A GENERALIZED TEMPORAL ROLE BASED ACCESS CONTROL MODEL FOR DEVELOPING SECURE SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

James B. D. Joshi

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2003

This thesis is dedicated to my parents, to Bhairab and Ganesh, and to my wife.

ACKNOWLEDGMENTS

I would like to sincerely thank my PhD advisor Prof. Arif Ghafoor for his invaluable guidance and support at each step of my graduate studies at Purdue University. I would also like to express my gratitude to Professors Mary P. Harper, Eugene H. Spafford and Hong Z. Tan for their participation in my PhD committee. Their guidance and suggestions have been very valuable. In particular, Prof. Spafford provided several stimulating ideas.

I am highly indebted to Center of Education and Research in Information Assurance and Security (CERIAS) at Purdue University for the unfettered support provided throughout my doctoral studies. Without such support my PhD studies would not have been possible. I would like to thank all my colleagues in the Distributed Multimedia Systems Lab at Purdue University for their cooperation and help. In addition, I would like to acknowledge the support provided by the National Science Foundation through the Grant# IIS-0209111.

I am forever indebted to Prof. Elisa Bertino for her unequivocal help. I relied heavily on her feedback throughout my research.

Finally, nothing would have been possible without the love and support of my family.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

James B. D. Joshi. Ph.D.. Purdue University, August 2003. A Generalized Temporal Role Based Access Control Model for Developing Secure Systems. Major Professor: Arif Ghafoor.

A key issue in computer system security is to protect information against unauthorized access. Emerging workflow-based applications in healthcare, manufacturing, the financial sector, and e-commerce inherently have complex, time-based access control requirements. To address the diverse security needs of these applications, a Role Based Access Control (RBAC) approach can be used as a viable alternative to traditional discretionary and mandatory access control approaches. The key features of RBAC include policy neutrality, support for least privilege, and efficient access control management. However, existing RBAC approaches do not address the growing need for supporting time-based access control requirements for these applications.

This research presents a Generalized Temporal Role Based Access Control (GTRBAC) model that combines the key features of the RBAC model with a powerful temporal framework. The proposed GTRBAC model allows specification of a comprehensive set of time-based access control policies, including temporal constraints on role enabling, user-role and role-permission assignments, and role activations. The model provides an event-based mechanism for supporting dynamic access control policies, which are crucial for developing secure workflow-based enterprise applications. In addition, the temporal hierarchies and separation of duty constraints facilitated by GTRBAC allow the development of security policies for commercial enterprises. The thesis provides various design guidelines for managing complexity and building secure systems based on this model. X-GTRBAC, an XML-based policy language has been developed to allow specification of GTRBAC policies.

1. INTRODUCTION

1.1 Research Motivation and Problem Statement

The rapid proliferation of the Internet and the cost effective growth of its key enabling technologies such as the World Wide Web, database systems, storage and end-systems, and networking are revolutionizing information technology and have created unprecedented opportunities for developing large scale distributed applications. The emerging trend indicates that information systems are increasingly being interconnected for sharing data and applications. Applications such as workflow management systems (WFMSs) are expected to play a critical role in many distributed applications, including e-commerce, finance and banking, manufacturing, corporate databases, on-line services and businesses, on-line health care services and many others. Such workflow-based applications are subject to time-based constraints [Alt96, Att93, Ber99b, Ede99, Tho97].

Information systems security refers to the protection of information systems against unauthorized access to or modification of information, whether in storage, processing or transit, and against denial of service to authorized users, including measures necessary to detect, document, and counter such threats. This is achieved by accomplishing the following set of security goals [Jos01b]:

Confidentiality: The goal of confidentiality is to ensure that an unauthorized person does not access information while it is in data storage, during processing and in transit.

Integrity: The goal of information integrity is to protect information from unauthorized modification done either intentionally or accidentally.

Availability: Information availability ensures that information is available when needed and is not made inaccessible by malicious data denial activities.

Accountability: Information accountability ensures that every action of an entity can be uniquely traced back to it.

Assurance: Security assurance is the degree of confidence in the security of the system with respect to predefined security goals.

Authentication, access control, and auditing have been traditionally considered as the key security services providing the foundation for information and system security [San96b]. Each access request is usually mediated by a reference monitor. Several models of access control have been proposed in the literature to address diverse security needs of information systems; however, these models have been found inadequate in addressing the complex security requirements of the emerging applications [Jos01b, San94, San96b]. In this research, we focus on the temporal access control requirements in large organizations and emerging applications.

Security models that support efficient security management and enforcement, and capture a wide range of time-based, dynamic access control requirements of applications, can provide an important framework for developing secure systems [Bac02, Ber01a, Cla87, Fer93]. Such a need has been highlighted by several surveys and reports, and technological needs of the e-commerce environments [Bar97, Gar96, Gar97]. We briefly discuss some of these issues that have motivated the research reported in this dissertation.

1. In large corporate information systems, the *insider-attack* is a growing security concern. A joint study on computer crimes conducted by the Computer Security Institute (CSI) and the FBI indicates that the most serious losses in enterprises occur through unauthorized access by insiders, and 71% of the respondents had detected unauthorized access by insiders [Pow00, Gho98]. The challenge is in developing new security models or extending existing ones that allow efficient access control management and administration of organizational information assets.
2. The Auditing Report published in November, 2001, by the US General Accounting Office for 24 of the largest federal agencies indicated that security management and access control were the most significant weaknesses in all these agencies [Gao02]. This report has a significant bearing in the need for a robust national defense in the light of growing threat to critical infrastructures of the country.
3. As mentioned earlier, many emerging applications have time-based access control requirements as they employ workflow management systems (WFMSs) where tasks have a temporal dimension [Ede99]. Furthermore, the size and complexity of these applications are increasing rapidly. Flexible security models that support viable security administration are essential to protect the organizational information assets. The traditional access control models such as Discretionary and Mandatory Access Control (DAC and MAC) models have several limitations when applied to emerging applications, in terms of supporting both security management and a wide range of

access requirements [Jos01b, San94, San96b]. Bhavani *et. al.* enlist the following crucial requirements for secure e-commerce and web based applications [Thu01].

a. *Tools and mechanisms to support access control policy specification and enforcement*
b. *Secure workflows that have time constrained security requirements*
c. *Secure federations of collaborating organizations.*

In other words, there is a crucial need for models that can express flexible access control policies and can be used in time-constrained application environments such as WFMSs. Such models should be able to provide support in environments supporting federations of organizations. While such a comprehensive model is yet to be developed, role based access control models have been perceived as the most promising approaches for addressing these challenges.

Role based access control (RBAC) models are receiving increasing attention as a generalized approach to access control [Fer01, Giu95, Giu97, Jos01a, Jos01b, Ker02, Nya93, Nya99, Osb00a, San95, San96a, San97, San98a, Tar97b]. A survey conducted by NIST [Fer93] shows that in many organizations the access control decision is based on a person's role and responsibilities within the organization, making role-based approaches suitable for expressing security requirements. In [Cla87], Clark *et. al.* show that the traditional DAC and MAC policies do not adequately address the diverse security needs of many organizations. RBAC approach can greatly simplify security administration. For example, if a user moves to a new function within the organization, he/she can simply be assigned to the new role and removed from the earlier role, whereas in the absence of an RBAC model, his/her old privileges need to be revoked, and new privileges need to be granted. An authorization constraint relevant and well known in commercial application environments is the *separation of duty* (SoD) constraint [Ahn00, Ber99b, Bew89, Kun99, Nya99, San91, Sim97, Tid98]. SoD constraints aim at reducing the risk of fraud by not allowing any individual to have sufficient authority within the system to single-handedly perpetrate a fraud. RBAC models allow expressing a wide variety of SoD constraints that are beneficial to many applications. Roles can be organized into hierarchies to capture organizational functional hierarchies and to define a role's permissions inherited by other roles. A role hierarchy can significantly reduce explicit permission assignments to a role and hence can considerably reduce the administration overhead. Furthermore, RBAC models are policy-neutral [Jos01b, San98b]. In particular, by appropriately configuring a role-based system, one can support different policies, including both DAC and MAC policies [Nya95, Osb00b]. Such flexibility of RBAC models is extremely significant, as the can be adapted to support the access control needs of enterprise-wide security administration and enforcement.

Although RBAC modeling is now a mature field of research, no existing RBAC models can handle fine-grained time-based access control requirements. Examples of time-based RBAC policies abound. For instance, a *part-time staff member* in a company may be authorized to work within the company only on working days between 9am and 1pm. If a *part-time staff* member is represented by a role, enforcing such rules requires that the part-time employee assume the role in that interval only. Similarly, an *external auditor* may need access to organizational financial resources for the assigned period of three months. Such requirements can be supported by specifying times when the role can be enabled so that a legitimate user can activate it. Roles can thus be enabled/disabled at a certain time. A *part-time staff member* or an *external auditor* role may be further restricted to only pre-specified hours of active time in one session. Development of such a temporal RBAC model is highly desirable in order to address the comprehensive security requirements of organizations and applications.

1.2 Summary of Contributions

In this research, we address the need for a powerful and flexible time-based access control model, pointed out in the earlier section. Our main objectives are:

1. to develop a model that can express a wide range of time-based access control requirements of organizations,
2. to develop a user friendly policy specification language and enforcement mechanism that can be used in a wide range of applications.

The contributions of the research reported in this thesis can summarized as follows:

1. We propose a Generalized Temporal Access Control (GTRBAC) model that extends the basic RBAC model by introducing a comprehensive set of temporal constraints. These constraints include periodicity and duration constraints on role enabling and assignments, as well as duration and cardinality constraints on role activation. The event-based framework of GTRBAC allows modeling run-time events and triggers, which can be used to express dynamic access control requirements.
2. In this research, we investigate the issue of how permission inheritance and role activation semantics can be captured when the hierarchically related roles have

temporal constraints. Contribution with respect to our work in role hierarchy include the following:

a. We introduce various types of role hierarchies and provide their permission-inheritance and role-activation semantics in the presence of various temporal constraints on the hierarchically related roles. We identify various scenarios where hierarchical structure can play an important role.
b. A *hybrid* role hierarchy consisting of different hierarchy relations among roles can induce different types of *derived* hierarchical relations among roles that are not directly related. We propose a set of inference rules for the *derived* hierarchical relations and prove that these rules are *sound* and *complete*. A security administration tool can use these rules to identify possible flaws in the policy specification of an enterprise. Such flaws may be induced by complex hierarchical relations among roles.
c. Given a complex *hybrid* hierarchy, we provide a mechanism for generating the sets of roles that a user can activate within a session simultaneously. Sessions in RBAC, in which the users activate one or more roles, correspond to subjects in traditional access control models [Osb97, Osb00b, San96c, San98b]. Hence, generating such sets of roles for a user in a session is of crucial importance to capture policies that are defined with respect to the traditional notion of subject.
d. Role hierarchies evolve with time. New roles may be added and existing ones deleted or modified. We present rules for handling such hierarchy evolution.

3. An open issue for a model with a constraint language is its expressiveness and minimality. In other words, it is important to determine whether the set of constraints for the model is minimal. If the model is not minimal, an important issue is to determine whether the non-minimal model provides any practical benefits over the minimal model. It is thus possible a non-minimal model or a model with all the constraints can be more flexible in terms of complexity and usability than the minimal model. Given the large variety of such languages that have been recently proposed, issues concerning expressive power and minimality for RBAC constraint languages are extremely relevant [Ahn00, Cra03, Neu03].

   In this dissertation, we show that there exists a minimal model that has a subset of constraint types defined in the GTRBAC model and yet has the same expressive power as the GTRBAC model. In addition, we show that the sets of different constraint types can be used to generate a family of GTRBAC models having the same expressive power. We show that the GTRBAC model, although is not minimal, has several advantages in terms of complexity of specification and usability.

4. Constraints have been considered as a very important aspect of policy specifications. In this thesis, we investigate the cardinality, dependency and SoD constraints within the framework of GTRBAC. In particular, we incorporate the following constraints in our model.
    a. We introduce a generic framework for expressing a wide range of time-based cardinality constraints. The cardinality constraint expression framework provides specifying cardinality control with respect to all the GTRBAC states.
    b. We develop an elaborate trigger expression that can capture complex dependencies among events and conditions. In particular, we define control flow dependency (CFD) constraints that can be used to express access control requirements that are typical in workflow types of applications. Furthermore, we show that the trigger-based framework and the CFD constraint expressions can be easily extended to provide an elaborate time-based RBAC model for context-based access control.
    c. We identify a comprehensive set of SoD constraints for using the GTRBAC framework. These SoDs subsume the SoDs that have been identified earlier in the RBAC literature, and provide a modeling capability at a finer level of granularity.

1.3 Outline of Dissertation

The dissertation is organized as follows. In Chapter 2, we present the related work in information system security. In Chapter 3, we present the GTRBAC model, and discuss various temporal constraints and their execution semantics. In Chapter 4, we first introduce various types of temporal role hierarchies and then present detailed analysis and techniques to compute sets of roles users can activate in a session, to derive induced hierarchical relations using inference rules and to handle hierarchy evolution efficiently. In Chapter 5, we present a comprehensive set of the time-based cardinality, dependency and separation of duty constraints. In Chapter, 6, we present the minimality results and discuss constraint design issues. Chapter 7 will present the X-GTRBAC policy specification language and discuss its syntax. Finally, in Chapter 8, we provide our conclusions and future work.

2. RELATED WORK

In this chapter, we briefly review work related to information system security. In particular, we present background on access control models and then describe work related to authorization constraints, role hierarchies and time-based access control.

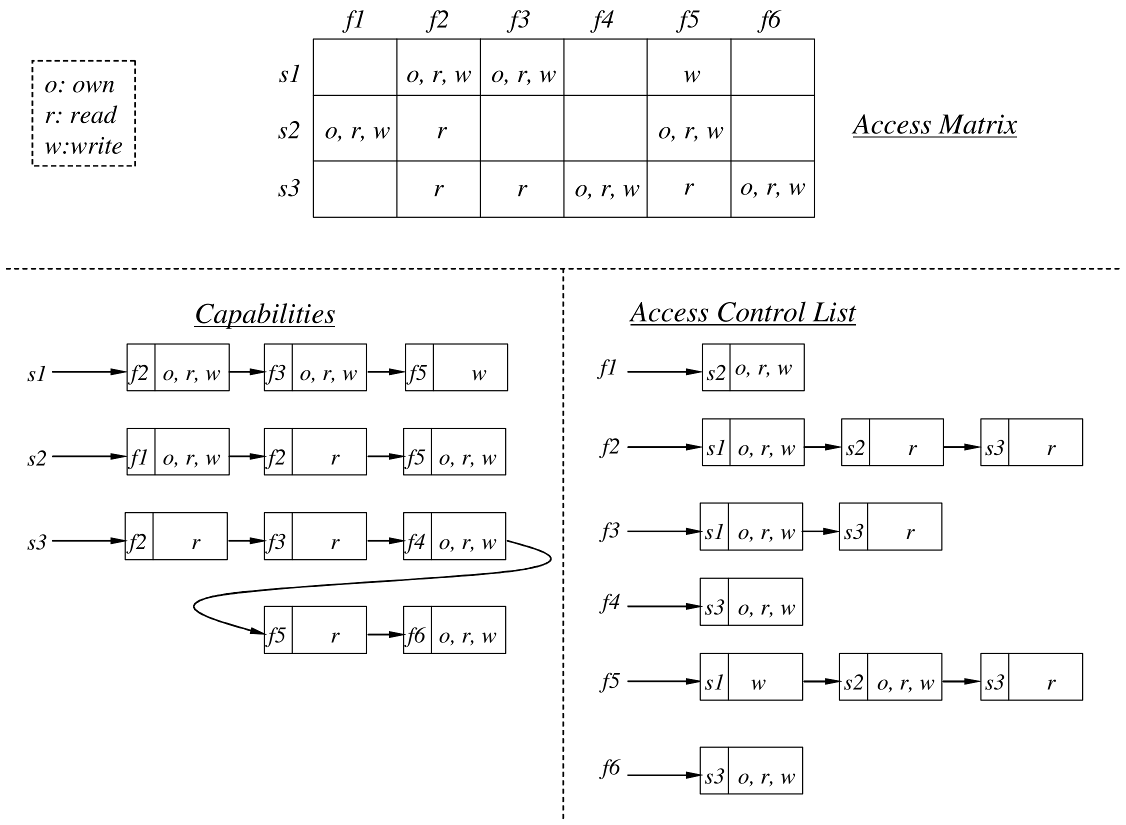
2.1 Traditional Access Control Models

Traditional access control approaches are broadly categorized as discretionary access control (DAC) [Gra72, Har76, Jaj97, Lam71, San94] and mandatory access control (MAC) [Bel76, Bib77, Den761, Lam73, Mcl90]. In the following sections, we briefly overview these approaches.

2.1.1 Discretionary Access Control (DAC)

In DAC, the basic premise is that subjects have ownership over objects of the system and subjects can grant or revoke access rights on the objects they own to other subjects at the original subject's discretion [Har76, San94]. Subjects can be users, groups, or processes that act on behalf of other subjects. DAC policies are flexible and the most widely used [San96b]. However, these policies do not provide high security assurance. For example, DAC allows copying of data from one object to another, which can result in allowing access to a copy of data to a user who does not have access to the original data. Such risks can propagate to the entire interconnected environment, causing a serious violation of security goals. This allows a *Trojan horse* program to easily leak confidential information without the knowledge of the subject accessing the object. A *Trojan horse* program is one that appears to be doing one thing on the surface but is actually doing something else without the knowledge of the person using it. The *Trojan horse* problem has been considered the main reason that has led to a distinction between DAC and MAC [Mcl94].

The genesis of DAC is generally considered to be the access control matrix (ACM) model of confidentiality formulated by Lampson [Lam71] and refined by Graham and Denning [Gra72]. Structurally, the model is a state machine with each triple (*S, O, M*) defining a state [Mcl94], where *S* is a set of subjects, *O* is a set of objects, and *M* is an access matrix. *M* has |*S*| rows and |*O*| columns, and the content of *M*[*s, o*] indicates the rights that *s* has over *o*. Fig. 2.1, shows an access matrix that contains the permissions that subjects *s1, s2* and *s3* have on the six different file objects.


o: own
r: read
w:write
Access Matrix
Capabilities
s1 → f2 o, r, w → f3 o, r, w → f5 w
s2 → f1 o, r, w → f2 r → f5 o, r, w
s3 → f2 r → f3 r → f4 o, r, w → f5 r → f6 o, r, w
Access Control List
f1 → s2 o, r, w
f2 → s1 o, r, w → s2 r → s3 r
f3 → s1 o, r, w → s3 r
f4 → s3 o, r, w
f5 → s1 w → s2 o, r, w → s3 r
f6 → s3 o, r, w

	f1	*f2*	*f3*	*f4*	*f5*	*f6*
s1		*o, r, w*	*o, r, w*		*w*	
s2	*o, r, w*	*r*			*o, r, w*	
s3		*r*	*r*	*o, r, w*	*r*	*o, r, w*

Fig. 2.1. An *access control matrix*, and its *access control list* and *capability list* representations

The HRU Access Control Model and Derivatives

Harrison, Ruzzo and Ullman [Har76] used the concept of the access matrix proposed by Lampson for the purpose of decidability analysis. The HRU access matrix model uses a set of commands to construct an authorization scheme, which is of the following form.

```
If
   a1 in M[s1, o1] and
     …
   am in M[sm, on]
then
   op1
   …
   opn
```

where each op_i is of form:

```
enter a into M[s, o], delete a from (s, o), create subject s,
create object o, destroy subject s, destroy object o.
```

Each command has a *body* part that contains primitive operations *op_is* and the *condition* part as shown above. The body part is allowed to execute if the rights specified in the conditions exist in the ACM. HRU's formulation of the safety problem is [Har76]:

(Safety Problem) *Is there a reachable state in which a particular subject possesses a particular privilege, which it did not previously possess*?

In [Har76], researchers show that the *safety* in HRU is, in general, *undecidable.* For the mono-operational case, where the body part consists of a single primitive operation, they further show that *safety* is *decidable*.

Resembling HRU closely is the Schematic Protection Model (SPM) by Sandhu *et. al.* [Amm92 San88] that introduces the notion of security types - *subject types* and *object types*. It has been shown that SPM is formally equivalent to monotonic HRU. Every subject or object is created to be of a particular type. The SPM has been shown to be flexible and able to formulate policies 'in between' the MAC and DAC policies. Another similar HRU extension can be found in Typed Access Matrix (TAM) [San92a] where again, the subjects and objects are strongly typed.

Yet another model closely related to the HRU access matrix model is the Dynamically Typed Access Control (DTAC) model by Tidswell *et. al.* [Tid98, Osb00a]. DTAC addresses the security issues in a highly dynamic environment. In DTAC, subjects and objects have no distinction and the access decisions are based on the *security types*.

Subjects are grouped into *security types* representing the subsystem to which they belong. Objects are grouped into security types, which encode the format of the information contained within the objects. DTAC uses dynamic typing, unlike the TAM and SPM models. A safety invariant is maintained by static analysis and the dynamic checks. Some benefits of DTAC include its use in modeling task-based security because of its ability to handle dynamic environments and the reduced size of configuration that can be achieved by grouping entities into types. These models are still in the theoretical stage of development.

An access matrix implementation in a large system often results in a big matrix with many empty entries [San94]. Because of this, an access matrix is rarely implemented in the actual form of a matrix. Two approaches commonly used to represent the information contained in an access matrix include [San94]:

- *Access Control List* (ACLs) and
- *Capabilities*

ACL is a popular implementation form of an access matrix. In this implementation, an ACL is associated with each object and it contains information about the rights each subject has on the object. It is easy to determine the access rights a subject has on an object using ACLs. Revoking access rights of a subject on an object is easy as it simply involves removing the entry in the ACL of the object. However, the disadvantage of this approach is that it is difficult to determine all the rights a particular subject has. To do this, one must check all the ACLs in the system. Similarly, if all the rights of a subject need to be revoked, again all the ACLs need to be traversed.

The dual approach of ACL is the *Capabilities*. In this approach, each subject is associated with a list, called a *Capability list*, which contains the information about the rights the subject has on each object of the system. Whereas ACL corresponds to storing the columns of access matrix with objects, *Capabilities* correspond to storing rows of the access matrix with the subjects. It is easy to check the total set of access rights of a subject using this approach. However, determining the subjects that have access rights to a particular object is difficult as this necessitates checking all the *Capability* lists in the system. Fig. 2.1 shows the ACL and *Capabilities* representations of an access control matrix.

2.1.2 Mandatory Access Control (MAC)

In MAC, all subjects and objects are classified based on some predefined sensitivity levels that are used in an access decision [Bel76, Bib77, Mcl90,, San94]. These levels generally form a lattice structure, and hence a MAC policy is sometimes known as a lattice-based policy [San92b, San93]. An important goal of MAC is to control information flow in order to ensure confidentiality and integrity of information, which DAC does not do. For example, to ensure information confidentiality in defense applications, Bell-LaPadula (BLP) model [Bel76], also known as multilevel model, can be used. BLP model is the best-known model for MAC [Mcl90]. It controls information flow by enforcing the *no read-up* and *no write-down* rules given as:

- *Simple security property*: a subject *s* is allowed to read an object *o* iff $l_s \geq l_o$ (*no-readup* property) where l_s and l_o are *clearance* and *classification* levels of *s* and *o*.
- **property*: a subject *s* is allowed to write an object *o* iff $l_s \leq l_o$ (*no-writedown* property)

To achieve information integrity, the access rules can be formulated as *no read-down* and *no write-up* [Bel76, San93] as first proposed by Biba [Bib77]. Biba's integrity model follows along the line of the *BLP* model in that its aim is to control flow of lower-integrity information to higher-integrity objects but allow the flow in the opposite direction [San93]. It is possible to combine the BLP model and the Biba model to get a composite model that provides both confidentiality and integrity using lattices [Bib77, San93].

Unlike DAC, MAC provides protection for data that is more robust, and deals with more specific security requirements, such as information flow control policy. However, enforcement of MAC policies is often a difficult task, and in particular, for many commercial organizations [Cla94], they do not provide viable solutions because they lack adequate flexibility. Furthermore, organizational security needs are often a mixture of policies that may need to use both DAC and MAC, which necessitates seeking solutions beyond those provided by DAC and MAC only [San94]. An example of such a policy is the *Chinese wall policy* [Bew89]. The *Chinese wall policy* has been proposed to mainly address access control requirements in *conflict of interest* situations that occur in commercial sectors such as in consulting firms. For example, a firm is providing a consulting service to two companies that are competitors e.g., banks A and B. A consultant should not be able to access confidential information of both the banks. Sandhu *et. al.* show that a lattice structure can be used to represent such a policy [San93].

2.2 Role-based Access Control

Role based access control (RBAC) is a flexible approach that has generated great interest in the security community [Fer01, Giu95, Giu97, Jos01a, Jos01b, Ker02, Nya93, Nya99, Osb00a, San95, San96a, San97, San98a, Tar97b]. In RBAC, users are assigned memberships to roles and these roles are in turn assigned permissions as shown in Fig. 2.2. A user can acquire all the permissions of a role of which he is a member. Role-based approach naturally fits into organizational contexts as users are assigned organizational roles that have well-defined responsibilities and qualifications [Fer93].

