector* pattern. With the RBAC

domain, this can be said as the event ET is constrained by the occurrence of the event EJ. Thus, in order to allow ET (detector), event EJ (initiator) should have happened. Thus, RBAC policies corresponding to the role activation of `Tom` are associated with the event EP1 in the form of a rule as opposed to event ET. As the event ET is controlled by EJ, there are no rules associated with ET. As shown in Figure 6.9, event node ET does not have any shadow event as it is constrained by event EP1 and policies for user `Tom` are verified only in node EP1 and not in the node ET.

Complex constraints restrict the occurrence of the *detector* event by requiring it to occur in an order defined by the event pattern. For example, in the above, event corresponding to `Tom` should be constrained. Thus, it is modeled as the detector event constrained by `Jim`'s event that act as the initiator. In other words, without the occurrence of the initiator, event registrar graph cannot initiate the event ordering process. In an ERG, leaf nodes propagate to the internal nodes which model a complex constraint either directly or using a shadow event. Similarly, internal nodes that act as a sub-pattern for other event patterns propagate it to them either directly or using a shadow event.

Let use assume that `Jim` has activated the role, then event EP1 is notified and is initiated. Now, if user `Tom` requests role activation it is notified to node ET. It is then propagated to event EP1 without any shadow node or any policy verification. In the rule associated with the node EP1 policies are verified for user `Tom` and the corresponding role is activated. Thus, node EP1 is the detector and determines whether `Tom` is allowed or denied activation. Let us assume that `Jim` has not activated any role, and `Tom` tries to activate a role. As event ET is the detector event of EP1, occurrence of ET without prior EJ will detect an uncomplete event EP1. Thus, `Tom`'s request will trigger the uncomplete rule and a denial message is sent.
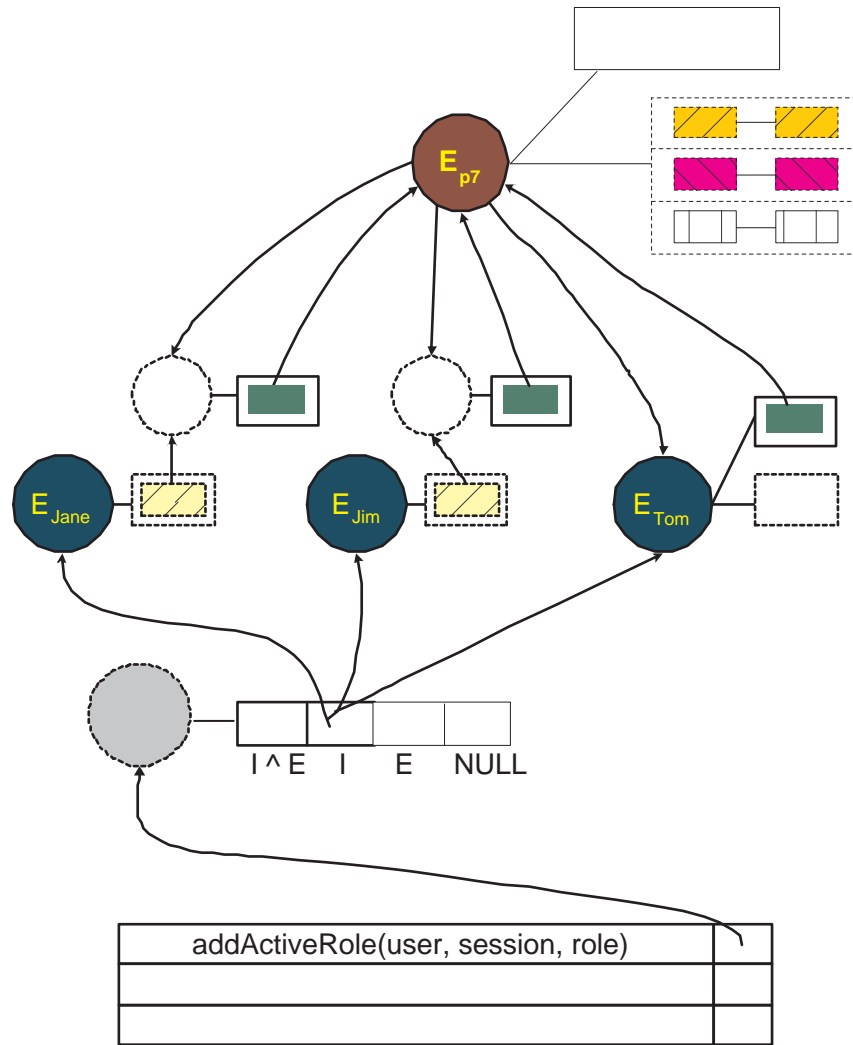
Figure 6.10. ERG for Policy 7.

### 6.3.3 Sample Policy Enforcement

In this section we will show the enforcement of Policy $E_P7$ defined previously. This policy requires to place non-occurrence precedent constraints on the role activation by the user Tom. We have shown the event corresponding to the policy, below;

$$\text{Event } E_{P7} = \quad NOT((E_{JaneAAR}, E_{JimAAR}, E_{TomAAR}),$$

$$(E_{JaneAAR}.\text{role} = \text{"Nurse"} \wedge$$

$$E_{JimAAR}.\text{role} = \text{"Training Nurse"}));$$

The ERG for the above shown events are illustrated in Figure 6.10. Events are raised from the virtual node that corresponds to the role activation function "addActive-Role()". As shown, events $E_{JaneAAR}$ and $E_{JimAAR}$, $E_{TomAAR}$ are the three leaf nodes. Virtual node for function "addActiveRole()" has three nodes connected from its (I) list. As $E_{TomAAR}$ is the event that is being constrained, there are no shadow events for it. Whenever user Jane is activating any role it is propagated to node $E_{Jane}$. Event $E_{JaneAAR}$ is propagated to event $E_{P7}$ from the shadow event when user Jane satisfies the constraints and activates any role. Propagation of the event initiates the event registrar graph corresponding to the event pattern $E_{P7}$. When Jane activates role "Nurse", capturing of event ordering is initiated. When user Tom is trying to activate, it is propagated from virtual node to event node $E_{Tom}$ and then to event $E_{P7}$ where the event ordering is verified. As the event $E_{P7}$ has been initiated and event $E_{JimAAR}$ has not occurred, user Tom is verified for constraints in the rule and if satisfied Tom is activated in the requested role. Consider three scenarios:

1. User Jane has activated role "Nurse" and user Jim has not activated role "Training Nurse". User Tom then requests for the role activation. In this scenario, event $E_{P7}$ is initiated by Jane's activation. When Tom request's a complete event is detected, as there is no occurrence of Jim's activation. Thus, Tom's request is allowed/denied based on the complete rule execution.

2. Users Jane and Jim have activated role "Nurse". User Tom then requests for the role activation. In this case, event $E_{P7}$ is initiated by Jane's activation but there
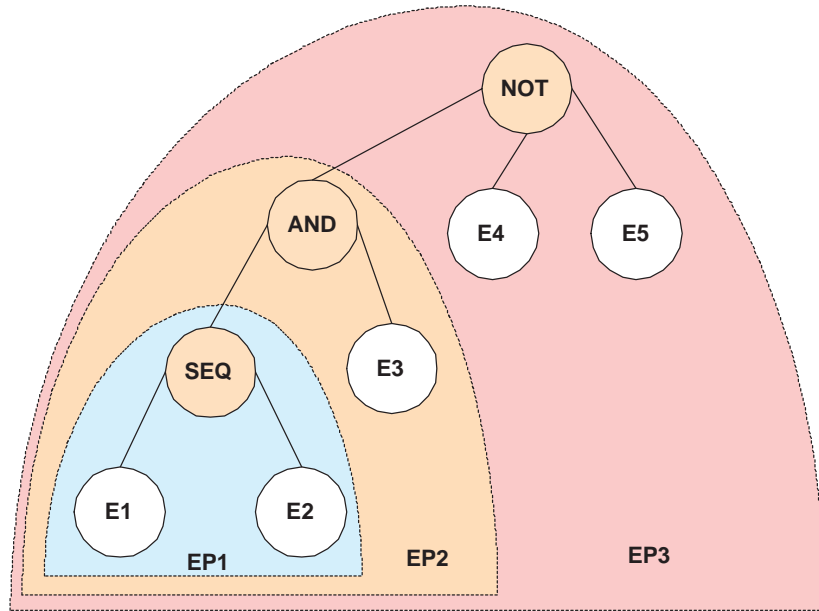
Figure 6.11. Complex Event Pattern Policy.

is an occurrence of Jim's activation. Thus, a failed event is detected and the failed rule is executed denying Tom's request.

3. User Jane has not activated any role, and Tom requests for the role activation. In this case, the event $E_P7$ is not initiated, thus detecting an uncomplete event. This event will execute the uncomplete rule and deny Tom's request.

Similar to event detection graph, event sub-patterns can be shared and composed in event registrar graphs to form complex graphs. Figure 6.11 illustrates a complex event pattern policy.

## 6.4 Policy Conflict Identification

When a user operation is associated with more than one access control policy, conflicts arise. For example, allowing user Tom to activate a particular role via two different policies introduces conflicts. These conflicts can be overcome by associating priorities with policies. Policy conflicts can arise in two different places in our enforcement. First, there can be conflicts with the basic RBAC policy specification. Second, there can be

conflicts with event pattern constraint specification. In this work, we assume that basic policy specification does not have any conflicts. In this section we discuss ways for identifying policy conflicts when using event pattern constraints. The discussion of conflict management is preliminary.

Policy conflicts arise when any event has a rule associated with it and the same event is constrained by another event. For example, consider a sequential event pattern A with two constituent events B and C. Let us assume that B is a role activation event and C is a role deactivation event. Let event B be the initiator and C be the detector. Thus, event C is constrained by the occurrence of B. In other words event C should follow event B. In this case, whenever a role deactivation event occurs it is allowed only when B has occurred. Allowing of role deactivation is carried out by the rule associated with the sequential event pattern A. Thus, if there is a rule associated with event C for deactivation, then there is a policy conflict. In general, whenever an event or event pattern is a sub-pattern of another event pattern and is the detector of that larger event pattern, then there cannot be any rules associated with the sub-pattern. On the other hand, if the sub-pattern is an initiator or terminator this does not cause any policy conflict. Below we discuss some event pattern policies from RBAC domain where conflicts arise.

Consider Policy $E_{P7}$ discussed previously. When user `Tom` is associated with more than one policy, conflicts arise. Below we show two events and their rule definitions that were previously discussed.

$$\texttt{Event}\, E_{TomAAR} = (addActiveRole(user, session, role), (userId = \text{``}Tom\text{''}));$$

```
RULE [  R_TomAAR
```

> ON    $E_{TomAAR}$
>
> WHEN TRUE
>
> THEN $< call > \_addActiveRole(user, session, role)$
>
> ELSE   raise error "Access Denied Cannot Activate"

```
]
```

Event $E_{TomAAR}$ defined above is raised/detected when user `Tom` requests for role activation. Rule $R_{TomAAR}$ enforces the policy corresponding to NIST RBAC with role hierarchies by invoking function $\_addActiveRole(user, session, role)$. Once policies are satisfied, `Tom` is activated in the requested role.

$$\texttt{Event}E_{P7} = \texttt{NOT}((E_{JaneAAR}, E_{JimAAR}, E_{TomAAR}),$$

$$(E_{JaneAAR}.role = \text{"}Nurse\text{"} \wedge$$

$$E_{JimAAR}.role = \text{"}TrainingNurse\text{"}));$$

```
RULE [  R_P7
```

> ON    $E_{P7}$
>
> WHEN TRUE
>
> THEN $< call > \_addActiveRole(user, session, role)$
>
> ELSE   raise error "Access Denied Cannot Activate"

```
]
```

Event $E_{P7}$ defines a policy that also restricts role activation by user `Tom`. As both the events try to restrict user `Tom` when activating roles, one with the simple constraint and other using the complex constraint, they have a policy conflict. Thus, when administrators define a policy that conflict with other policy the system should be able to identify.

In our approach, we can identify policies by analyzing the events and their rules either while instantiating events/rules or while constructing event registrar graphs. Policies' conflict when the same event acts as a detector in more than one event. When the same event acts as a detector in more than one event, in our example `Tom`'s role activation, each of them are attached to an authorization rule for enforcing a policy. Thus, allowing an event to act as only one detector event in the entire set of events, will avoid policy conflicts.

## 6.5  Summary

In Chapter 4, we have shown the modeling and enforcement of various policies based on RBAC and its extensions. Active authorization rules were used in the modeling and enforcement of the policies. In Chapter 5, we generalized event definitions and introduced complete, uncomplete, and failed events and rules, and discussed how they are used to generalize ANSI RBAC. In this chapter, we have analyzed the limitations of current event detection graph mechanism in LED. We then extended event detection graphs as event registrar graphs to incorporate all the generalization introduced in Chapter 5 and for capturing event occurrences and keeping track of event ordering. Event registrar graphs follow a bottom-up data flow paradigm and are efficient as they allow the sharing of event patterns and simple events. We have also shown how expressive event-based constraints can be enforced using event registrar graphs. Finally, we have explained how policy conflicts can be identified and resolved, though they require further investigation.

# CHAPTER 7

# USABILITY IN ROLE-BASED ACCESS CONTROL

In role-based access control, users and objects are *assigned* to one or more roles. An user should be *active* in the role that has the required permissions before the access to an object is granted. Thus, users should be aware of the role-permission (i.e., between roles and objects) assignments for activating the required roles. In other words, they have to know what roles are required to perform operations on objects. In general, with respect to the role activation, current systems follow the *human-active, system-passive* model. Users often get swamped with *role activations* due to numerous factors that include increase in the number of objects, multiple role assignments, and shifting roles often, and lean toward activating *all* the assigned roles violating the principle of least privilege ($\mathcal{PLP}$). In this chapter we introduce *SmartAccess*, a system based on the *system-active, human-passive* model, that allows users to concentrate on what objects they need, rather than what role should be activated in order to do their work efficiently. In other words, we make RBAC more usable, preserve the $\mathcal{PLP}$, and avert any information leak. We provide algorithms for discovering roles and analyze various associated factors.

## 7.1 Introduction

Currently, operating systems, database management systems, and various other systems support RBAC minimally in its primitive form. Current systems [19, 21, 71, 73, 76, 80, 89, 147] enforce RBAC in a binary mode; that is, they either *allow* or *deny* user access requests. In general, with respect to role activations, they follow the *human-active, system-passive* model, where users are required to know the role-permission assignments for activating the required roles.

190

Even though employees can be assigned to more than one role in an enterprise based on their job functions, they are *not* required to be active in all the assigned roles at *all* times to access objects, preserving the *principle of least privilege*[1] or $\mathcal{PLP}$. This introduces some problems while providing role-based authorization as users access requests are granted only if they are *active* in a role that has the required permission. For instance, consider a file *analysis.scr* that can be *read* only by the role *Project Manager*. Consider an user `Alice` *assigned* to two roles *Project Manager* and *Software developer*, but has only activated the role *Software Developer*. Thus, when `Alice` tries to *read* the file *analysis.scr* the access control system denies access as `Alice` is not active in the role *Project Manager*.

When users request object access, current systems check for the permissions based on the active roles, authorized roles (when role hierarchies are present), and other constraints (if defined in the enterprise security policy) that are *available* at the time when access requests are made, and provide binary replies i.e., *allow* or *deny*. Thus, users are required to know beforehand the role-permission assignments or $\mathcal{PA}$, which keeps track of the operations that can be performed on the objects by the roles, so that they can activate the required roles. However, being aware of the $\mathcal{PA}$ is cumbersome because of various factors and they include:

- thousands of objects in enterprises.

- number of objects is ever increasing.

- users shift roles due to promotions, demotions, relocations, and so forth.

- restructuring of roles in enterprises.

Binary decisions – either *allow* or *deny* alone are not sufficient in real-life situations where users often try to gain access without activating the required roles as they are $\mathcal{PA}$-unaware. Furthermore, with current systems, users tend to activate all the assigned roles

---

[1]At any given point in time no additional permissions are made available than required.

violating the $\mathcal{PLP}$. Thus, access control systems should allow users to concentrate on the *data* or the *object* that needs to be accessed, rather than the roles that fetch them those access permissions. However, the system should also be able preserve $\mathcal{PLP}$ and avert information leak.

To the best of our knowledge, this it the first work to introduce a system or an approach that follows the *system-active, human-passive* model in supporting RBAC and its extensions using *role discovery*. Previous works [113, 114, 115] deal with disclosure/release of policies, automatic trust negotiations, reasoning services in access control for autonomic communications, and so forth, but does not deal with RBAC and its extensions. In this work we provide *SmartAccess*, a system that provides role-based authorizations and overcomes the above mentioned problems. It allows users to be $\mathcal{PA}$-unaware, interacts with users without leaking information and preserve $\mathcal{PLP}$.



U - Users                R - Roles
OPR - Operations    OBJ - Objects
PRM - Permissions  UA - User-Role Assignments
                            PA - Role-Permission Assignments
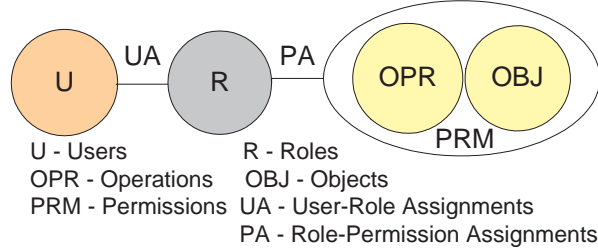
Figure 7.1. User-Role-Permission in RBAC.

## 7.2  Issues and Problems

Figure 7.1 illustrates the relationship between the basic element sets of RBAC – users, roles and permissions. As shown, users are assigned to roles and permissions combining operations and objects are assigned to roles (i.e., $\mathcal{PA}$). Thus, users can activate the required role, which in turn has permissions to access objects. This abstraction al-

lows users to shift (or switch between) roles seamlessly, and allows roles to be owners of the objects rather than individual users. On the other hand, it requires users to be $\mathcal{PA}$ aware, as users have to activate the required role in order to access objects, which is a burden on the user. Hence, users lean toward activating all the assigned roles violating $\mathcal{PLP}$. However, this can be remedied if the system can *transparently discover the role that needs to be activated.*
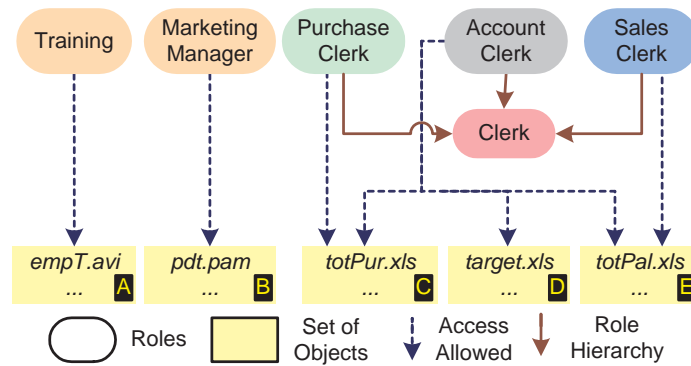


Figure 7.2. Role-based Access Policy.

Figure 7.2 illustrates the role-based access policy of an enterprise. As shown, there are six roles where roles *Purchase Clerk*, *Account Clerk* and *Sales Clerk* are senior to the role *Clerk*. For simplicity, objects that can be accessed by the roles are represented as sets ($\mathcal{A}$, ..., $\mathcal{E}$). For example, the object `pdt.pam` belongs to the set $\mathcal{B}$. The role *Marketing Manager* does not have any junior role and has permissions to access those objects belonging to the set $\mathcal{B}$. $\mathcal{PA}$ for RBAC policy defined in Figure 7.2 is shown[2] in Figure 7.3. For instance, the role *Account Clerk* has permissions to access objects belonging to the sets $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{E}$.

---

[2]For brevity, we have not shown the operations that can be performed over the objects.

| Objects / Roles | | Training | Marketing Manager | Clerk | Purchase Clerk | Sales Clerk | Account Clerk |
|---|---|---|---|---|---|---|---|
| A | (empT.avi, ...) | Y | N | N | N | N | N |
| B | (pdt.pam, ...) | N | Y | N | N | N | N |
| C | (toPur.xls, ...) | N | N | N | Y | N | Y |
| D | (target.xls, ...) | N | N | N | N | N | Y |
| E | (totPal.xls, ...) | N | N | N | N | Y | Y |

Y - Access Allowed    N - Access Denied

Figure 7.3. Role-Permission Assignments ($\mathcal{PA}$).

Consider a user `Tom` assigned to roles *Marketing Manager* and *Purchase Clerk*. Even though `Tom` is *assigned* to two roles, he has to be *active* in those roles in order to access the objects. For example, if `Tom` needs to access object `pdt.pam`, then he has to be active in the role *Marketing Manager*. Even though there are lot more scenarios that include context-aware constraints, separation of duty relations, etc, a set of sample requests from the user `Tom` are shown below, with only role *Marketing Manager* activated;

1. Tom requests for `pdt.pam` and the access is GRANTED.

2. Tom requests for `totPur.xls` and the access is DENIED. Even though he can access the same as he is assigned to role *Purchase Clerk*, he will not be permitted as he is currently NOT active in the role. Thus, he *needs to know* that `totPur.xls` can be accessed by the role *Purchase Clerk* and has to activate the role. Although this can be argued as a viable process, it is cumbersome as there are thousands of objects and hundreds of roles present in enterprises.

3. Tom requests for `empT.avi` and the access is DENIED. This object can be accessed by users in role *Training*, which is required by all employees in order to complete a training. Even though this role is not assigned to `Tom` he can acquire this role, as he might be delegated to activate this role temporarily for the duration of training

by some authorized authority. Thus, in addition to *assigned* and *authorized* roles, *delegated* roles should also be checked.