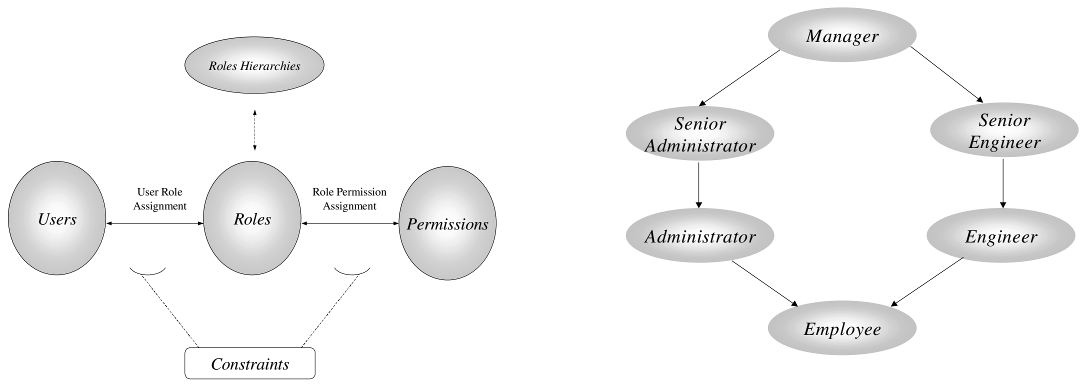


Fig. 2.2. Constraints and hierarchy in RBAC

According to a survey conducted by the U.S. National Institute of Standards and Technology (NIST) [Fer93], RBAC has been found to address many needs of the commercial and government sectors. This study showed that access control decisions in many organizations are based on "*the roles that individual users take on as part of the organization*." Many surveyed organizations indicated that they had unique security requirements and the available products did not have adequate flexibility to address them.

RBAC approach has several advantages, the key among which include [Jos01b, San94, San96a]:

- *Security management*: The *role in the middle* approach to access control removes the direct association of the users from the objects. This logical independence greatly simplifies management of authorization in RBAC systems. For example, when a user changes his role, all that needs to be done is to remove his membership from the current role and assign him to the new role. In case authorizations were specified in terms of direct associations between the user and the individual objects, this change would require revoking permissions granted to all the objects and explicitly granting permissions to the new set of objects. Using a role-based approach, the number of assignments of users to permissions is considerably reduced. Generally, a system has a very large number of subjects and objects, and hence, using RBAC has benefits in terms of managing permissions.

- *Role hierarchy*: Natural role hierarchies exist in many organizations based on the principle of generalization and specialization [San96c]. For example, there may be a general *Employee* role in a Consulting Firm as shown in Fig. 2.2: *Employee, Engineer, Senior Engineer, Administrator, Senior Administrator* and *Manager*. Since everyone is an employee, the *Employee* role models the generic set of access rights available to all. A *Senior Engineer* role will have all the permissions that an *Engineer* role will have, who in turn will have the permissions available to the *Employee* role. Thus, permission inheritance relations can be organized in role hierarchies. This further simplifies management of access permissions. Fig. 2.2 shows a simple hierarchy.
- *Principle of Least Privilege*: RBAC can be configured to assign the least set of privileges from a set of roles assigned to a user when that user signs on. Using least privilege set minimizes the damage incurred to a system if someone not assigned to a role acquires its permissions through other means, or if someone masquerades as another user [Jos01b, San94, san96a].
- *Separation of Duties*: Separation of duties (SOD) has been considered a very desirable organizational security requirement [Ahn00, Ber99b, Bew89, Kun99, Nya99, San91, Sim97, Tid98]. SOD constraints are enforced mainly to avoid possible fraud in organizations. RBAC can be used to enforce such requirements easily – both statically and dynamically. For example, a user can be prevented from being assigned to two roles to prevent possible fraud by using a *static* SOD which says that a user cannot be assigned to two roles, one of which prepares a check and the other authorizes it.
- *Grouping Objects*: Roles classify users according to the activity or the access needs based on the organizational functions they carry out. Similar classifications can also be possible for objects. For example, a *secretary* generally has access to all the memos and letters in his/her office, whereas an accountant has access to all the bank accounts belonging to his/her organization. Thus when permissions are assigned to roles, it can be based on object classes instead of individual objects [San96a]. This further increases the manageability of authorizations.
- *Policy-neutrality*: Role-based approach is policy-neutral and is a means for articulating policy [Jos01b, San96a]. Role-based systems can be configured to represent many useful DAC, MAC policies [Nay95, Osb97, Osb00b] and user-defined and organizational security policies.

2.2.1 The NIST RBAC Model

Recently, Ferrailo *et. al.* have proposed the NIST-RBAC (National Institute for Science and Technology RBAC) [Fer01] as a standard reference model. Depicted in Fig. 2.3, NIST-RBAC uses a four-level system in which each higher level includes the functional capabilities of all the levels below it. The levels correspond to four RBAC models: *flat*, *hierarchical*, *constrained*, and *symmetric*.

The flat RBAC model provides the minimal features essential for any RBAC mechanism. These include roles, user-role assignment, and role-privilege assignment. Hierarchical RBAC includes as a requirement role hierarchies that define relationships among roles in a domain. Constrained RBAC requires SOD.

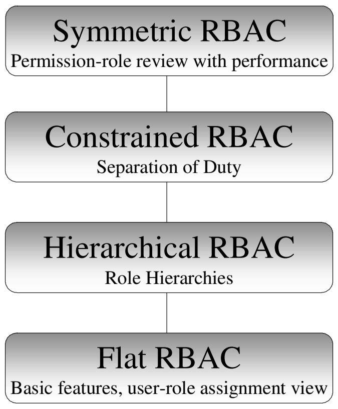


Fig. 2.3. Proposed NIST RBAC Model

The symmetric RBAC model adds a permission-role review requirement. As a result, the model allows identification of the permissions assigned to existing roles and vice versa. RBAC approach is an attractive candidate for use in a multidomain environment because of its flexibility, generality and easy manageability.

The NIST RBAC model as proposed by Ferraiolo *et. al.* consists of four basic components: a set of users `Users`, a set of roles `Roles`, a set of permissions `Permissions`, and a set of sessions `Sessions` [Fer01]. A user is a human being or an autonomous agent. A role is a collection of permissions needed to perform a certain function within an organization. A permission refers to an access mode that can be exercised on an object in the system and a session relates a user to possibly many roles. In each session, a user can request to activate some subset of roles he is authorized to assume. Such a request is granted only if the corresponding role is enabled at the time of the request and the user is entitled to activate the role at that time. Several functions are defined for the sets `Users, Roles, Permissions`, and `Sessions`. The *user role assignment* (UA) and the *role permission assignment* (PA) functions model the assignment of users to roles and the assignment of permissions to roles, respectively. The *user* function maps each session to a single user, whereas the *role* function establishes a mapping between a session and a set of roles activated by the corresponding user in the session. On `Roles`, a hierarchy is denoted by ≥. For roles r_i, $r_j \in$ `Roles`, if $r_i \geq r_j$, then r_i inherits the permissions of r_j. In such a case, r_i is a senior role and r_j a junior role.

The GTRBAC model proposed in this dissertation is an extension of the NIST RBAC model. The extensions are essentially with respect to the temporal constraints.

2.2.2 Role Hierarchy

Many researchers have highlighted the importance and use of role hierarchies in RBAC models [Giu95, Giu97, Mof98, Mof99, Nya99, San96c, San98]. A properly designed role hierarchy allows efficient specification and management of access control structures of a system. When two roles are hierarchically related, one is called the senior and the other the junior. The senior role inherits all the permissions assigned to the junior roles. The inheritance of permissions assigned to junior roles by a senior role significantly reduces assignment overhead, as the permissions need only be explicitly assigned to the junior roles.

Even though the notion of role hierarchy has been widely investigated, to our knowledge, no earlier work has addressed the implication of the presence of temporal constraints on role hierarchies, which is the focus of our work. In particular, in this dissertation, we present a detailed analysis of role hierarchy in the presence of various temporal constraints with respect to the GTRBAC model and show that there are various distinctions that need to be made about the inheritance semantics of a role hierarchy.

It is important to point out that Sandhu [San98] and Moffet [Mof98] have already recognized the limitations of the pure inheritance semantics proposed in the RBAC96 family of models [San96a]. Sandhu [San98] has proposed the ER-RBAC96 model that incorporates a distinction between two types of role hierarchy: *usage* hierarchy that applies *permission-inheritance* semantics and *activation* hierarchy that uses *activation-inheritance* semantics. In a *usage* hierarchy, the activation of a *senior* role allows a user to acquire all the permissions of all of its junior roles but no user assigned only to the senior role is allowed to activate the junior roles. An *activation* hierarchy extends "*permission inheritance hierarchy to roles that are stipulated to have dynamic separation of duty* (SoD)" [San98]. Our analysis further strengthens his arguments and shows that, in the presence of timing constraints on various entities, the separation of the *permission-inheritance* and the *activation-inheritance* semantics provides a basis for capturing various inheritance semantics of a hierarchy. We show that these hierarchies can further be divided into sub-types, to account for the subtle effects of temporal constraints. In another important work related to role hierarchies, Moffet *et al.* [Mof98, Mof99] have identified the need for three types of hierarchies – *isa* hierarchy, *activity* hierarchy and *supervision* hierarchies – in order to address the needs of control principles in an organization, which include *separation of duty*, *decentralization* and *supervision and review* [Mof99]. They show that the complete inheritance within a hierarchy can limit a hierarchy from achieving organizational control needs. Clearly, our temporal hierarchies as well as Sandhu's hierarchies provide a basis for limiting such complete inheritance in a hierarchy, making it possible to support separation of duty and restricted inheritance in a hierarchy. Furthermore, Moffett *et. al.* [Mof99] point out that the commercial organizations' demand for a *dynamic* access control model that can support dynamic authorization states as well as dynamic propagation of access rights has largely been neglected. As we show in this dissertation, the proposed GTRBAC's temporal framework and the trigger mechanism along with the temporal hierarchies provide a strong basis for such dynamic features in an access control model.

Nyanchama *et. al.* address the transformation of hierarchies in terms of the addition, deletion and partitioning of roles in the context of access rights administration [Nya94]. However, the analysis is limited to hierarchies that contains only one type of hierarchy among roles and does not indicate how the transformations are affected by the presence of other constraints on hierarchical roles. We analyze the transformation of a role hierarchy in the presence of multiple hierarchy types and constraints on hierarchically related roles.

2.2.3 Constraints in RBAC

Mainly two kinds of cardinality constraints are often mentioned in the literature - user cardinality and role cardinality [Ahn00, Atl96, Fer01]. In this dissertation, we introduce status predicates to capture all the states of a GTRBAC systems and present a set of functions to capture complex cardinality control on these states.

Several papers in the literature deal with separation of duty constraints, with efforts focused on identifying various forms of SoDs as well as to categorize them [Ahn00, Ber99b, Bew89, Kun99, Nya99, San91, Sim97, Tid98]. Simon and Zurko [Sim97] discuss informally a wide variety of SoD constraints that are required in systems. Gligor *et. al*. [Gli98] provide a formalism for these SoDs. One limitation of this work, however, is that it does not consider the session-based dynamic SoDs needed for simulating lattice-based access control and Chinese Wall policy in RBAC [Bew89, San17, San92b]. Another limitation is that the SoDs defined do not capture the hierarchical semantics. Improvements along these lines can be seen in the SoDs listed by Ahn *et. al.* [Ahn00]. Unlike these approaches, we follow a predicate-based definition of general exclusion and inclusion of various kinds of assignments and activations to define the SoD properties in GTRBAC. This approach, while subsuming the SoDs defined in the above-mentioned literature, also identifies the overall capability of an RBAC model to capture the separation of duty constraints that may exist.

Dependency constraints form a less explored aspect in RBAC. While some form of dependency is implied by role triggers, also used in GTRBAC, the control flow dependency constraints, where strict dependencies are implied, have not been included within an RBAC framework. Such control flow dependencies are typically used in workflow types of systems to define inter-dependencies between workflow tasks [Att93, Tho97]. We believe that using such dependency constraints, GTRBAC can better handle access control requirements in time-sensitive, workflow types of applications by providing a much broader framework for mapping tasks into roles and using these constraints to capture the interdependencies between these tasks.

No earlier work has addressed the issue of time-based cardinality, SoD, and dependency constraints. Applying periodicity/duration constraints for these SoDs is more suitable for supporting the access control needs of dynamically evolving systems that are prevalent today. We further show that when intervals, durations or periodicity expressions are associated with these constraints, different interpretations are possible.

2.3 Time-based Access Control

The importance of time-based access control requirements has only recently been recognized. The need for time-based access control requirement can be attributed to the growing importance of workflow-based applications, particularly in e-commerce and web based application environments. Furthermore, the humongous volumes of data available over the Internet based applications may have temporal characteristics which attach varying security risks or importance to the available data. Such applications require support for time-based authorization policies.

Few access control models have been proposed to address such requirements. In particular, Bertino *et. al.* propose a time-based access control model in [Ber01a]. In [Ber01a], Bertino *et. al* propose a Temporal RBAC model that extends the NIST RBAC model with periodicity expressions used in [Ber98, Nie92]. Atluri *et. al* have recently proposed a Temporal Data Access Model (TDAM) for addressing access control based on the temporal characteristics of data being accessed, such as the valid time and transaction time.

In the following sections, we briefly overview these models and compare with the proposed GTRBAC model. As GTRBAC is a generalization of limited temporal constraints introduced in the TRBAC model, the GTRBAC model borrows the periodic time expression used in the TRBAC model. We briefly overview the periodic time expression before discussing the time-based access control models.

2.3.1 Periodic Expression

Periodic time is represented through a symbolic formalism and is expressed as a tuple $\langle$[`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 the lower and upper bounds imposed on instants in P [Ner98, Nie92]. The periodic time uses the notion of *calendar* defined as a countable set of contiguous intervals representing their indices. A sub-calendar relationship can be established among calendars. Given two calendars C_1 and C_2, C_1 is said to be a sub-calendar of C_2, written as $C_1 \sqsubseteq C_2$, if each interval of C_2 is covered by a finite number of intervals of C_1. A set of calendars containing the calendars *Hours*, *Days*, *Weeks*, *Months*, and *Years* is assumed where *Hours* is the calendar that has the finest granularity. Calendars can be combined to represent more general *periodic expressions* denoting periodic intervals such as the set of *Mondays* or the set of *the third hour of the first day of each month.* A periodic expression is defined as: P = $\sum_{i=1}^{n} O_i.C_i \triangleright x.C_d$, where C_d, C_1, …, C_n are calendars and $O_1 = all$, $O_i \in 2^{\mathbb{N}} \cup \{all\}$, $C_i \sqsubseteq C_{i-1}$ for i = 2,.., n, $C_d \sqsubseteq C_n$, and $x \in \mathbb{N}$. Symbol $\triangleright$ separates the first part of the periodic expression that distinguishes the set of starting points of the intervals, from the specification of the duration of each interval in terms of calendar C_d. For example, {*all.Years* + {3,

$7\}.Months \rhd 2.Months\}$ represents the set of intervals having a duration of 2 months with their starting times synchronized with the same instant as the third or seventh month of every year. In practice, O_i is omitted if its value is *all*. In case O_i is a singleton, it is represented by its unique element. Similarly, $x.C_d$ is omitted when x is equal to 1. A set of time instants corresponding to a periodic expression P is denoted by $Sol(I, P)$. Similarly, the set of intervals in (I, P) is denoted by $\prod(P)$. For simplicity, in this dissertation the bounds begin and end, constraining a periodic expression, will be denoted by a pair of *date expressions* of the form mm/dd/yyyy:hh. The end point end can also be ∞. For instance, [1/1/2001, 12/31/2001] denotes all the instants in 2001.

2.3.2 Temporal Access Control Models

Bertino *et. al.* propose a time-based access control model that supports temporal authorization and derivation rules in a non-RBAC environment [Ber01a]. The periodicity constraint expression introduced earlier is used to describe how subjects are allowed time constrained access to resources. Their model also introduces high level operators such as WHENEVER, ASLONGAS and UPON to show temporal relations between temporal authorizations. However, the proposed model is not role based and hence does not provide the benefits of RBAC.

Atluri *et. al.* have recently proposed a *Temporal Data Authorization Model* (TDAM) that can express access control policies based on the temporal characteristic of data, such as valid and transaction time [Atl01]. The GTRBAC model proposed in this dissertation can capture this aspect of authorization by using dynamic role-permission assignments through periodicity and duration constraint, as well as triggers, although we do that at the abstraction of a permission, which is defined as a permitted operation on an object. As TDAM focuses on the temporal characteristic of data and not on the overall aspect of an authorization system, it again lacks benefits of an RBAC model. We believe that TDAM's capability to capture fine-grained temporal characteristics of data in an authorization decision, it can be used to supplement, at a more concrete level, the dynamic aspects of role-permission assignments in GTRBAC.

The Temporal-RBAC (TRBAC) model proposed by Bertino *et. al.* is the first model that extends an RBAC model with temporal constraints [Ber01a]. The TRBAC model, however, supports temporal constraints on role enabling only. The main features of the TRBAC model include periodic enabling of roles and dependencies among roles expressed by means of triggers**.** Priorities are associated with role events for handling potential conflicts. Precedence rules are used to resolve conflicts among events. The TRBAC model also allows an administrator to issue run-time requests for enabling and disabling a role.

The TRBAC model, however, cannot handle several other important temporal constraints. First, the model does not include temporal constraints for the user-role and role-permission assignments. It assumes that only roles are enabled and disabled at different time intervals. In this dissertation, we show that in some applications, roles are static in that they are enabled at all times, while users and permissions assigned to them can be transient. Second, the TRBAC model only handles the temporal constraints on role enabling and does not include any constraints on the actual activations of roles by the users. Thus, the TRBAC model does not support well-defined, separate notions of role enabling and role activation. Therefore, the TRBAC model cannot handle many constraints that are related to the activations of a role such as the constraints on the maximum active duration allowed to a user, the maximum number of activations of a role by a single user within a particular interval of time, etc. In this dissertation, a role is said to be enabled, if it can be assumed by a user. On the other hand, a role is active if there is at least one user who has assumed that role. Third, as the TRBAC model does not consider duration constraints as well as constraints on the actual activations of roles, it does not support the notion of enabling and disabling of constraints. The activation constraints need to be clearly defined with respect to the enabled time of a role. We, therefore, introduce the notion of constraint enabling/disabling. Finally, the TRBAC model does not address the time-based semantics of role hierarchies and SoD constraints.

In this dissertation, we illustrate the importance of the constraints mentioned above and accordingly propose a Generalized TRBAC (GTRBAC) model that subsumes TRBAC and can handle all the issues mentioned above. To incorporate various constraints regarding role activations, we distinguish between the notions of role activation and role enabling.

3. THE GTRBAC MODEL

In this chapter, we propose the Generalized TRBAC (GTRBAC) model that allows expressing a wide range of temporal constraints. We first discuss various types of temporal constraints relevant to role-based systems. In particular, we show how temporal constraints can be meaningfully applied to various components of RBAC systems. To incorporate various constraints regarding role activations, we distinguish between the notions of role activation and of role enabling. The proposed GTRBAC model provides duration and periodicity constraints, as well as other forms of specialized activation constraints. We present the syntax and semantics of these constraint expressions. In the subsequent sections, we discuss conflicts that may arise in a GTRBAC system and show to handle them to provide an execution model. We also use a notion of safeness to argue that the safe constraint sets ensure that certain undesirable or ambiguous execution semantics do not occur.

3.1 Temporal Constraints in GTRBAC

A key aspect of the proposed GTRBAC model is that it distinguishes between the notions of role enabling and role activation states. Such distinction leads to the notion of states of a role, as depicted in Fig. 3.1. In the proposed model, a role can assume one of the three states: *disabled*, *enabled* and *active*. The *disabled* state indicates that the role cannot be used in any user session, i.e., a user cannot acquire the permissions associated with the role. A role in the *disabled* state can be enabled. The *enabled* state indicates that users who are entitled to use the role at the time of the request may activate the role. Subsequently, if a user activates the role, the state of the role becomes *active*. A role in the *active* state implies that there is at least one user who has activated the role. Once in *active* state, further activations of the same role do not change its state. When a role is in *active* state, upon deactivation, the role transitions to the *enabled* state if there is only one session in which it is *active*, otherwise it remains in the *active* state. A role in *enabled* or *active* state transitions to the *disabled* state if a disabling event occurs.

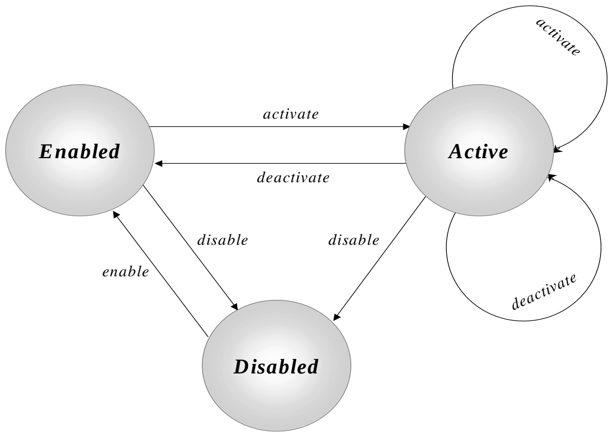


Fig. 3.1 States of a role

Table 3.1
Temporal constraint expressions

Categories	Constraints		Expression	Set/ Type
Periodicity Constraint	User-role assignment		$(I, P, pr{:}\texttt{assign}_{\texttt{U}}/\texttt{deassign}_{\texttt{U}}\ r\ \texttt{to}\ u)$	C_{Urp}
	Role enabling		$(I, P, pr{:}\texttt{enable/disable}\ r)$	C_{Rp}
	Role-permission assignment		$(I, P, pr{:}\texttt{assign}_{\texttt{P}}/\texttt{deassign}_{\texttt{P}}\ p\ \texttt{to}\ r)$	C_{PRp}
Duration Constraints	User-role assignment		$([(I, P)\| D], D_U, pr{:}\texttt{assign}_{\texttt{U}}/\texttt{deassign}_{\texttt{U}}\ r\ \texttt{to}\ u)$	C_{Urd}
	Role enabling		$([(I, P)\| D], D_{\mathrm{R}}, pr{:}\texttt{enable/disable}\ r)$	C_{Rd}
	Role-permission assignment		$([(I, P)\| D], D_{\mathrm{P}}, pr{:}\texttt{assign}_{\texttt{P}}/\texttt{deassign}_{\texttt{P}}\ p$ to $r)$	C_{PRd}
Duration Constraints (activation)	Total active duration	Per-role	$([(I, P)\| D], D_{active}, [D_{default}], pr{:}\texttt{active}_{\texttt{R\_total}}\ r)$	C^{a}_{dr}
		Per-user-role	$([(I, P)\| D], D_{uactive}, u, pr{:}\texttt{active}_{\texttt{UR\_total}}\ r)$	C^{a}_{dur}
	Max role dur. per activation	Per-role	$([(I, P)\| D], D_{max}, pr{:}\texttt{active}_{\texttt{R\_max}}\ r)$	C^{a}_{mr}
		Per-user-role	$([(I, P)\| D], D_{umax}, u, pr{:}\texttt{active}_{\texttt{UR\_max}}\ r)$	C^{a}_{mur}
Cardinality Constraint (activation)	Total no. of activations	Per-role	$([(I, P)\| D], N_{active}, [N_{default}], pr{:}\texttt{active}_{\texttt{R\_n}}\ r)$	C^{a}_{nr}
		Per-user-role	$([(I, P)\| D], N_{uactive}, u, pr{:}\texttt{active}_{\texttt{UR\_n}}\ r)$	C^{a}_{nur}
	Max. no. of con. activations	Per-role	$([(I, P)\| D], N_{max}, [N_{default}], pr{:}\texttt{active}_{\texttt{R\_con}}\ r)$	C^{a}_{nnr}
		Per-user-role	$([(I, P)\| D], N_{umax}, u, pr{:}\texttt{active}_{\texttt{UR\_con}}\ r)$	C^{a}_{nmur}
Constraint Enabling			$\texttt{enable/disable}\ c$ where $c \in \{(D, D_x, pr{:}E), (C), (D, C)\}$	C_c
Run-time request	*Users' activation request*		$(\mathrm{s}{:}\texttt{(de)activate}\ r\ \texttt{for}\ u\ \texttt{after}\ \Delta t))$	C_u
	Administrator's run-time request		$(\mathrm{pr}{:}\texttt{assign}_{\texttt{U}}/\texttt{de-assign}_{\texttt{U}}\ r\ \texttt{to}\ u\ \texttt{after}\ \Delta t)$	C_{admin}
			$(\mathrm{pr}{:}\texttt{enable/disable}\ r\ \texttt{after}\ \Delta t)$	C_{admin}
			$(\mathrm{pr}{:}\texttt{assign}_{\texttt{P}}/\texttt{de-assign}_{\texttt{P}}\ p\ \texttt{to}\ r\ \texttt{after}\ \Delta t)$	C_{admin}
			$(\mathrm{pr}{:}\texttt{enable/disable}\ c\ \texttt{after}\ \Delta t)$	C_{admin}
Trigger			$E_1, \ldots, E_n, C_1, \ldots, C_k \rightarrow pr{:}E\ \texttt{after}\ \Delta t$	C_{tr}

The proposed model allows the specification of the following types of constraints: (1) *temporal constraints on role enabling*, *user-role and role-permission assignments*, (2) *activation constraints*, (3) *run-time events*, (4) *constraint enabling expressions*, and (5) *triggers*. Table 3.1 summarizes the constraint types and expressions of the GTRBAC model, which are discussed in the following section.