Although the access for requests 2 and 3 is DENIED directly, it is possible to provide the same *indirectly*. It is evident from the above requests that user `Tom` should be aware of the roles that need to be activated in order to access *any* object in the underlying system. To be aware of $\mathcal{PA}$ relationships is cumbersome and is nearly impossible due to various reasons. Some of them were explained briefly in Section 7.1 and the others are;

1. Users are usually assigned to more than one role.

2. With users shifting (or switching between) roles it becomes harder, as the new users assigned to the roles have to be aware of the objects that were *created by the previous users* in those roles. For example, when user `Tom` is assigned to the role *Account Clerk*, he has to be aware of all the previously created objects that can be accessed.

Issues concerning $\mathcal{PA}$ awareness are critical and can be addressed in a number ways:

1. users activate *all* the assigned roles in *every* session.

2. system *discovers and activates all* the required roles based on the users' access request.

3. system discovers the required roles based on the users' access request and *requests* the user to activate the required role.

As explained earlier, adopting the first way violates the $\mathcal{PLP}$, whereas the other two ways require role discovery. With the second way, system can discover and activate all the roles, but it increases the security risks and is not necessary as only one role is required to grant the required access. Instead of activating all the roles, the system can activate only one of the required roles based on inference, and inform the user. But the process of inferring a particular role from a set of roles is not straightforward and is

complex. On the other hand, the system can activate the first role discovered and inform the user, but it involves some security risks and it can surprise the user as the first role discovered might be out of the context. With the third way, the system discovers the required roles, but *requests* the user to activate any of those roles. Again the system can notify the first role discovered, all the roles that have the required permission, or only the role with the least permission set, though it purely depends on the enterprise.

Current systems are non-interactive and have the same problem of the $\mathcal{PA}$ awareness burdening the user. Thus, systems should not *merely* provide binary replies based on the current active roles, but need to be more user friendly. However, discovering roles, notifying users, and requesting role activations require special attention as it should not leak information and should preserve the $\mathcal{PLP}$.

## 7.3 SmartAccess

Figure 7.4 illustrates the architecture of the *SmartAccess* system. As shown, user requests from the underlying system is sent to *SmartAccess*, where the role checking module checks for the access requests utilizing the RBAC server and authorization rule server. All modules are explained in detail (except RBAC Server [3]) in the following subsections. RBAC server can be any server that provide role-based authorizations. For instance, our event-based system discussed in Chapter 4 can be used as the RBAC server.

### 7.3.1 User Request and Response Handler

Input and output of access control systems are well defined, where the former is the user access requests and the latter is either allowing or denying those requests. Access requests contain the user who made the request, operation that needs to be performed, and the object that needs to be accessed. In addition, there may be other domain-based information such as time of request, context of request, delegations and so forth.

---

[3]It maintains the basic element sets of RBAC and their relationships.
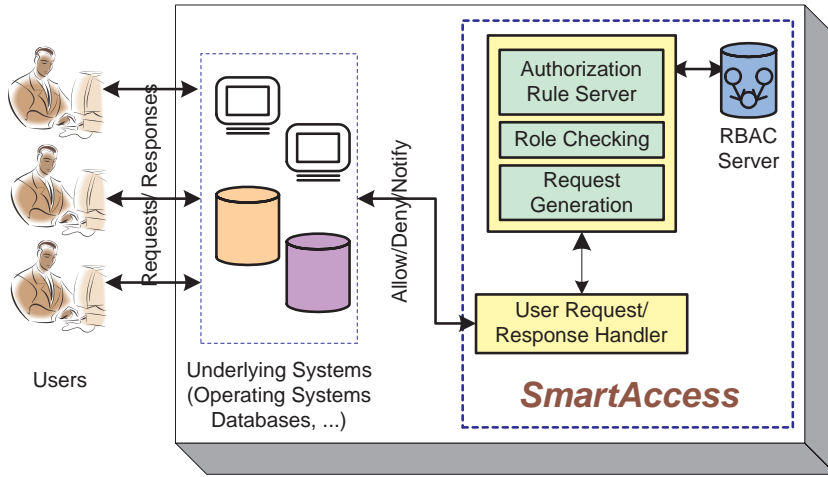
Figure 7.4. *SmartAccess* RB Authorizations.

In *SmartAccess* we represent the basic element sets (i.e., users, roles, objects, etc.) as follows:

- $\mathcal{U}$ - users, $\mathcal{S}$ - user sessions [4], and $\mathcal{R}$ - roles.

- $\mathcal{P}$ - permissions[5] (i.e., operations that can be carried out on objects), $\mathcal{OBJS}$ - objects, and $\mathcal{OPS}$ - operations.

- $\mathcal{ASRS}$ - users' assigned roles set, and $\mathcal{ASRS} \subseteq \mathcal{R}$.

- $\mathcal{ACRS}$ - users' active roles set in a $session \in \mathcal{S}$, and $\mathcal{ACRS} \subseteq \mathcal{ASRS}$.

- $\mathcal{PORS}$ - set consisting of potential roles that can be activated to gain permission, and $\mathcal{PORS} \subseteq \mathcal{ASRS}$. When role hierarchies are considered potential roles can also be from the authorized roles set.

- $\mathcal{REQS}$ - notifications or activation requests that are sent to the users' *session*. It contains instances of $\mathcal{S}, \mathcal{P}$, and $\mathcal{PORS}$. Both at the start and the end of a session $s1$, $\mathcal{REQS} = \mathcal{REQS} - \{s1reqs\}$, where $s1reqs$ contain all notifications corresponding to

---

[4]A user can have multiple sessions, but a session is associated with only one user and can have many active roles.

[5]In this work, we use basic permissions and not abstract permissions.

session $s1$. When $\mathcal{REQS}$ are based on a particular user or on the entire system (refer Section 7.3.5), the above condition may be slightly different.

A typical request include the *session* $\in \mathcal{S}$, the *object* $\in \mathcal{OBJS}$ and the *operation* $\in \mathcal{OPS}$. For example, consider user `Tom's` request for reading the object `pdt.pam`. This request is received by this module and is sent to the *Role Checking* module where the access permissions are checked. After the role checking, this module sends ALLOW, DENY, or NOTIFY/REQUEST to the user.

| Users |
|-------|
| Tom   |
| Jim   |
| Jane  |

(a)

| Roles |
|-------|
| Training |
| Marketing Manager |
| Purchase Clerk |
| Account Clerk |
| Sales Clerk |
| Clerk |

(b)

| User | Assigned Roles |
|------|----------------|
| Tom  | Purchase Clerk |
| Tom  | Marketing Manager |

(c)

| User | Active Roles |
|------|--------------|
| Tom  | Marketing Manager |

(d)

Figure 7.5. RBAC (a) Users; (b) Roles; (c) UA; (d) Active Roles.

## 7.3.2  RBAC Server

Enterprises utilizing RBAC for controlling accesses define their security policies in terms of the instances of RBAC basic elements sets, namely, $\mathcal{U}$, $\mathcal{R}$ and $\mathcal{P}$, and are maintained in the RBAC server. In other words, the server maintains all the users, roles, permissions, objects, and operations in the enterprise. For instance, lists that are maintained in the server for the RBAC policy defined in Figure 7.2 are shown in Figure 7.5. We assume three enterprise users and the corresponding list is shown in Figure 7.5(a). As illustrated in Figure 7.2 there are six roles and the corresponding list is shown in Figure 7.5(b).

In addition, the server also maintains the relationships between RBAC elements, for example, user-role assignments ($\mathcal{U}$ and $\mathcal{R}$) have a many-to-many relationship. Similarly, role-permission assignments represent the relationship between ($\mathcal{R}$ and $\mathcal{P}$), role hierarchies represent the relationship between ($\mathcal{R}$ and $\mathcal{R}$), and so forth. Figure 7.5(c) shows the user-role assignments and Figure 7.5(d) shows the active roles of users. Role-permission relationships are illustrated in Figure 7.6. Similarly, constraints such as separation of duty relations and others are placed on the relationships, and are maintained in the server.

### 7.3.3 Role Checking

Access requests received by the request handler are propagated to this module, where roles are discovered and access decisions are taken. In RBAC, users can request for activating roles, deactivating roles, accessing objects, and many others. Even though requests such as role activations can be made smart by asking the user to satisfy more constraints in order to activate the requested role, *object access requests* are more critical for the decision making as discussed in Section 7.2. First, we explain the types of roles that need to be discovered using the examples detailed in Section 7.2. Next, we explain the working of *object access request* handler in Section 7.3.4. Finally, we analyze the algorithms and associated factors in Section 7.3.5

**Assigned and Authorized Roles:** Whenever an user `Tom` requests for an object `totPur.xls` the system checks his active roles (i.e., *Marketing Manager* in this case) and authorized roles (if role hierarchies are present) for required permissions. As the role *Marketing Manager* does not have the required permission, the access control system can discover or infer the roles from the assigned/authorized roles that have the required permissions and those that can be activated by the user. In our example, only the assigned role *Purchase Clerk* has the necessary permissions that `Tom` can activate it. Thus, the

| Role | Permission | |
| --- | --- | --- |
| | Objects | Operation |
| Training | empT.avi | r |
| Marketing Manager | Pdt.pam | rwx |
| Purchase Clerk | totPur.xls | rwx |
| Account Clerk | totPur.xls | r |
| | target.xls | rwx |
| | totPal.xls | r |
| Sales Clerk | totPal | rwx |
| Clerk | | |

Figure 7.6. Role-Permission Relationships.

system notifies `Tom` that he should activate role *Purchase Clerk* in order to access the object.

**Delegated Roles:** There can be other object requests on which `Tom` has indirect permissions. As mentioned in section 7.2 `Tom` can request for an object `empT.avi` and can have indirect access permissions because of the role delegation. When there is a role delegation, then the access control system checks the origin authentication and integrity of the delegation, and grants the required access. Let us assume that the permission for accessing an object `empT.avi` is delegated to `Tom` by some authorized authority. In this case, the role delegation is verified and if it is valid then user `Tom` is given the required access. Apart from the above requests there can also be other requests that require the user to satisfy some constraints. For instance, let us assume there exists a context constraint that requires any user who needs to use the object `pdt.pam` should be accessing it from a secured network. Thus, when `Tom` requests for `pdt.pam` from an unsecured network the system denies his request even though he is active in *Marketing Manager*, and notifies that he cannot access.

**Separation of Duty Constraints:** With respect to static SoD constraints, users are not assigned to conflicting roles, thus, checking assigned roles is sufficient for role discovery.

When role hierarchies are considered with static SoD then authorized roles need to be considered. When dynamic SoD constraints are included, with or without hierarchies, discovering roles require some special cases to be handled, and they are explained in the following algorithms.

**Other Constraints:** Constraints such as content-based, context-aware, purpose-based, temporal, and so forth can be placed on the role operations. Although a user is active in the required role, the user can still be denied access as the user is yet to satisfy other constraints. For example, the user might be able to access only when he is accessing from location B. Thus, during role activations the system can check for these required constraints, but should notify the user without information leak.

1. check for access based on current credentials i.e., active roles, authorized roles, etc; if allowed goto step 5; else goto next step;

2. check if the operation requested on the object can be performed by any of the user's assigned roles; if allowed send a NOTIFY/REQUEST to the user in a secure way to activate the any role that has the required permissions (i.e., step 4); else goto next step.

3. similar to the above, check for the delegated roles, constraints that need to be satisfied, etc.; if allowed goto step 4; else deny access;

4. send a NOTIFY/REQUEST to the user in a secure way.

5. allow the user access.

### 7.3.4  *Object Access Request* Handler

In this section, we provide algorithms to handle users' object access requests. First, we will analyze the input and output of the algorithm. The input consists of a $session \in \mathcal{S}$, an $object \in \mathcal{OBJS}$ and an $operation \in \mathcal{OPS}$. Furthermore, requests can have other free attributes $\mathcal{A}_i$ $(i = 1 \ldots n)$ that are required to satisfy other external constraints, if

---

**1 INPUT**: *session, operation, object*
**2 OUTPUT**: *result*:BOOLEAN
**3** *session* $\in \mathcal{S}$; *operation* $\in \mathcal{OPS}$; *object* $\in \mathcal{OBJS}$; $\mathcal{PA}$;
**4** Retrieve all active roles in *session* as $\mathcal{ACRS}$
**5 foreach** *role* $R_i \in \mathcal{ACRS}$ **do**
**6**     **if** *(operation, object, $R_i$)* $\in \mathcal{PA}$ **then**
**7**         **return** TRUE /* allow object access request */
**8 if** *No role in* $\mathcal{ACRS}$ *satisfies the condition* **then**
**9**     **return** FALSE /* deny object access request */

Figure 7.7. CheckAccess *without* Role Discovery in Core RBAC.

---

defined in the enterprise security policy. Once the input is received, it is processed and the algorithm takes one of the following actions;

1. ALLOW access,

2. DENY access, or

3. NOTIFY the user and request for role activation, or other constraint satisfaction.

### 7.3.4.1   CheckAccess without Role Discovery in Core RBAC

Algorithm shown in Figure 7.7 handles object access requests[6] in Core RBAC *without* role discovery. Lines 1 and 2 show the input and output of the algorithm, where the output is a binary (ALLOW/DENY) reply. Line 3 shows the sets used. In Line 4, all the roles that are active in the *session* are retrieved as $\mathcal{ACRS}$. Lines 5 through 7 check if any role $R_i \in \mathcal{ACRS}$ has the required permission, and if it has (i.e., Line 6), it allows the access request. If none of the roles from $\mathcal{ACRS}$ have the required permission (i.e., Line 8), it denies the access request.

### 7.3.4.2   CheckAccess with Role Discovery in Core RBAC

Algorithm for handling object access requests in Core RBAC *with* role discovery is provided in Figure 7.8. In this algorithm, object access request *histories* are maintained,

---

[6]Identical to CheckAccess function detailed in [12].

```
1  INPUT: session, operation, object
2  OUTPUT: result:BOOLEAN, por
3  session ∈ 𝒮; operation ∈ 𝒪𝒫𝒮; object ∈ 𝒪ℬ𝒥𝒮; req ∈ ℛℰ𝒬𝒮; por ∈
   𝒫𝒪ℛ𝒮;
4  Retrieve all active roles in session as 𝒜𝒞ℛ𝒮
5  Retrieve all assigned roles for session as 𝒜𝒮ℛ𝒮
6  foreach role Rᵢ ∈ 𝒜𝒞ℛ𝒮 do
7      if (operation, object, Rᵢ) ∈ 𝒫𝒜 then
8          RESET all session role requests in ℛℰ𝒬𝒮
9          return TRUE /* allow object access request */
10 if No role in 𝒜𝒞ℛ𝒮 satisfies the condition then
11     if (∃Rⱼ ∈ 𝒜𝒮ℛ𝒮∧ ∃req ∈ ℛℰ𝒬𝒮) then
12         return FALSE /* deny object access request */
13     foreach role Rⱼ ∈ 𝒜𝒮ℛ𝒮 do
14         if (operation, object, Rⱼ) ∈ 𝒫𝒜 then
15             ℛℰ𝒬𝒮 = ℛℰ𝒬𝒮 ∪ Rⱼ
16             return Rⱼ /* notify por */
17     if No role in 𝒜𝒮ℛ𝒮 satisfies the condition then
18         return FALSE /* deny object access request */
```

Figure 7.8. CheckAccess with RoleDiscovery in Core RBAC.

so that the user is *not* notified if the request for the same operation and object is more than once in a *session*. In addition, this algorithm notifies the users, the *first role* discovered.

Lines 1 and 2 show the input and output of the algorithm, respectively, where the output is *not* a binary (ALLOW/DENY) reply. As shown, it outputs a role *por* ∈ 𝒫𝒪ℛ𝒮 that needs to be activated. Line 3 shows the sets containing the inputs, roles that can be activated, and the requests. In Lines 4 and 5, all the roles that are active in the *session* are retrieved as the 𝒜𝒞ℛ𝒮, and all the assigned roles are retrieved as the 𝒜𝒮ℛ𝒮. Lines 6 through 9 checks if any of the role $R_i$ ∈ 𝒜𝒞ℛ𝒮 has the required permission. If any role $R_i$ has the required permission, it allows the access and removes previous access request history from ℛℰ𝒬𝒮 for the corresponding *session*, *operation*, and *object*. If none of the roles from the 𝒜𝒞ℛ𝒮 have the required permission (i.e., Line 10), it checks for the

$\mathcal{ASRS}$ through Lines 11 to 16. In Lines 11 and 12 it check for previous notifications, and if there are any notification it sends a denial message. Through Lines 13 to 16 it checks for the $\mathcal{ASRS}$, and discovers role $R_j$ that has the permission. Even though there can be more than one assigned role that can be activated, this algorithm notifies the *first* role discovered.

When none of the roles in the $\mathcal{ASRS}$ satisfies (i.e., Line 17), it denies the access request. The denial actually means that there are NO roles that can be activated to make this request happen. This is a much *stronger* denial than what current systems provide and this is what is expected by an intelligent system.

Let us consider the access requests by the user `Tom` from Section 7.2. When the user is requesting for the `totPur.xls`, the input to the algorithm (Figure 7.8) is $\{session_1,$ $totPur.xls, read\}$. $\mathcal{ACRS}$ will contain the role *Marketing Manager*, and the $\mathcal{ASRS}$ will contain the roles *Marketing Manager* and *Purchase Clerk*. After checking the *Marketing Manager*, in Lines 6 through 9, it checks for the roles in $\mathcal{ASRS}$. As there are no previous notifications, it checks the roles in $\mathcal{ASRS}$. As the role *Purchase Clerk* has the required permission, it notifies the user to activate the role *Purchase Clerk*. If the user does not activate the role and requests again, it will DENY the request.

### 7.3.4.3 CheckAccess with Role Discovery in Dynamic SoD Without Hierarchies

Figure 7.9 shows the algorithm that handles object access requests and discovers the roles in RBAC with dynamic SoD and without role hierarchies. In contrast with the previous algorithm, *history* of object access requests is *not* maintained. Thus, discovered roles are notified for every request, when active roles do not have the permission. Instead of notifying the first role discovered, the *entire set* of roles that can be activated are notified to the user.

```
 1 INPUT: session, operation, object
 2 OUTPUT: result:BOOLEAN, 𝒫𝒪ℛ𝒮
 3 session ∈ 𝒮; operation ∈ 𝒪𝒫𝒮; object ∈ 𝒪ℬ𝒥𝒮; 𝒫𝒪ℛ𝒮;
 4 Retrieve all active roles in session as 𝒜𝒞ℛ𝒮
 5 Retrieve all assigned roles for session as 𝒜𝒮ℛ𝒮
 6 foreach role R_i ∈ 𝒜𝒞ℛ𝒮 do
 7    if (operation, object, R_i) ∈ 𝒫𝒜 then
 8       return TRUE /* allow object access request */
 9 if No role in 𝒜𝒞ℛ𝒮 satisfies the condition then
10    foreach role R_j ∈ 𝒜𝒮ℛ𝒮 do
11       if (operation, object, R_j) ∈ 𝒫𝒜 then
12          /*check if R_j can be activated */
13          if R_j can be added to 𝒜𝒞ℛ𝒮 then
14             𝒫𝒪ℛ𝒮 = 𝒫𝒪ℛ𝒮 ∪ R_j
15    /* check if there is a role that can be activated */
16    if ({PORS} ≠ ∅) then
17       return {PORS} /* output roles that can be activated */
18    else /* if {PORS} = ∅ */
19       return FALSE /* deny object access request */
```

Figure 7.9. CheckAccess with RoleDiscovery in Dynamic SoD Without Hierarchies.

Lines 1 through 5 are the same as in the algorithm shown in Figure 7.8. But, this algorithm outputs the entire set of roles $\mathcal{PORS}$ that can be activated for gaining the required permission. Lines 6 through 8 allows the access when any of the role $R_i \in \mathcal{ACRS}$ has the required permission. If none of the roles from $\mathcal{ACRS}$ have the required permission (i.e., Line 9), it checks for the $\mathcal{ASRS}$ in Lines 10 through 15. In Line 11 it checks for the $R_j \in \mathcal{ASRS}$ that has the permission. It then checks whether the role $R_j$ that has the permission can be activated in that *session*. Whether a role can be activated in the presence of dynamic SoD can be checked using a function similar to the AddActiveRole() from [12]. After discovering that $R_j$ can be activated, it adds it to the list $\mathcal{PORS}$ (Line 14). (Note: Lines 11 and 13 can also be interchanged.) After all the roles $R_j$ are checked,

it sends notification if $\mathcal{PORS} \neq \emptyset$ (Line 16-17). When none of the $\mathcal{ASRS}$ satisfies (i.e., Line 18-19) the access request is DENIED.

### 7.3.5   Analysis of the Algorithms

Users need not know all the role-permission assignments, as the above introduced algorithms notify the user instead of providing binary replies. Algorithms presented above handle users' object access requests when enterprise security policies are defined using core RBAC or dynamic SoD relations without role hierarchies. Similar to the above, algorithms can be developed for discovering roles in the presence of 1) role hierarchies, 2) static SoD with/without hierarchies, 3) dynamic SoD with hierarchies, and 4) constraints based on time, context, history, event and session.

For instance, when hierarchies are present, algorithm discovering roles should utilize authorized role sets of the user. Similarly, other constraints such as time of the day, IP address, quotas based on bandwidth or time, and so on can also be supported seamlessly in *SmartAccess*. Even though we consider only the notion of basic permissions for role discovery, we can utilize abstract permissions.

INFORMATION LEAK AND $\mathcal{PLP}$: Confidentiality, one of the three main principles of information security, requires systems to prevent the disclosure of information by unauthorized accesses. Thus, whenever a user makes a access request, information should not be disclosed. For example, when notifying the user of the required role the system should not disclose the role information which the user does not have access. Notifications generated by the algorithms are based on the discovered roles, which in turn are the set of assigned or authorized roles for a user. Thus, users are not notified with roles that they cannot activate preventing any information leak. Furthermore, as the users are notified based on their assigned or authorized roles, it keeps the interaction simple. In general, notification requests allow users to be $\mathcal{PA}$ unaware, preserving $\mathcal{PLP}$.

Below we analyze some of the important factors associated with the role discovery algorithms.

### 7.3.5.1 Single Vs. Multiple Role Discovery

When discovering the roles, either the *first* role that has the required permission or *all* the roles that have the required permissions can be sent to the user. Even though the user can activate the *first* role discovered, it can surprise the user as the first role might not fit the users' current context. In contrast, discovering *all* the roles increases the computation time of the algorithm, but it allows the user to activate any discovered role. In general, it depends on the enterprise requirement whether to discover all the roles or to stop with the first one.

We have shown both kinds of role extractions in the above algorithms, where we loop through all the roles to find which roles have permissions. With the roles as abstraction, it is not possible to check the objects for finding what roles can perform what operations and intersect it with assigned roles set to extract those roles that need to be activated. On the other hand, as mentioned earlier, inferring from the set of discovered roles and sending only one role that is the most appropriate is nearly impossible.

In addition to the above ways of role discovery, all the roles can be discovered, but the user can be notified with the role that has the least permission set or the user can be notified with the all the roles ordered by the permission set. This way of notification is a better way for satisfying the $\mathcal{PLP}$.

### 7.3.5.2 Stateful Vs. Stateless

All the notifications sent to the users for object access requests are tracked using $\mathcal{REQS}$. Keeping track of requests for a particular session avoids the unnecessary role discovery and the duplicate notification generation. In contrast, the cost of role discovery increases as it has to maintain all the user requests in all the *sessions*. We have presented

algorithms following both the stateful and stateless model. Performance optimizations are possible by caching the session request history while checking for roles and permissions. While maintaining request histories, changes to role policies should be propagated so that access is not granted for an unauthorized user.

### 7.3.5.3 Local Vs. Global

When algorithms follow stateful model they can keep track of the requests either locally or globally. Requests can be maintained based on *users* or *sessions*, or for the entire system. Maintaining local lists based on the users might include requests from more than one session. As algorithms access these lists for every request during role discovery they should be synchronized, which in turn increases the notification generation time. Even though maintaining the requests based on the sessions increase the number of lists, it reduces the notification generation time. Similar to the user based lists, global lists also have similar set of problems.

### 7.3.5.4 Overhead and Complexity

Roles are discovered only when the access decisions are FALSE. Thus, when users have an active role that has the required permission (i.e., TRUE), the runtime of algorithms *with or without* role discovery is *same*. In case of vanilla algorithms that does not involve role discovery it is the responsibility of the users to activate required roles, either following several iterations or in the first attempt. However, when the roles are discovered in a transparent manner and shown to the user, users can activate and request again. Even though there is an associated overhead because of the role discovery, it still supersedes the vanilla algorithms (e.g., Figure 7.7) as it *reduces* the user response time and allow users to be $\mathcal{PA}$ unaware. As *all* the roles in the entire system are not checked the delay is minimized.

Below we briefly analyze all the three algorithms explained before. Let us consider;

- $|\mathcal{R}| = n$, where "| |" represents cardinality

- $|\mathcal{ASRS}| = assigned$; $|\mathcal{ACRS}| = active$

- $assigned \leq n$; $active \leq assigned$

- $discover = assigned - active$

- $|\mathcal{PORS}| \leq discover$; $0 \leq discover \leq assigned$

Runtime of all the three algorithms are same if access decisions are TRUE. The best case running time is $O(1)$, where the first role in $\mathcal{ACRS}$ has the permission.Worst case time is $O(assigned)$, where last role in $\mathcal{ACRS}$ has the permission and $active ==$ $assigned$. When we consider $active < assigned$, the worst case time is $O\left(active\right)$. We use probabilistic analysis for the average case, for which, we need to have some idea of the input distribution [160]. Since a role that fetches the required object for a users' request is one of the roles from $\mathcal{ACRS}$, we can assume a random order in which the roles are selected. Thus, by utilizing the equation 5.6 of [160], we can compute the average case running time of the algorithm as $O\left(\ln active\right)$.

Algorithms presented in Figures 7.8 and 7.9 discover roles. As roles are discovered only if the access decision turns FALSE, we analyze the time taken for role discovery. When considering the best case scenario for role discovery it is again $O\left(1\right)$ when the first role in $(\mathcal{ASRS} - \mathcal{ACRS})$ is notified, and it is $O\left(discover\right)$ when all the roles have to be discovered. On the other hand, worst case time for both the cases is $O\left(discover\right)$. Similar to our previous average case analysis, it takes $O\left(\ln discover\right)$ when the first role is notified. Thus, runtime overhead for both the algorithms will be $O\left(\ln discover\right)$ when the first role is notified and will be $O\left(discover\right)$ when all the roles have to be notified.

### 7.3.6 Requests Generation

Requests can be either role-requests or constraint satisfaction requests. When any one of the roles from $\mathcal{PORS}$ has to be activated or any other constraints have to be

satisfied by the user in order to allow an access request, requests are generated and sent to the user. Requests that are generated should be sent to the user in a secure way. In general, systems can use any means [161] to notify the user, but it should provide both data integrity and data origin authentication. Secure way of communication play a major role when enterprises utilize distributed access control.

### 7.3.7  Authorization Rule Server

Enterprise security policies are maintained in terms of authorization rules (refer Chapter 4) in *SmartAccess*. All the user initiated operations are captured as events. Occurrence of the event triggers an authorization rule that enforces the policy. Authorization rules can be either integrated into the underlying system or can be maintained outside the system. By maintaining them outside the system, the same set of rules can be used to enforce access control in diverse systems.

### 7.3.7.1  Approaches for Role-Based Authorizations

Systems (Chapter 4) enforcing RBAC should be generalized and be able to provide access control to diverse underlying systems (i.e., operating systems, databases, etc.) in a uniform and transparent way. Although the system that provide access control act as the reference monitor it should be loosely coupled with the underlying system. Below we analyze two kind of approaches that can be exploited for providing role-based authorizations.

**Integrated Role-Based Authorization:** With the integrated approach the underlying system should be open to modification to incorporate access control modules (or authorization rules). In other words, this approach assumes that the kernel of the underlying system is understood so that it can be modified. There are a number of advantages to this approach and they are summarized as follows; *1)* flexibility to fine tune the access control modules resulting in good performance. *2)* only the minimum

amount of code that is required can be added. *3)* additional functionalities can be easily incorporated. *4)* applications can be easily modularized and maintained. There are also many disadvantages to this approach and they are; *1)* requires access to the internals of the underlying system including the source code. *2)* it is not cost and time effective. *3)* it is not generalized as it is tightly coupled and customized.

**Mediated Role-Based Authorization:** As opposed to the integrated approach, in this approach the internals of the underlying system need not be accessed. It is assumed that the underlying approach provides some hooks that can be exploited for triggering events and invoking authorization rules. When this approach is employed, the mediator or agent acts as the access control reference monitor for the underlying system. Thus, conditions are evaluated in the mediator when there is a user request. The main advantages of the mediated approach are; *1)* provides access control in a generalized and uniform way. *2)* not customized to any underlying system. *3)* it can provide role-based authorizations to diverse domains and not domain-specific. *4)* cost and time effective.

**SmartAccess Role-Based Authorizations:** As shown in Figure 7.4 *SmartAccess* follows the mediated approach and handles all the access requests that are generated in the underlying system. Whenever the underlying system receives an user request it triggers an event which in turn invokes an authorization rule. Once the rule is triggered, it is executed, which in turn evaluates the necessary conditions and triggers an action which can be allow/deny/notify.

## 7.4 Summary

Abstraction provided by the roles allows them to own objects and allow users to shift roles. On the other hand, for accessing objects users have to know the relationship between roles and objects. *SmartAccess*, provided in this chapter, is a system that provides role-based authorizations and overcomes the problem of $\mathcal{PA}$ awareness. *Smar-*

*tAccess* is proactive in that it provides the necessary notifications to the user acting in anticipation of future problems that the user may face when he is requesting for access. We have presented algorithms for discovering roles and analyzed various factors associated with them. When the users' get a DENY from our algorithms it actually means that there are no roles that can be activated to make this request happen. This is a much *stronger* denial than what the current systems provide. Even though we have explained the algorithms in the context of *SmartAccess*, these algorithms are general-purpose and can be used in any system that enforces RBAC. Notifications provided allow users to concentrate on what data needs to be accessed rather than the roles that are required for access, and thus preserving principle of least privilege. Roles are disclosed to the user without any information leak. Although role discovery has its associated overhead with respect to system response time, it reduces user response time, increases user satisfaction or usability and supersedes other algorithms that provide binary replies and follow the *human-active, system-passive* model.

## CHAPTER 8

## NOVEL APPLICATIONS

In this chapter, various novel applications are introduced for demonstrating the applicability of the results obtained in this thesis. First, we discuss a novel application where the role discovery (refer Chapter 7) in RBAC was leveraged for providing fine grained access control. Second, we discuss an advanced information filtering application where information from text streams are filtered. This application uses the interval-based semantics developed for complex event operators. Although this application is not directly relevant to RBAC domain it can be used for various applications including tracking of information flow among terrorist outfits. Finally, we discuss an event stream processing system that utilizes the attribute-based semantics of events ($E_{expr}$ detailed in Chapter 5) developed as part of the Snoop generalization. Furthermore, we show how the event stream processing can be utilized for network fault management. From the above it is evident that the results obtained from this thesis not only caters to the security domain, but to a larger class of applications.

### 8.1  Role-Based Security for Web Gateways

Efficient and effective web gateways or proxy servers are important to control the access privileges of users and to protect the private networks that are connected to the Internet, thus providing a productive and a safe web environment. Access control in the form of complex access rules based on the users or the user sets (groups) has been studied extensively. The objective of this work is to provide role-based (RB) security for the web gateways utilizing the Role-Based Access Control (RBAC). RB security reduces the administrative burden, provides fine grained access control, and supports various

constraints such as, context-aware and temporal, seamlessly. In this section we elaborate on the problems, issues that need to be addressed, and our approach for providing RB security for web gateways by leveraging the flexibility and expressiveness of RBAC. Our approach enables the proxy server to act smarter, rather than just allow or deny access based on access rules, meanwhile preserving the principle of least privileges.

### 8.1.1 Introduction

With the ever growing impact of the Internet on our daily activities, security and productivity have a greater role to play. Online shopping, stock trading, email communication, credit card management, job search, news, e-greetings, online auction, and bandwidth intensive applications (i.e., streaming media, online games, and MP3 downloads) are some of the capabilities of the Internet. When employees engage in any of these activities during the `work hours` it *threatens* the enterprise productivity and security, and makes the web a less safe environment for enterprises. Recent surveys [162, 163, 164, 165] show that the enterprises are losing 30-40% of their productivity every year due to the non-work (or non-business) related surfing during work hours. In addition, with its ever expanding networks, the Internet also raises various security concerns and legal liabilities of enterprises.

However, there are number of situations where certain legitimate users/departments in enterprises require access to a subset of the above mentioned activities. For instance, training and marketing departments might require streaming media access, the human resources department may require access to job search sites, and the high level users may also require some of these accesses. Web gateways or proxy servers control the access privileges of users and protect private networks that are connected to the Internet. Current gateways provide access control using complex access rules based on users or user sets (groups) [116, 117, 118, 119, 120, 121]. The number of access rules used are large, as

they are based on the number of users or groups that are spread over multiple repositories. Rules can be customized in order to support various policies such as, time/resource based quotas, network/workstation based, IP address based, and so on. Writing these rules based on the users or groups is a tedious job for administrators, and the number of rules increases with the number of users and groups.

RBAC, where object accesses (or operations) are controlled by roles (or job functions) in an enterprise rather than a user or group, has established itself as a viable alternative to traditional discretionary and mandatory access control. Employees can be assigned to more than one role in an enterprise where RBAC is employed. For example, user `Tom` can be assigned to roles *project manager* and *software developer*. Employees are not required to be active in all the assigned roles at all times (i.e., `Tom` can just be active in *software developer*) preserving the principle of least privileges (i.e., at any given point in time no additional permissions are made available than required) and this introduces some problems while providing RB security. Suppose an employee (e.g., `Tom`) needs to perform some activity, it might not be legitimate based on the current set of active roles (e.g., *software developer*) but can be legal based on some inactive roles (e.g., *project manager*). In addition, users might get access delegated for a session from other authorized users. Problems and issues that need to be addressed are elaborated in section 8.1.2. In essence, systems that provide RB security should address all the problems and issues, and be able to make additional decisions rather than just making traditional binary decisions (i.e., allow or deny).

In this work we leverage the flexibility and expressive power of RBAC for providing RB security in web gateways. To the best of our knowledge this is the first paper to provide RB security for web gateways or proxy servers. The previous work deals with providing RB security for web servers [122]. RB security provides fine grained access control, reduces administrative burden, and supports various constraints such as context-

aware, and temporal. *SmartGate*, discussed in this work overcomes the problems outlined above and those in section 8.1.2 by using a smart push-pull approach.
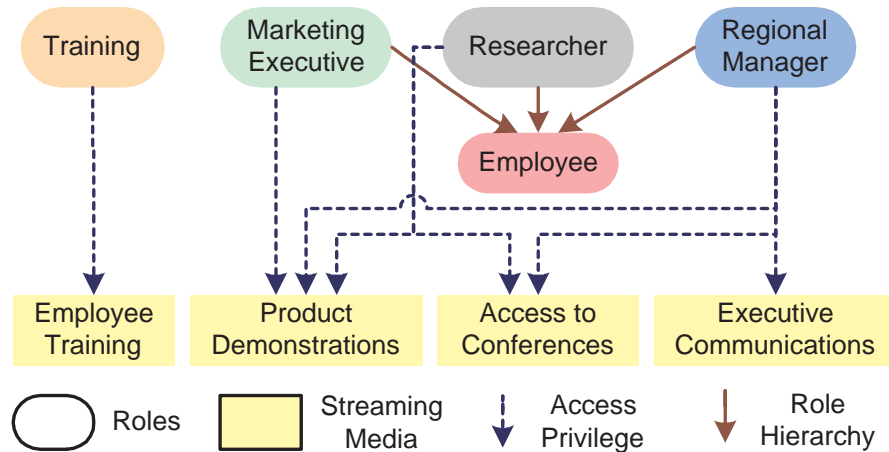


Figure 8.1. Role Hierarchy with Access Policy.

## 8.1.2   Problems and Issues

Traditionally, web gateways were used for caching, to minimize the response time of user requests. Over the years additional functionalities were supported such as controlling the user or group accesses based on rules. Designing and specifying these rules are cumbersome as these are based on specific users or user sets. On the other hand, RBAC reduces the administrative burden and increases the cost savings of enterprises in authorization management. This section discusses the issues that arise while providing RB security. Formulation of access policies using RBAC, and how roles are checked for users and permissions are explained in Sections 8.1.4.2 and 8.1.4.3 respectively. Supporting RB security as a middle-ware rather than a tight integration with proxy servers is beneficial, as it can be used seamlessly with other underlying systems such as firewalls, and so on.

Key findings in a recent study [162] show that 21% of Americans watch or listen to Internet broadcasting monthly as compared to 10% in January 2000. Moreover, various studies and observations show the increasing use of Internet for non-business usage during work hours. Nonetheless, streaming media is effectively used by enterprises for various necessary purposes and some of them are

1. `Training:` specialized training are provided to employees,

2. `Web Casts:` providing important events, seminars, etc.,

3. `Product Demos:` demonstrating products of the enterprise and its competitors,

4. `Presentations:` streaming conference presentations, and

5. `Communications:` used by enterprise executives for effective discussions.

Besides the streaming media, enterprises may need to restrict access for job searches, online shopping, and so forth. They may also need other kind of application proxies which restricts the *put* command in **FTP** accesses to avoid information leak, etc. In this work our examples are based on the web proxy, but our approach is applicable to applications and other proxies as well. Key issues that need to be addressed when RB security is introduced in web gateways are explained below. Consider an enterprise *SmartWeb* that needs to support streaming media for some of its employees as shown in Figure 8.1. Roles *Marketing Executive*, *Researcher*, *Regional Manager*, and *Training* have access to the streaming media. Access permissions for all the roles are shown in Figure 8.2. The role *Employee* is a junior role to the first three roles.

As shown in Figure 8.2 the role *Employee* cannot access any of the streams, whereas the stream `Employee Training` can be accessed only by the role *Training*. Let us assume that the user `Tom` is assigned to roles *Marketing Executive* and *Regional Manager*. When a user access request arrives, access policies are checked for permissions based on the users' active role. Following are the requests from the user `Tom` when he is only active in the *Marketing Executive* role;

1. `Tom` requests for `Product Demonstrations` and the access is granted.

2. The role *Marketing Executive* has no permission for `Access for Conferences`. Even though he can request for the same as he is assigned to the role *Regional Manager*, he will not be permitted as he is currently not active in the role.

3. Let us assume that the role *Employee* has permission to access a generic object `XY`. Even though he cannot access the generic object `XY` directly, he can be granted the required permissions as he is active in a role that is senior to the role *Employee*.

4. `Tom` is denied access for the stream `Employee Training`. The stream can be accessed by the users active in the role *Training*. This role is required by all the employees in order to complete a training. On the other hand, it is appropriate to delegate this role temporarily to the user for the duration of training by some authorized authority.

| Roles<br><br>Stream | Employee | Researcher | Marketing Executive | Regional Manager | Training |
|---|---|---|---|---|---|
| Employee Training | N | N | N | N | Y |
| Product Demonstrations | N | Y | Y | Y | N |
| Access to Conferences | N | Y | N | Y | N |
| Executive Communications | N | N | N | Y | N |

Y - Access Allowed     N - Access Denied

Figure 8.2. Streaming Access Privileges.

From the above requests it is evident that only the first request is granted, directly. Although the access for the remaining is denied directly, it is possible to provide the same indirectly. These requests are crucial in the case of web gateways as users can try to perform legitimate access even when they are not active in a role. Thus, RB

security enabled web gateways *cannot* just provide binary replies (i.e., either allow or deny), but need to address additional issues. A straightforward manner to do so is by activating all the roles that are assigned to the user, but it *contradicts* the principle of least privileges and increases the security risks. However, granting accesses indirectly requires special attention. *SmartGate* provides RB security and addresses the problems and issues discussed above.

### 8.1.3  Role-Based Security

This section addresses the issues that arise while providing RB security. Formulation of access policies using RBAC and the checking of the roles are explained in Sections 8.1.4.2 and 8.1.4.3, respectively. With traditional web gateways, object requests by the users are granted if all the access rules are satisfied, else they are denied. In the case of RB security, user's access requests cannot be denied directly as discussed in section 8.1.2. Thus, when a user request arrives, the system that provides RB security should make additional decisions and not just match access rules.

Let us take our previous example where user `Tom` is assigned to roles *Marketing Executive* and *Regional Manager*, and is active only in the former role. All the issues that need to addressed are explained using the role hierarchy shown in Figure 8.1 and access permissions shown in Figure 8.2. Whenever `Tom` requests for the stream `Product Demonstrations` he is granted access for the same as he is active in role *Marketing Executive* that has the required permissions. When he requests for the stream `Access for Conferences` the system checks his active roles (i.e., *Marketing Executive*) for permissions. As role *Marketing Executive* does not have the required permission the system should make additional decisions. It should check if any of his assigned roles/junior roles have the required permissions. In other words, the system should *pull* the information from the server where roles are stored and check for permissions. In our example, he is

assigned to role *Regional Manager* that has the necessary permissions. Thus, the system should convey to the user in a secure way that he should be active in certain roles in order to access the object. If the user activates the role and requests again, then he can be granted the required access.

However, there can be some other object requests to which he has indirect permissions. As mentioned in section 8.1.2 he can request for an object `XY` or a stream `Employee Training`. He can have indirect access permissions via one of the junior roles or role delegation. When such requests come, the system should verify whether any of the junior roles has the required permissions and grant the access accordingly. In the case of object `XY` he should be granted access as the junior role *Employee* has the required permissions. When there is a role delegation, then the system should check the origin authentication and integrity of the delegation and grant the required access. Let us assume that the permission for accessing stream `Employee Training` is delegated to `Tom` by some authorized authority. In this case, the role delegation is verified and if it is valid then user `Tom` is given the required access. Thus, a smart push-pull approach, where the user identity is pulled from the server for checking and appropriate messages are pushed to the user in a secure manner, is more appropriate for providing RB security for web gateways. *SmartGate* discussed in the next section provides RB security using a smart push-pull approach.

### 8.1.4 SmartGate Architecture

*SmartGate* is an access privilege control system that provides role-based security for web gateways. *SmartGate* aims at investigating the support for role-based security using smart push-pull. Figure 8.3 summarizes the high level architecture of *SmartGate*. The *SmartGate* architecture is described in the following sections.
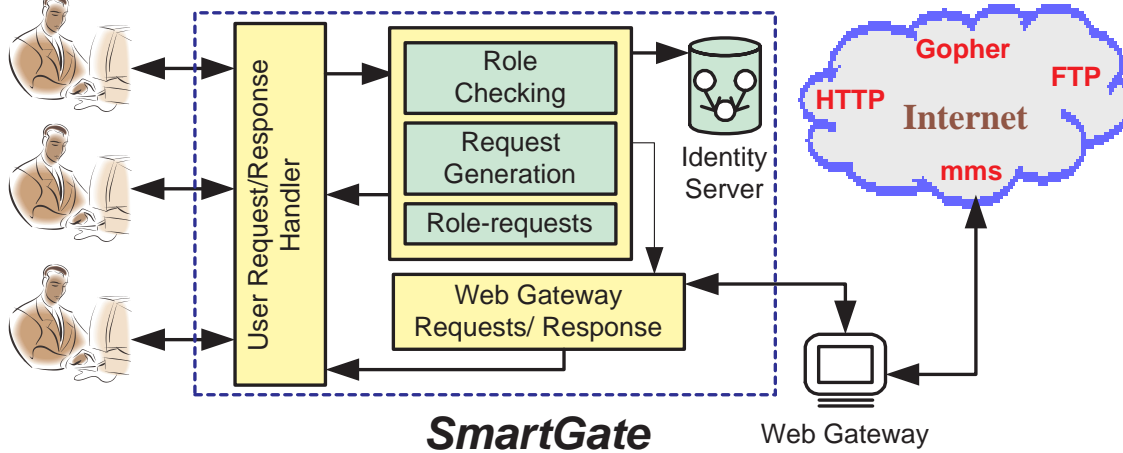
Figure 8.3. SmartGate Architecture.