3.1.1 Temporal Constraints on Role Enabling and Assignment

An important feature of the proposed GTRBAC model is that periodicity and duration constraints can be applied to various components of RBAC. Specifically, by constraining the times when roles are enabled or active, these constraints can be applied to roles themselves, as well as to user-role and role-permission assignments. Depending on the requirements, role enabling and assignments can be restricted to particular intervals or to a specified duration.

Periodicity constraints are used to specify the exact intervals during which a role can be enabled or disabled, and during which a user-role assignment or a role-permission assignment is valid. Duration constraints, on the other hand, are used to specify durations for which enabling or assignment of a role is valid. When an event occurs, the duration constraint associated with the event validates the event for the specified duration only. In case no duration constraint exists for the event, the event remains valid until it is disabled by some other means, e.g., by a trigger.

3.1.2 Temporal Constraints on Role Activation

Role activations are the result of granting users' requests to activate roles. Such requests are made at the discretion of a user at arbitrary times and hence periodicity constraints on role activations should not be imposed. However, duration constraints can be imposed on role activations. In the proposed model, duration constraints on role activations can be classified into two types: *total active duration constraint* and *maximum duration per activation constraint*. The *total active duration constraint* on a role restricts the span of the role's activation duration in a given period to a specified value. After the users have utilized the specified total active duration for a role, the role cannot be activated again, even though it may still be enabled. It can be noted that the *total active duration* allowed for a role may span a number of intervals in which the role is enabled. The *total active duration* may be specified on *per-role* and *per-user-role* basis. *Per-role* constraint restricts the total active duration for a role. Once the sum of all activation durations of the role reaches the maximum allowed value, no further activation of the role is allowed. *Per-user-role* constraint restricts the total active duration for a role by a particular user. Once a user utilizes the total active duration of the role specified for him, he cannot activate the role further, whereas other users may still activate the role.

The *maximum duration constraint per activation* restricts the maximum duration for each activation of a role. Once such a *duration* expires for a user, the role activation for that user becomes void. However, there may still be other activations of the same role in the system, including one by the same user in some other session. This constraint can also be specified on *per-role* or *per-user-role* basis. The *per-role* constraint restricts the maximum active duration for each activation of a role for any user, unless there is a *per-user-role* constraint specified for that user. The *per-user-role* constraint restricts the maximum active duration allowed for each activation of a role by a particular user.

In some applications, restrictions on the number of concurrent activations of a role may be required for controlling access to critical objects or resources. For example, we may want to ensure that a single user does not access all the resources while others are denied the access. Such a cardinality restriction on role activation can be categorized into two types: *total n activations constraint*, and *maximum n concurrent activations constraint*. In the first category, a role is limited to a total of *n* activations. This constraint may also be specified on a *per-role or per-user-role* basis. The *per-role* constraint allows at most *n* activations of a role in a given period of time, irrespective of whether these activations occur simultaneously in different sessions or at different times. Similarly, the *per-user-role* constraint restricts a total of *n* activations of a role by a specified user.

In the second category a role is restricted to *n* concurrent activations at any time. Constraint on a *per-role* basis may be specified to restrict the number of concurrent activations of a role to a maximum value. The activation of these roles may be associated with the same or different users. On the other hand, the *per-user-role* constraint restricts the total number of concurrent activations of a role by a particular user to a given value. Different users may have different permissible upper limits on the number of concurrent activations of the same role.

3.1.3 Run-time Requests, Triggers, and Constraint Enabling

As mentioned earlier, a user's request to activate a role is made at his discretion. In GTRBAC, a user's role activation request is modeled as a run-time event. Similarly, the administrators' run-time requests to initiate events that may override any existing valid events are also modeled. Such events can be used to override a pre-defined policy to make useful changes in the policy. For example, an administrator may initiate events to disable roles found to be in use by some malicious users. A relevant requirement in many application domains is the need of automatically executing certain actions as an occurrence of an event, such as the enabling or disabling of a role. In GTRBAC, we model such dependencies among events by using triggers. In addition, the duration constraints on role enabling and assignments and role activation constraints can be enabled for a pre-specified interval or duration. GTRBAC includes constraint enabling expressions to enable or disable such constraints.

3.2 Formal Syntax and Semantics of the GTRBAC Model

In this section, we discuss the formal syntax and semantics for the constraint expressions used in the GTRBAC model. Basic event expressions used by the GTRBAC constraint language are depicted in Table 3.2. Priorities are associated with each event in the proposed model. We define ($\mathtt{Prios}$, $\preccurlyeq$) as a totally ordered set of priorities and assume that $\mathtt{Prios}$ contains two distinct elements $\bot$ and $\top$ such that, for all $x \in \mathtt{Prios}$, $\bot \preccurlyeq x \preccurlyeq \top$. We use $x \prec y$, if $x \preccurlyeq y$ and $x \neq y$. Status predicates, listed in Table 3.3, are used to capture the state information associated with roles. In GTRBAC, event expressions, priorities and status predicates are used to express the constraints listed in Table 3.1. Next, we present the syntax and semantics of the constraint expressions listed in Table 3.1 and illustrate their use in expressing an access control policy in a medical application domain.

Table 3.2
Prioritized event expressions

Simple Event ($r \in$ `Roles`, $u \in$ `Users`, and p $\in$ `Permissions`)
`enable` r *or* `disable` r
`assign`$_U$ r `to` u *or* `de-assign`$_U$ r `to` u,
`assign`$_P$ p `to` r or `de-assign`$_P$ p `to` r,
`enable` c *or* `disable` c,
Prioritized Events
pr:E, *where pr* $\in$ `Prios` *and E is a* simple event expression

Periodicity Constraints (*I, P, pr*:*E*)

As shown in Table 3.1, the periodicity constraint expressions have the general form (*I, P*, *pr*:*E*). The pair (*I, P*) specifies the intervals during which an event *E* takes place. *E* can be a role enabling event: "`enable/disable` *r*", or either of the assignment events: "`assign`$_U$`/deassign`$_U$ *p* `to` *r*" or "`assign`$_P$`/deassign`$_P$ *u* `to` *r*".

Table 3.3
Status predicates

Status Predicate (*C*)	**Status Predicate with time** (C_t)	**Semantics [*for time*]**
`enabled`(*r*)	`enabled`(*r, t*)	*r is enabled* [*at time t*]
`u_assigned`(*u, r*)	`u_assigned`(*u, r, t*)	*u is assigned to r* [*at time t*]
`p_assigned`(*p, r*)	`p_assigned`(*p, r, t*)	*p is assigned to r* [*at time t*]
`active`(*r*)	`active`(*r, t*)	*r is active* [*at time t*]
`u_active`(*u, r*)	`u_active`(*u, r, t*)	*r is active in u's session* [*at time t*]
`s_active`(*u, r, s*)	`s_active`(*u, r, s, t*)	*r is active in u's session s* [*at time t*]
`acquires`(*u, p*)	`acquires`(*u, p, t*)	*u acquires p* [*at time t*]

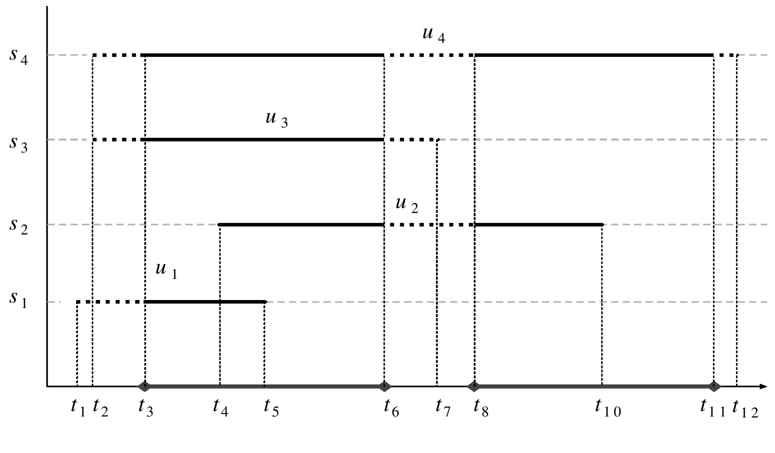


Fig. 3.2. Periodicity constraint on user-role assignment

Fig. 3.2 shows an example of periodicity constraints on user-role assignments. The two thick lines at the time axis represent the intervals (t_3, t_6) and (t_8, t_{11}) in which role *r* is enabled. The lines above the axis indicate intervals in which users are assigned to role *r*. The dotted portions of these lines indicate intervals in which user-role assignments are valid, although their assignment may not be in effect because the role is disabled in these intervals. For example, when user u_1 is assigned to role r in interval (t_1, t_5), he can activate role r only in the interval (t_3, t_5), as the role is disabled in the remaining part of interval (t_1, t_5). Similarly, user u_2 is assigned to r in interval (t_4, t_{10}) but can activate the role only in intervals (t_4, t_6) and (t_8, t_{10}). User u_3 is assigned to r in interval (t_2, t_7) but can assume r only in interval (t_3, t_6).

Duration Constraints ([(*I*, *P*,)|*D*], *D*$_x$, *pr*:*E*)

The general form of the duration constraint expressions for role enabling and assignment is ([(I, P,)|D], D_x, pr:E), where x is either `R`, `U`, or `P`, corresponding to events, respectively:

"`enable/disable` r",
"`assign`$_{\mathtt{U}}$`/deassign`$_{\mathtt{U}}$ r `to` u" and
"`assign`$_{\mathtt{P}}$`/deassign`$_{\mathtt{P}}$ p `to` r".

D and D_x refer to the durations such that $D \geq D_x$. The symbol "|" between (I, P) and D indicates that either (I, P) or D is specified. The square bracket in [(I, P,)|D] implies that this parameter is optional. Accordingly, we have three types of duration constraints: (I, P, D_x, pr:E), (D, D_x, pr:E) and (D_x, pr:E).

The expression (I, P, D_x, pr:E) indicates that event E is valid for the duration D_x within each valid periodic interval specified by (I, P). (D_x, pr:E) implies that the constraint is valid at all times. Therefore, if event E is caused at any time, it is restricted to duration D_x. The constraint c = (D, D_x, pr:E) implies that there is a valid duration D within which the duration restriction D_x applies to event E. In other words, the constraint c is enabled for duration D. The constraint enabling expressions as shown in Table 3.1 can be used to enable such constraints and the activation constraints discussed later. The constraint enabling/disabling event has the expression of the form "`enable/disable` c", where c is a constraint expression (D, D_x, pr:E). A constraint enabling event corresponds to either a run-time request or a triggered event. The duration constraint expression has the same general form as that of the activation constraint expression. Hence, the semantics of the duration constraints on role enabling and assignments is similar to that of the activation constraints. The example about activation constraints in Fig. 3.3 also illustrates how duration constraints mentioned here are imposed.

Activation Constraints ([(*I, P*,)|*D*], *C*):

Activation constraints have the general form ([(*I, P*)| *D*], *C*), where *C* represents the restriction applied to a role activation. For example, $C = (D_{\text{active}}, [D_{\text{default}}], \texttt{active}_{\texttt{R\_total}}\ r)$ corresponds to the *total active role duration per-role* constraint. [(*I, P*)| *D*] is an optional temporal parameter and has the same meaning as given by the duration constraints. Therefore, similar to the duration constraints, an activation constraint assumes one of the three forms: (*I, P, C*), (*D, C*) or (*C*). The first two expressions are semantically similar to those for duration constraints. Constraint (*C*) implies that the activation restriction specified by *C* applies to each enabling of the associated role. If *C* is a *per-role* constraint, it has an optional default parameter that can be used to specify the default value corresponding to the *per-user-role* restriction. For example, if $C = (D_{\text{active}}, [D_{\text{default}}], \texttt{active}_{\texttt{R\_total}}\ r)$ then D_{default} indicates that the *default per-user-role* active duration value is applied to all the users assigned to the role. In case D_{default} is not specified, it is assumed to be equal to the *per-role* value, D_{active}. Parameters of other activation constraints can be similarly interpreted.

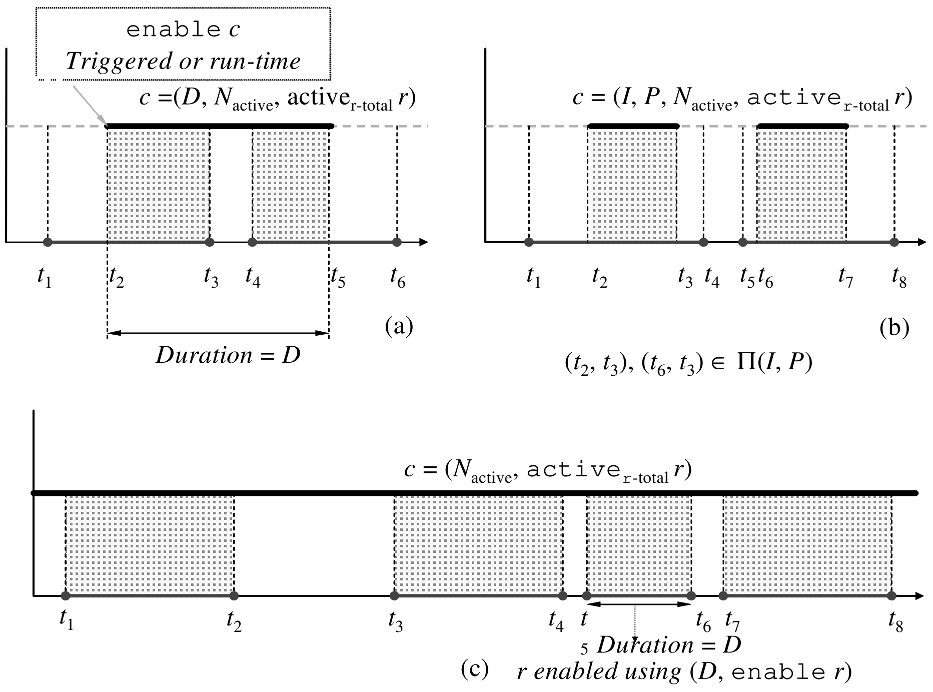

enable c
Triggered or run-time
c =(D, N_active, active_r-total r)
t1 t2 t3 t4 t5 t6
(a)
Duration = D
c = (I, P, N_active, active_r-total r)
t1 t2 t3 t4 t5 t6 t7 t8
(b)
(t2, t3), (t6, t3) ∈ Π(I, P)
c = (N_active, active_r-total r)
t1 t2 t3 t4 t5 t6 t7 t8
Duration = D
(c) r enabled using (D, enable r)

Fig. 3.3. Constraint enabled (a) for a specified duration (b) in specified intervals (c) at all times

Fig. 3.3 illustrates the three different forms of an activation cardinality constraint *C*. In Fig. 3.3(a), the constraint *c* is of form (*D, C*). In this case, the role is enabled in the intervals (t_1, t_3) and (t_4, t_6). A trigger or a run-time request can enable this constraint at time t_2 (i.e., event "`enable` *c*" occurs). Subsequently, *c* becomes valid for duration *D,* which in this case corresponds to interval (t_2, t_5). However, within interval (t_2, t_5), a subinterval (t_3, t_4) can exist in which role *r* is not enabled. The cardinality constraint *c* implies that the total number of activations of role *r* in the intervals (t_2, t_3) and (t_4, t_5) combined should not exceed N_{active}.

Fig. 3.3(b) illustrates an activation constraint of the form *c* = (*I, P, C*). Here, (t_2, t_3) and (t_6, t_7) are intervals in (*I, P*) and hence, during each of these intervals the total number of activations of role *r* is restricted to N_{active}. Fig. 3.3(c) shows a constraint of the form *c* = (*C*), where, for each enabling period of *r*, constraint (*C*) is valid. For example, role *r* is enabled by a periodicity constraint in the intervals (t_1, t_2), (t_3, t_4) and (t_7, t_8). During each of these intervals, at most N_{active} activations of role *r* are allowed. Furthermore, role *r* can also be enabled in interval (t_5, t_6) because of the duration constraint (*D*, `enable` *r*). The activation constraint *c* is then also applicable to this interval, for which only N_{active} activations of role *r* are allowed.

Run-time Requests and Triggers:

As shown in Table 3.1, a user's run-time request to activate or deactivate a role can be expressed as: (1) `s:activate` *r* `for` *u* `after` Δt, and (2) `s:deactivate` *r* `for` *u* `after` Δt. The priority associated with this request is assumed to be the same as that of event "`assign` *r* `to` *u*" that authorizes the activation of role *r* by user *u*. Similarly, *an administrator's run-time request* expression, written as *pr:E* `after` Δt is a prioritized event that occurs Δt time units after the request. In case the priority and the delay need to be omitted, we set $pr = \top$, where $\top$ represents the highest priority, and Δt = 0.

The trigger expression has the form E_1 ,…, E_n , C_1 ,…, C_k $\rightarrow$ *pr:E* `after` Δt, where E_i*'s* are simple event expressions or run time requests, C_i*'s* are status predicates, *pr:E* is a prioritized event expression with $pr \prec \top$, *E* is a simple expression such that $E \notin$ {*s*`:activate` *r* `for` *u*}, and Δt is a duration expression. It can be noted that because an activation request is made at a user's discretion, the event *E* should not be "*s*`:activate` *r* `for` *u*". However, event "*s*`:activate` *r* `for` *u*" can trigger other events and hence can be a part of the body of a trigger. Note that the event "*s*`:de-`

`activate` *r* `for` *u*" is allowed to appear in the head of a trigger as it can be used to enforce system controls. We illustrate the GTRBAC specification of an access control policy through the following example for a medical information system.

`Example 3.2.1`: Consider the GTRBAC access control policy of Table 3.4, from a medical information system. In row 1a, the enabling times of DayDoctor and NightDoctor roles are specified as a periodicity constraint. The (*I, P*) forms for *DayTime* (9am-9pm) and *NightTime* (9pm-9am) are as follows: *DayTime* = ([12/1/2003, ∞], *all.Days, +* 10.*Hours* ⊳ 12.*Hours*), and *NightTime* = ([12/1/2003, ∞], *all.Days, +* 12.*Hours* ⊳ 12.*Hours*).

Table 3.4.
Example GTRBAC access policy for a medical information system

1	A	(*DayTime*, `enable` DayDoctor), (*NightTime*, `enable` NightDoctor)
	B	((M, W, F), `assign`$_U$ *Adams* `to` DayDoctor), ((T, Th, S, Su), `assign`$_U$ *Bill* `to` DayDoctor),
	C	(*Everyday between* 10am - 3pm, `assign`$_U$ *Carol* `to` DayDoctor)
2	A	(`assign`$_U$ *Ami* `to` NurseInTraining); (`assign`$_U$ *Elizabeth* `to` DayNurse)
	B	*c*1 = (6 *hours*, 2 *hours*, `enable` NurseInTraining)
3	A	(`enable` DayNurse → `enable` *c*1)
	B	(`activate` DayNurse `for` *Elizabeth* → `enable` NurseInTraining `after` 10 *min*)
	C	(`enable` NightDoctor → `enable` NightNurse after 10 *min*); (`disable` NightDoctor → `disable` NightNurse after 10 *min*)
4	A	(10, `active`$_{R\_n}$ DayNurse);
	B	(5, `active`$_{R\_n}$ NightNurse);
	C	(2 *hours*, `active`$_{R\_total}$ NurseInTraining)

In 1b, *Adams* is assigned to role DayDoctor on *Mondays*, *Wednesdays* and *Fridays*, whereas *Bill* is assigned to it on *Tuesdays*, *Thursdays, Saturdays* and *Sundays*. The assignment in 1c indicates that *Carol* can assume the DayDoctor role everyday between 10am and 3pm. In 2a, users *Ami* and *Elizabeth* are assigned roles NurseInTraining and DayNurse respectively, without any periodicity or duration constraints. In other words, their assignments are valid at all the times. 2b specifies a duration constraint of 2 *hours* on the enabling time of the NurseInTraining role, but this constraint is valid for only 6 *hours* after the constraint $c1$ is enabled. Consequently, once the NurseInTraining role is enabled, *Ami* will be able to activate the NurseInTraining role at the most for two hours.

Trigger 3a indicates that constraint $c1$ is enabled once the DayNurse is enabled. As a result, the NurseInTraining role can be enabled within the 6 *hours*. Trigger 3b indicates that 10 *minutes* after *Elizabeth* activates the DayNurse role, the NurseInTraining role is enabled for a period of 2 *hours*. As a result, a nurse in training can then have access to the system only if *Elizabeth* is present in the system. In other words, once the roles are assumed, *Elizabeth* acts as a training supervisor for a nurse in training. It is possible that *Elizabeth* activates the DayNurse role a number of times within 6 hours after the DayNurse role is enabled. The activation constraint 4c limits the total activation time associated with the NurseInTraining role to 2 *hours*. The constraint set 4 shows additional activation constraints. For example, constraint 4a indicates that there can be at most 10 users activating DayDoctor role at a time, whereas 4b shows that there can be at most 5 users activating the NightDoctor role at a time.

3.3 GTRBAC Conflict Resolution and Execution Semantics

In this section, we address issues related to conflicts that may arise in the GTRBAC model and propose an approach for conflict resolution and generating an execution model. We define set Γ consisting of all the event expressions, constraints and triggers in a GTRBAC system as the Temporal Constraint and Activation Base (TCAB). The set Γ is essentially a set of constraints of the types listed in Table 3.1. Furthermore, we assume users' and administrators' run-time requests as a sequence $RQ = \langle RQ(0), RQ(1),\ldots, RQ(t), \ldots\rangle$. Note, $RQ(t) \in RQ$ is a set of run-time requests at time t and may be empty.

3.3.1 Conflicts in GTRBAC

Various types of conflicts may arise in a GTRBAC system. A clear semantics is needed to capture such conflicting scenarios. For example, a role enabling event caused by a periodicity constraint, and a role disabling event caused by the firing of a trigger, can correspond to the same role and may occur at the same time. Such a scenario gives rise to conflicts. Essentially, there are three categories of conflicts that may occur for a given Γ and a request sequence *RQ*, as depicted in tables 3.5-3.7. These include:

1. *Conflicts between events of the same category* (*type* 1 conflicts): Events in the same category are associated with the same pair of states of a role or assignment. For example, event "`enable` *r*" results in changing the *disabled* state of role *r* to an *enabled* state whereas event "`disable` *r*" corresponds to changing the *enabled* state of a role to the *disabled* state. Similarly, events "`assign` *r* `for` *u*" and "`de-assign` *r* `for` *u*" are of the same category. The entries in Table 3.5 refer to conflicts among the same category of events. A pair of events E_1 and E_2 in a row is said to conflict (*written as* E_2 = Conf (E_1)) if the corresponding *condition C* holds.

Table 3.5

Type 1 conflicts: conflicts between events of same category

Conflicting Events	E_1	E_2 = **Conf** (E_1)	***Condition*** (*C*)
Role Enabling conflicts	`enable` *r*	`Disable` *r'*	($r = r'$)
	`disable` *r*	`enable` *r'*	
Assignment conflicts	`assign`$_U$ *r* `to` *u*	`de-assign`$_U$ *r'* `to` *u'*	($r = r'$ and $u = u'$)
	`de-assign`$_U$ *r* `to` *u*	`assign`$_U$ *r'* `to` *u'*	
	`assign`$_P$ *p* `to` *r*	`de-assign`$_P$ *p'* `to` *r'*	($r = r'$ and $p = p'$)
	`de-assign`$_P$ *p* `to` *r*	`assign`$_P$ *p'* `to` *r'*	
Activation conflicts	*s*`:deactivate` *r* `for` *u*	*s'*`:activate` *r'* `for` *u'*	($s = s'$, $r = r'$ and $u = u'$)
	s`:activate` *r* `for` *u*	*s'*`:deactivate` *r'* `for` *u'*	($s = s'$, $r = r'$ and $u = u'$)
Constraint enabling conflicts	`enable` *c*	`disable` *c'*	($c = c'$)
	`disable` *c*	`enable` *c'*	

2. *Conflicts between events of different categories*: (*type* 2 conflicts): Conflicts may also arise between events of different categories. For instance, an activation request "`activate` *u* `for` *r*" and a role disabling event "`disable` *r*" are conflicting events if they attempt to occur simultaneously, as a *disabled* role cannot be *active*. Similarly, an activation event "`activate` *u* `for` *r*" and a user-role de-assignment event "`de-`

`assign` *r* `to` *u*" cannot occur at the same time as a user may activate a role only if he is assigned to the role. We also note that events "`enable` *r* " and "*s*`:deactivate` *r* `for` *u*" do not conflict even if both occur simultaneously.

Table 3.6

Type 2 conflicts: conflicts between events of different categories

Conflicting Events	E_1	E_2 = **Conf** (E_1)	***Condition*** (C)
Activation vs. role disabling	*s*`:activate` *r* `for` *u*	`disable` *r'*	($r = r'$)
Activation vs. deassignment	*s*`:activate` *r* `for` *u*	`De-assign` *r'* `to` *u'*	($r = r'$ & $u = u'$)

3. *Conflicts between constraints* (*type* 3 conflicts): Conflicts may also occur between two constraints defined for role enabling or role assignment (*type* 3a shown in Table 3.7). For example, a duration constraint on role enabling, (D_R, `enable` *r*) and a duration constraint on role disabling (D_R, `disable` *r*) may occur at the same time, if both "`enable` *r*" and "`disable` *r*" events are valid at the same time. It can be noted that such conflicts occur because of the underlying conflicting events.