### 8.1.4.1 User Request/Response Handler

Users (or user applications) request access to perform certain activities in the web. For example, a client browser may ask permission to access a particular URL using the HTTP protocol on behalf of the user who is logged in that machine. As shown in the Figure 8.4(a) web gateways accept the request from user and match it against the access rules and either allow or deny the request. *SmartGate* shown in the Figure 8.4(b) accepts the user request and checks for permission. As a result of the check it can take three actions as opposed to the traditional allow or deny and they are 1) allow access, 2) deny access, and 3) role-request (or ask for) if one of the assigned role has the necessary permission that is not currently active.

In *SmartGate*, user's access request comprises of three elements:

- user identity ($\mathcal{UI}$) (e.g., user name).
- other attributes (specified in the enterprise policy) such as ($\{\mathcal{ARS}|\mathcal{DRS}\}$ [$,\mathcal{IP}, \mathcal{T}, ...$]) where $\mathcal{ARS}$ is the current set of active roles, $\mathcal{DRS}$ is the current set of delegated roles, $\mathcal{IP}$ is the IP address, $\mathcal{T}$ is the time of occurrence.

- object for which the access is requested ($\mathcal{O}$) (e.g., URL).

In a request, $\mathcal{UI}$ is required along with either the $\mathcal{ARS}$ or the $\mathcal{DRS}$ while the other parameters are optional and are based on the enterprise policy. Delegate role set $\mathcal{DRS}$ is always accompanied with the role-delegate and can be verified based on the issuing authority (refer Section 8.1.4.4). Once the user request is received, this handler forwards it to the role checking module.
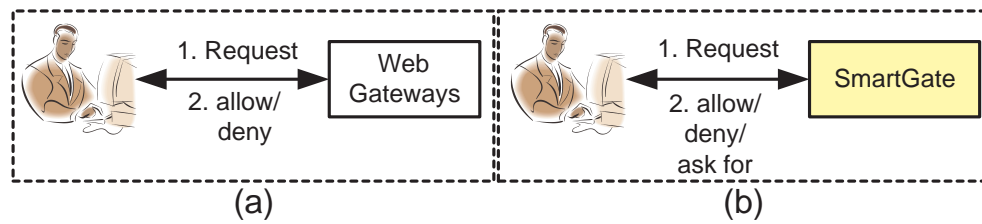


Figure 8.4. User Request/Response (a) General; (b) SmartGate.

After the request is processed, results are passed on to this handler which in turn forwards it to the user. As shown in the Figure 8.4(b), response from the handler can be an outcome of the request (i.e., allow), denial message, or request for some conditions that need to be satisfied. These are further elaborated in the following subsections. In order to simplify the discussions, we are not addressing the issue of how the user identity is passed as this can be done in different ways.

### 8.1.4.2 Identity Server

Identity server is responsible for checking the existence of the user identity and the corresponding access policy. It supports enterprise policies based on RBAC. Event-Based RBAC allows enterprises to form access control policies based on RBAC and its extensions (refer Chapters 4, 5, and 6). Figure 8.5(a) shows the roles and their associated permissions. For example, the role *researcher* can access objects in categories 2, 3, and 5

whereas the role *employee* can just access category 6. Each category can contain many types and are shown in Figure 8.5(b). For example, category 5 contains all the news and job websites whereas category 2 contains the streaming media corresponding to the `Product Demonstrations`.

| Role | Permission | | Category | Type |
|---|---|---|---|---|
| | Objects | Operation | 1 | Employee Training |
| Marketing Executive | Category (2) | Allow | 2 | Product Demonstrations |
| Regional Manager | Category (2,3,4,5) | Allow | 3 | Access to Conferences |
| Researcher | Category (2,3,5) | Allow | 4 | Executive Communications |
| Training | Category (1) | Allow | 5 | News, Jobs |
| Employee | Category (6) | Allow | 6 | Internal Websites |
| (a) | | | (b) | |

Figure 8.5. Assignments (a) PA; (b) Categories.

Figure 8.1 shows roles and their corresponding hierarchies where roles *Marketing Executive*, *Regional Manager*, *Researcher* and *Training* are senior to role *employee*. With hierarchical RBAC [12] all objects that can be accessed by the junior role *employee* are also accessible by its senior roles. Thus, all of the above mentioned four roles can access category 6. Each role is assigned to a set of users, and the Figure 8.6(a) shows users and their assigned roles. For example, user `Tom` is assigned to roles *Marketing Executive* and *Regional Manager*. Thus, user `Tom` has the required permissions to access categories 2, 3, 4, and 5 directly, and category 6 indirectly via role hierarchy. Identity server can be expanded with separation of duty constraints or based on other attributes, for example, certain roles can access certain objects only from 9 a.m. to 5 p.m. on weekdays.

When the input $(\mathcal{UI}, \{\mathcal{ARS}|\mathcal{DRS}\}, \mathcal{O})$ is provided, assigned roles $\{\mathcal{AR}\}$ corresponding to a user identity $\mathcal{UI}$ can be retrieved from the server. For example, set of

| User | Assigned Role |
|------|---------------|
| Tom | Marketing Executive |
| Tom | Regional Manager |
| Jim | Employee |

(a)

| User | Active Role |
|------|-------------|
| Tom | Marketing Executive |
| Jim | Employee |

(b)

Figure 8.6. Users and Roles (a) Assigned; (b) Active.

assigned roles {*Marketing Executive* and *Regional Manager*} is retrieved by providing `Tom` as the input. Figure 8.6(b) shows the user identity and its active roles. As illustrated, user `Tom` is active only in the role *Marketing Executive* and not in the role *Regional Manager*. Thus, access is directly `allowed` when the input ($Tom$, {$Marketing\ Executive$}, *Product Demonstrations*) is provided to the server, but not ($Tom$, {$Marketing\ Executive$}, *Access To Conferences*).

### 8.1.4.3 Role Checking

Values corresponding to ($\mathcal{UI}$, {$\mathcal{ARS}|\mathcal{DRS}$}, $\mathcal{O}$) are received by this module from the user request/response handler. Once received these values are passed to the identity server. If the user has the necessary permission then the role checking module forwards the request to the Web Gateway Request/Response module. Role checking module receives a deny access from the identity server when the user does not have the necessary permission either with his active role set {$\mathcal{ARS}$} or delegate role set {$\mathcal{DRS}$}. In this case it might be possible that the user may still have the permission based on the assigned roles but may not be active in those roles preserving the principle of least privilege. Thus, the user should be notified regarding the current state of denial and the conditions that need to be satisfied in order to grant access rather than completely denying his request.

Instead of just denying the request this module performs the above required actions using a smart pull from the identity server. For example, if the values corresponding to $(\mathcal{UI}, \{\mathcal{ARS}|\mathcal{DRS}\}, \mathcal{O})$ are $(Tom, \{Marketing\ Executive\}, Access\ To\ Conferences)$, then identity server denies the access as the role $Marketing\ Executive$ does not have the permissions for $Access\ to\ Conferences$. Based on the result, role checking module performs a smart pull and retrieves the assigned roles $\{Marketing\ Executive$ and $Regional\ Manager\}$ from the identity server based on the user identity `Tom` and checks if the assigned roles have the required permissions. If the user's assigned roles has the required permission, then it forwards the corresponding information to the role-request generation module. In our example, the role $Regional\ Manager$ has the required permission. On the other hand, if the user's assigned roles do not have the required permissions then the user can be either allowed if he provides a role-delegation or can be denied. In both cases it is propagated to the user request/response handler, which in turn sends the denial message to the user. For example, if $(Jim, \{Employee\}, Product\ Demonstrations)$ is provided, then this module returns a `deny or false` to the user request/response handler. The algorithms for discovering roles are detailed in Chapter 7.

### 8.1.4.4 Role-request Generation

When one of the roles that has the required permissions should be activated by the user in order to allow an access request, role-requests are generated and are pushed to the user. Role-requests that are generated are pushed from the $SmartGate$ to the network user. It can be done in many ways, such as, using digital signatures [161], etc. However, the mechanism chosen should provide both data integrity and data origin authentication.

**Role-request flag:** Role-request generation creates the request and sends it to the Request/Response handler which forwards it to the user. In some cases, the user might not want to activate a role or he might not have a delegate certificate, in which

case the user issues the same request again. Thus, when the same user request is received again, the role checking module should not check for access if the values corresponding to $\{\mathcal{ARS}|\mathcal{DRS}\}$ are not updated after the role-request is sent. In this case, role checking is not performed in order to avoid cycles, and is carried out by setting an access flag for the user request.

### 8.1.4.5 Web Gateway Request/Response

This module interacts with the current web gateways or proxy servers that reduces the user response time by providing caching. When the input is ($Jim$, $\{Employee\}$, $Internal\ Websites$), the role checking verifies and forwards the request to this module. If the requested object is available, this module gets the object from the web gateway. Objects requested can be either in the cache or not in the cache and it depends on the web gateway on how to retrieve it. In this example, the requested $InternalWebsite$ is retrieved and given to the user request/response handler to be forwarded to the user `Jim`.

### 8.1.5 Future Directions

$SmartGate$ is generalized and can support RBAC Standard and other constraints such as time of the day, IP address, quotas based on bandwidth or time, and so on. All of the above functions can be supported seamlessly in $SmartGate$ as it takes a general approach for providing RB security for web gateways. In general, any type of constraint can be supported when the required attribute values are provided. For example, assume that user `Jack` is allowed to perform any action only between 9.00 a.m. to 5.00 p.m. on weekdays. When Jack requests at 6.00 p.m. using the following values ($Jack$, $\{\mathcal{ARS}|\mathcal{DRS}\}$, $\mathcal{O}$), then identity server checks the enterprise policy and denies access.

There may be a slight increase in user response time as $SmartGate$ checks the roles for every request that comes from the user in addition to the normal access rule checking in traditional web gateways. Nevertheless, RB security provides fine-grained access poli-

cies and reduces administrative burden while formulating access policies. Performance optimization is possible by caching the user request history while checking for roles and permissions. While maintaining the user request history, if there is any change in the role policy then it should be propagated to the role checking module so that the access is not granted for an unauthorized user. On the other hand, objects can be in different granularity. For example, if streaming media can be categorized then it can be assigned separate roles or all the streaming media can be assigned to a single category. Role delegation is important for specialized tasks like employee training, where roles are delegated occasionally and exclusively depending on enterprise policies.

### 8.1.6 Summary

In this section we have provided *SmartGate*, a smart push-pull approach for supporting RB security in web gateways. *SmartGate* has enhanced the access control decisions (i.e., allow, deny, ask for) from traditional web gateways or proxy servers that provide just binary access decision (i.e., either allow or deny). In current systems, access rules are based on the users or user sets, which are complex and large in number. RB security realizes the concept of RBAC; provides a practical and elegant mechanism for controlling accesses in web gateways. By leveraging RBAC, the number of access rules and their complexity is greatly reduced in *SmartGate*, thus reducing the administrative burden. Moreover, there are additional advantages such as seamless constraint specification (e.g., time of the day, quotas based on bandwidth or time, IP address, Location, etc.), and fine grained access control decisions. *SmartGate* uses smart pull for getting the required information from identity server and pushes the role-requests to the network users. Providing RB security by leveraging RBAC increases the level of security and productivity, creates a more secure perimeter around enterprise networks, and makes web a safer environment.

## 8.2 Advanced Information Filtering

Information filtering includes monitoring text streams to detect patterns that are more complex than those handled by search engines. Text stream monitoring and pattern detection have far reaching applications such as tracking information flow among terrorist outfits, web parental control, and business intelligence. InfoFilter, a content-based information filtering system, presented in this section, allows users to specify complex patterns and detects these patterns in incoming text streams from various sources such as news feed, emails, web pages and caption text from streaming videos. Complex patterns such as combinations of sequential, structural patterns, wild cards, word frequencies, proximity, Boolean operators and synonyms are formulated using the expressive pattern specification language, PSL, proposed in this section. Once specified, these complex patterns are detected using a data flow paradigm over Pattern Detection Graphs (PDGs).

### 8.2.1 Introduction

Recent advancements in computing have led to a digitized world with an ever increasing data available online. Users often find themselves swamped with colossal amount of information while retrieving `task relevant` data. Information filtering is the process of extracting relevant or useful portions of information/documents from large data repositories or continuous streams of textual data based on relatively static user patterns (or queries). In this process, expressiveness of pattern (or query) specification by a user and its detection play a significant role. Typically, a user profile in the form of one or more patterns is created and submitted to the system, and patterns in such a profile are filtered from the incoming text streams. In order to extract useful or meaningful information, the user needs to have the flexibility to specify complex and meaningful patterns using an expressive pattern specification language.

Based on the similarity between information filtering and information retrieval [166, 167, 125], most of the existing filtering tools such as personalized information filtering systems use Information Retrieval Query Languages (IRQLs) [166] for user query specification. Thus, this work has been motivated by several observations on the IRQLs and current query languages and the amount of expressiveness or flexibility desirable in user pattern specification. As observed from the characteristics of the query languages proposed in the literature, they support single-word, Boolean, context, natural language, pattern matching and structural queries, and their compositions in a very restricted manner.

Consider a real world example where a federal agent is tracking terrorist-related information streaming from various resources. He/she is interested in the occurrence of the word "bomb" followed by the word "ground zero" occurring twice, along with the word "automotive" or its synonyms (i.e., *(("bomb" FOLLOWED BY "ground zero") occurring twice) AND "automotive" (or its synonyms)*). This pattern contains keywords, sequence (FOLLOWED BY), phrase, frequency, synonyms, and a Boolean operator. This pattern cannot be expressed using current query specifications as they do not support the following: 1) quantification of multiple occurrences (or frequency) of patterns and complex compositions, and 2) a user cannot include synonyms in the pattern, and is required to explicitly list all the synonyms as separate patterns . Thus, current query languages are quite restrictive in their expressive power and need to be extended and generalized to address the specification of meaningful complex user patterns. Pattern specification alone does not suffice as detecting these complex patterns is equally important in order to use these systems for real-world applications. The aim of this work is to overcome the limitations mentioned above.

### 8.2.2  User Specification

In InfoFilter, users can specify simple and complex patterns using the Pattern Specification Language, PSL. It supports the following operators and options: frequency, synonyms, sequence, Boolean operators, structural, wild card, and proximity. Furthermore, any arbitrary complex pattern can be composed using the above operators. Some of the operators in PSL have some similarities with event specification languages (refer Chapter 3) used for the specification of events. Even though some of the operators are similar, semantics of pattern operators are different as it includes the notion of proximity, which is crucial in information filtering. In addition, PSL supports pattern operators such as regular expressions, frequency and synonyms. Also, PSL allows composition of all the above operators for specifying complex patterns.

### 8.2.2.1  Pattern Types

In PSL, a pattern $\boldsymbol{P}$ is formally represented as $P_i^j$, where $i$ is the pattern identifier and $j$ is the instance of the pattern identifier. A pattern $\boldsymbol{P}$ is a function that maps from the offset interval domain onto the boolean values, "True/False" corresponding to the occurrence or non-occurrence of the pattern. $O_s$ is the start offset, and $O_e$ is the end offset of the pattern, where offset is the position of the word relative to the beginning of the text stream. For example, in the phrase "user pattern", if "user" occurs at offset 50 then "pattern" occurs at 51, and $O_s$ is 50 and $O_e$ is 51.

According to the semantics of PSL, patterns are classified as:

**Simple patterns**: These are the basic building blocks and can be either *System-defined* (i.e., built into the system), or *User-defined*. System-defined patterns are predefined and they correspond to the structural elements present in text streams, such as the beginning of a sentence, a paragraph, or a document/stream. For example, two system-defined patterns *BeginPara* and *EndPara* are used to define the beginning and

end of a paragraph. On the other hand, possible user-defined patterns include a single word or any of its synonyms (e.g., *"filter"*), multi-word or phrases ( e.g., *"information filtering system"*), or simple regular expressions (e.g., *"filter\*"*).

**Complex Patterns**: These are composed of simple patterns, complex patterns, pattern operators and options. PSL provides a comprehensive set of pattern operators and they are: Boolean (OR, NOT, NEAR), sequence (FOLLOWED BY), structure (WITHIN), frequency (FREQUENCY), proximity (NEAR/N, FOLLOWED BY/N) and the option synonym (SYN).

### 8.2.2.2 Operators for Pattern Specification

Semantics of PSL operators and options are explained below (formal definitions for these operators and options are provided in [168]).

**OR**: Disjunction of two simple or complex patterns $P_1$ and $P_2$, denoted by $(P_1 \ OR \ P_2)$, occurs when either $P_1$ or $P_2$ occurs. For example, *"information" OR "filtering"* will be detected when either one of the keywords occurs. Since simultaneous occurrences of the same patterns are not possible in a stream (essentially a sequence), exclusive OR semantics is used.

**NOT**: Non-occurrence of the simple or complex pattern $P_2$ in the range formed by the end offset of $P_1$ and the start offset of $P_3$, where $P_1$ and $P_3$ can also be simple or complex patterns, is denoted by $(NOT \ [/F](P_2)(P_1, P_3))$. "F" indicates the minimum number of occurrences and its default value is 1. For example, *NOT ("filtering")("information", "retrieval")* will be detected whenever *"information"* is followed by *"retrieval"* without the word *"filtering"* occurring at least once in between them.

**NEAR**: Conjunction of two simple or complex patterns $P_1$ and $P_2$, denoted by $(P_1 \ NEAR \ [/D] \ P_2)$, occurs when both $P_1$ and $P_2$ occur, irrespective of their order of occurrence. "$D$" is the maximum distance allowed between the patterns $P_1$ and $P_2$.

Default value of "$D$" is the scope of the operator (which can be the entire document), and it refers to the *AND* operator of the Boolean model. The minimum value of D is 1. For example, *"information" NEAR/10 "filtering"* will be detected whenever both these words co-occur within a distance of 10.

**FOLLOWED BY**: Sequence of two simple or complex patterns $P_1$ and $P_2$, denoted by *($P_1$ FOLLOWED BY [/D] $P_2$)*, occurs when the occurrence of $P_1$ is followed by the occurrence of $P_2$. The end offset of $P_1$ is less than the start offset of $P_2$; that is, the occurrence interval of $P_1$ and $P_2$ should not overlap. "$D$" is the maximum distance allowed between the two patterns $P_1$ and $P_2$. If "$D$" is not specified, the distance is bounded by the scope of the operator (can be the entire document). If the value of "$D$" is 1 (minimum value), this indicates that the patterns $P_1$ and $P_2$ form a phrase. For example, *"information" FOLLOWED BY /10 "filtering"* will be detected whenever the word *"information"* precedes *"filtering"* within a distance of 10 words.

**WITHIN**: Occurrence of a simple or complex pattern $P_2$ in the range formed by the end offset of the pattern $P_1$ and the start offset of $P_3$, denoted by *($P_2$ WITHIN ($P_1$, $P_3$))*. The pattern is detected each time pattern $P_2$ occurs in the range defined by patterns $P_1$ and $P_3$. For example, *"information filtering" WITHIN (BeginPara, EndPara)* will be detected whenever the phrase *"information filtering"* occurs within a paragraph. When an expression is specified without a system-defined pattern, the default structure (e.g., a document) is used as the default. This operator is crucial while expressing the scope of the stream being processed.

**FREQUENCY**: Multiple occurrences of a simple or complex pattern that exceed or equal to $F$, denoted by *(FREQUENCY /[F] (P))*. A pattern $P$ is detected each time $P$ occurs at least F times, where "$F$" is the minimum number of occurrences specified by the user. The default value of $F$ is 1, which is the minimum value. All the occurrences that are used for detection should be disjoint (i.e., the end offset of each pattern occurrence

should precede the start offset of the subsequent pattern occurrence). The same set of occurrences is not used for detecting multiple patterns. For example, *FREQUENCY/10 ("information filtering")* will be detected whenever the phrase *"information filtering"* occurs at least 10 times. Frequency can be applied to any pattern expression.

**SYN**: This is an option and is specified along with a single-word pattern (currently), denoted by *(P [SYN])*, to indicate multiple single-word patterns that have the same meaning, in a succinct manner. In PSL, specifying a single-word pattern with SYN option is equivalent to specifying $N$ simple patterns that carry the same meaning (synonyms) as the original pattern. For example, if you specify the word *"bomb"[SYN]* is equivalent to specifying *"bomb" OR "explosives device" OR "weaponry" OR "arms" OR "implements of war" OR "weapons system" OR "munition"* . If any of these words or phrases appears in the text, the pattern *"bomb"[SYN]* is detected. This option adds simplicity and flexibility to the specification of single-word patterns. The same is true for complex patterns with embedded synonym specification, e.g. *"Bomb"[SYN] NEAR "Ground Zero"*.

Using the above operators, users can specify complex and meaningful patterns (refer [168] for more patterns). For instance, a complex pattern *("bomb" occurring prior to "ground zero" occurring twice, with a single occurrence of "automotive" or its synonyms)*, can be specified using the PSL as

Pattern $P_1$ = "bomb" FOLLOWED BY "ground zero"

Pattern $P_2$ = FREQUENCY /2 ($P_1$)

Pattern $P_3$ = $P_2$ NEAR "automotive" [SYN]

### 8.2.3 InfoFilter

InfoFilter analyzes text streams based on the content and structural information, and notifies users when their patterns of interest are detected. Figure 8.7 shows various
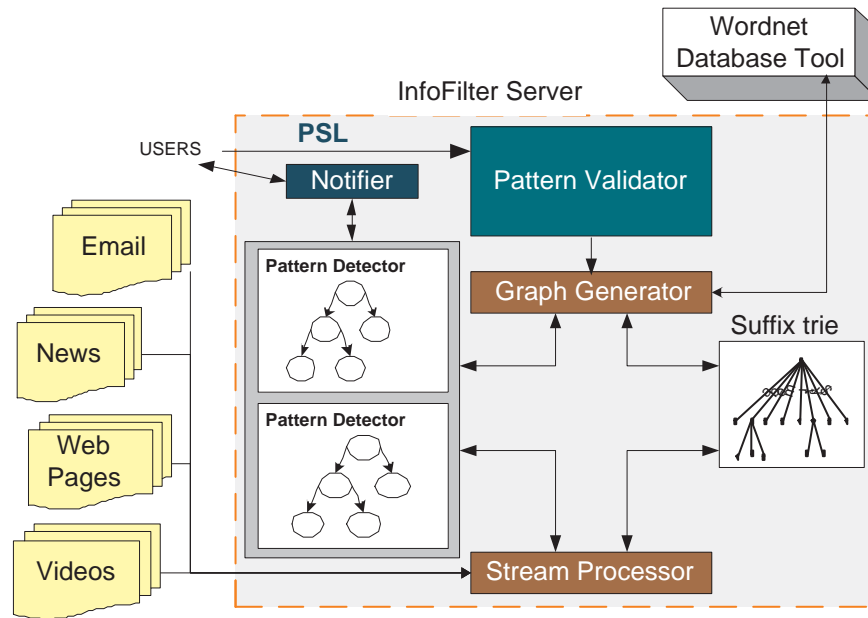
Figure 8.7. Architecture of InfoFilter.

modules of the InfoFilter server: Pattern Validator, Graph Generator, Stream Processor, Pattern Detector, Notifier and other external components. Patterns can be associated with different types of text streams (e.g., documents, web pages, and video captions). User patterns are handled by certain modules and incoming streams are handled by other modules. Pattern flow and its corresponding modules are shown in Figure 8.8, and stream flow and its corresponding modules are shown in Figure 8.9, and both are explained below.

Pattern flow in the InfoFilter is illustrated in Figure 8.8. Users submit patterns using PSL to the InfoFilter server. These patterns are validated and processed upon submission by the pattern validator. Once processed, these patterns are sent to the graph generator. The graph generator constructs the PDGs corresponding to the patterns in the pattern detector, and interacts with the WordNet Database tool to extract the synonyms of single words if specified. It also stores the keywords, phrases and regular expressions, embedded in these patterns in a shared suffix trie.
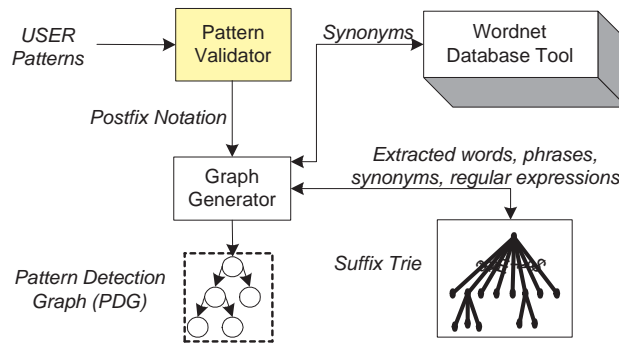
Figure 8.8. Illustration of Pattern Flow in InfoFilter.

Figure 8.9 depicts the stream flow in InfoFilter. As shown, stream processor interacts with the shared suffix trie for pattern matching. In addition, it also interacts with various pattern detectors associated with each stream type (i.e., email, text). Suffix trie stores extracted keywords, phrases and regular expressions, whereas pattern detectors store all the user patterns in form of PDGs. Incoming streams are tokenized using stream based tokenizers (i.e., email stream tokenizer is different from text tokenizer). Once tokenized, the lookup module matches the tokens against the stored patterns in the shared suffix trie. Once a pattern is matched, it is sent to the corresponding PDG in the pattern detector.
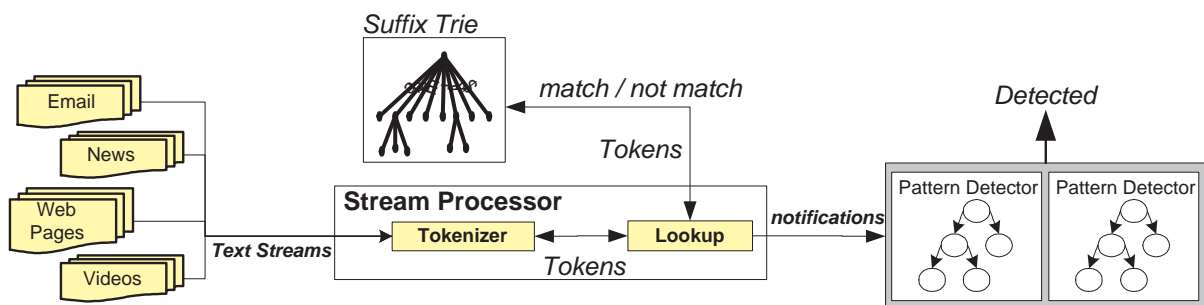


Figure 8.9. Illustration of Stream Flow in InfoFilter.

InfoFilter continuously monitors various types of streaming text, detects simple pattern occurrences and notifies the corresponding pattern detectors. For instance, a user specifies the pattern *"information" NEAR "filtering"* to be detected in an email stream. A separate pattern detector is used to detect the pattern in the email stream. If the pattern is detected, the PDG constructed in this pattern detector is notified. The pattern detector in turn alerts the notifier to send notifications to a user when a pattern is detected.

### 8.2.4 Pattern Detection

User patterns are required to be detected as the incoming text streams flow into the system. For simple patterns, detecting the pattern occurrence is straightforward. For example, the keyword "Language" is detected just by simple pattern matching. However, for complex patterns composed of complex sub-patterns, the detection should conform to the semantics of PSL pattern operators where the order of pattern occurrence is significant (e.g., FOLLOWED BY operator). For example, in order to detect a complex pattern "Information" FOLLOWED BY/10 "Filtering", the flow of pattern occurrences should be preserved. Thus, a data flow paradigm in the form of a PDG is used to detect the patterns according to the operator semantics and to maintain the flow of pattern occurrences. In this section we address how patterns are detected using pattern detection graphs and modes. How patterns are detected effectively using shared approaches are explained in [168].

### 8.2.4.1 Pattern Detection Graphs (PDGs)

For each pattern or sub-pattern, a corresponding PDG is constructed. In a PDG (Figure 8.10), leaf nodes represent simple patterns and internal nodes represent PSL pattern operators. For example, Figure 8.11(b) shows the PDG corresponding to the complex pattern "Query" FOLLOWED BY "Language". Both the leaf and internal
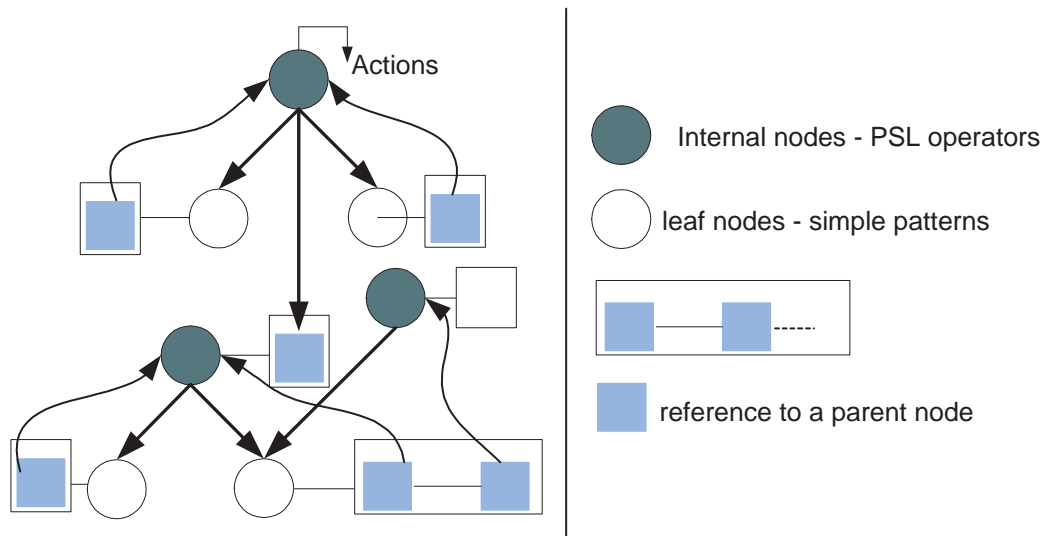
Figure 8.10. Pattern Detection Graph (PDG).

nodes store information about their parent nodes in a subscriber list. Typically, during the construction of a PDG, the parent node subscribes to its child nodes by placing its reference in the subscriber list of the child nodes. In addition to the subscriber list, the leaf node contains the name of the simple pattern it corresponds to. For complex patterns, the internal node contains the name of the complex pattern, references to the nodes of the sub-patterns and their parameters such as the offset of the pattern occurrence and reference to the text stream in which it occurs.

The pattern occurrences flow through the PDG in a bottom-up manner. Once a simple pattern occurrence is detected, the corresponding leaf node propagates it with the associated parameters to the parent node as leaf nodes do not have storage capabilities. For example, in Figure 8.11(b) when an occurrence of the simple pattern "Query" is detected, it is propagated to the FOLLOWED BY node. Each parent node allocates a space for storing the pattern occurrences that belong to its child nodes. Similarly, when a sub-pattern occurrence is detected, the corresponding internal node propagates it to the parent nodes, using the subscriber list.
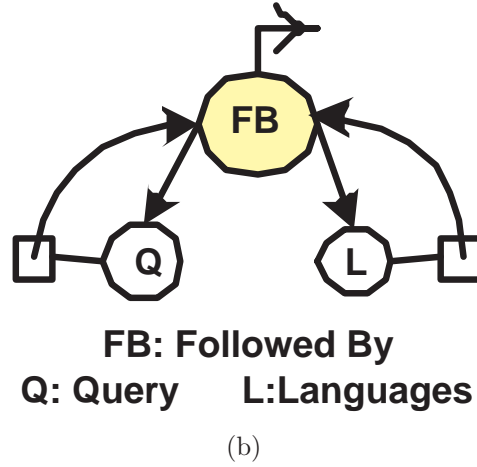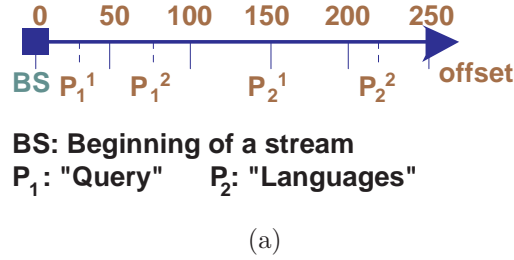
Figure 8.11. Pattern Detection (a) Pattern Occurrences; (b) PDG.

### 8.2.4.2 Pattern Detection Modes

According to the characteristics of the PDGs, the detection of a complex pattern requires the detection of its sub-patterns. The sub-pattern that starts the detection of a complex pattern is termed the *"initiating sub-pattern/pattern"*. Similarly, the sub-pattern that ends the detection of a complex pattern is termed the *"terminating sub-pattern/pattern"*. In the most general case (termed unrestrictive mode), the complex pattern occurrences are detected using all the occurrences of the sub-patterns. This may generate a large number of pattern occurrences which can contain duplicates and may not be meaningful.

Let us take a sample pattern "Query" FOLLOWED BY "Languages", represented as $\mathcal{Q}$ FOLLOWED BY $\mathcal{L}$. Simple pattern occurrences $\{\mathcal{Q}^1\ (P_1^1),\ \mathcal{Q}^2\ (P_1^2),\ \mathcal{L}^1\ (P_2^1),\ \mathcal{L}^2$

$(P_2^2)$} are shown in Figure 8.11(a), where $P_j^i$ indicates the pattern occurrence and offset. For these simple pattern occurrences, four combinations of complex pattern occurrences are generated and they are:

$\{\mathcal{Q}^1, \mathcal{L}^1\}, \{\mathcal{Q}^2, \mathcal{L}^1\}, \{\mathcal{Q}^1, \mathcal{L}^2\}, \{\mathcal{Q}^2, \mathcal{L}^2\}.$

This poses a question as to which pattern occurrences can participate in the detection of the complex pattern (i.e., which occurrence of $\mathcal{Q}^i$ should be paired with $\mathcal{L}^j$). As singularity (i.e., a single pattern cannot detect two different patterns, thus creating duplicates) and proximity (i.e., words that co-occur near each other are considered highly correlated providing a semantic meaning) of pattern occurrences play a significant role in information filtering domain, `Proximal-Unique` detection mode was developed to provide an accurate pattern detection.

**Proximal-Unique**: Only the initiating sub-pattern/pattern occurrence that is closest to the terminating sub-pattern/pattern occurrence is used for pattern detection, ensuring the proximity property. From the above example, pattern occurrence $\mathcal{Q}^1$ $(P_1^1)$ is discarded as $\mathcal{Q}^2$ $(P_1^2)$ will be the closest to the terminating sub-pattern/pattern occurrence $\mathcal{L}^1$ $(P_2^1)$. On the other hand, the initiating sub-pattern/ pattern occurrence is discarded immediately after it has been used in any pattern detection, ensuring that no duplicate pattern occurrences are generated. In other words, this mode entails that an occurrence of a sub-pattern can participate only once in detecting a complex pattern. Therefore, once the pattern occurrence $\mathcal{Q}^2$ is paired with $\mathcal{L}^1$ it cannot be paired with $\mathcal{L}^2$. In this mode only the pattern occurrences $\{\mathcal{Q}^2, \mathcal{L}^1\}$ are used to detect the complex pattern shown in Figure 8.11(a). Since this detection mode emphasizes the proximity and uniqueness of pattern occurrences, it is termed as Proximal-Unique.

### 8.2.5  Summary

In this section, we have presented the InfoFilter[1] system, a content-based system for filtering text streams. InfoFilter has been developed with an intent to support expressive user patterns using PSL and to provide filtering on streams and notification. PSL, proposed in this section, with its expressiveness and well-defined semantics, overcomes the limitations of the current information filtering systems used for specifying and detecting user patterns. It provides a complete set of pattern operators and options such as *frequency*, *synonyms*, *followed by*, *Boolean operators*, *structural*, *wild card*, and *proximity*.

## 8.3  Event Streams

Although research seems to address event and stream data processing as two separate topics there are a number of similarities between them. For many advanced stream applications, both event and rule processing are needed and are not currently well-supported. Extant event processing systems concentrate primarily on complex events and rules and stream processing systems concentrate on stream operators, scheduling, and quality of service issues. Synergistic integration of these models will be better than the sum of its parts. We propose an integrated model to combine the capabilities of both models for applications that not only need to monitor changes through continuous queries (CQs), but also to express and process complex events from the simple events generated by CQs.

### 8.3.1  Introduction

Event processing [43, 53, 54, 55, 56, 58, 61, 62, 63, 65, 66, 67] and lately stream data processing [130, 131, 132, 133, 134, 135] have evolved independently based on situation monitoring application needs. Several event specification languages [42, 65, 93, 169,

---

[1]Please refer [52, 168] for detailed information.

170, 171] for specifying composite events have been proposed and triggers have been successfully incorporated into relational databases. Different computation models [43, 53, 60, 61, 62, 63, 64, 65] for processing events, such as Petri nets [64, 65], extended automata [60, 61, 170], and event graphs [43, 53, 62] – have been proposed and implemented. Various event consumption modes [42, 43, 62, 64, 65] (or parameter contexts) have been explored. Similarly, stream data processing has received a lot of attention lately, and a number of issues – from architecture [130, 132, 133, 134, 172] to scheduling [173, 174, 175] to Quality-Of-Service (QoS) management [176, 177, 178] – have been explored. Although both of these topics seem to be different on the face of it, we believe, based on the applications we have analyzed that they augment/complement each other in terms of computational needs of real-world applications. As it turns out, the computation model used for stream data processing (data flow model) is not very dissimilar from some of the event processing models (e.g., event graph), but developed with a different emphasis.

As many of the stream applications are based on sensor data, they invariably give rise to events that need to be composed to detect composite (or complex) events on which some actions need to be taken. A number of sensor database projects, Cougar [136, 137], TinyDB [138, 139] have tried to integrate the event processing with query processing under a sensor database environment. However, the event-driven queries proposed in TinyDB, for example, are used to activate queries based on events from underlying operating systems. Our focus in this work is to develop an end-to-end system that can process highly dynamic event streams generated from continuous query (CQ) processing stage for applications that need complex event processing as well.

Current event processing systems assume that primitive events are domain specific and are assumed to be detected by the underlying system such as a database, operating system (OS), or an application. Treating the output of complex computations in the form of a stream query processing as events have not been addressed. Furthermore, combining

these two computational models - one window based and the other non-window based (but using event consumption modes or parameter contexts) requires enhancements to *both models*. This work addresses synergistic integration of these two strands of research and development into a more expressive and powerful model of computation.

### 8.3.1.1 Motivating Example

We have analyzed several real-world applications [179] to understand the requirements and issues that need to be solved in order to have an end-to-end system. We summarize one of them which we will use as a running example.

**Example 1.** *In a car **accident detection and notification** system (adapted from the linear road benchmark [180]), each expressway in an urban area is modeled as a linear road, and is further divided into equal-length segments (e.g., 5 miles). Each registered vehicle on an express way is equipped with a sensor and reports its location periodically (say, every 30 seconds). Based on this location stream data, we want to detect a car accident in a near-real time manner. If a car reports the same location (or with speed zero mph) for four consecutive times, FOLLOWED BY at least one car in the same segment with a decrease in its speed by 30% during its four consecutive reports, then it is considered as a potential accident. Once an accident is detected, some actions may have to be taken immediately: i) notify the nearest police/ambulance control room about the car accident, ii) notify all the cars in 5 upstream segments about the accident, and iii) notify the toll station so that all cars that are blocked in the upstream for up to 20 minutes by the accident will not be tolled.*

Every car in the express way is assumed to report its location every 30 seconds forming the primary input data for the above example. The format of car location data stream (i.e., *CarLocStr*) is given below:

```
CarLocStr(timestamp, car_id, speed, exp_way,
```

```
         lane, dir, x-pos)
```

**CarSegStr** is the car segment stream (or the input *CarLocStr* stream), but with the location of the car replaced by the segment corresponding to the location. Query shown below produces the *CarSegStr* from the *CarLocStr* stream.

```
SELECT timestamp, car\_id, speed, exp\_way,
       lane, dir, (x-pos/5 miles) as seg
  FROM CarLocStr;
```

Detecting an accident in the above CAR ADN example has three requirements:

(1) IMMOBILITY: checking whether a car is at the same location for four consecutive time units (i.e., over a 2 minutes window, in our example, as the car reports its location every 30 seconds).

(2) SPEED REDUCTION: finding whether there is at least one car that has reduced its speed by 30% or more during four consecutive time units.

(3) SAME SEGMENT: determining whether the car that has reduced its speed (i.e., car identified in (2)) is in the same segment and it follows the car that is immobile (i.e., car identified in (1)).

Immobility of a car can be computed using CQs that are supported by the current data stream processing systems as shown below:

```
SELECT car_id, AVG(speed) as avg_speed
  FROM CarLocStr [2 minutes sliding window]
GROUP BY car_id
  HAVING avg_speed = 0;
```

With the current event and stream processing models using a non-procedural language[2], it is difficult or impossible to efficiently compute the speed reduction. Whether

---

[2]Models that are based on procedures may compute this, but they are more difficult to use than those models that are based on non-procedural languages (i.e., SQL). In this work we consider the latter one.

the cars that are found in requirements (1) and (2) are from the same segment can be readily determined in an event processing model using a `sequence` operator [65, 62, 93]. As the cars that are identified in requirement (3) can be separated by more than 4 time units, it requires an efficient, meaningful and less redundant approach to notifications. In other words, number of times the accident is reported should be kept to a minimum. This can be done efficiently using the current event processing models using the notion of contexts (e.g., recent context for this case), but not the current stream processing models. Although JOIN operator can be used to compute it in a roundabout manner, the number of notifications (or the number of times an event is raised) are not minimized.

The real-world examples we have analyzed clearly illustrate the need for stream processing followed by event processing to accomplish the task in an elegant manner[3]. In addition, the above notifications have to meet some Quality of service (QoS) require-ments. Processing of events using event detection graphs (analogous to a query tree) and a data flow architecture is similar to the processing of data streams. We have ana-lyzed the characteristics of event and data stream processing models and analyzed their similarities and differences [98]. This will form the basis of our integrated model and for identifying the extensions needed for each model. Although the computation models are similar and events can be viewed as timestamp-based streams, operator semantics, contexts, and processing requirements have a number of differences. Eventually, a num-ber of extensions have to be made on the event processing side to accommodate high input rates and QoS requirements. Some extensions need to be made on the stream side

---

[3]Although the literature from which the example is taken seems to indicate that event processing can be combined into stream processing directly, we believe that it will be a clumsy way to approach this problem and other complex problems as well. Semantically, stream and event processing play different roles and hence need to be brought together in a synergistic way that preserves their individual semantics. Furthermore, specification of events as continuous queries needs to be expressive and flexible.

to generate events (changes/values of interest) which are less in number as compared to stream output and are meaningful. Also, we have provided constructs for defining streams as events, event masks, and addressed architectural issues of coupling the two models.
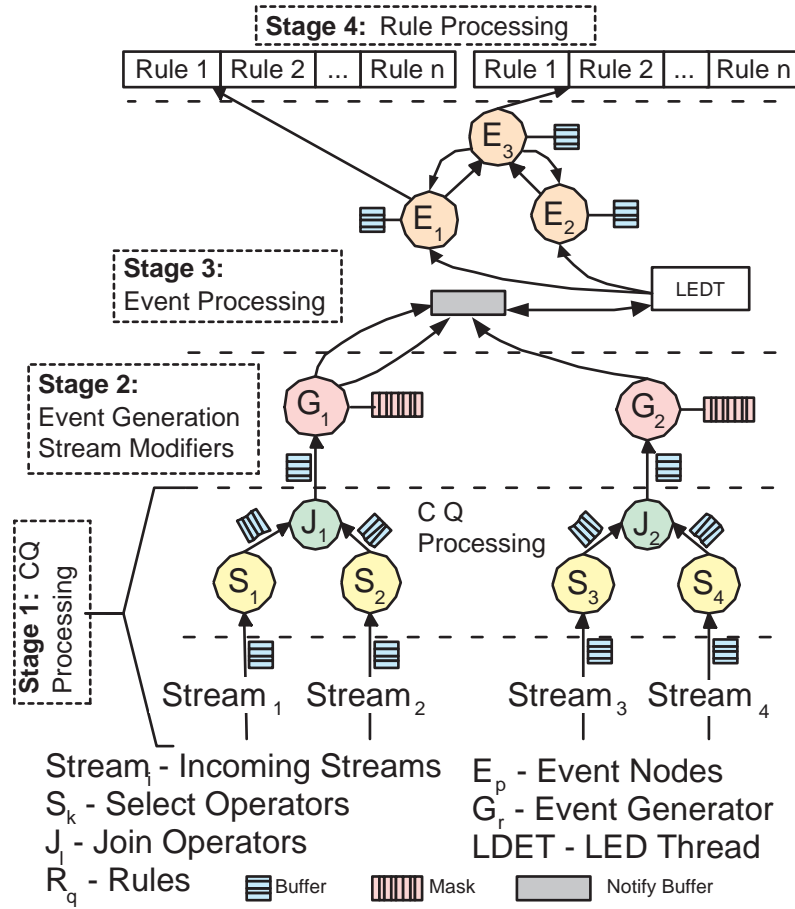


Figure 8.12. MavEStream: Four Stage Integration Model.