Table 3.7

Type 3 conflicts: conflicts between constraints

Conflicting Constraints	C_1	C_2 = **Conf** (C_1)	***Condition*** (C)
Type 3a	(X_1, d_1, E_1)	(X_2, d_2, E_2)	E_2 = Conf (E_1) & occurrence of d_1 and d_2 overlap
Type 3b	(d_a, d_d, $\texttt{active}_{\texttt{R\_total}}$ r)	(d_{ua}, u, $\texttt{active}_{\texttt{UR\_total}}$ r')	($r = r'$) and ($d_a \neq d_{ua}$ or $d_d \neq d_{ua}$)
	(d_{max}, $\texttt{active}_{\texttt{R\_max}}$ r)	(d_{umax}, u, $\texttt{active}_{\texttt{UR\_max}}$ r')	($r = r'$) and ($d_{max} \neq d_{umax}$)
	(n_a, n_d, $\texttt{active}_{\texttt{R\_n}}$ r)	(n_u, u, $\texttt{active}_{\texttt{UR\_n}}$ r')	($r = r'$) and ($n_a \neq n_{ua}$ or $n_a \neq n_{ua}$)
	(n_{max}, $\texttt{active}_{\texttt{R\_max}}$ r)	(n_{umax}, u, $\texttt{active}_{\texttt{UR\_max}}$ r')	($r = r'$) and ($n_{max} \neq n_{umax}$)

A conflict can occur between the *per-user activation* constraint and the *per-role activation* constraint (*type* 3b) as shown in Table 3.7. For example, consider the *per-role* constraint (D_{active}, [$D_{default}$], $\texttt{active}_{\texttt{R\_total}}$ *r*) and the *per-user-role* constraint ($D_{uactive}$, *u*, $\texttt{active}_{\texttt{UR\_total}}$ *r*). The first constraint indicates that role *r* is allowed for an activation duration of $D_{uactive}$, whereas the second constraint specifies that user *u* is allowed to assume role *r* for the total activation duration of D_{ua}. If duration $D_{default}$ is specified, then all the users are restricted to a total activation time of $D_{default}$. There is an inherent ambiguity whether the user *u* should be allowed a total activation time of $D_{uactive}$ or $D_{default}$. Note, in the *per-user* constraint if $d_{default}$ is not specified then we assume $D_{default} = D_{active}$. In other words, any single user may activate role *r* for the entire activation duration of D_{active}. Therefore, the *per-user-role constraint* will again conflict with the per-role constraint.

The GTRBAC model uses the notion of blocked events to resolve conflicts of types 1 and 2, as defined below. When priorities cannot resolve conflicts, the model uses a *negative*-takes-precedence principle to resolve type 1 conflicts. According to this principle, disabling of a role takes precedence over enabling the role and the deactivation of a role takes precedence over the activation of the role. Similarly, for type 2 conflicts, we prefer the role disabling and user-role deassignment event over the activation event, as an enabled role and a valid assignment are prerequisites for a role activation. The following definition states these conflict resolution rules.

Definition 3.3.1 (**Conflict resolution for Type 1 and Type 2**) *Let S be a set of prioritized event expressions and constraints. Let pr:E be a prioritized event expression, where E is an event and pr* $\in$ `Prios`. *pr:E is said to be blocked by S, if the following conditions hold:*

1. *if there exists a q* $\in$ `Prios`, *such that q*:Conf(*E*) $\in$ *S and the following holds*
    a. *If pr:E and q*:Conf(*E*) *result in a type* 1 *conflict*, *then either*
        i. *E* corresponds to E_1 or in Table 3.5, *and pr* $\preccurlyeq$ *q, or*
        ii. *E* corresponds to E_2 in Table 3.5 and *pr* $\prec$ *q;*
    b. *If pr:E and q*:Conf(*E*) *result in a type* 2 *conflict*, *and E* = *s:*`activate` *r* `for` *u*
2. *if there exists a valid constraint* ([(*I*, *P*)| *D*], *X*) *that does not permit event pr:E to occur.*

The set of non blocked events in S is denoted by `Nonblocked`(*S*). *Furthermore, if both type 1 and type 2 conflicts occur, events blocked by type 1 conflicts are removed prior to removing events blocked by type 2 conflicts. In addition, if S has valid* *constraints of the form* ([(*I*, *P*)| *D*], *X*), *events blocked by these constraints are evaluated last*.

In definition 3.3.1, 1(a)(i) implies that event "*q*:disable *r*" blocks "*pr*:enable *r*" if $pr \preccurlyeq q$. If, however, $pr \prec q$ then according to condition 1(b)(ii), the event "*q*:enable *r*" would instead block the event "*pr*:`disable` *r*". Condition 1(a) applies to all the conflicts of *type* 1. Rule 1(b) applies to type 2 conflicts depicted in Table 3.6. According to this rule, events associated with role disabling or user-role de-assignment events override the role activation events, as role activation by a user depends on both the role enabling and user-role assignments. It is important that a role disabling or user-role de-assignment event is not blocked if either one aims to block an activation event. By resolving the type 1 conflicts first, we ensure that an activation event is blocked by a role disabling or user-role de-assignment that has not been blocked. Parts (b) and (c) of Example 3.3.1 presented below illustrate the necessity of handling type 1 conflicts prior to handling type 2 conflicts. The second part of the definition indicates that an event may also be blocked by the duration constraints on role enabling and assignments, and activation constraints on roles. When several activation requests for a role are present, some of these activation requests may need to be blocked to enforce an activation constraint. For example, assume that there is a cardinality constraint that says only five activations of role *r* are to be allowed at a time. If, at a particular time, multiple requests associated with role *r* are present, the cardinality constraint on the role will block two of these events. In such cases, a predefined selection criterion is needed to select the activation requests that are to be blocked. Such a selection criterion may depend, for example, on the priority of the activation requests, or the duration for which the activation has existed, or their combinations. Furthermore, note the general form of the activation request is "activate *r* for *u* after Δt", which indicates that a user may request role activation in advance. The selection criteria can use the value of Δt to determine activation requests that should be blocked. Furthermore, once the type 1 and type 2 conflicts have been resolved, events blocked by constraints following the resolution rule defined for the type 3b conflicts are selected. The following example further illustrates the notion of the blocked events.

`Example 3.3.1:` Assume a system with two priorities `H = High` and `VH = Very High` with `H < VH.` Now, consider the following three cases of increasing complexity.

(a) Let *S* = {`H:enable` r_0`,H:disable` r_0`,VH:enable` r_1`,` `H:disable` r_1}. Thus, by condition 1(a)(i) of Definition 3.3.1, `Nonblocked`(*S*) =

{H:disable r_0, VH:enable r_1}, since event "H:enable r_0" is blocked by event "H:disable r_0". Similarly, according to condition 1(a)(i), event "H:disable r_1" is blocked by "VH:enable r_1."

(b) Next, we consider a more complex case for S = {H:enable r_0,H:disable r_0, VH:enable r_1, H:disable r_1 VH:(s:activate r_1 for u)}. Assume we first resolve type 2 conflicts and then type 1 conflicts. In this case, event "VH:(s:activate r_1 for u)" is removed first as it is blocked by the event "H:disable r_1" as per condition 1(b)(i). We then encounter the case where Nonblocked(S) = {H:disable r_0, VH:enable r_1}. Note, event "H:disable r_1," that blocks event "VH:(s:activate r_1 for u)", which itself is a blocked event. Hence, blocking of event VH:(s:activate r_1 for u) by H:disable r_1 is not correct.

Alternatively, assume we first remove type 1 conflicts, which results in Nonblocked(S) = {H:disable r_0, VH:enable r_1, H:(s:activate r_1 for u)}. In the next step, we remove any type 2 conflicts. As event "H:(s:activate r_1 for u)" is not blocked by any event, the final result is Nonblocked(S) = {H:disable r_0, VH:enable r_1, H:(s:activate r_1 for u)}.

(c) S can be further extended as follows: S ={H:enable r_0, H:disable r_0, VH:enable r_1, H:disable r_1, VH:(s:activate r_1 for u_1), H:(s:activate r_1 for u_2), enable c}, where c= (1, H:active$_{\texttt{R\_Total}}$ r_1). After resolving type 1 and type 2 conflicts, we generate Nonblocked(S) = {H:disable r_0, VH:enable r_1, VH:(s:activate r_1 for u_1), H:enable c, H:(s:activate r_1 for u_2)}. Note that constraint c implies that only one activation of r_1 is permitted. Thus, one of the activation requests must be blocked. Because of the low priority event "H:(s:activate r_1 for u_2)" is blocked. Hence, the final set of non-blocked events generated is Nonblocked(S) = {H:disable r_0, VH:enable r_1, VH:(s:activate r_1 for u_1)}.

It can be noted that *type* 3a conflicts associated with constraints are mainly due to the underlying conflicting events associated with the constraint expressions. Hence, the resolution of *type* 1 conflicts in Definition 3.3.1 is applicable to *type* 3a conflicts as well. To resolve *type* 3b conflicts, we us a combination of the "*per-role*-takes-precedence over the *per-user-role* constraint" and the "*more-specific constraint* takes precedence" rules. These rules are formally defined below:

Definition 3.3.2 (**Conflict resolution for Type 3b conflicts**): Let (dn_a, [$dn_{default}$,] *pr*:$\texttt{active}_{\texttt{R\_x}}$ *r*) *be a per-role constraint and* (dn_{ua}, *u*, $\texttt{active}_{\texttt{UR\_x}}$ *r*) *be a per-user-role constraint defined for the same role r*, and $\texttt{R\_x} \in \{\texttt{R\_Total}, \texttt{R\_Max}, \texttt{R\_n}, \texttt{R\_con}\}$. *Then the following rules apply*:

1. *If there exist the same type of activation constraints for a role, the highest priority constraint blocks the others as per definition* 3.3.1.
2. *With respect to the per-role parameter dn_a and the per-user-role parameter dn_{ua}, the per-role constraint overrides the later one.*
3. *With respect to the default parameter $dn_{default}$ and the per-user-role parameter dn_{ua}, the* more-specific *per-user-role constraint overrides the later. In other words, when per-role activation constraint* (dn_a, $dn_{default}$, *pr*:$\texttt{active}_{\texttt{R\_x}}$ *r*) *and per-user-role activation constraint* (dn_{ua}, *u*, $\texttt{active}_{\texttt{UR\_x}}$ *r*) *are both specified, user u has constraint* dn_{ua}, *but not* $dn_{default}$.
4. *The following conditions hold: (1) $d_a \geq d_{ua}$, and (2) $d_a = n.d_{ua}$, for some $n > 0$. In other words, the value of per-user-role parameter should not exceed the value of per-role parameter.*

3.3.2 The GTRBAC Execution Model

Based on the rules for conflict resolution defined in the previous section, we now discuss the execution semantics of the GTRBAC model. In this section, we define system states and traces, and construct an execution model for GTRBAC. We also provide a definition to capture events that are caused at each instant of time, and present a state generation algorithm for constructing new states from the existing states based on the current set of the valid constraints.

The dynamics of occurrences of events and various states of role enablings and activations in GTRBAC are represented as a sequence of snapshots. Each snapshot provides the current set of prioritized events and the status of role, user-role and role-permission assignments as well as that of the activation constraints. To efficiently represent status information in the form of snapshots, we first define the following two structures, called *u-snapshot* and *r-snapshot*:

Definition 3.3.3 (**u-snapshot/ r-snapshot**): *We define:*

1. *a u-snapshot for user u with respect to a role r as a tuple* (*u*, *r*, d_{ua}, n_{ua}, d_{m}, n_{m}, S_{u}, D_{u}), *where* $r \in \texttt{Roles}$, $u \in \texttt{Users}$ *such that u is assigned to r, and the constraint parameters are as defined in* Table 3.8.

2. *an r-snapshot for a role r as a tuple* (r, d_{ra}, n_{ra}, d_{rm}, n_{rm}, *status*, P_r, U_r) *where*, $r \in$ `Roles` *and the other constraint variables are as defined in* Table 3.8.

In particular, these structures are used to model events, various role and assignment status, and the status of constraints obtained by two distinct sequences *EV*, and *ST*, respectively. The model in the form of a system trace is defined below.

Table 3.8.
Constraint parameters of *u-snapshot* and *r-snapshot*

u-snapshot parameter		*r-snapshot parameter*	
d_{ua}	*remaining total duration for which u can activate r*	d_{ra}	*remaining total active duration for r,*
n_{ua}	*remaining number of times that u can activate r,*	n_{ra}	*remaining total number of activations of r,*
d_m	*maximum duration for which u can activate r at one time*	d_{rm}	*remaining maximum active duration for r,*
n_m	*maximum number of concurrent activations of r that u can have*	n_{rm}	*remaining maximum number of activations of r,*
S_u	$S_u = (s_1, s_2, \ldots, s_k)$ *is the list of sessions in which u is currently using r and* $D_u = (d_1, d_2, \ldots, d_k)$ *is the list of durations of activations of r by u in each of these sessions.*	*status*	*current status of r*
		P_r	*is the set of permissions that are assigned to r.*
		U_r	*is the set of u-snapshots such that, for all ut* $\in U_r$, $ut.r = r$; *where ut.r refer to the element r of the u-snapshot ut.*

Definition 3.3.4 (**System Trace**) *A* system trace *- or simply a* trace *– consists of infinite sequences of EV and ST, such that for all integers* $t \geq 0$:

- *the* t^{th} *element of EV, denoted as EV(t), is a set of prioritized event expressions. Intuitively, this is the set of events which occur at time t*;
- *the* t^{th} *element of ST, denoted as ST(t), is a set of r-snapshots corresponding to existing roles at time t. Algorithm* `ComputeST` *in* Fig. 3.4 *is used to compute ST(t) for each t.*

A trace *is called* canonical *if* $ST(0)$ *= set of r-snapshots of the form* (r, ∞, ∞,, ∞,, ∞,, `disabled`, ∅, ∅) *for all roles r in the system, i.e., all r-snapshots are initialized to* (r, ∞,, ∞,, ∞,, ∞,, `disabled`, ∅, ∅). We assume that a system starts from an initial state where all the roles are disabled and there are no user-role assignments, role-permission assignments or valid activation constraints. Such a state exists at time $t = 0$. As the time progresses, the events listed in Table 3.2 take place thus changing various role and assignment states. The notion of a GTRBAC trace with such an initial state is represented by a canonical trace.

The above definition of a trace enforces the intended semantics of events. The set `Nonblocked`(*EV*(*t*)) contains the maximal priority events that occur at time *t*. We note that Γ and *RQ* determine a unique state. It can also be noted that the state information contained in *ST*(*t*) concerning the active state of roles depends on activation constraints enabled at time *t*. A duration constraint or role activation constraint (*c*) is valid if event "`enable c`" is in `Nonblocked`(*EV*(*t*)). Therefore, given a previous state, event set and the valid activation constraint set, the following proposition holds.

Proposition 3.1 [7]. *Given a sequence EV, and an initial status* S_0*, a unique trace* (*EV, ST*) is *generated with* $ST(0)=S_0$.

The proposition implies that a procedure to generate a unique trace can be developed. Accordingly, we describe an algorithm `ComputeST`, shown in Fig. 3.4 that, based on a given set of events and valid constraints, computes the next state from an existing state. Based on the unblocked events and the current set of valid constraints, the algorithm updates the state information contained in *r-snapshots* and *u-snapshots*. All the events in `Nonblocked`(*EV*(*t*)) happen at time *t*. The state information *ST*(*t*) contains the effect of the events in `Nonblocked`(*EV*(*t*)) on state *ST*(*t*-1).

In step 1, all non-blocked assignment/de-assignment and deactivation events are processed. In step 2, the role disabling events are processed. Note, when a role is disabled, the role-specific and the user-specific parameters are reset to ∞, which indicates that if there are no per-role or per-user-role constraints then the activation duration and number of concurrent activations are unlimited.

Note, the conflict resolution rules for type 2 conflicts indicate that the role disabling and the user-role de-assignment events affect the active sessions related to corresponding roles and users. Hence, it is important to first process these events and then update the information related to active roles that remain active for the next unit duration. In step 3, the values of per-role parameters in *r*-snapshots are reverted to the initial value ∞ corresponding to those activation constraints that become invalid. In step 4, per-role constraint variables in *r*-snapshots of the newly enabled roles are initialized. In step 5, new activations of roles are processed. In this process, first, the cardinality variables *per-role* and *per-user-role* are decremented to find the remaining number of activations allowed after this activation request has been granted. Next, users' constraint variables are initialized and session information is entered to the session list.

```
Algorithm ComputeST
  Input   : $t$, $EV$, $ST$, $CT$;
  Output  : $ST(t)$;
/ * Initially $ST(0) = (r, \infty, \infty, \infty, \infty,$ disabled, $\emptyset, \emptyset)$. For each
   pair $(r, u)$ we use the associated snapshots $rt$ and $ut \in U_r$.
   Assume that $CT(t) = \{c \mid$ enable c $\in$ Nonblocked($EV(t)$)$\}$*/
Step 1: Handle assignments */
FOR each $E \in$ Nonblocked($EV(t)$) DO
  Case (E) of   de-assign $r$ to $u$   : $U_r = U_r - \{ut\}$;
        de-assign $p$ to $r$   : $P_r = P_r - \{p\}$;
        assign $p$ to $r$      : $P_r = P_r \cup \{p\}$;
        assign $r$ to $u$      : $U_r = U_r \cup \{(u, \infty, \infty, \infty, \infty, \emptyset, \emptyset)\}$;
        a: deactivate $r$ for $u$ : remove($s$, $S_u$, $D_u$)
/* Step 2: Handle role disabling event */
FOR each (disable $r$) $\in$ Nonblocked($EV(t)$) DO
    $rt.status$ = disabled;
    IF ($C_x \in CT(t)$) THEN
        Set per-role parameters of $rt$ to $\infty$
        FOR each $ut \in U_r$ DO
            Set $(S_u, D_u)$ to $(\emptyset, \emptyset)$;
            IF ($C_x \in CT(t)$) OR ($C_{x-1} \in CT(t)$) THEN
                Set per-user-role parameters of $rt$ to $\infty$
/* Step 3: Handle valid constraints */
FOR each ((X, C) $\in CT(t-1)$ and (X, C) $\notin CT(t)$)
where X $\in \{(I, P), D\}$ & C is a per-role activation constraint DO
    IF (C = $C_x$) THEN
      Set per-role parameters of the corresponding $rt$ to $\infty$
/* Step 4: Handle role enabling events */
FOR each (enable $r$) $\in$ Nonblocked($EV(t)$) DO
    IF ($rt.status \neq$ enabled) /* role is being enabled */
        $rt.status$ = enabled;
        FOR each ([(I, P)|D], C) $\in CT(t)$) Set the per-role parameter of $rt$ to per-role value specified in $C$
/* Step 5: Handle valid activation requests*/
FOR each (s:activate $r$ for $u$) $\in$ Nonblocked($EV(t)$) DO /* assume $rt$ for $r$ and $ut$ for $u$ in $rt$ * /
  $rt.n_{ra} = rt.n_{ra} - 1$; $ut.n_{ua} = ut.n_{ua} - 1$;        /* decrement the values */
  FOR each ([(I, P)|D], C) $\in CT(t)$) such that $C$ is a constraint on $r$ DO
    IF ($C$ is per-user-role constraint) THEN
    Set the per-user-role parameter of the corresponding $ut$ to that in $C$.
    ELSE /* $C$ is a per-role constraint */
        IF (per-user-role default value is specified in $C$)
            Set the per-user-role parameter of the $ut$ to default value;
        ELSE /* per-user-role default value is not specified in $C$*/
            Set the per-user-role parameter of $ut$ to the per-role value in C;
    $d = \min(d_{ua}, d_m)$;  /* update the remaining role value */
    add($s$, $d$, $S_u$, $D_u$);
/* Step 6: Process constraint variables for the currently active roles and user-role activation*/
FOR each r-snapshot DO
    IF status = enabled THEN
        decrement (durations($r$));    $d_{ra} = d_{ra} - |$sessions($r$)$|$;
    ELSE
        $d_{ra} = d_{ra} - 1$;
        FOR each user assigned to $r$ DO $d_{ua} = d_{ua} - 1$;
```

Functions used are defined as follows:

remove(s, S, D), where, s is a session id, $S = \{s_1, s_2, \dots, s_k\}$ and $D = \{d_1, d_2, \dots, d_k\}$ is a procedure that computes (S, D) such that $S = S - \{s\}$ and $D = D - \{d\}$, where d corresponds to s.

add(s, d, S, D), where, s is a session id, d is the duration of activation related to s, $S = \{s_1, s_2, \dots, s_k\}$ and $D = \{d_1, d_2, \dots, d_k\}$; after processing, we get $S = S \cup \{s\}$ and $D = D \cup \{d\}$.

decrement(D), where $D = \{d_1, d_2, \dots, d_k\}$ is a set of integers; after processing we get $D = \{d_1\text{-}1, d_2\text{-}1, \dots, d_k\text{-}1\}$.

sessions (r) returns a set of sessions $\{s_1, s_2, \dots, s_k\}$in which role r is currently activated.

durations(r) returns a set of active durations $\{d_1, d_2, \dots, d_k\}$that corresponds to the sessions in sessions(r)

Fig. 3.4. Algorithm ComputeST

In step 6, the remaining active duration of each role is decremented. The total role duration is also adjusted accordingly. For disabled roles, the duration constraint variables, for both roles and users assigned to them, are decremented. Decrementing duration constraint variables takes care of any activation constraint that is valid at the time the associated role is disabled. The following theorem shows that the algorithm terminates correctly. Also, the theorem provides the complexity of the algorithm.

Theorem 3.1 (Correctness and complexity of `ComputeST`): *Given EV(t), CT(t), ST(t-1) and Γ, the algorithm* `ComputeST`:

1. *produces ST*(t) *such that the updated status of r-snapshots and u-snapshots in ST*(t) *satisfies all the constraints in Γ and the valid activation constraints for the interval* (t, t+1).
2. *terminates, and has complexity* $O(n_R\ (n_U + n_P + n_{Sm}))$, *where* n_R, n_P, n_U *and* n_{Sm} *represent the number of roles, permissions, users, and the maximum allowable number of sessions, respectively, in a system.*

Proof of the theorem is presented in the appendix A.

Given a *Γ* and a request stream *RQ*, we need to identify events in *EV*. Intuitively, each event should be caused by some element of *Γ* or *RQ*. When a trigger causes a prioritized event, the event expressions in the body of the trigger should not be blocked. Events in *EV* are formally defined as follows.

Definition 3.3.5 (**Caused Events**) *Given a trace, a Γ and a request sequence RQ, the set of caused prioritized events at time t, is the least set* `Caused`(*t, EV, ST, Γ, RQ*) (in short, written as `CSet`(t) below) *that satisfies the following conditions*:

C1. If $(I, P, pr{:}E)$ *and* $t \in Sol(I, P)$ *then* $pr{:}E$ = `CSet`(t) (*for periodicity constraint*)

C2. If ($pr{:}E$ `after` Δt) $\in RQ(t-\Delta t)$ $\Delta t \le t$) then (*for run-time request*)
$pr{:}E$ = `CSet`(t);

C3. If $[E_1, \ldots, E_n, C_1, \ldots, C_k \rightarrow p{:}E$ `after` $\Delta t\ ] \in \Gamma$ *and the following conditions hold then* $pr{:}E \in$ `CSet`(t); (*for triggers*)

   a. $0 \le \Delta t \le t$ &

   b. $\forall C_i$, *such that* $(1 \le i \le k)$, C_i *holds* (C_i *is* C *or* C_t *as shown in* Table 4.1 &

   c. $\forall E_i$, such that $(1 \le i \le n)$, $pr{:}E_i \in EV(t - \Delta t)$ *not blocked by* $EV(t - \Delta t)$

C4. a. If $c = (I, P, X) \in \Gamma$ and $t \in Sol(I, P)$ & (*for duration/activation constraints*)

      i. $0 \le \Delta t = (t - t_1) \le D_x$ &

ii. $[B \rightarrow pr{:}E$ `after` $\Delta t] \in \Gamma$ *or a run-time request* $pr{:}E \in RQ(t - t_1)$, *as a result of which* $pr{:}E \in$ `CSet`$(t - t_1)$ *not blocked* $(EV(t - t_1)))))$

then $pr{:}$`enable` $c \in$ `CSet`(t);

b. *If* $c = (D, X) \in \Gamma$ where $x \in \{$`U, R, P`$\}$, *and if there exists a pair* t_1, t_2 *such that*

i. $t_1 \leq t_2$ & $\Delta t_1 = (t - t_1) \leq D$ &

ii. $(\exists\ [B \rightarrow pr{:}$`enable` c `after` $\Delta t_1] \in \Gamma$ OR $pr{:}$`enable` $c \in RQ(t - t_1)$ *as a result of which* `enable` $c \in$ `CSet`$(t - t_1)$ *and is not blocked by* $EV(t - t_1))$

then $pr{:}$`enable` $c \in$ `CSet`(t);

Furthermore, in addition to (a) and (b), If $X = (D_x, pr{:}E) \in \Gamma$ is a duration constraint *such that* $x \in \{$`U, R, P`$\}$, *and the following condition holds*

i. $(\exists\ [B \rightarrow pr{:}E$ `after` $\Delta t_2] \in \Gamma$ OR $pr{:}E \in RQ(t - t_2)$, *as a result of which* $pr{:}E \in EV(t - t_2)$ *and is not blocked by* $EV(t - t_2)$

then $pr{:}$`enable` $c \in$ `CSet`(t) and $q{:}$`enable` $c \in$ `CSet`(t) *where q is the priority specified for c*;

Condition C1 implies that all events scheduled via a periodic event are added into the set `Caused`(t, EV, ST, Γ, RQ). Condition C2 shows that all the explicit run-time requests are added into the set `Caused`(t, EV, ST, Γ, RQ). Similarly, condition C3 implies that all the events scheduled through a trigger are added to `Caused`(t, EV, ST, Γ, RQ), provided that the conditions C_is specified in the body of the trigger are satisfied and each of the events E_i's occurs at time $t\text{-}\Delta t$. Furthermore, it is necessary that events E_i's are not blocked by any other concurrent event, as indicated by condition C3(c).