## 8.3.2 MavEStream: An Integrated Model

The proposed integrated model, termed MavEStream is shown in Figure 8.12 and it consists of four stages: 1) CQ processing stage used for computing CQs over data

streams, 2) coupling stream output with event processing system, 3) event processing stage that is used for detecting events, and 4) rule processing stage that is used to check conditions, and to trigger predefined actions once events are detected. The seamless nature of our integrated model is due the compatibility of the chosen event processing model [4] (i.e., an event detection graph) with the model used for stream processing.

### 8.3.2.1 Continuous Query Processing

This stage processes normal CQs where it takes streams as inputs and gives computed continuous streams. The scheduling algorithms and QoS delivery mechanisms (i.e., load shedding techniques) along with other techniques developed for stream processing model can be applied directly. In many cases, final results of stream computations need to be viewed as events for defining situations that use multiple streams and composite events. A CQ may give rise to multiple events based on the attribute values of the output stream. In Figure 8.12, operators $S_1$, $S_2$, and $J_1$ form a CQ. Similarly, operators $S_3$, $S_4$, and $J_2$ form a CQ.

```
CREATE CQ CQName AS (Normal CQ statements)
```

**Named Continuous Queries:** In order to express computations clearly, CQs are named. The name of a CQ is analogous to the name of a table in a DBMS and it has the same scope and usage as that of a table. The queue (buffer) associated with each operator in a CQ supports the output of a named CQ to be fed into the input queue of another named CQ. A named CQ is defined by using the CREATE CQ statement shown above. However, the FROM clause in a named CQ can use any previously defined CQs through their unique names. The meta information of a named CQ is maintained in a CQ_dictionary in the system. The meta information includes the *query name*, its *input*

---

[4]It will be difficult to integrate either the Petri net event processing model of SAMOS or the extended automata model of ODE with stream processing models

*sources*, all *output attributes* ordered by their order in final output tuples, and its *output destination(s)*. Events can be specified by using named CQs and in addition providing conditions on attributes to generate multiple event types.

### 8.3.2.2 Event Processing

Below we discuss two limitations of current event processing systems. Event detection graphs (or EDGs) in the current event processing systems do not have input queues/buffer for event operators as the input rate of an event stream is not assumed to be very high and highly bursty. Thus, in our integrated model, input queues/buffers are added to event operator nodes (shown in Figure 8.12) to handle the highly bursty input generated by the CQs from the CQ processing stage.

In a traditional event processing system, primitive events can be either class or instance level, but both of them are based on *timestamps*. Instance level events play an important role for events generated by stream processing, but with the dynamic nature of incoming streams it is difficult or *impossible* to determine the instance level events ahead of time. The example discussed below highlights the limitations of the current event operators that operate solely on timestamp.

Consider the CAR ADN (Section 8.3.1.1) example. Event *Eimm* represents IMMOBILITY and event *Edec* represents SPEED REDUCTION. Event *Eacc* represents the accident and is detected when an event *Eimm happens before* event *Edec*. In addition, *Eacc* is detected only when events *Eimm* and *Edec* are generated by cars from the SAME SEGMENT.

```
CQ1 <timestamp, car_id, speed, exp_way, lane, dir, segment_id>
CQ2 <timestamp, car_id, speed, exp_way, lane, dir, segment_id,
     decrease_in_speed>
```

Stream *CarSegStr* (Section 8.3.1.1) sends inputs to the named continuous queries CQ1 and CQ2. CQ1 checks the car for IMMOBILITY and CQ2 checks for SPEED REDUCTION. Attributes of both CQ1 and CQ2 are shown above. We define events *Eimm* and *Edec* on CQ1 and CQ2, respectively.

```
Eimm <9.00 am, 1,  0 mph, EW1, 3, NW, 104>

Edec <9.03 am, 2, 40 mph, EW1, 1, NW, 109, 45%>

Edec <9.04 am, 5, 20 mph, EW1, 4, NW, 104, 40%>
```

From the above, *Eimm* occurs at `9.00 am` and *Edec* occurs at `9.03 am` and `9.04 am`. *Eacc* is detected when *Eimm* precedes *Edec* in time. From the above tuples, two accidents are detected; `car_id` 1 and `car_id` 2, and `car_id` 1 and `car_id` 5. Thus, it is evident that in current event processing systems, the important condition that both the cars should be from the SAME SEGMENT is checked only after the event *Eacc* is detected. This introduces a high overhead on the event computation as there can be many unnecessary detection of event *Eacc* with nature of data stream applications.

The above example can be modeled using instance level events, but all the instances of a class should be predefined (or known previously). This may be impossible in a system where the data streams' attribute values are dynamic. Even if the values are predefined, they require large number of event nodes, which introduce high computation and memory overhead. Hence, event processing needs to be burdened less to support *efficient* detection. We have generalized event expression computation, so that attribute conditions are checked *before* the events are detected. This generalized expression allows both primitive and composite event nodes to detect events based on attribute-based constraints (or MASKS). MASKS are pushed to the event generator node with primitive events (i.e., for leaf nodes in EDG) and are pushed into the event operator nodes (i.e., internal nodes in EDG) for other events. For instance, in Figure 8.12 MASKS corresponding to CQ with $J_1$ as the root node is pushed to event generator node $G_1$. Thus,

when multiple events are defined on the same CQ but with different MASKS, all of them are pushed to the corresponding event generator node.

```
CREATE EVENT   Ename

       SELECT   A₁, A₂, ..., An

         MASK   AC₁, AC₂, ..., ACn

         FROM   Es | Ex
```

Users can specify events based on `CQs` (for primitive events) or on `Events` using the CREATE EVENT statement shown above.

- CREATE EVENT creates a named event $\mathcal{E}_{name}$

- SELECT selects attributes $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$

- MASK applies conditions on the attributes

- $\mathcal{E}_S$ is a named `CQ` or a CREATE CQ statement

- $\mathcal{E}_X$ is an event expression that combines more than one event using event operators

Below we show how we model the CAR ADN example (Section 8.3.1.1) using multiple[5] CREATE EVENT statements.

```
CREATE EVENT Eimm

      SELECT CQ1.car_id, CQ1.seg_id FROM CQ1


CREATE EVENT Edec

      SELECT CQ2.car_id, CQ2.seg_id FROM CQ2


CREATE EVENT Eacc

      SELECT Eimm.car_id, Edec.car_id

            Eimm.seg_id, Eimm.timestamp

        MASK Eimm.seg_id = Edec.seg_id
```

---

[5]We can also model the same using a nested statement.

```
FROM Eimm SEQUENCE Edec
```

*Eimm* is created from the CQ1 and *Edec* is created from the CQ2. If we need to select cars from a particular seg_id then it should be specified as a MASK in events *Eimm* and *Edec*. In our integrated model, event generator nodes are created on CQ1 and CQ2 so that events can be raised when ever there is an output from CQ1 or CQ2. Event expression $\mathcal{E}_X$ for the accident is $Eacc = Eimm$ SEQUENCE $Edec$, where SEQUENCE is an event operator that is detected when the first event precedes the second event in time. In addition, MASK specifies that cars should be from the SAME SEGMENT and is checked in the SEQUENCE operator node.

### 8.3.2.3   Coupling Event and Stream Processing

The local event detector (LED) has a common *notify buffer* (or event processor buffer) into which all events that are raised are queued. A single queue is necessary as events are detected and raised by different components of the system (CQs in this case) and they need to be processed using their time of occurrence. Briefly, a new operator is added to every stream query at the root if an event is associated with that CQ. This operator can take any number of MASKS and for each MASK, a different event tuple/object is created and sent to the notify buffer. This operator is activated only when an ECA rule associated with that CQ is enabled. This operator is similar to the select operator except that when it generates an event, it invokes an API of LED to queue that event in the notify buffer. CQs output data streams in the form of tuples and *event generator* operator nodes are attached to the root node of the CQ. As shown in Figure 8.12, nodes $J_1$ and $J_2$ are attached to event generator nodes $G_1$ and $G_2$. In addition, nodes $G_1$ and $G_2$ are also associated with MASKS. Thus, stream tuples from $J_1$ and $J_2$ are converted to events by nodes $G_1$ and $G_2$.

### 8.3.3 Summary

In this work, we argued for keeping the semantics of event and stream processing systems intact and integrating them *synergistically* to provide an end-to-end system for advanced applications[6]. Our goal was to provide an integrated model for advanced stream applications that supports not only stream processing, but also complicated event and rule processing. We analyzed the similarities and differences between the stream processing and the event processing models and identified a number of enhancements needed. We elaborated on coupling the two systems, specifying ECA rules where the events are generated by CQs. By using masks it is easier to specify expressive rules (car example).

### 8.4 Network Fault Management

Network fault management has been an active research area for a long period of time because of its complexity, and the returns it generates for service providers. However, most fault management systems are currently custom-developed for a particular domain. As communication service providers continuously add greater capabilities and sophistication to their systems in order to meet demands of a growing user population, these systems have to manage a multi-layered network along with its built-in legacy logical processing procedure. Stream processing has been receiving a lot of attention to deal with applications that generate large amounts of data in real-time at varying input rates and to compute functions over multiple streams, such as network fault management. In this section, we provide an integrated inter-domain network fault management system for such a multi-layered network based on *data stream and event processing techniques* explained in Section 8.3. We discuss briefly (please refer [99] for more details) various

---

[6]Please refer [98] for more information.

components in our system and how data stream processing techniques are used to build a flexible system for a sophisticated real-world application.

### 8.4.1 Introduction

In telecommunication network management, Network Fault Management (NFM) is defined as the set of functions that (a) detect, isolate, and correct malfunctions in a telecommunication network, (b) compensate for environmental changes, and (c) maintain and examine error logs, accept and act on error detection notifications, trace and identify faults, carry out sequence of diagnostic tests, correct faults, report error conditions, and localize and trace faults by examining and manipulating database information.
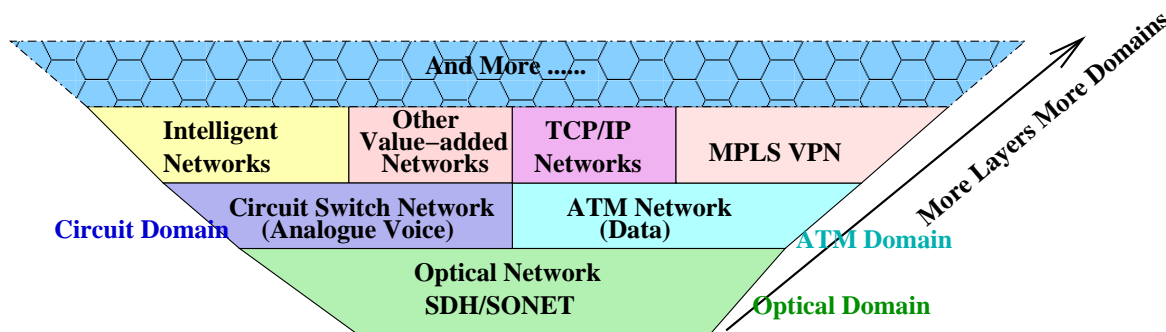


Figure 8.13. A Typical Telecomm Network.

A typical telecommunication network illustrated in Figure 8.13 is a multi-layered network, in which the bottom layer provides transport service through SDH/SONET networks. Above that, a PSTN switch network with a SS7 signaling network is used to provide traditional voice services, and an ATM network is used to provide Internet data service. Intelligent networks and other value-added networks can be added above the PSTN switch networks, and BGP/MPLS VPN network can be added above the ATM network. The NFM in such a multi-layered telecommunication network has been

an interesting research problem [140, 141, 142, 143, 181, 182, 183] in both industry and academia for a long time because of its high cost and complexity. The recent convergence of data network, cable network, and telecommunication network has further compounded this problem.

Each network element (NE) in this multi-layered network reports the status of each of its components and the status of its environment (e.g., temperature) periodically (e.g., every 5 minutes). Some NEs have simple capabilities such as summarizing its status by processing the status message locally in order to decrease the number of messages to be reported or to identify trivial messages. Hence, these messages arrive in a form of a message stream, and each NE can be considered as a message stream source. These status and alarm messages from each NE, each operation system (OS), and each link are continuously collected in a network operation center (NOC) to be further analyzed by experts to detect and to isolate faults. Once a fault is identified, sequences of actions need to be taken locally and remotely. Due to the complexity of the network and different interfaces of multiple-vendor's devices, each layer has one or more independent NFM systems [182, 183]. For example, there is a SDH/SONET fault management system for the transport layer in the network as illustrated in Figure 8.13. There are also individual fault management systems for circuit switch and Internet data networks. Similarly, all vendors have their own fault management systems when multiple-vendors' devices are used. As a result, when a failure happens at a lower level, it is propagated to all the components above, and a large volume of failure messages are reported to those independent NFM systems. Moreover, there is a demand for providing an integrated view of the whole network system and to process faults centrally.
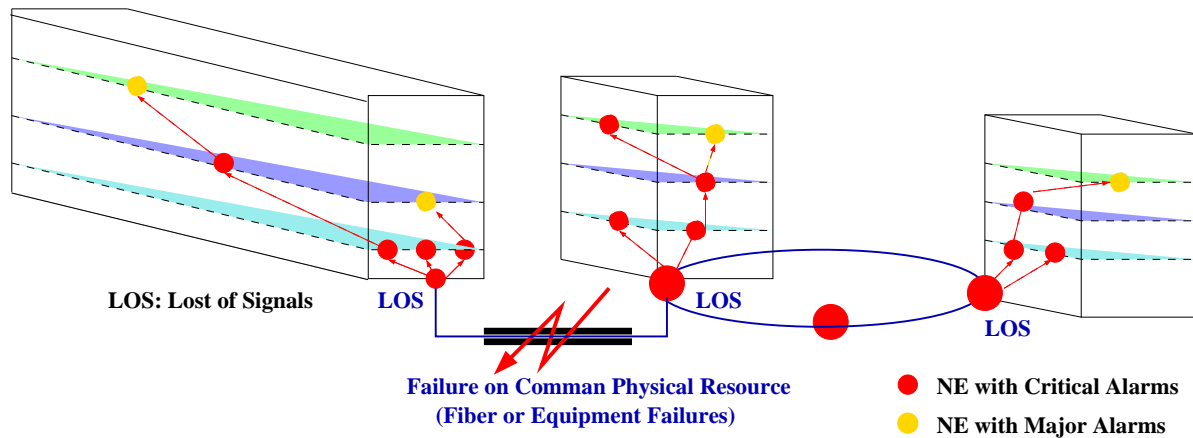
Figure 8.14. Motivation Example.

## 8.4.2 Problem Definitions

Currently, for each independent NFM system, due to the large volume of messages that are continuously reported by each NE and the complex message processing requirements, it is impossible to employ a traditional database management system (DBMS) plus trigger mechanisms as the data processing paradigm for NFM. Current NFM systems have to hard code their data processing logic and specific monitoring rules (queries) in the system. As a result, various filters, pattern languages, regular expressions are employed to find their interesting alarm messages and group those messages into multiple subgroups based on various criteria. These subgroups are finally presented to experts to diagnose root causes or route to an event correlation system to identify causes automatically. Once the causes are identified, a ticket is placed to a trouble ticket system to trace the problem and have corresponding engineers fix it. There are several major shortcomings of these legacy systems. First, current systems have difficulty adapting to new requirements from their customers because of the hard-coded queries. To add a new query or to add a new monitoring rule, the system has to be reconfigured partially. Second, current systems are very complicated, and their performance is poor because there

is almost no query optimization. Third, the scalability of the systems is limited because of the tight integration between query processing and other logical components. Fourth, there is no standard interface or language such as SQL to access those systems, which makes them hard to use and manage. Finally, it is difficult to integrate different NFM systems at different layers because of the hard-coded queries and different implementation techniques. As there is a dramatic growth in both the volume of message stream and the number of interesting alarms, there is an increasing demand to process and manage message streams for these applications. This motivates us to investigate various aspects of a data stream, and to exploit various efficient and effective data stream processing technologies, eventually to build an integrated network fault management system for a large-scale telecommunication network.

Some of the challenges are unique to our system and are different with those issues in current data stream processing models. First, the input stream consists of semi-structured text messages, which include numerical data, date/time, place, phone number, and other more critical and complicated information. Second, the computation required to detect, isolate, and correct malfunctions in a fault management system is far more complicated than the computation discussed in current data stream management systems through a set of traditional relational operators. Finally, sequences of actions that need to be take when a fault is detected is complicated and involves experts and can last for long periods of time. These challenges make our system more critical and complicated than current data stream processing systems.

### 8.4.3  Proposed $NFM^i$ system

Inter-domain network fault management ($NFM^i$), illustrated in Figure 8.15, is based on stream processing techniques to provide an integrated NFM system for a multi-layered telecommunication network with on-line processing and near-real time response to

faults. The system has many advantages compared to a traditional domain specific fault management system: 1) $NFM^i$ provides an integrated view of the status of the entire network, and correlates alarms in a global domain, which greatly decreases the number of alarms shown to network administrators, 2) it is more flexible than a traditional fault management system. In a traditional fault management system, the alarm processing is hard coded in the system, and as a result any change to the alarm processing (e.g., addition of new computations or rules to monitor) needs reconfiguration of the system. The stream-based system proposed in this paper can easily add new computations by adding new operators, and monitor new rules by issuing new continuous queries, and 3) the system is easier to use and maintain because of the flexibility of continuous queries (extended SQL), and the clear separation of alarm processing and alarm expression.

On the other hand, our system is quite different from a traditional data stream processing system in the following sense: 1) it processes semi-structured messages in contrast to well-defined tuples assumed in most stream-based systems. Actually, the processing to convert a semi-structured message to a well-defined tuple can be done as stream processing, which usually takes at least one third or even more computational power to handle this part of the work in our system, 2) the complicated processing requirements cannot be addressed using the basic select-project-join and aggregation operators. It requires computation of the correlations among alarms and an intelligent decrease in the number of alarms shown to the administrators, and 3) it also involves update operations, which are not discussed in the current data stream processing systems. For example, when an administrator takes some actions for an alarm, certain information (i.e., when and what kind actions have been performed) has to be added to the alarm. Also when the problems related to that alarm have been fixed, the status of the alarm has to be updated.

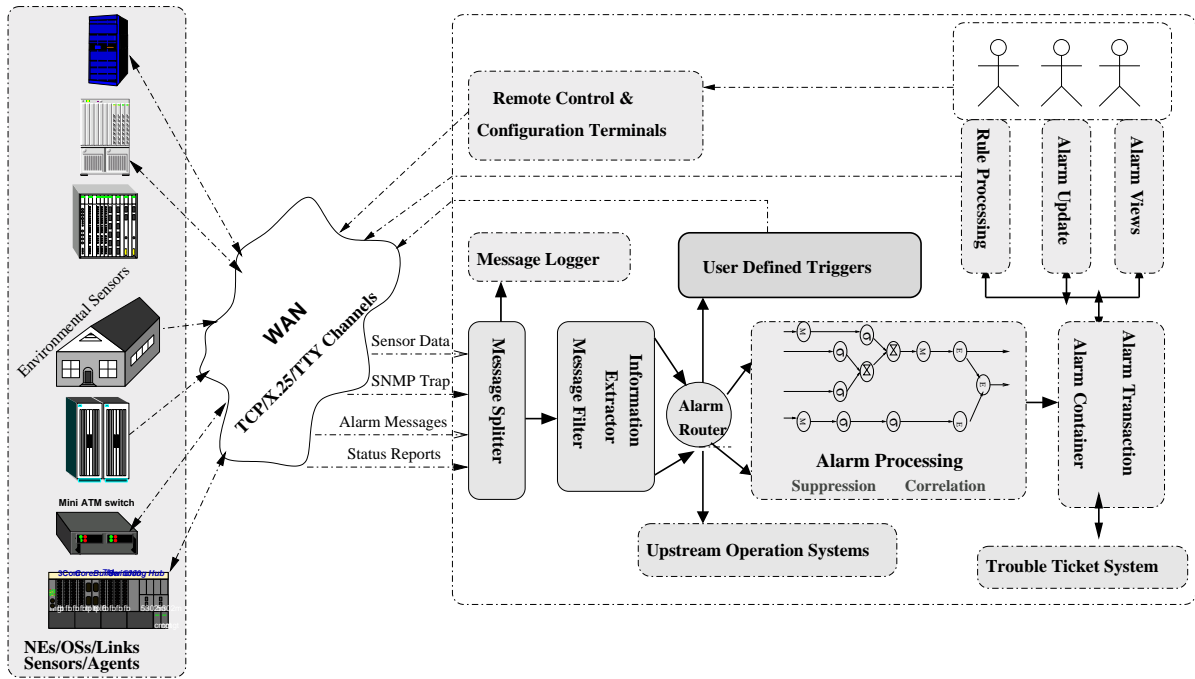Below we explain the architecture of our proposed NFM system.