Condition C4 implies that all the events not blocked by valid duration or activation constraints are added to Caused(t, EV, ST, Γ, RQ). C4(a) defines the condition that must be satisfied by caused events associated with either a duration or activation constraint. Note that events restricted by a duration or activation constraint are caused by either the run-time requests or by the triggers and are not activated by any periodicity constraints. Furthermore, such events must not be blocked by any concurrent event. These conditions are ensured by condition C4(a)(ii).

Condition C4(a)(i) ensures that an event is still valid only if the duration D_x associated with the event has not expired. Similarly, C4(b) implies that all events that are associated with the duration or activation constraints of the form $c = (D, X)$ are considered. Note, as the start time of D is not known, semantically we require that c itself be enabled for a duration *D*. In other words, "enable *c*" is a caused event for *D* duration. Furthermore, "enable *c*" should not be blocked by any concurrent event at that time. The condition C4(b)(ii) ensures that these conditions hold. Condition C4(b)(iii) defines those events which are restricted by the constraint *c*.

It can be noted that the TCABs and request streams determine changes in system state at each time instant. Next, we define the system behavior induced by TCABs and request streams and address the safeness issue. Intuitively, safeness implies that for each event in *EV*(*t*), there is a definite and known cause.

Definition 3.3.6 (**Execution Model**) *A trace (EV, ST,) is an execution model of a TCAB Γ and a request stream RQ, if for all* $t \geq 0$, *EV* (*t*) = Caused(*t, EV, ST, Γ, RQ*) .

It is possible that some specifications may yield no execution model, whereas some ambiguous specifications may admit two or more such models [Ber01a]. For instance, if an event in *EV*(*t*), say enable *r*, triggers another event which in turn causes event disable *r* to occur, the later one is added in *EV*(*t*). By the conflict resolution rule, event disable *r* blocks enable *r*. Such a situation is undesirable as the event enable *r* that is the cause of event disable *r* is itself being blocked by the event disable *r*. However, if such cases are excluded, the GTRBAC specifications yield exactly one model for all possible run-time requests. There are simple syntactic conditions that prevent undesirable behavior as a result of conflicting events. Such syntactic conditions - called *safeness* – are introduced next.

3.3.3 Safe Temporal Constraint and Activation based (TCAB)

We use a safeness condition that can be verified in polynomial time and that guarantees that a given TCAB has one and exactly one execution model. The notion of a dependency graph is essential to analyze the safeness of the execution model. Each TCAB *Γ* can be represented as a directed labeled dependency graph $DG_R = \langle N, ED \rangle$ where *N,* a set of nodes, represents the set of all prioritized event expressions *pr:E* that occur in the head of a trigger [*B*→*pr:E*]∈ *Γ; and ED* (*the set of edges*) *consists of the following triples*, *for all triggers* [*B* → *pr:E*] ∈ *Γ*, *for all events E' in the body B, and for all nodes q:E'* ∈ *N*,

1. ⟨*q:E'*, +, *pr:E*⟩ and
2. ⟨*r*:Conf(*E'*), - , *pr:E*⟩, *for all* [*r*:Conf(*E'*)] ∈ *N* such that $q \preccurlyeq r$.

Each triple (N_1, l, N_2) represents an edge from node N_1 to N_2, labeled by l. *Given the initial status of the roles and assignments*, *safeness of Γ implies that the system's behavior is unambiguously determined by Γ*, and *RQ*. Accordingly, *Γ* is safe if its dependency graph DG_R contains no cycles in which some edge is labeled '-' [Ber01a]. Based on this notion of safeness of *Γ*, we extend the formal results of TRBAC to GTRBAC, stated as follows:

Theorem 3.2: *If a Γ is safe, then for a given RQ and ST*(*0*), *there exists exactly one execution model* <*EV*, *ST*>.

As the GTRBAC model essentially extends the TRBAC model with a large set of events, and the *safeness* of these models is determined with respect to the event dependencies through the triggers in *Γ*, the formal proof of this theorem follows directly from that of the theorem applied to the TRBAC model, as the rule based semantics of the TRBAC model can be easily extended to capture that of the GTRBAC model by simply incorporating the new events [Ber01a].

It can be noted that safeness is a sufficient condition for predictable system behavior. Although it is difficult to find the necessary conditions, even if found, they offer little practical help, because such syntactic properties fail to recognize that the ill-formed portions of a program may be harmless because they can never be activated. Furthermore, checking the existence and uniqueness of a model are, in general, NP-hard problems [Ber01a]. Accordingly, if *Γ* is safe then for a given *RQ*, there exists exactly one execution model [Ber01a].

Next, we present algorithm `SafetyCheck` to determine the safety of a given *Γ*, as shown in Fig. 3.5. The first part of the algorithm builds the dependency graph associated with *Γ*, and the second part checks for cycles with a negative edge. The correctness of the algorithm can be proven from the results reported in [Ber01a]. If *Γ* is found to be unsafe then we need to remove a trigger to ensure that a cycle with a negative edge does not exist in the dependency graph of *Γ*.

Algorithm `SafetyCheck` illustrated in Fig. 3.5 is used for the safeness verification of a TCAB. The first part of the algorithm builds the dependency graph associated with *Γ*, and the second part checks for cycles with a negative edge. The correctness of the algorithm can be simply proven from the results reported in [Cor90]. We note the following with respect to the labeled dependency graph:

- Dependency graph construction takes polynomial time. Such complexity can be reduced to O(|Γ|.|N|) by representing the graph as an ordered vector, which can be sorted in time O(|N|.log|N|).
- The strongly connected components of the graph can be determined in O(|N| + |ED|) time (cf. [Co90]). As the total number of edges is bounded by |E|, the second phase of the algorithm has a cost of O(|N| + |ED|).
- As each node must occur in some trigger's head, |N| = |Γ| and |ED| is in O(|Γ|^2).

From this, we see that the algorithm's complexity is in O(|Γ|^2). We note that the number of iterations of the innermost loop of the graph construction phase is bounded by a constant (i.e., |`Prios`|) for a fixed set of priorities. Hence, for a given set of priorities, the cost of the safeness verification is O(|Γ|. log|Γ|).

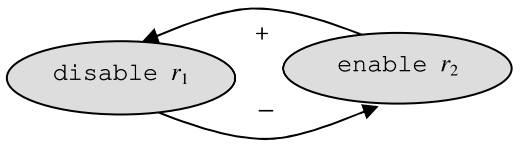
```
Algorithm SafetyCheck
Input:: a TCAB $\Gamma$
Output: true if $\Gamma$ is safe, false otherwise
/* construction of the dependency graph */
$N$ := 0;  $ED$:= 0;
FOR all $[B \to pr{:}E] \in \Gamma$ DO
  IF ($E$ = activate $r$ for $u$) THEN return false;
  $N := N \cup \{pr{:}E\}$;
FOR all $[B \to pr{:}E] \in \Gamma$ DO
  FOR all $E' \in B$ such that $\exists q$; $q{:}E' \in N$ DO
      $ED := ED \cup \{\langle q{:}E', +, pr{:}E\rangle\}$;
  FOR all $r{:}\mathrm{conf}(E') \in N$ such that $q \preccurlyeq r$ DO
      $ED := ED \cup \{\langle r{:}\mathrm{conf}(E'), -, pr{:}E\rangle\}$
/* cycle generation and checking */
SCC :=strongly connected components of $\langle N, ED\rangle$
FOR all $\langle N', ED'\rangle \in$ SCC DO
   FOR all $\langle X, l, Y\rangle \in ED'$ DO
        IF $l$ = '-' THEN return false;
return true;
```

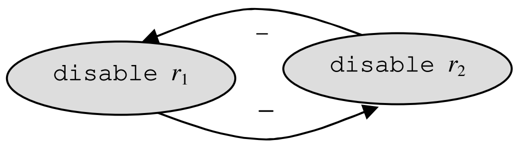
Fig. 3.5. Algorithm `SafetyCheck`

The examples presented below illustrate the working of algorithm `SafetyCheck.`

`Example 3.3.2`: First, let Γ = {`enable` r_1 → `enable` r_2; `enable` r_2 → `disable` r_1} and $RQ(t)$ = {`enable` r_1}. Hence, initially, $EV(t)$ = {`enable` r_1}. Because of event "`enable` r_1" the first trigger fires resulting in $EV(t)$ = {`enable` r_1, `enable` r_2}. Next, because of event "`enable` r_2" the second trigger fires resulting in

$EV(t)$ = {`enable` r_1, `enable` r_2, `disable` r_1}. Here, event "`disable` r_1" blocks event "`enable` r_1" (assuming the same priority). As event "`disable` r_1" was caused by event "`enable` r_1" such blocking is undesirable or ambiguous. Algorithm `SafetyCheck` detects such unsafe cases by detecting a cycle shown in Fig. 3.6(a).





(a) (b)

Fig. 3.6. Example dependency graphs

Next, consider Γ= {t_1: `enable` r_1 → `disable` r_2; t_2:`enable` r_2 → `disable` r_1} and initial $EV(t)$ = {`enable` r_1, `enable` r_2}. Now, for the firing of the triggers there are two alternatives:

1. First, the trigger t_1 fires. In this case, we get, $EV(t)$ = {`enable` r_1, `enable` r_2, `disable` r_2}. Assuming equal priorities, once the trigger t_1 fires, we get `Nonblocked`($EV(t)$) = {`enable` r_1, `disable` r_2}. As a result, trigger t_2 does not fire.
2. First, the trigger t_2 fires. In this case, we get, $EV(t)$ = {`enable` r_1, `enable` r_2, `disable` r_1}. Assuming equal priorities, after the trigger t_2 fires, we get `Nonblocked`($EV(t)$) = {`enable` r_2, `disable` r_1}. As a result, trigger t_1 cannot fire.

Such ambiguous system behavior is captured by the algorithm by indicating the cycle shown in Fig. 3.6 (b).

3.4 Authentication and Clock Synchronization Issues

An important issue related to access control is the authentication of users. In practice, authentication is a prerequisite for access control [Jos01b, San96a]. A comprehensive access control solution requires an authentication mechanism which ensures that the users are who they claim to be. In this chapter, although we have not explicitly addressed this issue, whenever we address an activation of a role by a user, we have implicitly assumed that the user has been properly authenticated. This assumption is made throughout this dissertation. In practice, several mechanisms can be used for authenticating users [And01, Sch96].

Another assumption for the enforcement of the temporal constraints that has been made in this chapter is also the granularity and accuracy of the underlying clock. For the purpose of modeling, we have assumed that the temporal constraints are specified at the granularity of the smallest calendar – *hours*. Ensuring accurate time to support the GTRBAC system can be a challenging issue. By manipulating the system clock used by the GTRBAC framework, attacks on the system security may be possible. Existing techniques to ensure clock accuracy can be used to augment the existing implementation of the GTRBAC framework to provide secure time for practical applications. For example, in a networked system, a clock synchronization mechanism such as Network Time Protocol (NTP) can be used. The NTP provides a moderate amount of protection with clock voting and authentication of time servers and is dependable enough for many applications [And01]. In case the GTRBAC system is used in a database application, the database clock can be used to provide the basic timing support for the GTRBAC system. When the proposed model is employed in an open internet environment, the issue of how the users' time and the system time are synchronized can provide a daunting challenge. In this dissertation, we assume that such synchronization problem in such distributed environment has been properly addressed [Lam77].

3.4 Conclusions

In this chapter, we have presented a generalized temporal role based access control model that can handle a comprehensive set of temporal constraints. The model allows temporal constraints on role enablings and role activations, as well as triggers to specify dependencies among GTRBAC events. Various temporal restrictions can be specified on user-role and role-permission assignments. We have also presented various time-based semantics of hierarchies and SoD constraints. We have described a notion of safeness that has been used to generate a safe execution model for a GTRBAC system.

4. ROLE HIERARCHIES IN GTRBAC

Many researchers have highlighted the importance and use of role hierarchies in RBAC models. A properly designed role hierarchy allows the efficient specification and management of access control policies of a system. When two roles are hierarchically related, one is called the senior and the other the junior. The senior role inherits all the permissions assigned to the junior roles. The inheritance of permissions assigned to junior roles by a senior role, i.e., permission-inheritance, significantly reduces assignment overhead, as the permissions need only be explicitly assigned to the junior roles. Even though the notion of role hierarchy has been widely investigated, no earlier work has addressed the implication of the presence of temporal constraints on role hierarchies.

Sandhu [San98] distinguishes a role hierarchy into two types: *usage* hierarchy and *activation* hierarchy. Sandhu's *usage* hierarchy allows only permission-inheritance, whereas the *activation* hierarchy allows role-activation semantics as well, allowing a user assigned to the senior role to activate its junior roles also. In particular, he shows that the distinction allows capturing *dynamic* SoD constraints that may exist between hierarchically related roles. Our analysis further strengthens his arguments and shows that, in the presence of timing constraints on various entities, the separation of the permission-inheritance and the role-activation semantics provides a basis for capturing the various inheritance semantics of a hierarchy.

Moffet *et al.* [Mof99] have identified the need for three types of hierarchies: *isa* hierarchies, *activity* hierarchies and *supervision* hierarchies. These hierarchies are needed to address the needs of control principles in an organization, principles which include *separation of duty*, *decentralization* and *supervision and review* [Mof98]. They show that combining permission-inheritance and role-activation within a hierarchy can its ability to achieve organizational control needs. Clearly, hierarchies that capture temporal characteristics of roles presented in this chapter provide a basis for limiting such complete inheritance in a hierarchy, making it possible to support separation of duty and restricted inheritance. Furthermore, Moffet *et. al.* [Mof99] point out that the commercial organizations' demand for a dynamic access control model that can support a dynamic authorization state as well as a dynamic propagation of access rights has largely been neglected. The GTRBAC model's temporal framework and the trigger mechanism along with the temporal hierarchies presented in this chapter provide a strong basis for such dynamic features in an access control model.

In this chapter, we formally define the various types of temporal hierarchies and then analyze the effects of various temporal constraints on them. We show that the different types of hierarchies need to be further divided into subtypes in order to capture the complete inheritance semantics introduced due to different temporal properties associated with the roles of the hierarchies. We then present an analysis of hybrid hierarchies and inference rules for derived hierarchical relations among roles. Finally we present the transformation rule for hybrid hierarchies when roles are added, deleted, or modified in an existing hierarchy.

4.1 Temporal Role Hierarchy

Here, we take a slightly different approach than in [San98]. We explicitly define the hierarchy that allows only permissions to be inherited as an *inheritance-only hierarchy* or *I*-hierarchy (same as the *usage* hierarchy in [San98]) and the one that allows only the *activation-inheritance* semantics as *activation-only hierarchy* or *A*-hierarchy. We further refer to a hierarchy combining both the *inheritance* and *activation* semantics as a *general inheritance* or *Inheritance-Activation hierarchy* (*IA*-hierarchy for short). Finally, we extend the notion of hierarchical relations with respect to a time instant *t* in order to capture the fact that such semantics are time dependent. We use the predicate status expressions introduced in the previous chapter (refer to Table 3.3). In order to define the semantics of the hierarchies, we first introduce the following axioms to capture the key relationships among various status predicates:

Axioms: *For all* $r \in$ `Roles`, $u \in$ `Users`, $p \in$ `Permissions`, $s \in$ `Sessions`, *and time instant* $t \geq 0$, *the following implications hold*:

1. `assigned`$(p, r, t) \rightarrow$ `canbe_acquired`(p, r, t)
2. `assigned`$(u, r, t) \rightarrow$ `can_activate` (u, r, t)
3. `can_activate` $(u, r, t) \wedge$ `canbe_acquired`$(p, r, t) \rightarrow$ `can_acquire` (u, p, t)
4. `active`$(u, r, s, t) \wedge$ `canbe_acquired`$(p, r, t) \rightarrow$ `acquires`(u, p, s, t)

Axiom (1) states that if a permission is assigned to a role, then it "*can be acquired*" through that role. Axiom (2) states that all users assigned to a role *can activate* that role. Axiom (3) states that if a user *u can activate* a role *r*, then all the permissions that *can be acquired* through *r can be acquired* by *u*. Thus, for the simple case where user *u* and permission *p* are assigned to *r*, the axioms indicate that *u can acquire p*. Similarly, axiom (4) states that if there is a session in which a user *u* has activated a role *r*, then *u acquires* all the permissions that *can be acquired* through role *r*. We note that axioms (1) and (2) indicate that permission-acquisition and role-activation semantics are governed by explicit user-role and role permission assignments.

4.1.1 Unrestricted Hierarchies

Semantically, the purpose of a role hierarchy is to extend the possibility of permission-acquisition and role-activation beyond the explicit assignments, as we shall show next. Below, we define the formal semantics of the time-dependent role hierarchies. The following definitions do not consider the enabling times of the hierarchically related roles, and hence are termed *unrestricted* hierarchies. The restricted forms will be introduced in later sections.

Definition 4.1.1 (*Unrestricted inheritance-only hierarchy* or *I-hierarchy*): *Let x and y be roles such that* $(x \geq^t y)$, *that is, x has an inheritance-only relation over y at time t. Then the following holds:*

$\forall p, (x \geq^t y) \wedge \texttt{canbe\_acquired}(p, y, t) \rightarrow \texttt{canbe\_acquired}(p, x, t)$ $(c1)$

x is said to be a senior role of y, and conversely y is said to be a junior role of x, with respect to the inheritance-only hierarchy.

The condition characterizing the *inheritance-only* relation provides a new way of acquiring a permission through a role by using its relation with other roles. Its semantics indicates that a permission *can be acquired* through a role by direct inheritance of all the permissions of junior roles. Thus if $(x \geq^t y)$, the permissions that can be acquired through *x* include all the permissions assigned to *x* (by axiom (1)) and all the permissions that *can be acquired* through role *y* (by *c*1), which in turn include all the permissions assigned to *y* as well as all the permissions that *can be acquired* through *y*'s juniors (by axiom (1) and condition *c*1). This shows that the *I*-hierarchy is *transitive*. Note that the axioms and condition $c1$ do not allow u to activate y. Hence, the hierarchical relation $\geq^t$ is restricted to the *permission-inheritance* semantics only.

Definition 4.1.2 (*Activation hierarchy* or *A-hierarchy*): *Let x and y be roles such that* $(x \succcurlyeq^t y)$*, that is, x has an activation-only relation over y at time t. Then the following holds:*

$\forall u, (x \succcurlyeq^t y) \wedge \texttt{can\_activate}(u, x, t) \rightarrow \texttt{can\_activate}(u, y, t)$ $(c2)$

x is said to be a senior role of y, and conversely y is said to be a junior role of x, with respect to the activation inheritance.

Here, the *activation-only* semantics introduces a new "*can activate*" semantics between a user and a role . Axiom (2) states that a user is able to activate a role through explicit assignment, whereas the *A*-relation allows such activation through relations between roles, without a need for explicit user-role assignment. Condition ($c2$) states that if user *u can activate* role *x*, and *x* has an *A*-relation over *y*, then s/he *can activate* role *y* too, even if *u* is not explicitly assigned to *y*. However, note that an explicit assignment of *u* to *y*, while possible, would be redundant here. The set of axioms and condition $c2$ together allow a user *u* assigned to role *x* to activate all of *y*'s juniors. However, as condition $c1$ does not apply to an *A*-hierarchy, if $(x \succcurlyeq^t y)$, then *u* cannot acquire *y*'s permissions just by activating *x*. Note that the $\texttt{can\_activate}(u, x, t)$ predicate makes *A*-hierarchy transitive the same way the $\texttt{canbe\_aquired}(p, y, t)$ makes an *I*-hierarchy so.

Definition 4.1.3 (*General inheritance hierarchy* or *IA-hierarchy*): *Let x and y be roles such that* $(x \succsim^t y)$*, that is, x has a general inheritance relation over y at time t. Then the following holds*

$$(x \succsim^t y) \rightarrow (x \geq^t y) \wedge (x \succcurlyeq^t y)$$

The *IA*-hierarchy is the most common form of hierarchy and contains both *permission-inheritance* and *activation-inheritance* aspects of a hierarchy. In particular, a user assigned to a role can acquire the permissions of its junior roles without activating them. At the same time, s/he may activate the junior roles even though he is not explicitly assigned to them. Note that the definitions do not account for the enabling times of the roles that are hierarchically related.

On a given set of roles, there may be various inheritance relations. Therefore, we require that the following *consistency* property be satisfied in a role hierarchy:

Property (*Consistency of hierarchies*): *Let* $\langle f \rangle \in \{\geq^t, \succcurlyeq^t, \succsim^t\}$ *and* $\langle f' \rangle \in \{\geq^t, \succcurlyeq^t, \succsim^t\}/\{\langle f \rangle\}$. *Let x and y be distinct roles such that* $x\langle f \rangle y$; *then the condition* $\neg(y\langle f' \rangle x)$ *must hold.*

The main purpose of a hierarchical relation is the acquisition of permission of junior roles by a senior role using of any of the three hierarchy types. The *consistency* property ensures that a senior-junior relation between two roles in one type of hierarchy is not reversed in another type of hierarchy.

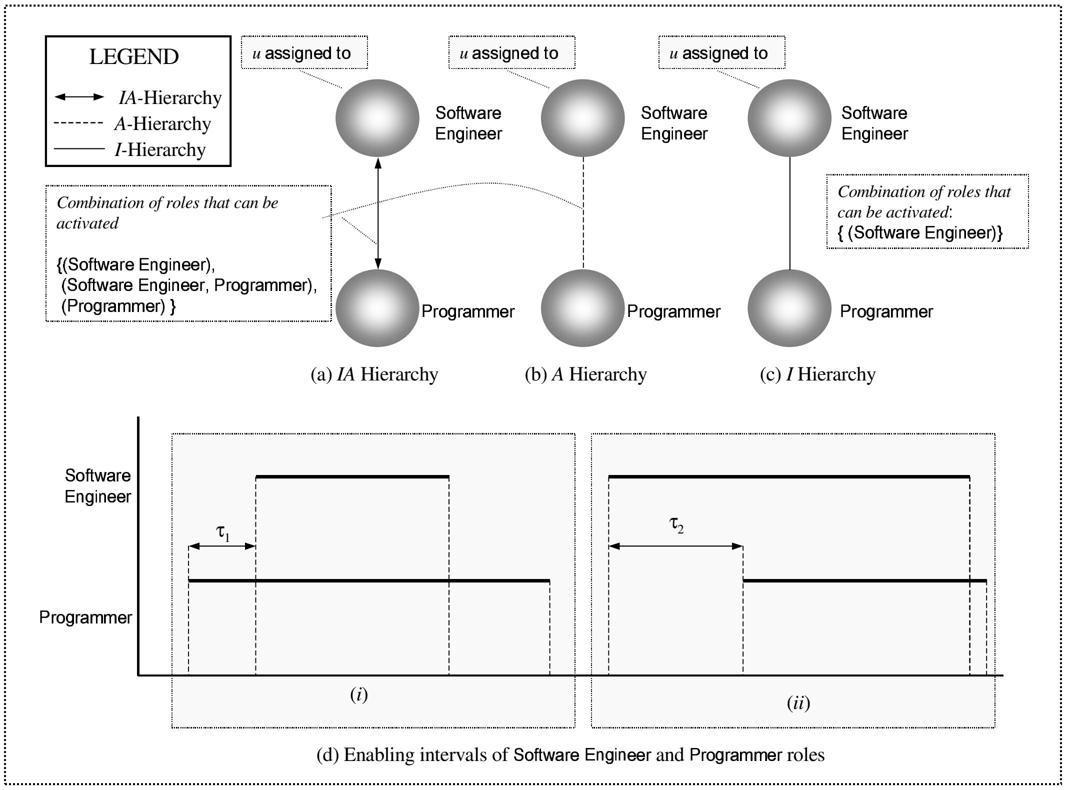

LEGEND
IA-Hierarchy
A-Hierarchy
I-Hierarchy
u assigned to
Software Engineer
Programmer
Combination of roles that can be activated
{(Software Engineer), (Software Engineer, Programmer), (Programmer) }
Combination of roles that can be activated:
{ (Software Engineer)}
(a) IA Hierarchy
(b) A Hierarchy
(c) I Hierarchy
τ1
τ2
(i)
(ii)
(d) Enabling intervals of Software Engineer and Programmer roles

Fig. 4.1. Hierarchy examples

Examples of the three hierarchies are given in Fig. 4.1, where the Software Engineer role is senior to the Programmer role. In Fig. 4.1(a) and 4.1(b), the combination of roles that a user *u*, assigned only to the Software Engineer role, can activate is {(Software Engineer), (Software Engineer, Programmer)

(Programmer)}. However, the permissions associated with the same combinations in the two cases are not the same. For example, if *u* activates the Software Engineer role, the permissions acquired by *u* under an *IA*-hierarchy (see Fig. 4.1(a)) is maximal, that is, both the roles' permissions are acquired. On the other hand, only the permissions assigned to the Software Engineer role are acquired in the case of an *A*-hierarchy (see Fig. 4.1(b)). Furthermore, the activation of the combination (Software Engineer, Programmer) is redundant in an *IA*-hierarchy in terms of what permissions are acquired, while it is significant in an A-hierarchy.