Figure 8.15. Inter-Domain Network Fault Management System Architecture.

*Message splitter* is the first function in our system that is applied over the incoming alarm message streams. Each stream is a character stream, in which the messages are sent continuously to the system character by character. Message splitter processes over the character stream, distinguish each message and wraps a segment of consecutive characters as a message. The *message filter* and *information extractor* module provides an intelligent way to find interesting messages and extract required information from those messages. It takes the message streams from message splitters as its input and outputs formatted alarm messages which are similar to a tuple in a relation.

The main role of the *alarm processing module* is to detect and isolate the faults by analyzing the relationships between input alarm messages. In the multi-layered network shown in Figure 8.13, when the lower layer has a fault, it is usually propagated to all its upper components. This fault propagation causes the number of alarms reported by the entire network to increase exponentially. A single NE or OS reports the same

alarm periodically. Obviously, an administrator does not want to see the same problem periodically before the problem is solved. However, the administrator wants to increase the severity of the alarm if the problem is not fixed within a predefined time period in order to draw more attention to the alarms that persists (or alive) in the system for a long time. A fault in one component causes multiple NEs, OSs, and Agents to report the same problem from their point of view. Some of those alarms have the same format and similar content; while the others may have significantly different formats and different contents. However, there exists certain relationships among these alarms, and based on the relationships, an administrator wants the fault management system to automatically process (NOT to discover the relationships) the relationships and decrease the number of alarms presented to the administrators. Both problems require the fault management system to *continuously monitor* the alarm streams and evaluate the relationships of these alarms. By considering each alarm as an *event*, we propose to use our four phase model explained in Section 8.3 to evaluate those relationships.

Both Event-Condition-Action (ECA) and stream processing models have their limitations for handling applications that require a combination of stream and event/alarm processing, such as fault management. The ECA model has its strength in expressing and processing complex events. The CQ processing phase takes the output streams from the message filter and information extractor as inputs and outputs computed continuous streams to the event detection phase. In our NFM system, final results of stream computations need to be viewed as events for detecting faults and isolating faults that use multiple alarm streams and composite events. However, current stream processing model is short of event and rule processing. It is also difficult to detect changes to one or more attributes and to suppress the number of outputted alarms.

Every alarm that is produced by the *alarm processing* phase should be processed scrupulously by the experts. An alarm container is proposed to contain all active alarms

[7] outputted from the continuous query processing stage. This alarm container is implemented as a large block of shared memory. Once an alarm is cleared, it is deleted from the container. Alarm container can also be considered as a data source of streaming alarms, and various CQs can be defined over it to present alarms to experts and to update the status of alarms. An in-memory DBMS such as Times-ten [184] can also be used for this purpose.

Since different experts have different domain knowledge, they only monitor a small portion of alarms based on their knowledge and their interest. Therefore, various *views* have to be defined to select alarms and to project most useful fields of an alarm. Current fault management systems categorize alarms into a large number of small-groups based on various fields, which are similar to GROUP BY clause in SQL, and then experts subscribe to their groups of interest and define which fields should be shown on their screens. Currently, all of this is done by defining various configuration files and there is no methodology making the process inflexible and difficult to manage and maintain. For example, when a new expert joins the team, those configuration files have to be updated to select alarms for this expert. In our proposed system, rather than using various configuration files to select alarms and to select fields from alarms, we propose to use continuous queries to define various views for each expert, and the system can further optimize those CQs to achieve a better performance. This set of CQs can be defined to provide a set of snapshots of the alarm container. Once a new alarm is appended to the container, it is also shown on the screen of relevant experts. If *read* is the only operation over an alarm container, CQs work as intended. However, *update* and *delete* operations over the alarm container are necessary during the course of alarm processing.

Though some human interventions are required to correct some malfunctions identified by the alarm continuous queries, there is a real need for *rule processing* which

---

[7]those waiting for further processing.

triggers a sequence of actions automatically based on some conditions and alarms. Some malfunctions can be corrected without human interaction and some actions can be done automatically but others require human intervention. For example, when a very critical alarm is detected, an audio broadcast (in addition to sending an alarm to a group of experts) may be necessary to get the attention of corresponding experts and administrators in order to respond to it as soon as possible. One alarm can trigger multiple sequences of actions, and diverse alarms can trigger the same sequence of actions. In order to handle multiple actions, Event-Condition-Action (or ECA) rules are used by the system. Whenever an event arrives (i.e., outputs from our alarm continuous queries in our system), corresponding condition is checked, and an action is triggered (if the condition evaluates to true). In network fault management applications, most of the rules should be immediately triggered as the problem needs to be brought to the attention of various experts.

### 8.4.4 Summary

In this work, we have chosen a real-world problem and proposed a state-of-art network fault management system based on *event and stream processing techniques.* We proposed an architecture, set of operators, alarm processing techniques for an inter-domain network fault management system for a multi-layered telecommunications network. In order to do this, we had to come up with an integrated system that combines stream data processing with event processing[8]. We also developed a set of solutions for the issues in the proposed inter-domain network fault management system. Some are general solutions that are applicable to most stream-based applications. However, some of the solutions proposed in this chapter are specifically targeted to the special requirements of network fault management domain.

---

[8]Please refer [99] for detailed information.

# CHAPTER 9

## CONCLUSIONS AND FUTURE WORK

In this thesis, we have focused on several aspects of RBAC including generalization and enforcement by exploiting and extending Sentinel – a well-established event-based framework, the usability of RBAC, and novel applications of RBAC. Specifically, we have addressed the following problems and made the following contributions:

- Enforcement of Existing RBAC Approaches: We have shown how active authorization rules or extended ECA rules can be used to enforce RBAC and its extensions, such as temporal, and control flow dependency constraints in a uniform way [85, 94]. The generated rules have different granularities and classifications based on their functionality. The mapping of RBAC standard and its extensions to extended ECA rules provide a practically applicable view of RBAC. The set of rules used to realize all of the role-based models can also be used in an underlying system (that provides some hooks) to support Role-Based models. We have also carried out a feasibility study and analyzed alternative approaches for enforcing role-based models. We have provided interval-based semantics [91, 92, 93] for Snoop event operators in continuous and cumulative consumption modes to avert incorrect event detection.

- Generalization of RBAC and Event Specification: Constraints play a vital role in providing fine-grained access control and realizing RBAC over diverse domains. First, we motivated the need for generalizing RBAC based on event pattern constraints with some critical examples. We identified several advantages and limitations of Snoop and LED, and proposed several generalizations to overcome those limitations. In particular, we discussed how making event pattern specification and

its enforcement using a uniform approach is beneficial (typically they are disjoint or orthogonal in all systems we are familiar with). We have generalized the traditional simple and complex event definitions. We then identified the simple or domain events that are required for constraint specification in RBAC. We illustrated how policy checking is carried out via authorization rules. We have shown how constraints can be placed on simple events using authorization rules. We then generalized RBAC with event pattern constraints. Event patterns with complex events and simple events as constituent events were used to model constraints, such as temporal, context, precedence, dependency, non-occurrence, and their combinations. Even though we have discussed various pattern operators that are useful in constraint specification, new operators can be plugged in seamlessly into our framework.

- Enforcement of Generalized RBAC: We have analyzed the limitations of current event detection graph mechanism in LED. We then extended event detection graphs as event registrar graphs to incorporate all the event generalizations and for capturing event occurrences and keeping track of event ordering. Event registrar graphs follow a bottom-up data flow paradigm and are efficient as they allow the sharing of event patterns and simple events. We have also shown how expressive event-based constraints can be enforced using event registrar graphs. We introduced complete, uncomplete, and failed events and discussed how rules associated with them can be used to check policy constraints based on the complex constraint satisfaction. Finally, we have explained how policy conflicts can be identified and resolved, though it requires further investigation.

- Usability in RBAC: The abstraction provided by roles allows them to own objects and allows users to shift roles. On the other hand, for accessing objects, users have to know the relationship between roles and objects. Our approach is proactive in

that it provides the necessary notifications to the user acting in anticipation of future problems that the user may face when the user is requesting access. We have presented algorithms for discovering roles and have analyzed their complexity and effectiveness [95]. When a user gets access DENIAL from our algorithm, it actually means that there are no roles that can be activated to make this request happen. This is a much *stronger* denial than what the current systems provide. Notifications in our approach allow users to concentrate on what data needs to be accessed rather than the roles that are required for access, thus preserving the principle of least privilege. Roles are disclosed to the user without any information leak. Although role discovery has its associated overhead with respect to system response time, it reduces user response time, increases user satisfaction or usability and is more effective than algorithms that provide binary replies and follow the *human-active, system-passive* model.

- Novel Applications: We have developed several novel applications for demonstrating the applicability of the results obtained in this thesis.

  i) We have provided a smart push-pull approach for supporting Role-Based security in web gateways [96]. The role-based security realizes the concept of RBAC and provides a practical, elegant mechanism for controlling accesses in web gateways. By leveraging RBAC, the number of access rules and their complexity is greatly reduced, thus reducing the administrative burden. Moreover, there are additional advantages, such as seamless constraint specification (e.g., time of the day, quotas based on bandwidth or time, IP address, location, etc.), and fine-grained access control decisions.

  ii) InfoFilter, a content-based system for filtering text streams has been developed with an intent to support expressive user patterns using a pattern specification language and to provide filtering on streams and notification [52, 97]. We have shown

how event operators with interval-based semantics can be utilized for information filtering. Although this application is not directly relevant to the RBAC domain, it can be used for various applications including tracking of information flow among terrorist outfits.

iii) We introduced an integrated model for advanced stream applications that supports, not only stream processing, but also complicated event and rule processing [98]. The integrated model uses attribute-based semantics of events. In other words, this model requires explicit expressions (E) developed as part of our work in the generalization of Sentinel. We chose a real-world problem to propose a state-of-the-art network fault management system based on *event and stream processing techniques* [99].

## 9.1 Future Work

We have shown how active authorization rules are used in the enforcement of existing RBAC approaches in a uniform manner. Mapping of higher level policies into ECA rules need to be examined. Complete set of rules that are required for the complete GTR-BAC and other policy specifications need to be generated. In general, it will be interesting to explore how extended ECA rules can be used in supporting other security models and providing distributed and cross-domain access control for enterprises. Benchmarking and the overhead of using ECA rules needs to be investigated. We have generalized RBAC with event pattern constraint specification. Showing whether the event pattern approach subsumes other approaches (individually) needs to be investigated. The formal semantics for attribute-based evaluation of snoop operators in different contexts need to be evaluated. We have also identified policy conflicts from our event pattern constraint specifications, but this requires further investigation. Consistency checking of ECA rules (cyclic and other undesirable properties) needs to be investigated. In addition, access to

these rules and their management (meta-level) need to be investigated. In general policy verification, validation and conflict resolution should be explored further. Incorporation of event registrar graphs into different systems in different ways, such as integrated, and middle-ware, needs to be investigated. Update, redefinition, and flexibility of the event-based approach need to be investigated, and applying this approach to common information model (CIM) with respect to services is another direction. Need to investigate the applicability of generalized event patterns to other domains to provide better expressiveness. Finally, newer application domains, where RBAC with event pattern constraints can be leveraged, need to be explored.

## REFERENCES

[1] G. S. Graham and P. J. Denning, "Protection - Principles and Practice," in *Proceedings, AFIPS Spring Joint Computer Conference*, Montvale, New Jersey, 1972, pp. 417–429.

[2] D. E. Bell and L. J. LaPadula, "Secure Computer System: Unified Exposition and MULTICS Interpretation," The MITRE Corporation, Bedford, MA 01730, Tech. Rep. MTR-2997 Rev. 1 and ESD-TR-75-306, rev. 1, Mar. 1976.

[3] *A Guide to Understanding Discretionary Access Control in Trusted Systems*, NCSC-TG-003-87, National Computer Security Center, Sep. 1987.

[4] R. S. Sandhu, "Lattice-Based Access Control Models," *IEEE Computer*, vol. 26, no. 11, pp. 9–19, 1993.

[5] R. S. Sandhu and P. Samarati, "Access Control: Principles and Practice," *IEEE Communications Magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[6] R. S. Sandhu, "Access Control: The Neglected Frontier," in *Proceedings, First Australian Conference on Information Security and Privacy*, Wollong, Australia, 1996, pp. 219–227.

[7] R. S. Sandhu and P. Samarati, "Authentication, Access Control, and Intrusion Detection," in *The Computer Science and Engineering Handbook*, A. B. Tucker, Ed. CRC Press, 1997, pp. 1929–1948.

[8] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," *Proc. IEEE Symposium on Security and Privacy, Oakland, California*, pp. 184–194, April 1987.

[9] D. F. Ferraiolo and D. R. Kuhn, "Role-Based Access Control," in *Proc. of the 15th National Computer Security Conference*, 1992, pp. 554–563.

[10] D. F. Ferraiolo, J. A. Cugini, and D. R. Kuhn, "Role-Based Access Control: Features and Motivations," in *Proceedings, Annual Computer Security Applications Conference*, 1995.

[11] R. S. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-Based Access Control Models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[12] *RBAC Standard, ANSI INCITS 359-2004*, ANSI INCITS 359-2004, InterNational Committee for Information Technology Standards, 2004.

[13] W. Eßmayr, S. Probst, and E. Weippl, "Role-based access controls: Status, dissemination, and prospects for generic security mechanisms." *Electronic Commerce Research*, vol. 4, no. 1-2, pp. 127–156, 2004.

[14] F. Chen and R. S. Sandhu, "Constraints for role-based access control," in *Proceedings, ACM Workshop on Role-Based Access Control.* New York, NY, USA: ACM Press, 1996, p. 14.

[15] E. Lupu and M. Sloman, "Reconciling role based management and role based access control," in *Proceedings, ACM Workshop on Role-Based Access Control.* New York, NY, USA: ACM Press, 1997, pp. 135–141.

[16] L. Giuri and P. Iglio, "Role templates for content-based access control," in *Proceedings, ACM Workshop on Role-Based Access Control.* New York, NY, USA: ACM Press, 1997, pp. 153–159.

[17] T. Jaeger, "On the increasing importance of constraints," in *Proceedings, ACM Workshop on Role-Based Access Control.* New York, NY, USA: ACM Press, 1999, pp. 33–42.

[18] G.-J. Ahn and R. Sandhu, "Role-based authorization constraints specification," *ACM Transactions on Information and System Security*, vol. 3, no. 4, pp. 207–226, 2000.

[19] M. J. Moyer and M. Ahamad, "Generalized role-based access control," in *Proceedings, International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2001, p. 391.

[20] E. Bertino, P. A. Bonatti, and E. Ferrari, "TRBAC: A temporal role-based access control model," *ACM Transactions on Information and System Security*, vol. 4, no. 3, pp. 191–233, 2001.

[21] J. Crampton, "Specifying and enforcing constraints in role-based access control," in *Proceedings, ACM Symposium on Access Control Models and Technologies*. New York, NY, USA: ACM Press, 2003, pp. 43–50.

[22] M. Strembeck and G. Neumann, "An integrated approach to engineer and enforce context constraints in rbac environments," *ACM Transactions on Information and System Security*, vol. 7, no. 3, pp. 392–427, 2004.

[23] J. B. D. Joshi, E. Bertino, U. Latif, and A. Ghafoor, "A Generalized Temporal Role-Based Access Control Model," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 1, pp. 4–23, Jan. 2005.

[24] *Internet Security Glossary - RFC 2838*, Network Working Group, 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2828.txt

[25] *Security Requirements for Cryptographic Modules, FIPS PUB 140-1*, U.S. Department of Commerce, 1994.

[26] *Glossary of Computer Security Terms, NCSC-TG-004, ver. 1*, National Computer Security Center, (Part of the Rainbow Series.), 1998.

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

[28] M. Nyanchama and S. L. Osborn, "Modeling Mandatory Access Control in Role-Based Security Systems," in *Proceedings, IFIP Workshop on Database Security*, 1995, pp. 129–144.

[29] R. S. Sandhu and Q. Munawer, "How to do discretionary access control using roles," in *ACM Workshop on Role-Based Access Control*, 1998, pp. 47–54.

[30] S. L. Osborn, R. S. Sandhu, and Q. Munawer, "Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies," *ACM Transactions on Information and System Security*, vol. 3, no. 2, pp. 85–106, 2000.

[31] *The Economic Impact of Role-Based Access Control*, RTI Project Number: 07007.012, National Institute of Standards and Technology (NIST), 2002. [Online].

[32] "Role Based Access Control Case Studies and Experience," National Institute of Standards and Technology (NIST). [Online]. Available: http://csrc.nist.gov/rbac/RBAC-case-studies.html

[33] K. Beznosov, "Requirements for access control: US healthcare domain," in *RBAC '98: Proceedings of the third ACM workshop on Role-based access control*. New York, NY, USA: ACM Press, 1998, p. 43.

[34] M. Evered and S. Bögeholz, "A case study in access control requirements for a health information system," in *CRPIT '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2004, pp. 53–61.

[35] J. B. D. Joshi, E. Bertino, and A. Ghafoor, "Hybrid role hierarchy for generalized temporal role based access control model." in *COMPSAC*, 2002, pp. 951–956.

[36] J. B. D. Joshi, E. Bertino, and A. Ghafoor, "Temporal hierarchies and inheritance semantics for gtrbac." in *Proceedings, ACM Symposium on Access Control Models and Technologies*, 2002, pp. 74–83.

[37] J. B. D. Joshi, B. Shafiq, A. Ghafoor, and E. Bertino, "Dependencies and separation of duty constraints in GTRBAC." in *Proceedings, ACM Symposium on Access Control Models and Technologies*, 2003, pp. 51–64.

[38] B. Shafiq, A. Masood, A. Ghafoor, and J. B. D. Joshi, "A Role-Based Access Control Policy Verification Framework for Real-Time Systems," in *Proc. of the IEEE Workshop on Object-oriented Real-time Databases*, 2005.

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

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

[41] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite Events for Active Databases: Semantics, Contexts, and Detection," in *Proceedings, International Conference on Very Large Data Bases.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 606–617.

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

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

[44] J. Bailey, A. Poulovassilis, and P. T. Wood, "An Event-Condition-Action Language for XML," in *Proceedings, International Conference on World Wide Web.* ACM Press, 2002, pp. 486–495.

[45] M. Bernauer, G. Kappel, and G. Kramler, "Composite Events for XML," in *Proceedings, International Conference on World Wide Web*. ACM Press, 2004, pp. 175–183.

[46] G. Papamarkos, A. Poulovassilis, and P. T. Wood, "RDFTL: An Event-Condition-Action Language for RDF," in *Proc. of The Hellenic Data Management Symposium*, 2004.

[47] G. Papamarkos, A. Poulovassilis, and P. T. Wood, "Event-Condition-Action Rule Languages for the Semantic Web," in *Proc. of the International Workshop on Semantic Web and Databases, at the VLDB*, 2003, pp. 309–327.

[48] M. Zoumboulakis, G. Roussos, and A. Poulovassilis, "Active Rules for Sensor Databases," in *Proc. of the Workshop on Data management for Sensor Networks*. ACM Press, 2004, pp. 98–103.

[49] T. Terada, M. Tsukamoto, K. Hayakawa, T. Yoshihisa, Y. Kishino, A. Kashitani, and S. Nishio, "Ubiquitous chip: A rule-based i/o control device for ubiquitous computing." in *Proceedings, International Conference on Pervasive Computing*, 2004, pp. 238–253.

[50] S. Chakravarthy *et al.*, "WebVigiL: An approach to Just-In-Time Information Propagation In Large Network-Centric Environments," Hawaii, US, Aug. 2002.

[51] V. Kantere and A. Tsois, "Using ECA Rules to Implement Mobile Query Agents for Fast-Evolving Pure P2P Database Systems," in *Proceedings, International Conference on Mobile Data Management*, Ayia Napa, Cyprus, 2005, pp. 164–172.

[52] L. Elkhalifa, R. Adaikkalavan, and S. Chakravarthy, "InfoFilter: A System for Expressive Pattern Specification and Detection over Text Streams," in *Proceedings, Annual ACM SIG Symposium On Applied Computing*, Santa Fe, NM, USA, Mar. 2005, pp. 1084–1088.

[53] H. Engstrom, M. Berndtsson, and B. Lings, "Acood essentials," University of Skovde, Tech. Rep., 1997.

[54] O. Diaz, N. Paton, and P. Gray, "Rule Management in Object-Oriented Databases: A Unified Approach," in *Proc. of VLDB*, Sep. 1991.

[55] U. Schreier *et al.*, "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS," in *Proc. of VLDB*, 1991.

[56] E. N. Hanson, "The Design and Implementation of the Ariel Active Database Rule System," *IEEE TKDE*, vol. 8, no. 1, 1996.

[57] E. N. Hanson, "Ariel," in *Active Rules in Database Systems*, Norman W.Paton, Ed.  New York: Springer, 1999, pp. 221–232.

[58] N. H. Gehani, H. V. Jagadish, and O. Shmueli, "COMPOSE: A System For Composite Event Specification and Detection," AT&T Bell Laboratories, Tech. Rep., Dec. 1992.

[59] U. Dayal *et al.*, "The HiPAC Project: Combining Active Databases and Timing Constraints," *SIGMOD Record*, vol. 17, no. 1, pp. 51–70, Mar. 1988.

[60] N. H. Gehani and H. V. Jagadish, "Ode as an Active Database: Constraints and Triggers," in *Proc. of VLDB*, Sep. 1991, pp. 327–336.