Under the *I*-hierarchy reported in Fig. 4.1(c), the user can activate only the Software Engineer role (unless of course, the user is also explicitly assigned to the Programmer role). However, he acquires maximal permissions, that is, permissions assigned to both roles.

4.1.2 Enabling Time Restricted Hierarchies

A hierarchy in the presence of various temporal constraints becomes dynamic as permissions and users can be assigned or de-assigned to any junior roles at times when a senior role is enabled. Furthermore, there are activation constraints that need to be accounted for when either of the hierarchy types is considered. Here, we consider the effect of the presence of temporal assignment constraints on both *inheritance* and *activation* hierarchies.

Inheritance-only hierarchy (I-hierarchy)

As we can see, in an *I*-hierarchy, the permissions of a junior role are implicitly assigned to the senior role itself. However, in the presence of temporal constraints, we need to be able to capture various dynamic aspects of the hierarchy.

Let us revisit the *I*-hierarchy of Fig. 4.1(c). Fig. 4.1(d) shows two possible intervals associated with the enabling times of the two roles. In Fig. 4.1(d)-(*i*), we see that the enabling interval of the Software Engineer role is a subset of that of the Programmer role. In this case, the *I*-hierarchy has the semantics similar to the non-temporal RBAC; that is, whenever *u* activates the Software Engineer role s/he also acquires the permissions of the Programmer role, because at that time the Programmer role is also enabled. Thus, in interval τ_1, *u* cannot acquire any permissions of the Programmer role even if this role is enabled, as the Software Engineer role is disabled at that time. It is also possible that there is a temporal interval in which the Software Engineer role is enabled but the Programmer role is not, as indicated by interval τ_2 in Fig. 4.1(d)-(*ii*). In such a case, we can see that the following two approaches can be used to capture the inheritance semantics:

1. *Weakly restricted approach* (I_w): The permissions of the Programmer role are inherited by the Software Engineer role in interval $\tau_{2,}$
2. *Strongly restricted approach* (I_s): The permissions of the Programmer role are not inherited by the Software Engineer role in interval τ_2.

Under the *weakly restricted* approach, every permission that can be acquired through a junior role can also be acquired through its senior roles under an *I*-hierarchy, irrespective of whether the junior role is enabled or disabled. Under the *strongly restricted* approach, each permission that can be acquired through a junior role can also be acquired through its senior roles only in intervals where the junior role is also enabled.

Table 4.1 summarizes the inheritance semantics of an *I*-hierarchy in the presence of temporal constraints. I_w refers to the *I*-hierarchy that adopts the *weakly restricted* approach above, whereas I_s refers to adopting the *strongly restricted* approach. Note that the two types of hierarchy act differently only in intervals where the senior role is enabled while the junior role is disabled.

Activation-only hierarchy (*A*-hierarchy)

We see that when we have an *A*-hierarchy, it is natural to just use the second approach given above. That is, there is no *activation*-inheritance allowed in interval τ_2. This is because of an explicit need for activating a junior role by a user assigned to its senior role in order to acquire the junior role's permissions, and in τ_2, the junior role cannot be activated. If we also try to enforce the first possibility mentioned above then it will conflict with the semantics of an enabled role, as only enabled roles can be activated.

However, as an *activation* hierarchy needs a user who is assigned to the senior role to activate a junior role in order to acquire the junior role's permissions, the issue of the propagation of temporal user-role assignment down the *A*-hierarchy needs to be considered. For example, consider the roles Software Engineer and Programmer forming the *A*-hierarchy in Fig. 4.1(b). Consider again the same enabled times of the two roles as in Fig. 4.1(d). We need to determine whether the user is to be allowed to activate the junior role at the time when the senior role he is assigned to is not enabled, as indicated by the interval τ_1 in Fig. 4.1(d)-(*i*). For such a case, we can again delineate the following two approaches:

1. *Weakly restricted approach* (A_w): The user u is allowed to activate Programmer role in the A-hierarchy at any time the Programmer role is enabled.
2. *Strongly restricted approach* (A_s): The user u is allowed to activate the Programmer role only if both the Software Engineer and Programmer roles are enabled (note that he does not need to activate the Software Engineer role).

In both approaches, when a user tries to activate a role in an *activation* hierarchy, additional checks need to be carried out. The first check is to determine if the user is assigned to any role, up the hierarchy, starting from the role it is attempting to activate. The second check is required to determine if the senior role that a user is assigned to is also enabled. If the senior role is disabled, we then need to deactivate all activations of junior roles by the user assigned to the senior role.

In Table 4.1, A_w refers to the *activation* hierarchy that adopts the *weakly restricted* approach, whereas A_s refers to that adopting the *strongly restricted* approach. We note that the two types of hierarchy act differently only in intervals where the senior role is disabled whereas the junior role is enabled.

Table 4.1.
Inheritance semantics of enabling time restricted hierarchy

r_1 is senior of r_2→ ↓Hierarchy Type		τ r_1 `disabled,` r_2 `enabled`	τ r_1 `enabled'` r_2 `disabled`
I-hierarchy	I_w	No inheritance in τ	*Permission*-inheritance in τ (*by activating* r_1)
	I_s	No inheritance in τ	No inheritance in τ
A-hierarchy	A_w	*Activation*-inheritance in τ (*by activating* r_2)	No inheritance in τ
	A_s	No inheritance in τ	No inheritance in τ
IA-hierarchy	IA_w	*Activation*-inheritance in τ (*by activating* r_2)	*Permission*-inheritance in τ (*by activating* r_1)
	IA_s	No inheritance in τ	No inheritance in τ

General inheritance hierarchy (*IA*-hierarchy)

As general inheritance embodies both the *permission inheritance* and *role-activation* semantics of a role hierarchy, it is simply a combination of the two. In other words, in interval τ_1, the general hierarchy can benefit from the use of role-activation semantics and activate the junior role using the *weakly restricted* semantics. Similarly, in interval τ_2, the *inheritance-only* semantics can be used and inheritance through the senior role using *weakly restricted* semantics can be utilized. This is shown in Table 4.1.

We now formally define the *weakly restricted* and *strongly restricted* forms of each hierarchy type discussed in the previous section.

Definition 4.1.4 (*Weakly restricted inheritance-only hierarchy* or *I_w-hierarchy*): *Let x and y be roles such that* ($x \geq_{w,t} y$); *that is, x has a weakly restricted inheritance-only relation over y at time t. Then the following holds:*

$$\forall p, (x \geq_{w,t} y) \wedge \texttt{enabled}(x, t) \wedge \texttt{canbe\_acquired}(p, y, t) \rightarrow \texttt{canbe\_acquired}(p, x, t)$$

We note that for a $x \geq_{u,t} y$ relation, only role x needs to be enabled at time t. Role y may or may not be enabled at that time. Similarly, for the *weakly restricted A*-hierarchy, $x \succcurlyeq_{w,t} y$, only role y needs to be enabled as shown in the following definition.

Definition 4.1.5 (*Weakly restricted activation hierarchy* or *A_w-hierarchy*): *Let x and y be roles such that* ($x \succcurlyeq_{w,t} y$); *that is, x has a weakly restricted activation-only relation over y at time t. Then the following holds*

$$\forall p, (x \succcurlyeq_{w,t} y) \wedge \texttt{enabled}(y, t) \wedge \texttt{can\_activate}(u, x, t) \rightarrow \texttt{can\_activate}(u, y, t)$$

Definition 4.1.6 (*Weakly restricted general inheritance hierarchy* or *IA_w-hierarchy*): *Let x and y be roles such that* ($x \succsim_{w,t} y$); *that is, x has a weakly restricted general inheritance relation over y at time t. Then the following holds:*

$$\forall p, (x \succcurlyeq_{w,t} y) \rightarrow (x \geq_{w,t} y) \wedge (x \succcurlyeq_{w,t} y)$$

The *strongly* restricted forms of the hierarchies allow inheritance semantics to be valid only when both the hierarchically related roles are enabled. The following definitions formalize these hierarchies.

Definition 4.1.7 (*Strongly restricted inheritance-only hierarchy* or *I_s-hierarchy*): *Let x and y be roles such that* ($x \geq_{s,t} y$); *that is, x has a strongly restricted inheritance-only relation over y at time t. Then the following holds:*

$$\forall p, (x \geq_{s,t} y) \wedge \texttt{enabled}(x, t) \wedge \texttt{enabled}(y, t) \wedge \texttt{can\_be\_acquired}(p, y, t) \rightarrow \texttt{can\_be\_acquired}(p, x, t)$$

Definition 4.1.8 (*Strongly restricted activation hierarchy* or A_s *-hierarchy*): *Let x and y be roles such that* ($x \succcurlyeq_{s,t} y$); *that is, x has a strongly restricted activation-only relation over y at time t. Then the following holds:*

$$\forall p, (x \succcurlyeq_{s,t} y) \wedge \texttt{enabled}(x, t) \wedge \texttt{enabled}(y, t) \wedge \texttt{can\_activate}(u, x, t) \rightarrow \texttt{can\_activate}(u, y, t)$$

Definition 4.1.9 (*Strongly restricted general inheritance hierarchy* or IA_s-*hierarchy*): *Let x and y be roles such that* ($x \gtrsim_{s,t} y$); *that is, x has a strongly restricted general inheritance relation over y at time t. Then the following holds:*

$$(x \succcurlyeq_{s,t} y) \rightarrow (x \geq_{s,t} y) \wedge (x \succcurlyeq_{s,t} y)$$

The *weakly restricted* and *strongly restricted* forms of hierarchies deal with the cases where at least one of the two roles is enabled. The hierarchies defined in section 4.1.1 do not consider the enabling times of the related roles. In this sense, the *weakly restricted* and *strongly restricted* hierarchies are specializations of the *unrestricted* hierarchy types with an additional requirement that one or both roles be enabled for the inheritance semantics to be valid.

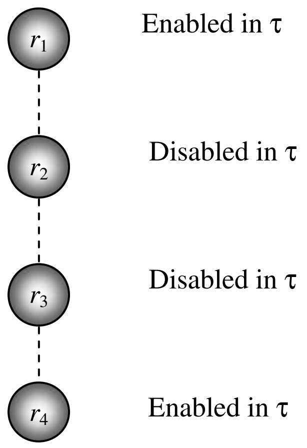


Fig. 4.2. Inheritance through disabled roles

It is important to note that if the inheritance between two roles is defined just by using one of the *unrestricted* types, the inheritance semantics applies even when the roles are not enabled. The benefit of such a case is in the propagation of the inheritance semantics along the hierarchy, as illustrated in Fig. 4.2. Assume that the hierarchy is an *unrestricted A*-hierarchy, and consider an interval τ in which only roles r_1 and r_4 are enabled. We can see that Definition 4.1.2 applies to each pair and the result is that any user assigned to r_1 can also activate r_4. Now suppose it is a *weakly restricted A*-hierarchy.

As r_2 and r_3 are both disabled, the activation-inheritance semantics does not apply between them. And hence, it blocks the activation-inheritance semantics between r_1 and r_4 also. Thus, no user assigned to r_1 will be able to activate role r_4.

We illustrate with the examples reported in Fig. 4.2 the practical uses of the various kinds of restricted hierarchies.

Example 4.1.1: Consider the I_w-hierarchy in Fig. 4.3(a). Note, the SeniorSecurityAdmin role is enabled only in interval (8pm, 11pm). Neither of the junior roles is enabled in the entire interval (8pm, 11pm). But the I_w relation allows a user who activates the SeniorSecurityAdmin role to acquire all the permissions of the junior roles too. This may be desirable if the SeniorSecurityAdmin role is designed to perform special security operations for checking and maintenance. In such a case, it is reasonable to think that the user assigned to the SeniorSecurityAdmin role will need all the administrative privileges of the junior roles. The temporal restrictions on SecurityAdmin1 and SecurityAdmin2 restrict the users assigned to them in carrying out corresponding system administration activities only in the specified intervals. However, here, the user assigned to SeniorSecurityAdmin cannot assume the role of the junior roles SecurityAdmin1 and SecurityAdmin2. To remove this limitation, we can use the IA_w-hierarchy instead.

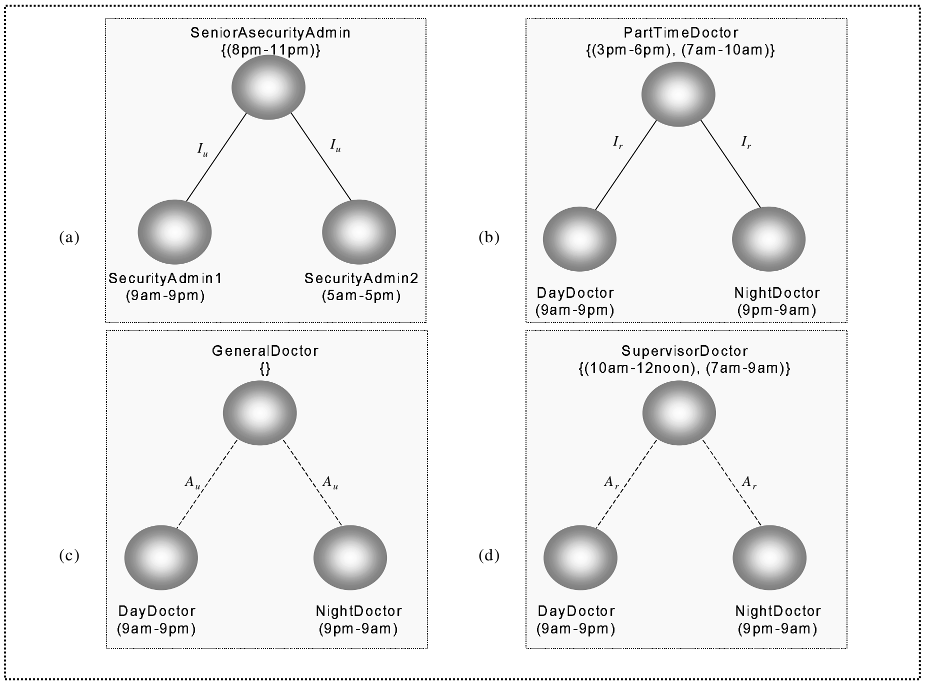


Fig. 4.3. Examples of hierarchy types

The hierarchy in Fig. 4.3(b), on the other hand, is of type I_s. The senior role is the PartTimeDoctor role, which has two intervals in which it can be enabled, (3pm, 6pm) and (7am, 10am). If a user activates the PartTimeDoctor role in the first interval, according to the I_s relation, he essentially gets all the privileges of only the DayDoctor role, as the NightDoctor role is disabled at that time. Now, consider the second interval. We see that it overlaps with the enabling times of the two junior roles. Hence, if the user activates the PartTimeDoctor role in the second interval, he acquires the privileges of only the NightDoctor role in the sub-interval (7am-9am) and that of only the DayDoctor role in the interval (9am, 10am). Thus, we see that the two different semantics of an *inheritance* hierarchy can be used to achieve different needs. Again, a part time doctor cannot work as a DayDoctor or a NightDoctor, although, he can acquire the permissions assigned to them. If a user is also to be allowed to use the junior roles, we can use IA_s-hierarchy instead.

Now, consider Fig. 4.3(c). Note that in this case, there is no interval in which the GeneralDoctor role can be enabled. However, since the activation hierarchy is of type A_w, any user assigned to the GeneralDoctor role can activate either of the junior roles when they are enabled. In effect, any one assigned to the GeneralDoctor role can activate both the DayDoctor and the NightDoctor roles whenever they are enabled.

Fig. 4.3(d) illustrates the use of an activation hierarchy of type A_s. Here, a doctor supervisor can assume the SupervisorDoctor role in intervals (10am, 12noon) and (7am, 9am). In the first interval, the supervisor will be able to acquire all the privileges of the DayDoctor role by activating it, and in the second interval, he will be able to acquire all the privileges of the NightDoctor role by activating it along with the SupervisorDoctor role. The SupervisorDoctor role may simply contain some extra privileges that are required for the supervision task during day and night.

Activation Constraints and Enabling Time Restricted Role Hierarchies

Each individual role in a hierarchy may have its own activation constraints. These constraints provide a way of limiting resource use by limiting access to resources. In either of the *inheritance* or *activation* hierarchies, the question of whether such activation constraints have any effect on the permission-inheritance becomes an issue. Next, we consider a hierarchy in the presence of cardinality constraints and then generalize the discussion to the other activation constraints.

Assume that the Programmer role has a permission set, say *P*, associated with a licensed software package. Suppose that there are five user licenses for the package indicating that only five users can concurrently execute any program of the package. Such a constraint could be directly expressed as a cardinality constraint on the Programmer role. Software Engineer, being senior to Programmer, can inherit *P*. However, at any time the number of concurrent executions of any particular program by users assigned to the Software Engineer role and Programmer role needs to be restricted to five. If we adopt an *I* or *IA*- hierarchy, we observe that correctly enforcing such a constraint is not straightforward:

- As the cardinality constraint is applied on the Programmer role, it cannot capture the use of the permission set *P* by the Software Engineer role. Hence, there may be five concurrent activations of the Programmer role and some activations of the Software Engineer role at any time, allowing more than five users to have access to the programs. In such a situation extra measures need to be taken to enforce the cardinality constraint.
- An alternative solution may be to develop a constraint expression on the combination of roles, such as the one that says "*the number of concurrent activations of* Software Engineer *and* Programmer *roles should be at most five*". However, this introduces other problems because of the fact that *P* could be only a subset of the permission set associated with the Software Engineer role. In such a case, the constraint will enforce the same cardinality constraint on all the permissions assigned to the Software Engineer role and not only to *P*. For example, six concurrent activations of the Software Engineer role will not be permitted and therefore permissions other than *P* assigned to it cannot be used, which may not be what we want.

We note that the cardinality constraint on a role is aimed at controlling the concurrent use of permissions and, hence, we say that the cardinality constraint is *permission-oriented*.

Now suppose that the role hierarchy is an *A*-hierarchy. As users need to explicitly activate junior roles in order to acquire its permissions, the above problems do not arise. Hence, in the example, if we use the *activation* hierarchy rather than the *inheritance* hierarchy, the intended cardinality control on the use of *P* is easily enforced. Furthermore, if there is another role Programmer2 that is also a junior to the Software Engineer role and that has a permission set P_2 and cardinality constraint (*permission-oriented* as in Programmer) of *n*, the simple overall *activation* hierarchy is an effective solution.

As another example, suppose we want at the most five nurses and three doctors on active duty at a time, and we create two roles, Doctor and Nurse, such that Doctor is senior to Nurse. Here, the cardinality constraints are *user-oriented* rather than being *permission-oriented* in that, by imposing the cardinality constraint of three on the Doctor role and five on the Nurse role, we want to restrict scheduling at the most three doctors and five nurses at a time. We can assume that there is no need to control the permission distribution associated with the Doctor and Nurse roles, as in the previous case.

Now assume that we use an *A*-hierarchy. This means, when there are three doctors and five nurses in active duty, the doctors do not have permissions that are associated with the Nurse role, as they cannot activate the Nurse role. If we want each doctor to also be able to use permissions associated with the Nurse role every time s/he is active, by making her/him activate both the roles, then only two nurses will be able to activate the Nurse role. This is not what we intend to enforce. However, if we adopt an *I*-hierarchy or an *IA*-hierarchy, the problem does not arise, because, the permissions associated with the Nurse role are implicitly assigned to the Doctor role too. There is no need to explicitly enable the Nurse role by a user assigned to the Doctor role.

Note that an *I*-hierarchy or an *IA*-hierarchy can capture any activation constraint on roles when the cardinality control implies the control on the number of users. An *A*-hierarchy, on the other hand, captures any activation constraint on roles when the activation control implies control on the distribution of permissions.

Similar to the cases in cardinality constraint, an *I*-hierarchy or an *IA*-hierarchy is appropriate when other activation constraints imply a *user-oriented* control, whereas an *A*-hierarchy is appropriate when the activation constraints imply a *permission-oriented* control. Furthermore, the prevalent concept of a role as a "*set of permissions*" implies that the *permission-oriented* activation control is a phenomenon that is closer to the RBAC concepts than the *user-oriented* activation control.

Periodicity and Duration Constraint Expression

A hierarchical relation between two roles is essentially a constraint on them. Hence, the GTRBAC model's constraint enabling/disabling expression can be used to *enable* or *disable* a hierarchical relation. Thus, if *h* is a hierarchical relation ($rs\langle f\rangle\ r$), its enabling/disabling can be done by the event "`enable/disable` *h*". This allows administrators to dynamically change the hierarchical relationships on a set of roles through predefined periodicity constraints, run-time requests, and triggers.

For specifying the periodicity constraints on a hierarchy, we simply use the GTRBAC model's periodicity expression framework. Thus, we use (I, P, `enable/disable` h) to mean that the enabling or disabling of hierarchical relation h is constrained by the interval expression (I, P); i.e., *for all* $t \in Sol(I, P)$, h is enabled/disabled, where $Sol(I, P)$ is the set of valid time instants denoted by (I, P).

Similarly, we use the constraint expression c_d = (D_h, `enable/disable` h) to define the duration constraint on a hierarchy h. D_h indicates how long the hierarchical relation h may hold. In other words, if $D_h = t_{end} - t_{start}$, where t_{start} is the time at which h becomes valid, then *for all* $t \in (t_{end}, \ t_{start})$, the relation h holds. Note that t_{start} is not known in advance and is therefore determined by the firing of the event `enable/disable` h by a trigger or a run-time request. For example, suppose we have the following trigger and a duration constraint on a hierarchical relation:

`enable` $r \rightarrow$ `enable` h `after` 10 *min*

(*1 Hour*, `enable` h)

Here, only 10 minutes after role r is enabled will role rs become the senior of r. Furthermore, the duration constraint allows rs to remain senior of r for only 1 *hour*.

We also note that the duration constraint can also be of forms (I, P, D_h, `enable/disable` h) and (D, D_h, `enable/disable` h). Constraint c = (I, P, D_h, `enable/disable` h) implies that the enabling/disabling of h can be done for duration D_h only within the intervals defined by (I, P). Now, suppose that constraint (c) above is replaced by ([Mondays, Fridays], D_h, `enable/disable` h). In that case, if the trigger tr is fired, then, on the days other than Mondays and Fridays, role rs is not the senior of r. But on Mondays and Fridays, the firing of tr makes rs the senior of r for 1 *hour*.

If the duration constraint (c) is (D, D_h, `enable/disable` h), it needs to be first enabled by a constraint enabling expression "`enable` c". If tr fires after constraint (c) has been enabled, then the hierarchical relation is enabled and rs becomes the senior of r. Compared to this, constraint c_d = (D_h, `enable/disable` h) indicates that the duration constraint c_d is enabled at all times.

A practical use of such a dynamically changing hierarchical relation is in the case where a senior (acting as a *supervisor*) is allowed to inherit the *read-only* permissions of its juniors. For example, a particular end of the week period can be specified when the *supervisor* can read all his juniors' documents, by enabling the senior-junior hierarchical relations. This will allow her/him to carry out a progress review of the project as well as the weekly progress of each individual team member that he is supervising. Moffet *et. al.*

[Mof99] have identified such a supervision-review capability as an important organizational control principle.

4.2 Uniquely Activable Set (UAS) of Role Sets

In a role hierarchy containing multiple hierarchy types, referred to as a *hybrid* hierarchy, a user may be able to activate different sets of junior roles in a session. Sets of roles that can be activated or permissions that can be acquired by a user at a particular time indicate the overall access capabilities of the user. From the perspective of the principle of the least privilege, it may be necessary to ensure that such activable sets of roles do not result in giving a user unnecessary access capabilities. Determining such sets can become very complex in a hybrid hierarchy. Furthermore, we may want to know what indirect relations may exist between roles that are not directly related so that when modifications are made to the hierarchy, original relations are not violated. For example, consider the relatively simple hybrid hierarchy of Fig. 4.4. Here, determining sets of roles that can be activated in a single session by a user assigned only to role, say r_3, is not straightforward. Similarly, when we delete a role, say s_1, we need to make sure that desirable relations between r_3 and t_1, r_3 and s_2 or r_3 and x_1 are retained.

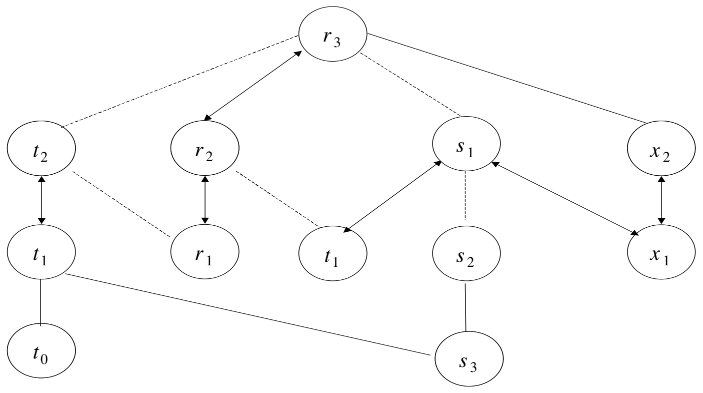


Fig. 4.4 An example hybrid hierarchy

A flexible model, such as GTRBAC, needs formal tools to analyze hierarchies in order to determine such activation and permission acquisition capabilities of users who are assigned to the roles in a hierarchy. The model also needs to identify how roles are related to each other indirectly through the permission-inheritance and role-activation semantics in the presence of different hierarchical relations between roles. Such tools are very essential for an efficient security administration and management function. In this section, we present an extensive analysis of hybrid temporal role hierarchies. The results provide a formal basis for developing support tools for analyzing hybrid temporal hierarchies in GTRBAC.

In this section, we present the following:

- We define the notion of the *uniquely activable set* of a hierarchy that can be used by security administrators for determining the access capabilities that a user can obtain from a role hierarchy in a single session. We also show formally how the set can be determined in a hybrid temporal role hierarchy.
- We introduce a set of inference rules that allows inferring the hierarchical relationships between pairs of roles that are not directly related and show that the inference rule set is sound and complete.
- We develop a set of hierarchy transformation algorithms to assist in administering role hierarchies when the roles are added, deleted or modified.

We introduce the notion of a *uniquely activable set* (UAS) and present formal results for characterizing it for a hierarchy. The UAS associated with a hierarchy is essentially the *set of role sets* that can be activated by a user assigned to a role of the hierarchy. In a hierarchy that allows the coexistence of the multiple hierarchy types, the permission-inheritance and role-activation semantics can be complex, thus making administration and management of large hierarchies difficult. As UAS gives the role combinations that can be activated by a user in a single session, it helps in determining the granularity of permission sets that can be acquired by users through a role in a hierarchy. Thus, UAS is mainly relevant from the perspective of the *principle of the least privilege*. Here, we first determine the UAS characteristics of a *monotype* hierarchy with only one type of hierarchical relation over the roles, followed by that of a *hybrid* linear path and then formalize the results for the more general role hierarchy. We then introduce the notion of *acquisition equivalence* to characterize equivalent hierarchies in order to address the usefulness of a hybrid hierarchy. Here onwards we will only use the *unrestricted* forms of hierarchies. The results can be extended easily to *restricted* forms by considering additional requirements associated with them. Furthermore, although we consider *unrestricted* forms of temporal hierarchies, the results directly apply to the non-temporal case with the same three different hierarchy types, as non-temporal cases are simply the special cases of temporal hierarchies in which the hierarchical relations apply at all times.

4.2.1 Computing UAS of a Hierarchy

We represent by *UAS*(H, t) the UAS associated with a user assigned to the senior-most role of a hierarchy H at time instant t. For a given role set $X = \{x_1, x_2, \ldots, x_n\}$ and a set of hierarchy relations $[f] \subseteq \{\geq^t, \succcurlyeq^t, \succsim^t\}$, we represent a general hierarchy H over X as (X, [f]). If $[f] = \{<f>\}$ is a singleton set with hierarchy relation $<f>$, then we call H a *monotype hierarchy* and write (X, $<f>$), else we call H a *hybrid hierarchy*. Furthermore, H is a linear path over X if $(X, [f]) = \{x_i <f_{i,j}> x_j \mid i = 1 \text{ to } n\text{-}1, j = i\text{+}1, \textit{and } <f_{i,j}> \in [f]\}$, and we represent it as LH (i.e., $LH = H$). LH may be either monotype, represented as $L = (X, <f>)$, or a hybrid type, represented as $Lh = (X, [f])$. We use Roles(H) to indicate the set of roles in a hierarchy H. In this dissertation, we assume that

(a) the set of permissions assigned to each role in Roles(H) is distinct, and
(b) for each hierarchy H, there is only one senior-most role, indicated by S_H. The results can be easily extended to deal with a general hierarchy. We use J_H to denote the set of junior-most roles of H.

We use notation $P(r, t)$ to refer to the *set of permissions assigned to role r at time t*. Similarly, given a set X of roles, we use $P(X, t)$ to denote $\bigcup_{r \in X} P(r,t)$. Now, we formally define the UAS of a hierarchy as follows:

Definition 4.2.1 ***(Uniquely Activable Set of a Hierarchy H)***: *Let* $H = (X, [f])$ *be a hierarchy. Then,* $UAS(H, t) = \{Y_1, Y_2, \ldots, Y_m\}$, *where* $\varnothing \subset Y_i \subseteq$ Roles(H) *for each* $i \in \{1, 2, \ldots, m\}$, *is the* uniquely activable set *of role sets for a user assigned only to role* S_H *at time t, if the following conditions hold*:

i. $i, j \in \{1, 2, \ldots, m\}$ *and* $i \neq j$, $P(Y_i, t) \neq P(Y_j, t)$, *and*
ii. $\forall Z \subseteq$ Roles(H) *s.t.* $Z \notin UAS(H, t)$, *if* $P(Y, t) = P(Z, t)$ *for a* $Y \in UAS(H, t)$, *then* $(|Y| < |Z|)$;

where $|A|$ *is the cardinality of set A.*

Note that each element Y_i is a subset of Roles(H). As condition (i) indicates, *UAS*(H, t) is unique because for any pair of role sets of *UAS(H, t)*, the permission sets associated with them are not the same as per assumption (*a*). Condition (ii) considers the possibility of different role sets associated with the same set of permissions. In such a case *UAS(H, t)* contains the role set that has the least number of roles. In conjunction with assumption (*b*), condition (ii) prevents a pair of senior and junior roles, e.g. of an *IA*-hierarchy, to be in a role set of *UAS(H, t)*. For instance, if relation $(x \succsim^t y)$ is in H, then the set $\{x\}$ and not $\{x, y\}$ will be in $UAS(H, t)$, as $P(\{x\}, t) = P(\{x, y\}, t)$. The UAS values for I, A and IA-hierarchy can differ significantly because of the difference in *permission-inheritance* and *role-activation* semantics associated with them.

As a *hybrid* linear path may have different types of hierarchical relations it can be decomposed into a set of *monotype* linear paths. We term such a decomposition of a *hybrid* linear path into a set of *monotype* components as a *horizontal partition*. The following definition formalizes these concepts.

Definition 4.2.2 (*Horizontal Partition*) *Let* $Lh = (X, [f])$ *be a hybrid linear path over role set X. Then Lh can be represented by an* ordered set of monotype linear paths*, that is,* $Lh = \{L_1, L_2, \ldots, L_n\}$ with $X = X_1 \cup X_2 \cup .. \cup X_n$, *provided that the following conditions hold*:

1. *for all* $i \in \{1, .., n\text{-}1\}$, (*i*) *if* $L_i = (X_i, \langle f_i \rangle)$ *then* $\langle f_i \rangle \neq \langle f_{i+1} \rangle$, (*ii*) $X_i \cap X_{i+1} = \{J_{Li}\} = \{S_{L(i+1)}\}$, *and*
2. *for all* $i \in \{1, .., n\}$ *and* $(i + 1 < j \leq n)$ or $(1 \leq j < i - 1)$, $X_i \cap X_j = \emptyset$,

Here, we say that $\{L_1, L_2, \ldots, L_n\}$ *is the* complete horizontal partition *of* Lh. *Lh can also be written as* $\{L_1, Lh'\}$, $\{Lh'', L_n\}$, $\{Lh_x, Lh_y\}$, *etc*., *each of which is a horizontal partition of L, not necessarily complete. We denote the senior-most and the junior-most roles of a hybrid hierarchy Lh as* S_{Lh} *and* J_{Lh}. It is easy to see that $S_{Lh} = S_{L1}$, and $J_{Lh} = J_{Ln}$.

As indicated, we will use L to represent a *monotype* linear path, Lh to represent a *hybrid* linear path, and LH to mean either of them. H represents any hierarchy, which may simply be a linear path. As indicated by definition 4.2.2, we can break a *hybrid* linear path into an ordered set of *monotype* linear paths. Such a horizontal partition of a *hybrid* path into its *monotype* components allows us to use the UAS of the *monotype* linear paths to determine the UAS of a *hybrid* linear path. Note that a *complete horizontal partition* consists of monotype linear paths that are maximal in the sense that combining any consecutive pair of component linear paths will give a *hybrid* linear path, as indicated by condition (a) of the definition. The use of horizontal partitions that are not complete allows expressing a hybrid linear path as a combination of smaller linear paths that may be of *hybrid* type. Furthermore, the complete horizontal partitioning of a hybrid linear path allows us to determine its UAS by using the UASs of the component monotype linear paths. Example 4.2.1 illustrates partitioning of a hybrid linear path into its components.

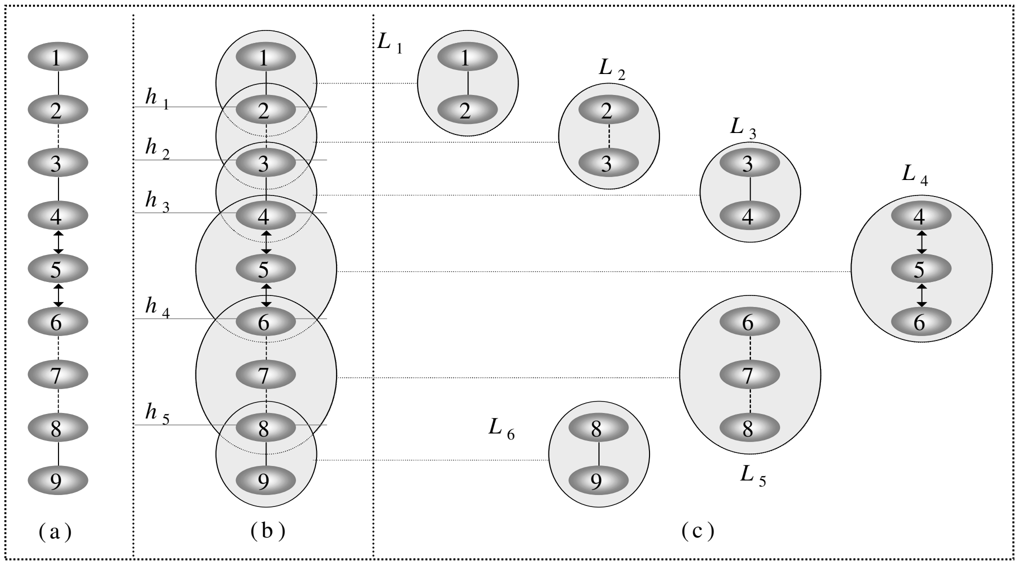


Fig. 4.5 Horizontal partition of a *hybrid* linear path

Example 4.2.1: Consider the role hierarchy of Fig. 4.5. The complete horizontal partition of the hybrid linear path is $\{L_1, L_2, L_3, L_4, L_5, L_6\}$. We note that if L_4 is split into $L_{4,1} = (\{4, 5\}, IA\text{-type})$ and $L_{4,2} = (\{5, 6\}, IA\text{-type})$, then $\{L_1, L_2, L_3, L_{4,1}, L_{4,2}, L_5, L_6\}$ is not a *complete horizontal* partition, as $L_{4,1}$ and $L_{4,2}$ do not satisfy condition (1) of definition 4.2.2.

In this dissertation, we also use functions $\text{sub}_L(LH)$ and $\text{sub}_U(LH)$ that return the lower and upper parts of a linear path *LH* That is, if $L = (\{x_1, x_2, \ldots, x_n\}, \langle f \rangle)$, then,

- $\text{sub}_L(L) = (\{x_2, \ldots, x_n\}, \langle f \rangle)$; $\text{sub}_U(L) = (\{x_1, x_2, \ldots, x_{n-1}\}, \langle f \rangle)$; *For* $L = (\{x, y\}, \langle f \rangle)$, $\text{sub}_L(L) = \text{sub}_U(L) = \varnothing$;
- $\text{sub}_L(Lh)=\{\text{sub}_L(L_1), L_2, \ldots, L_n\}$ and $\text{sub}_U(Lh)=\{L_1, L_2, \ldots, \text{sub}_U(L_n)\}$, where $Lh = \{L_1, L_2, \ldots, L_n\}$ *is the complete horizontal partition of Lh.*

Here, $\text{sub}_L(LH)$ and $\text{sub}_U(LH)$ return the lower and the upper sub-paths of *LH*. $\text{sub}_L(x\langle f \rangle y) = \text{sub}_U(x\langle f \rangle y) = \varnothing$ indicates that path $(x\langle f \rangle y)$ has no sub-paths. Because of the different activation semantics associated with each hierarchy type, the UAS associated with each type is also different. The following theorem formally characterizes the UAS of a *monotype* linear hierarchy:

Theorem 4.1: *Let* $H = (X, \langle f \rangle)$ *be a monotype hierarchy on role set* $X = \{x_1, x_2, \ldots, x_n\}$ *and the hierarchy relation* $\langle f \rangle \in \{\geq^t, \succcurlyeq^t, \succsim^t\}$. *Then,*

$$UAS(H, t) = \begin{cases} \{\{S_H\}\} = \{\{x_1\}\} & if\ (\langle f\rangle = \geq^t) \\ 2^X / \varnothing & if\ (\langle f\rangle = \succcurlyeq^t) \\ \{\{x_1\}\} \cup \{2^Z \mid Z = \bigcup_i z_i\ ;\ z_i \in UAS(SubH_i(H), t)\} / \varnothing & if\ (\langle f\rangle = \succsim^t) \end{cases}$$

where $SubH_i$(H) *represents the* i^{th} *subhierarchy of H.*

The theorem basically states that for a monotype linear hierarchy over an *I*-relation, the UAS only contains the senior-most role. For a monotype hierarchy with an *A*-relation, the UAS contains the power set of the role set *X* without the empty element; i.e., a user assigned to the senior-most role can activate every combination of the roles in the hierarchy. For a monotype hierarchy with an *IA*-relation, the UAS contains set elements containing individual roles of the hierarchy and the combinations of roles that occur in the different sub-hierarchies of the seniormost role. The proof for the theorem follows directly from the transitive properties of the hierarchical relations and the permission inheritance-only and/or role activation-only semantics of the three hierarchies. Example 4.2.2 illustrates the use of the results of Theorem 4.1.

Example 4.2.2: Consider the monotype hierarchies of Fig. 4.3. In each of the monotype hierarchies in figures 4.3(i)(a) and 4.3(i)(b), the UAS only contains the set with the senior-most role of the hierarchy, as each of them has the senior-most role related to its junior(s) by *I*-relation(s). For hierarchies in figures 4.3(i)(c) and 4.3(i)(d), assuming *unrestricted* forms in both the cases, instead of the *restricted* forms indicated in the figure, the UASs are as follows:

In the hierarchy of Fig. 4.3(c), UAS = {{GeneralDoctor}, {DayDoctor}, {NightDoctor}, {GeneralDoctor, DayDoctor}, {GeneralDoctor, NightDoctor}, {DayDoctor, NightDoctor},{GeneralDoctor, DayDoctor, NightDoctor}}. However, GeneralDoctor is never enabled. Furthermore, if we take the periodicity constraints on the roles, we have, in interval (9am – 9pm), UAS = {{DayDoctor}}, and in interval (9pm – 9am), UAS = {{NightDoctor}}.

In the hierarchy of Fig. 4.3(c), UAS = {{SupervisorDoctor}, {DayDoctor}, {NightDoctor}, {SupervisorDoctor, DayDoctor}, {SupervisorDoctor, NightDoctor}, {DayDoctor, NightDoctor}, {SupervisorDoctor, DayDoctor, NightDoctor}}. However, the effective UAS, because of the temporal constraints, differ. Hence, in intervals (9am – 10am) and (12noon-9pm), UAS = {{DayDoctor}}; in interval (10am - 12noon), UAS = {{DayDoctor}, {SupervisorDoctor}, {SupervisorDoctor, DayDoctor}}; in interval (7pm–9am), UAS = {{NightDoctor}, {SupervisorDoctor},{SupervisorDoctor, NightDoctor}}; in interval (9pm–7am), UAS= {{NightDoctor }}.

Next, we present a formal basis for characterizing the UAS for a *hybrid* linear path. We first present the results for a *hybrid* linear path consisting of only two *monotype* linear components in the following Lemma and then use it to characterize arbitrary hybrid linear paths.

Lemma 4.1: *Let* $Lh = \{L_1, L_2\}$ *be a hybrid linear path* Lh *such that* $L_1 = (X_1, <f_1>)$ *and* $L_2 = (X_2, <f_2>)$, *where* $X = \{x_1, x_2, \ldots, x_n\} = X_1 \cup X_2$, *and* $<f_1> \neq <f_2>$. *Then for a user u assigned only to* S_{L1}, *we have*:

$$UAS(Lh, t) = \begin{cases} UAS(L_1,t), & \text{if } \geq^t \in \{<f_1>, <f_2>\} \\ UAS(L_{1U},t) \cup UAS(L_2,t) \cup (UAS(L_{1U},t) \otimes UAS(L_2,t)) & \text{if } (<f_1>,<f_2>)=(\succcurlyeq^t,\succsim^t) \\ UAS(L_1,t) \cup UAS(L_{2L},t) \cup (UAS(L_1,t) \otimes UAS(L_{2L},t)) & \text{if } (<f_1>,<f_2>)=(\succsim^t,\succcurlyeq^t) \end{cases}$$

where, L_{2L}= $\mathsf{sub}_L(L_2)$, L_{2U}= $\mathsf{sub}_U(L_2)$) and $(A \otimes B) = \{ \{x \cup y\} \mid x \in A \text{ and } y \in B\}$.

Note that, in the computation involving *UAS(Lh, t)*, the components on the right side are disjoint with respect to each other and hence $|UAS(Lh, t)|$ is simply the sum of the cardinalities of the components on the right side. Theorem 4.2 determines the UAS for an arbitrary hybrid linear path.

Theorem 4.2: *Let* $Lh = \{L_1, LH_2\}$ *be a hybrid linear path such that* $L_1 = (X_1, <f_1>)$, LH_2 *is a linear path over* X_2, *and* $X = X_1 \cup X_2$, *where* X_1 and X_2 *are role sets. Furthermore, let* $LH_2 = \{L_x, LH'\}$, *where* $L_x = (X_x, <f_x>)$ *over role set* X_x *such that* $<f_x> \neq <f_1>$ *and LH' is a linear path, possibly empty. Then, we have the following*:

1. if $<f_1> = \geq^t$ then $UAS(Lh, t) = UAS(L_1, t)$
2. if $<f_1> = \succcurlyeq^t$ then

$$UAS(Lh, t) = \begin{cases} UAS(L_1, t) & \text{if } (<f_x> = \geq) \\ UAS(L_{1U}, t) \cup UAS(LH_2, t) \cup (UAS(L_{1U}, t) \otimes UAS(LH_2, t)) & \text{if } (<f_x> = \succsim^t) \end{cases}$$

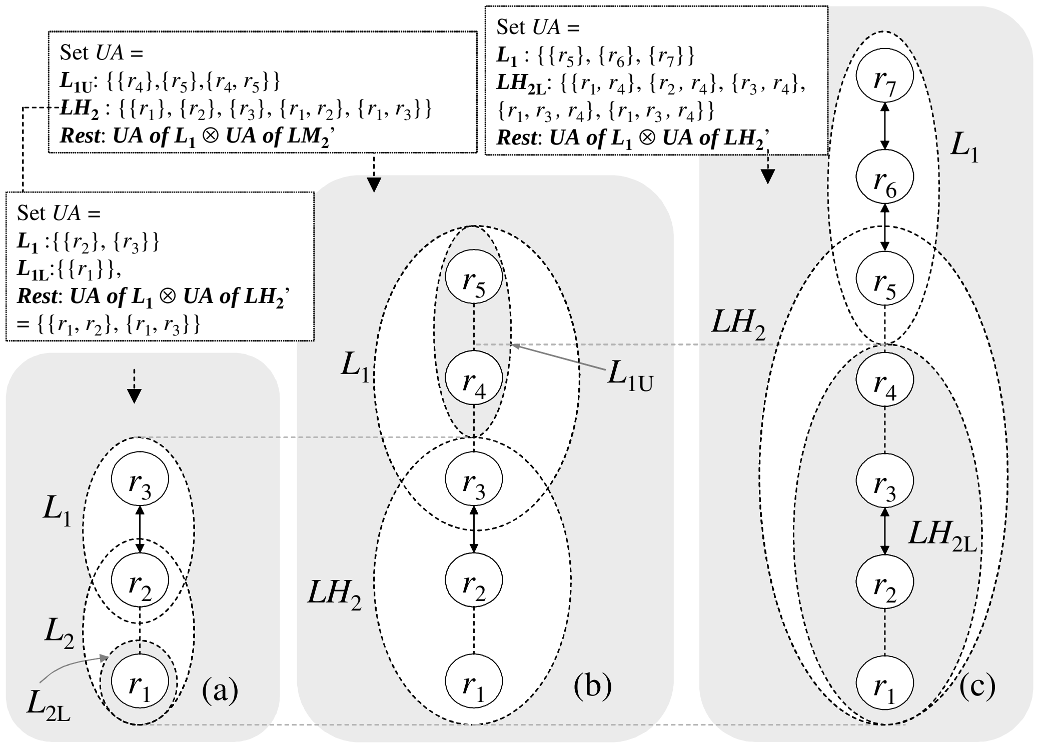
3. if $<f_1> = \succsim^t$ then

$$UAS(Lh, t) = \begin{cases} UAS(L_1, t) & \text{if } (<f_x> = \geq) \\ UAS(L_1, t) \cup UAS(LH_{2L}, t) \cup (UAS(L_1, t) \otimes UAS(LH_{2L}, t)) & \text{if } (<f_x> = \succsim^t) \end{cases}$$

The next example illustrates the use of the above theorem and refers to Fig. 4.6.

`Example 4.2.3`: Consider Fig. 4.6. We note that the hierarchy in (a) is part of the hierarchy in (b), which in turn is a part of the hierarchy in (c). We look at each case separately**.**

Case (a): Here $L_1 = r_3 \succsim^t r_2$, and $L_2 = r_2 \succcurlyeq^t r_1$. Hence, $(<f_1>,<f_2>) = (\succsim^t, \succcurlyeq^t)$ applies. Therefore, by Lemma 4.1, we have, $UAS(Lh, t) = UAS(L_1, t), \cup UAS(L_{2L}, t) \cup (UAS(L_1, t) \otimes UAS(L_{2L}, t)) = \{\{r_1\}\} \cup \{\{ r_2\}, \{r_3\}\} \cup (\{\{ r_2\}, \{r_3\}\} \otimes \{\{ r_1\}\}) = \{\{r_1\}, \{ r_2\}, \{ r_3\}, \{ r_1, r_2\}, \{ r_1, r_3\}\}$.



Set UA =
L_{1U}: {{r_4},{r_5},{r_4, r_5}}
LH_2 : {{r_1}, {r_2}, {r_3}, {r_1, r_2}, {r_1, r_3}}
Rest: UA of L_1 ⊗ UA of LM_2'
Set UA =
L_1 : {{r_5}, {r_6}, {r_7}}
LH_{2L}: {{r_1, r_4}, {r_2, r_4}, {r_3, r_4}, {r_1, r_3, r_4}, {r_1, r_3, r_4}}
Rest: UA of L_1 ⊗ UA of LH_2'
Set UA =
L_1 :{{r_2}, {r_3}}
L_{1L}:{{r_1}},
Rest: UA of L_1 ⊗ UA of LH_2'
= {{r_1, r_2}, {r_1, r_3}}
r_7
r_6
L_1
r_5
LH_2
L_1
r_5
r_4
L_{1U}
r_4
r_3
r_3
L_1
r_3
LH_{2L}
r_2
LH_2
r_2
r_2
L_2
L_{2L}
r_1
(a)
r_1
(b)
r_1
(c)

Fig. 4.6. Computing UAS of a hybrid linear hierarchy

Case(b): Here $L_1 = r_5 \succcurlyeq^t r_4 \succcurlyeq^t r_3$, and LM_2 is the hierarchy in (a). Now, we apply Theorem 4.1. As $<f_1> = \succcurlyeq^t$, case (2) of the theorem applies. Thus, $UAS(Lh, t) = UAS(L_{1U}, t), \cup UAS(LH_2, t) \cup (UAS(L_{1U}, t) \otimes UAS(LH_2, t)) = \{\{r_4\}, \{r_5\}, \{r_4, r_5\}\} \cup \{\{r_1\}, \{ r_2\}, \{ r_3\}, \{r_1, r_2\}, \{ r_1, r_3\}\} \cup (\{\{r_1\}, \{r_2\}, \{r_3\}, \{r_1, r_2\}, \{r_1, r_3\}\} \otimes \{\{r_4\}, \{r_5\}, \{r_4, r_5\}\}\}) = \{\{r_1\}, \{ r_2\}, \{ r_3\}, \{r_1, r_2\}, \{r_1, r_3\}, \{r_4\}, \{r_5\}, \{r_4, r_5\}, \{r_1, r_4\}, \{r_1, r_5\}, \{r_1, r_4, , r_5\}, \{r_2, r_4\}, \{r_2, r_5\}, \{r_2, r_4, , r_5\}, \{r_3, r_4\}, \{r_3, r_5\}, \{r_3, r_4, , r_5\}, \{r_1, r_2,$

r_4}, {r_1, r_2, r_5}, {r_1, r_2, r_4, r_5}, {r_1, r_3, r_4}, {r_1, r_3, r_4}, {r_1, r_3, r_4, r_5}} . Thus, the total number of elements is 23.

Case(c): Here $L_1 = r_7 \succsim^t r_6 \succsim^t r_5$, and LH_2 is the hierarchy in (a). Again, we apply Theorem 4.1. Computation can be carried out similarly using *UAS(Lh, t)* = *UAS(L*$_1$*, t)*, ∪ *UAS(L*$_{2L}$*, t)* ∪ (*UAS(L*$_1$*, t)* ⊗ *UAS(L*$_{2L}$*, t)*) as depicted in Fig. 4.2(c). Note that the computation of *UAS(Lh, t)* involves computing *UAS(L*$_{2L}$*, t)*, which can be done as in case(b).