[61] D. L. Lieuwen, N. H. Gehani, and R. Arlein, "The Ode Active Database: Trigger Semantics and Implementation," in *Proc. of ICDE*, Mar. 1996, pp. 412–420.

[62] A. P. Buchmann *et al.*, *Rules in an Open System: The REACH Rule System.*  Rules in Database Systems, 1993.

[63] A. Dinn, M. H. Williams, and N. W. Paton, "ROCK & ROLL: A Deductive Object-Oriented Database with Active and Spatial Extensions," in *Proc. of ICDE*, 1997.

[64] S. Gatziu and K. R. Dittrich, "SAMOS: An Active, Object-Oriented Database System," *IEEE Quarterly Bulletin on Data Engineering*, vol. 15, no. 1-4, pp. 23–26, Dec. 1992.

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

[66] P. Seshadri, M. Livny, and R. Ramakrishnan, "The Design and Implementation of a Sequence Database System," in *Proc. of VLDB*, 1996, pp. 99–110.

[67] A. Kotz-Dittrich, "Adding Active Functionality to an Object-Oriented Database System - a Layered Approach," in *Proc. of the Conference on Database Systems in Office, Technique and Science*, Mar. 1993.

[68] I. Motakis and C. Zaniolo, "Formal Semantics for Composite Temporal Events in Active Database Rules," *Journal of System Integration*, vol. 7, no. 3-4, pp. 291–325, 1997.

[69] I. Motakis and C. Zaniolo, "Temporal Aggregation in Active Database Rules," in *Proc. of SIGMOD*, 1997, pp. 440–451.

[70] V. Krishnaprasad, "Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation," Master's thesis, Database Systems R&D Center, CIS Department, The University of Florida, Gainesville, 1994. [Online].

[71] R. T. Simon and M. E. Zurko, "Separation of Duty in Role-Based Environments," in *Proc. of IEEE CSF Workshop*, 1997, pp. 183–194.

[72] R. Bhatti, A. Ghafoor, E. Bertino, and J. B. D. Joshi, "X-gtrbac: an xml-based policy specification framework and architecture for enterprise-wide access control." *ACM Transactions on Information and System Security*, vol. 8, no. 2, pp. 187–227, 2005.

[73] J. Bacon, K. Moody, and W. Yao, "A Model of OASIS Role-Based Access Control and its Support for Active Security," *ACM Transactions on Information and System Security*, vol. 5, no. 4, pp. 492–540, Nov. 2002.

[74] R. Chandramouli and R. S. Sandhu, "Role-Based Access Control Features in Commercial Database Management Systems," in *Proc. of NISSC*, 1998.

[75] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn, "A Role Based Access Control Model and Reference Implementation within a Corporate Intranet," *ACM Transactions on Information and System Security*, vol. 2, no. 1, pp. 34–64, 1999.

[76] R. Oppliger, G. Pernul, and C. Strauss, "Using Attribute Certificates to Implement Role-Based Authorization and Access Controls," in *Proc. of Fachtagung Sicherheit in Informationssystemen (SIS 2000)*, 2000.

[77] W. Yao, K. Moody, and J. Bacon, "A Model of OASIS Role-Based Access Control and its Support for Active Security," in *Proceedings, ACM Symposium on Access Control Models and Technologies*, 2001.

[78] J. Bacon, M. Lloyd, and K. Moody, "Translating Role-Based Access Control Policy Within Context." in *Proc. of POLICY*, 2001.

[79] L. Zhang, G.-J. Ahn, and B.-T. Chu, "A Role-based Delegation Framework for Healthcare Information Systems," in *Proceedings, ACM Symposium on Access Con-trol Models and Technologies*, June 2002.

[80] G. Neumann and M. Strembeck, "An Approach to Engineer and Enforce Context Constraints in an RBAC Environment," in *Proceedings, ACM Symposium on Access Control Models and Technologies*, 2003.

[81] T. M. Chalfant, "Role Based Access Control and Secure Shell — A Closer Look At Two Solaris™ Operating Environment Security Features," Enterprise Server Group, Sun Microsystems, Inc., July 2003. [Online].

[82] A. Galton and J. Augusto, "Two Approaches to Event Definition," in *Proceedings, International Conference on Database and Expert Systems Applications*. Springer-Verlag, 2002, pp. 547–556.

[83] R. Adaikkalavan, "Snoop Event Specification: Formalization, Algorithms, and Implementation using Interval-based Semantics," Master's thesis, Information Technology Laboratory, CSE Dept., The University of Texas at Arlington, Arlington, TX, U.S.A, 2002. [Online].

[84] R. Adaikkalavan and S. Chakravarthy, "ED-RBAC: A Flexible Event-Based Framework for Enforcing RBAC and its Extensions," CSE Dept., The University of Texas at Arlington, Tech. Rep. CSE-2004-2, Feb. 2004. [Online].

[85] 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 Man-agement)*, Tokyo, Japan, Apr. 2005, p. 1197.

[86] B. Shaiq, E. Bertino, and A. Ghafoor, "An optimal conflict resolution strategy for event-driven role based access control policies," CERIAS, Purdue University, Tech. Rep. 2005-08, 2005.

[87] "Committing to Security Benchmark Study: A CompTIA Analysis of IT Security and the Workforce (A White Paper Developed By TNS Prognostics)," The Computing Technology Industry Association, Mar. 2004.

[88] L. F. Cranor and S. Garfinkel, *Security and Usability: Designing Secure Systems that People Can Use.* O'Reilly, Aug. 2005.

[89] M. A. Al-Kahtani and R. Sandhu, "A Model for Attribute-Based User-Role Assignment," in *Proceedings, Annual Computer Security Applications Conference*, 2002.

[90] M. A. Al-Kahtani and R. Sandhu, "Induced Role Hierarchies with Attribute-Based RBAC," in *Proceedings, ACM Symposium on Access Control Models and Technolo-gies*, 2003.

[91] R. Adaikkalavan and S. Chakravarthy, "Formalization and Detection of Events Over a Sliding Window in Active Databases Using Interval-Based Semantics," in *Proceedings, East-European Conference on Advances in Databases and Information Systems*, Budapest, Hungary, Sep. 2004, pp. 241–256.

[92] R. Adaikkalavan and S. Chakravarthy, "Formalization and Detection of Events Using Interval-Based Semantics," in *Proceedings, International Conference on Management of Data*, Goa, India, Jan. 2005, pp. 58–69.

[93] R. Adaikkalavan and S. Chakravarthy, "SnoopIB: Interval-Based Event Specification and Detection for Active Databases (in press)," *Data and Knowledge Engineering*, 2005. [Online]. Available: http://dx.doi.org/10.1016/j.datak.2005.07.009

[94] R. Adaikkalavan, S. Chakravarthy, R. A. Liuzzi, and L. Wong, "Information Security: Using A Novel Event-Based Approach," in *International Conference on Information and Knowledge Engineering*, Nevada, USA, Jun. 2004, pp. 33–38.

[95] R. Adaikkalavan and S. Chakravarthy, "Discovery-Based Role Activations in Role-Based Access Control," in *Proceedings, IEEE International Performance Computing and Communications Conference (Workshop on Information Assurance)*, Phoenix, Arizona, USA, Apr. 2006, pp. 455–462.

[96] R. Adaikkalavan and S. Chakravarthy, "SmartGate: A Smart Push-Pull Approach to Support Role-Based Security in Web Gateways," in *Proceedings, Annual ACM SIG Symposium On Applied Computing*, Santa Fe, NM, USA, Mar. 2005, pp. 1727–1731.

[97] S. Chakravarthy, L. Elkhalifa, N. Desphande, R. Adaikkalavan, and R. Liuzzi, "Pattern Search over Streaming and Stored Data," in *Proc. of the ICAI*, Nevada, USA, Jun 2006.

[98] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "Towards an Integrated Model for Event and Stream Processing," CSE Dept., The University of Texas at Arlington, Tech. Rep. CSE-2004-10, 2004.

[99] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "$NFM^i$: An Inter-domain Network Fault Management System," in *Proceedings, International Conference on Data Engineering*, Tokyo, Japan, Apr. 2005, pp. 1036–1047.

[100] Q. He., "Privacy Enforcement with an Extended Role-Based Access Control Model," Department of Computer Science, NCSU, Tech. Rep. TR-2003-09, 2003.

[101] J. Hoagland, "Adage," 1999. [Online].

[102] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca, "A System to Specify and Manage Multipolicy Access Control Models," in *Proc. of POLICY*, 2002.

[103] M. Koch, L. V. Mancini, and F. Parisi-Presicce, "On the specification and evolution of access control policies," in *Proceedings, ACM Symposium on Access Control Models and Technologies*, 2001.

[104] "Authorization Manager (AzMan)." [Online]. Available: http://www.microsoft.com

[105] "Linux Intrusion Detection System (LIDS)." [Online].

[106] "Rule Set Based http://www.rsbac.org Access Control (RSBAC)." [Online]. Available:

[107] "Security Enhanced Linux." [Online].

[108] "grsecurity." [Online]. Available: http://www.grsecurity.net

[109] B. Shafiq, J. B. D. Joshi, and A. Ghafoor, "Petri-net Based Modeling for Verification of RBAC Policies," CERIAS, Purdue University, Tech. Rep. 2002-33, 2002.

[110] C. Liebig, M. Cilia, and A. Buchmann, "Event composition in time-dependent distributed systems," in *Proceedings, Fourth International Conference on Coopera-tive Information Systems.* Washington, DC, USA: IEEE Computer Society, 1999,

p. 70.

[111] P. Rönn, "Two Approaches to Event Detection in Active Database Systems," Master's thesis, Department of Computer Science (M.Sc. Dissertation), University of Skövde, 2001. [Online].

[112] 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.

[113] P. A. Bonatti and P. Samarati, "A uniform framework for regulating service access and information release on the web," *Journal of Computer Security*, vol. 10, no. 3, pp. 241–271, 2002.

[114] T. Yu, M. Winslett, and K. E. Seamons, "Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation," *ACM Transactions on Information and System Security*, vol. 6, no. 1, pp. 1–42, 2003.

[115] H. Koshutanski and F. E. Massacci, "Deduction, abduction and induction, the reasoning services for access control in autonomic communication," in *Proceedings, IFIP TC6 WG6.6 International Workshop on Autonomic Communication (WAC)*, 2004.

[116] "Web Filtering Products Feature Comparison," SurfControl, 2004. [Online].

[117] "Proxy SG$^{TM}$: Advanced Web Proxy," Blue Coat Systems. [Online]. Available: http://www.bluecoat.com/

[118] "SurfControl Web Filter," SurfControl, 2004. [Online].

[119] "Websense Enterprise," Websense, 2004. [Online].

[120] "Miscrosoft Internet Security and Acceleration (ISA) Server," Microsoft, 2004. [Online].

[121] "Border Manager," Novell, Inc., 2004. [Online].

[122] J. S. Park, R. S. Sandhu, and G.-J. Ahn, "Role-Based Access Control on the Web," *ACM Transactions on Information and System Security*, vol. 4, no. 1, pp. 37–71, 2001.

[123] T. Yan and H. Garcia-Molina, "The SIFT Information Dissemination System," *ACM Transactions on Database Systems (TODS)*, vol. 24, no. 4, pp. 529 – 565, December 1999.

[124] K. Aas, "Survey on personalized information filtering systems for the world wide web," Norwegian Computing Center, P.B. 114 Blindern, N-0314 Oslo, Norway, pp. 1–30, December 1997. [Online]. Available: http://citeseer.ist.psu.edu/466499.html

[125] M. Berry, *Survey of Text Mining : Clustering, Classification, and Retrieval*. New York : Springer-Verlag, 2004.

[126] U. Manber and S. Wu, "GLIMPSE: A Tool to Search Through Entire File Systems," in *Proceedings of the USENIX Winter 1994 Technical Conference*, San Fransisco, CA, USA, October 1994, pp. 23–32.

[127] M. Araújo, G. Navarro, and N. Ziviani, "Large text searching allowing errors," in *Proceedings of the South American Workshop on String Processing*, R. Baeza-Yates, Ed. Carleton University Press, 1997, pp. 2–20.

[128] J. P. Callan, W. B. Croft, and S. M. Harding, "The INQUERY Retrieval System," in *Proceedings of the International Conference on Database and Expert Systems Applications*, Spain, 1992, pp. 78–83.

[129] "Structured Query Retrieval in Lemur," http://www.lemurproject.org.

[130] Q. Jiang and S. Chakravarthy, "Data Stream Management System for MavHome," in *Proc. of ACM SAC*, Mar. 2004.

[131] S. Babu and J. Widom, "Continuous Queries over Data Streams," in *ACM SIG-MOD RECORD*, Sep. 2001.

[132] D. Abadi *et al.*, "Aurora: A New Model and Architecture for Data Stream Management," *VLDB Journal*, vol. 12, no. 2, Aug. 2003.

[133] J. Chen *et al.*, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases," in *Proc. of SIGMOD*, 2000.

[134] S. Madden and M. J. Franklin, "Fjording the Stream: An Architecture for Queries over Streaming Sensor Data," in *Proc. of ICDE*, 2002.

[135] M. F. Mokbel *et al.*, "PLACE: A Query Processor for Handling Real-time Spatio-temporal Data Streams," in *Proc. of VLDB*, Sep. 2004.

[136] P. Bonnet, J. E. Gerhke, and P. Seshadri, "Towards Sensor Database Systems," in *Proc. of MDM*, Jan. 2001.

[137] Y. Yao and J. E. Gehrke, "Query Processing in Sensor Networks," in *Proc. of CIDR*, Jan. 2003.

[138] S. R. Madden *et al.*, "The Design of an Acquisitional Query Processor for Sensor Networks," in *Proc. of SIGMOD*, 2003.

[139] S. R. Madden *et al.*, "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks," *In Proc. of OSDI*, Dec. 2002.

[140] L. H. Bjerring, D. Lewis, and I. Thorarensen, "Inter-Domain Service Management of Broadband Virtual Private Networks," *Journal of Network and Systems Management*, vol. 4, no. 4, pp. 355–373, 1996.

[141] R. Diaz-Caldera, J. Serrat-Fernandez, K. Berdekas, and F. Karayannis, "An Approach to the Cooperative Management of Multitechnology Networks," *Communications Magazine, IEEE*, vol. 37, no. 5, pp. 119–125, 1999.

[142] M. A. Mountzia and G. D. Rodosek, "Using the Concept of Intelligent Agents in Fault Management of Distributed Services," *Journal of Network and Systems Management*, vol. 7, no. 4, 1999.

[143] D. Medhi *et al.*, "A Network Management Framework for Multi-Layered Network Survivability: An Overview," in *IEEE/IFIP Conf. on Integrated Network Management*, May. 2001, pp. 293–296.

[144] R. Adaikkalavan and S. Chakravarthy, "SnoopIB: Interval-Based Event Specification and Detection for Active Databases," in *Proceedings, East-European Conference on Advances in Databases and Information Systems*. Germany: LNCS 2798, Sep. 2003, pp. 190–204.

[145] J. Allen, "Towards a general Theory of action and time," *Artificial Intelligence*, vol. 23, no. 1, pp. 23–54, 1984.

[146] J. Allen and G. Gerguson, "Action and Events in Interval Temporal Logic," *Journal of Logic and Computation*, vol. 4, no. 5, pp. 31–79, 1994.

[147] M. Strembeck, "Conflict Checking of Separation of Duty Constraints in RBAC - Implementation Experiences," in *Proc. of the Conference on Software Engineering*, 2004.

[148] R. Dasari, "Events And Rules For JAVA: Design And Implementation Of A Seamless Approach," Master's thesis, Database Systems R&D Center, CIS Department, The University of Florida, Gainesville, 1999.

[149] S. Castano, M. G. Fugini, G. Martella, and P. Samarati, *Database Security (ACM Press Book)*. Addison-Wesley, 1994.

[150] N. Dhanjani, "Hacking with Linux Kernel Modules," in *Proceedings of the Hack In The Box Conference*, Dec. 2003. [Online]. Available: http://dhanjani.com/presentations/hwlkm/hwlkm-hitb-2003.pdf

[151] L. Li and S. Chakravarthy, "An agent-based approach to extending the native active capability of relational database systems," in *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society, Mar. 1999, pp. 384–391.

[152] A. Vasudevan and R. Yerraballi, "SAKTHI: A Retargetable Dynamic Framework for Binary Instrumentation," in *Proceedings of the Hawaii International Conference on Computer Sciences*, Jan. 2004.

[153] G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," in *Proceedings of the 3rd USENIX Windows NT Symposium*, Jul. 1999.

[154] S. Chakravarthy, "Sentinel: an object-oriented dbms with event-based rules," in *Proceedings of the ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 1997, pp. 572–575.

[155] L. Bauer, J. Ligatti, and D. Walker, "Composing security policies with polymer," in *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, Chicago, 2005.

[156] R. Ramakrishnan and J. Gehrke, *Database Management Systems (3rd ed.)*. McGraw-Hill, 2003.

[157] J. B. D. Joshi *et al.*, "Generalized Temporal Role-Based Access Control Model - Specification and Modeling," CERIAS, Purdue University, Tech. Rep. 2001-47, 2001.

[158] J. B. D. Joshi, E. Bertino, and A. Ghafoor, "An Analysis of Expressiveness and Design Issues for the Generalized Temporal Role-Based Access Control Model," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 2, pp. 157–175, Apr-Jun. 2005.

[159] J. Widom and S. Ceri, Eds., *Active Database Systems: Triggers and Rules for Advanced Database Processing.* MK, 1996, ch. The HiPAC Project, pp. 177–206.

[160] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (2nd ed.).* McGraw-Hill, 2001.

[161] C. Adams and S. Lloyd, *Understanding PKI (2nd ed.).* Addison-Wesley, 2003.

[162] Join Bill Rose and Joe Lenski, "Internet and Multimedia 12: The Value of Internet Broadcast Advertising," Arbiton IBS and Edison Media Research, 2004. [Online].

[163] "Surfing at Work: Corporate Networks Are Paying the Price," SurfControl, 2004. [Online].

[164] Brian E. Burke, "Content Security: The Business Value of Blocking Unwanted Content," IDC, 2003. [Online]. Available: http://www.idc.com

[165] "Surfing at Work: Ethics in Computing," Dept of CSC, North Carolina State University (Raleigh, NC). [Online].

[166] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval.* New York: ACM Press / Addison-Wesley, 1999.

[167] G. Salton and M. McGill, *Introduction to Modern Information Retrieval.* New York: McGraw-Hill, Inc., 1983.

[168] L. Elkhalifa, "InfoFilter: Complex Pattern Specification and Detection Over Text Streams," Master's thesis, Information Technology Laboratory, CSE Dept., The

University of Texas at Arlington, Arlington, TX, U.S.A, 2004. [Online].

[169] 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.

[170] N. H. Gehani, H. V. Jagadish, and O. Shmueli, "Event Specification in an Object-Oriented Database," in *Proc. of SIGMOD*, San Diego, CA, June 1992, pp. 81–90.

[171] S. Gatziu and K. R. Dittrich, "Detecting Composite Events in Active Databases using Petri Nets," in *Proceedings of Workshop on Research Issues in Data Engineering*, Feb. 1994.

[172] R. Motwani *et al.*, "Query Processing, Resource Management, and Approximation in a Data Stream Management System," in *Proc. of CIDR*, Jan. 2003.

[173] Q. Jiang and S. Chakravarthy, "Scheduling Strategies for Processing Continuous Queries over Streams," in *Proc. of BNCOD*, Jul. 2004.

[174] B. Babcok *et al.*, "Operator scheduling in data stream systems," *The VLDB J.*, vol. 13, pp. 333–353, 2004.

[175] D. Carney *et al.*, "Operator Scheduling in a Data Stream Manager," in *Proc. of VLDB*, Sep. 2003.

[176] N. Tatbul *et al.*, "Load Shedding in a Data Stream Manager," in *Proc. of VLDB*, Sep. 2003.

[177] B. Babcock, M. Datar, and R. Motwani, "Load Shedding for Aggregation Queries over Data Streams," in *Proc. of ICDE*, Mar. 2004.

[178] A. Das, J. Gehrke, and M. Riedewald, "Approximate Join Processing over Data Streams," in *Proc. of SIGMOD*, 2003.

[179] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "$NFM^i$: An Inter-domain Network Fault Management System," in *Proc. of ICDE*, Apr. 2005.

[180] A. Arasu *et al.*, "Linear Road: A Stream Data Management Benchmark," in *Proc. of VLDB*, Sep. 2004.

[181] J. Baras, H. Li, and G. Mykoniatis, "Integrated, Distributed Fault Management for Communication Networks," University of Maryland, Tech. Rep. CS-TR 98-10, Apr. 1998.

[182] D. Gambhir, M. Post, and I. Frisch, "A Framework for Adding Real-Time Distributed Software Fault Detection and Isolation to SNMP-based Systems Management," *Journal of Network and Systems Management*, vol. 2, no. 3, 1994.

[183] P. Frohlich and W. Nejdl, "Model-based Alarm Correlation in Cellular Phone Networks," in *Proc. of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, Jan. 1997.

[184] C. T. Team, "In-memory Data Management for Consumer Transactions the Timesten Approach," in *Proc. of SIGMOD*, Jun. 1999.

## BIOGRAPHICAL STATEMENT

Raman Adaikkalavan was born in Pudukkottai, India, in 1978. He received his Bachelor of Engineering degree in Computer Science Engineering from Bharathidasan University, Tamilnadu, 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. He has worked as graduate teaching assistant, faculty associate, and instructor in the Computer Science and Engineering department. He received his Doctor of Philosophy in Computer Science and Engineering from The University of Texas at Arlington, in August 2006. He is a member of TBP, UPE, ACM and IEEE. He is also a recipient of the University Scholar award twice and is listed in the Who's Who Among Students in American Universities and Colleges. His current research interests include security and privacy in databases, information retrieval, grid, distributed and pervasive environments and complex event processing.