A hybrid general hierarchy can have complex inheritance and activation semantics. We note that each hierarchical structure can be broken down, or *vertically partitioned*, into a list of linear paths. In the following, we consider a general hierarchy rooted at a role and represent it using a *vertically partitioned* set of linear components. A *vertical partition*, defined next, represents a general hierarchy as an ordered list of its component linear paths.

Definition 4.2.3 *(Hybrid Hierarchy rooted at a role as a set of linear paths): Let* $H = (X, [f])$ *be a* hierarchy *over role set X rooted at role* S_H *with relation set* $[f] \subseteq \{\geq^t, \succcurlyeq^t, \succsim^t\}$. *We say that H is representable by* an ordered set of linear paths (*hybrid* or *monotype*), *that is*, $H = \{LH_1, LH_2, \ldots, LH_m\}$, *if, for* $i, j \in \{1, 2, \ldots, m\}$, $i \neq j$ *and* LH_i *is a linear path over* X_i, *the following conditions hold*

1. $S_{LHi} = S_H$; $J_{LHi} \subseteq J_H$,
2. $X_i \neq X_j$; $X = \bigcup_{i=1}^{m} X_i$
3. *for all* $J \in J_H$, *there exists no linear path* $LH = (\{S_H, x_{\pi 1}, x_{\pi 2}, \ldots, x_{\pi i}, J\}, [f'])$, *where* $[f'] \subseteq [f]$ *and* $\{x_{\pi 1}, x_{\pi 2}, \ldots, x_{\pi i}\} \subseteq X/\{S_H, J\}$, *such that* $LH \notin \{LH_1, LH_2, \ldots, LH_m\}$.

We say that $\{LH_1, LH_2, \ldots, LH_m\}$ *is the* complete vertical partition *of H. H can also be written as* $\{LH_1, H'\}$, $\{H'', LH_m\}$, $\{H_x, H_y\}$, *etc., each of which is (simply) a vertical partition of H.*

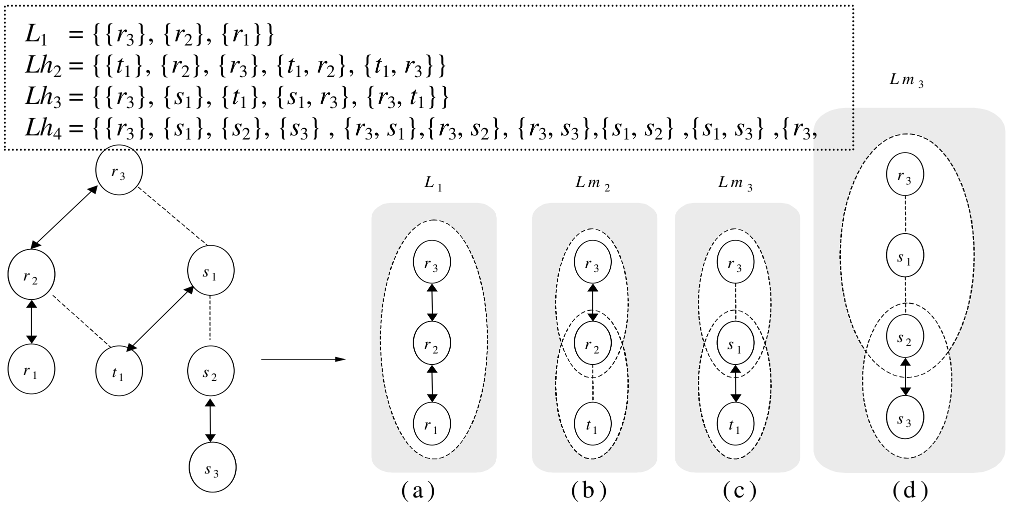
Based on the notion of vertical partitioning of a general hybrid hierarchy, the following theorem shows how we can formally determine UAS of a general hybrid hierarchy that is not a simple linear path.

Theorem 4.3: *Let* $H = (X, [f]) = \{LH_1, H_1\}$ *be a hierarchy such that the following condition holds*: $\exists\ x, y, z \in X,\ (x{<}f{>}y) \wedge (x{<}f{>}z)$. *Then,* $UAS(H, t) = I/C$, *where*

- $I = (UAS(LH_1, t) \cup UAS(H_1, t) \cup (UAS(LH_1, t)/B \otimes UAS(H_1, t)/B))$,
- $B = (UAS(LH_1, t) \sqcap UAS(H_1, t)) = \{X, Y \mid X \in UAS(LH_1, t)\ , Y \in UAS(H_1, t)$ and $X \cap Y \neq \emptyset\}$, and
- $C = \{Z \mid Z \in I,\ such\ that\ \exists\ x, y\ \in Z, x \succsim^t y\}$.

The theorem determines the UAS of a general hierarchy that has at least one role having multiple juniors, hence making it different from the linear paths. The computation is based on the partition of the hierarchy into two components, in which one is a linear component and the other is the remaining part of the hierarchy. This allows us to compute the UAS recursively once we have the linear components. The next example illustrates the working of Theorem 4.3.

Example 4.2.3: Consider the hierarchy in Fig. 4.7. The linear components of the hierarchy are shown in (a)-(d). Each component's *UAS* computed using Theorem 4.3 is shown in Fig. 4.7. Now, we apply Theorem 4.3 to generate the *UAS* of the overall hierarchy. We will write H_{12} to mean the hierarchy formed by components L_1 and Lh_2, H_{13} to mean the hierarchy formed by components L_1, Lh_2 and Lh_3, and H_{14} to mean the overall hierarchy.



$L_1 = \{\{r_3\}, \{r_2\}, \{r_1\}\}$
$Lh_2 = \{\{t_1\}, \{r_2\}, \{r_3\}, \{t_1, r_2\}, \{t_1, r_3\}\}$
$Lh_3 = \{\{r_3\}, \{s_1\}, \{t_1\}, \{s_1, r_3\}, \{r_3, t_1\}\}$
$Lh_4 = \{\{r_3\}, \{s_1\}, \{s_2\}, \{s_3\}\ , \{r_3, s_1\},\{r_3, s_2\}, \{r_3, s_3\},\{s_1, s_2\}\ ,\{s_1, s_3\}\ ,\{r_3,$

Fig. 4.7. Computing UAS of a general hierarchy

***Step* 1**: Consider components L_1 and Lh_2. Here, $B = UAS(L_1, t) \sqcap UAS(Lh_2, t) = \{\{r_3\}, \{r_2\}, \{t_1, r_2\}, \{t_1, r_3\}\}$. Therefore, $(UAS(L_1, t)/B) \otimes (UAS(Lh_2, t)/B) = \{\{r_1\}\} \otimes$

$\{\{t_1\}\} = \{\{r_1, t_1\}\}$. Note that C is empty. Thus, $UAS(H_{12}, t) = I/C = I = \{\{r_3\}, \{r_2\}, \{r_1\}, \{t_1\}, \{r_1, t_1\}, \{r_3, t_1\}, \{t_1, r_2\}\}$.

***Step* 2**: Consider component H_{12} (result from *Step* 1) and Lh_3. Here, $B = UAS(H_{12}, t) \sqcap UAS(Lh_3, t) = \{\{r_3\}, \{t_1\}, \{r_1, t_1\}, \{r_3, t_1\}, \{t_1, r_2\}\}$.Therefore, $(UAS(H_{12}, t) - B) \otimes (UAS(Lh_3, t) - B) = \{\{r_2\}, \{r_1\}\} \otimes \{\{s_1\}\} = \{\{r_2, s_1\}, \{r_1, s_1\}\}$. Hence $I = \{\{r_3\}, \{r_2\}, \{r_1\}, \{t_1\}, \{r_1, t_1\}, \{r_3, t_1\}, \{t_1, r_2\}, \{s_1\}, \{r_2, s_1\}, \{r_1, s_1\}\}$. Note that Lh_3 introduces the *IA*-relation between r_3 and t_1, but we have $\{r_3, t_1\} \in I$. Thus we need to remove $\{r_3, t_1\}$. Therefore, $UAS(H_{13}, t) = I/C = \{\{r_3\}, \{r_2\}, \{r_1\}, \{t_1\}, \{r_1, t_1\}, \{t_1, r_2\}, \{s_1\}, \{r_2, s_1\}, \{r_1, s_1\}\}$.

***Step* 3**: Consider component H_{13} (result from *Step* 2) and Lm_4. Here, $B = UAS(H_{13}, t) \sqcap UAS(Lm_4, t) = \{\{r_3\}, \{s_1\}, \{r_2, s_1\}, \{r_3, s_2\}, \{s_1, s_2\}\}$. Therefore, $(UAS(H_{13} - B) \otimes (UAS(Lm_4, t) - B) = \{\{r_2\}, \{r_1\}, \{t_1\}, \{r_1, t_1\}, \{t_1, r_2\}\} \otimes \{\{s_2\}\} = \{\{r_2, s_2\} \{r_1, s_2\}, \{t_1, s_2\}, \{r_1, t_1, s_2\}, \{t_1, r_2, s_2\}\}$. We also note that C is empty. Hence, $UAS(H_{13}, t) = I/C = I = \{\{r_3\}, \{r_2\}, \{r_1\}, \{t_1\}, \{r_1, t_1\}, \{t_1, r_2\}, \{s_1\}, \{r_2, s_1\}, \{r_1, s_1\}, \{s_2\}, \{r_3, s_2\}, \{s_1, s_2\}, \{r_2, s_2\} \{r_1, s_2\}, \{t_1, s_2\}, \{r_1, t_1, s_2\}, \{t_1, r_2, s_2\}\}$.

Table 4.2

Supporting functions for algorithm in Fig.4.8

`ComputeI`(*UAStail*, *UAShead*)	Compute *I* according to theorem 4.3
`ComputeC`(*I*, *H*	Compute *C* according to theorem 4.3
`isLinear`(*H*)	if (there is a *x* in `Roles`(*H*) s. t. *x* is senior of two other roles in `Roles`(*H*)) then returns FALSE else returns TRUE
`isEmpty`(*H*)	if (`Roles`(*H*) is empty) then returns TRUE else returns FALSE
`isMonotypeLinear`(*LH*)	if (there are two relations in *H*) then returns FALSE else returns TRUE
`Type`(*L*)	returns the type of linear hierarchy *L* (returns *I*, *A* or *IA*)
`hHead`(*Lh*)	if $\{L_1, L_2, \ldots, L_n\}$ is the complete horizontal partition of *Lh* then it returns $\{L_2, \ldots, L_n\}$
`vHead`(*H*)	if $\{LH_1, LH_2, \ldots, LH_n\}$ is the complete vertical partition of *H* then it returns LH_1
`vTail`(*H*)	if $\{L_1, L_2, \ldots, L_n\}$ is the complete horizontal partition of *Lh* then it returns $\{L_2, \ldots, L_n\}$}(returns ∅ if $n = 1$).

```
Algorithm ComputeUASHierarchy($H$)
1. if (isMonotype($H$)) then return ComputeUASMonotype($H$)
2. else if (isLinear($H$)) then return ComputeUASLinear($H$)
3. $UAStail$ = ComputeUASHierarchy(vTail($H$))
4. $UAShead$ = ComputeUASLinear(vHead($H$))
5. $I$ = ComputeI($UAStail$, $UAShead$); $C$ = ComputeC($I$, $H$)
6. return ($I - C$)                                  // Refer to Theorem 4.3
End ComputeUASHierarchy

Algorithm ComputeUASLinear($LH$)
1. if (isEmpty($LH$)) then return $\varnothing$;
2. if (isMonotype(hTail($LH$))) then
       return Compute2MonotypeLinear(hHead ($LH$), hTail($LH$))
3. $f_1$ = Type(hHead($LH$))
4. $f_x$ = Type(hHead(hTail($LH$)))
5. $UAShead$ = ComputeUASLinear(hHead($LH$))                  // Refer to Theorem 4.2
6. Case: ($f_1$ = 'I') or ($f_1$ = 'A', $f_x$ = 'I') or ($f_1$ = 'IA', $f_x$ = 'I')
       return ComputeUASLinear(hHead($LH$))
7. Case: ($f_1$ = 'A', $f_x$ = 'IA')
       return ( ComputeUASMonotype(sub$_U$(hHead($LH$)))$\cup$ComputeUASLinear(hTail($LH$))
               (ComputeUASMonotype(sub$_U$(hHead($LH$)))$\otimes$ComputeUASLinear(hTail($LH$))) )
8. Case: ($f_1$ = 'IA', $f_x$ = 'A')
       return ( ComputeUASMonotype(hHead($LH$))$\cup$ComputeUASLinear(sub$_L$(hTail($LH$)))
              (ComputeUASMonotype((hHead($LH$)))$\otimes$ComputeUASLinear(sub$_L$(hTail($LH$))))
End ComputeUASHierarchy

Algorithm Compute2MonotypeLinear($L_1$, $L_2$)
1. $f_1$ = Type($L_1$); $f_2$ = Type($L_2$)
2. Case: ($f_1$ = 'I') or ($f_2$ = 'I')                       // Refer to Lemma 4.1
       return ComputeUASMonotype($L_1$)
3. Case: ($f_1$ = 'A', $f_2$ = 'IA')
       return ( ComputeUASMonotype(sub$_U$($L_1$)) $\cup$ ComputeUASMonotypeLinear($L_2$)
               (ComputeUASMonotype(sub$_U$($L_1$)) $\otimes$ ComputeUASMonotypeLinear($L_2$) )
4. Case: ($f_1$ = 'IA', $f_2$ = 'A')
       return ( ComputeUASMonotype($L_1$) $\cup$ ComputeUASMonotypeLinear(sub$_L$($L_2$))
              (ComputeUASMonotype($L_1$) $\otimes$ ComputeUASMonotypeLinear(sub$_L$($L_2$)) )
End Compute2MonotypeLinear

Algorithm ComputeUASMonotype($H$)
1.     $f$ = Type($H$)
2.     Case: ($f_1$ = 'I') return $\{\{S_H\}\}$                 // Refer to Theorem 4.3
3.     Case: ($f_1$ = 'A') return ($2^{Roles(L)}$ /$\varnothing$)
4.     Case: ($f_1$ = 'IA') return $\{\{x_1\}, \{x_2\}, \ldots, \{x_n\}\}$ where, Roles($L$) = $\{x_1, x_2, \ldots, x_n\}$
End ComputeUASMonotypeLinear
```

Fig. 4.8 Algorithm for computing the uniquely activable set.

Based on the theorems, we derive the algorithms depicted in Fig. 4.8 for generating the UAS of a hierarchy rooted at a role. `ComputeUASHierarchy` essentially implements Theorem 4.3 and constructs UAS recursively. For this, it uses algorithm `ComputeUASLinear` which recursively constructs the UAS of linear paths. Algorithm `ComputeUASLinear`, on the other hand, implements Theorem 4.2 and makes use of algorithm `Compute2MonotypLinear` that implements Lemma 4.1. Theorem 4.1 is implemented by algorithm `ComputeUASMonotypeLinear`.

4.2.2 Acquisition Equivalent Hierarchies

In earlier sections, we have characterized the UAS of a general hierarchy. An important issue is whether or not we can use a hierarchy of one type to achieve what a hierarchy of another type allows. To address such an issue, we need an appropriate notion of equivalence between different hierarchies, as they are structurally and semantically different. We note that central to the use of hierarchies in a GTRBAC system is the efficient management of permission acquisition by users assigned to various roles in the hierarchy. Thus, a notion of equivalence between two types of hierarchy can be established if we show that the maximum set of permissions that can be acquired by a user in the two hierarchies is the same. The significance of using the maximum set of permissions is that within the equivalent hierarchies, the user can carry the same accesses, even though, in each hierarchy, the user may have to activate a different set of roles. Here, we introduce the notion of *acquisition-equivalence* between two hierarchies. We say that two hierarchies are *acquisition-equivalent* if they allow the same maximum set of permissions to be acquired by a user assigned to the senior-most role. We use $P_{max}(H, t)$ to refer to the maximum set of permissions that a user can acquire through the senior-most role of the hierarchy H in a session at time instant t. The notion of *acquisition-equivalence* is formally defined as follows:

Definition 4.2.4 *(Acquisition equivalence or AC-equivalence of two hierarchies): Let H_1 and H_2 be two hierarchies over role set* `Roles`*. Then we say that H_1 and H_2 are acquisition-equivalent* or *AC-equivalent* (*written as $H_1 =_{AC} H_2$*), *if $P_{max}(H_1, t)=P_{max}(H_2, t)$.*

The following theorem provides the formal characteristics of an *AC-acquisition - equivalent* set of hierarchies

Theorem 4.4 (***AC*-equivalent hierarchies**): *Let $H_1 = (X, [f_1]) = (LH_1, LH_2, \ldots , LH_n)$ and $H_2 = (X, \langle f_2 \rangle)$ be two hierarchies over role set X. If, for roles $x, y \in X$ and a* *relation* $<f> \in [f_1]$*, the condition* ($x<f>y \in H_1$ iff $x<f_2>y \in H_2$) *holds, then* $H_1 =_{AC} H_2$ (*i.e.* H_1 *and* H_2 *are* AC-equivalent) *provided the following holds*

- *for all* $i \in \{1, 2, \ldots, n\}$, *and hierarchies* LH', LH_{mid}, LH'', *each possibly empty, the following is satisfied*
  $\neg\exists\, L_x, L_y$ *such that* $LH_i = (LH', L_x, LH_{mid}, L_y, LH'')$, *where* $<f_x> = \geq^t$ *and* $<f_y> = \succcurlyeq^t$

The condition $LH_i = (LH', L_x, LH_{mid}, L_y, L'')$ implies that in the linear component LH_i, there is an *I*-relation that precedes (not necessarily immediately, as LH_{mid} may not be empty) an *A*-relation. All hierarchies that do not have such a component are *AC*-equivalent to a *monotype* hierarchy. As a consequence, first, the theorem implies that any two *monotype* hierarchies are *AC*-equivalent, as the condition $LH_i = (LH', L_x, LH_{mid}, L_y, L'')$ cannot occur in a *monotype* hierarchy. Furthermore, the theorem says that every hierarchy that does not contain such a linear component is *AC*-equivalent to a *monotype* hierarchy and hence to each other. This is because if an *I*-relation precedes an *A*-relation in the hierarchy, then the permissions associated with the roles below the *A*-hierarchy cannot be acquired by any user assigned to the senior-most role, thus, reducing the permissions that can be acquired. The significance of this result is that, in systems where the principle of least privilege is not of much concern, any monotype hierarchy can be used instead of a more complex hybrid hierarchy.

4.3. Derived Hierarchical Relations

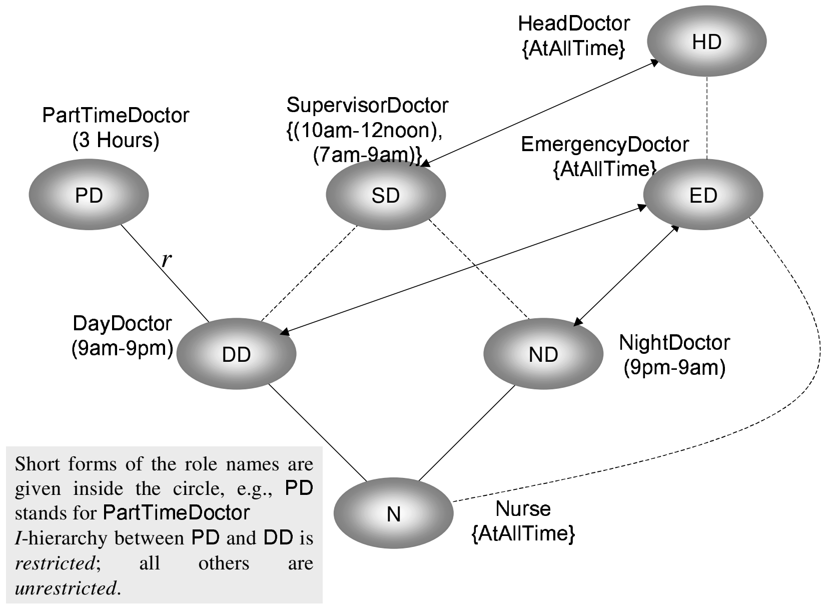
In a hierarchy where all three types of hierarchies can co-exist, a hierarchical relation between a pair of roles that are not directly related may be derived. While most derived relations fall into the three hierarchy types discussed earlier, we introduce a special derived type called a *conditioned* derived relation, written as ($x[A](B)<f>y$), and defined as follows:

Definition 4.3.1 (*Conditioned Derived relation*): *Let H be a role hierarchy,* $x, y \in$ Roles(H) and A, $B \subseteq$ Roles(H). *Then* $x[A](B)<f>y$ *is called a* Conditioned Derived Relation (*also read as* "the derived relation $x<f>y$ is *conditioned* on roles in A and B"), *if, for all* $a \in A$ and $b \in B$, *the following holds:*

$$\textit{for all } a \in A,\ b \in B,\ (x \succcurlyeq^t a) \wedge (a<f>y) \wedge ((x=b) \vee (x \succcurlyeq^t b)) \wedge (b \succcurlyeq^t y),$$

where $<f> \in \{\geq^t, \succsim^t\}$, $|A| > 0$, $|B| \geq 0$, *and* ($b \succcurlyeq^t y$) *is a direct relation.*

Here, the condition indicates that x is related to each $a \in A$, directly or through a derived relation, by an *A*-relation, whereas each a is related to y by the $<f>$ relation. This implies that a permission that can be acquired through role y can be acquired by a user assigned to role x, without activating y, if he activates any of the roles in A. We note that B may be empty, in which case, the *conditioned* derived relation is simply written as $x[A]\langle f\rangle y$. If B is not empty then for each $b \in B$, there is an A-path from x to y through b. If $C = A \cap B$ then, for all $c \in C$, both $(c \geq^t y)$ and $(c \succcurlyeq^t y)$ hold; i.e., for all $c \in C$, we have $(c \succsim^t y)$. It is possible that $x[A](\{x\})\langle f\rangle y$, which means $x[A]\langle f\rangle y$, and $(x \succcurlyeq^t y)$ is a direct relation. As we shall see, it is not necessary that the hierarchical path from x to each $a \in A$ contain all A-relations; it is only required that a user who is assigned to or can activate x can also activate a. This, however, implies that the hierarchical path from x to each a does not contain any I-relation as it prohibits activation of a junior role by users assigned to the senior. Furthermore, we note that in $x[A](B)\langle f\rangle y$, $\langle f\rangle$ is either $\geq^t$ or $\succsim^t$.



Short forms of the role names are given inside the circle, e.g., PD stands for PartTimeDoctor
I-hierarchy between PD and DD is *restricted*; all others are *unrestricted*.

Fig. 4.9 A hybrid hierarchy for a medical department

Example 4.3.1: Consider the hierarchy of Fig. 4.9, representing a medical department. PD can be enabled for three hours only. Since it has *restricted*-inheritance over DD, a user assigned to PD can acquire DD's permissions only in daytime. SD's relation to DD and ND are as discussed in Fig. 4.3(d). N can be *I*-inherited by DD and ND. ED is enabled at all times. The *A*-relation between ED and N allows a user assigned to ED to explicitly act as a nurse besides inheriting N's permissions through DD or ND. Assume that the HD role represents the head doctor of the medical department, which is enabled at all times. HD can act as the supervisor role of doctors, because of the unrestricted relations through SD. Two *conditioned derived relations* are as follows.

1. SD[DD, ND]$\geq^t$ N: This is because users assigned to SD can acquire permissions of N only by activating SD or ND.
2. HD[DD, ND, ED](ED) $\geq^t$ N: This is because users assigned to HD can acquire permissions of N by activating SD, ND or ED, without activating N. Furthermore, the users can directly activate N (because of the *A*-path through ED).

4.3.1 The Inference Rules for Hybrid Hierarchies

We now introduce the inference rules that allow the derivation of indirect relations between roles from explicitly specified relations between roles. Such derived relations can be used to determine the permissions that can be acquired through the activation of a role in a hierarchy by a user. We use ISen(y) = $\{x|\ \ x \succcurlyeq^t y$ is a direct relation$\}$ to denote the set of immediate seniors of role y through *A*-relation. The inference rules are as follows:

Inference Rules: *Let H be a role hierarchy, x, y, $z \in$ Roles(H), and A, A_1, A_2, B_1, B_2 $\subseteq$ Roles(H). Then the inference rules for deriving indirect relations are as shown in* Table 4.3.

***R*1** is a trivial case of transitivity using a single hierarchy type. Thus, if $<f>$ is $\succcurlyeq^t$, then from the two relations $x \succcurlyeq^t y$ and $y \succcurlyeq^t z$, relation $x \succcurlyeq^t z$ is inferred. ***R*2** applies to all the pairs of relations $x<f>y$ and $y<f>z$ that may not be direct relations. This can result in a *conditioned* derived relation of the form $x[A]<f>z$. ***R*3** deals with each of the cases in which an *unconditioned* relation follows a *conditioned* derived relation. In a hierarchy, there may be more than one relation between a pair of roles. Such a situation arises when there are multiple hierarchical paths between a given pair of roles. ***R*4** deals with such cases. Rule ***R*4.1** is a trivial case in which both the hierarchical paths are the same *unconditioned* relation (derived or direct). Rule ***R*4.2** captures all the possible combinations of two different hierarchical *unconditioned* relations between the same pair. Similarly, rule ***R*4.3** captures all the possible combinations of two different hierarchical relations between the same pair in which one is an *unconditioned* derived relation. Lastly, ***R*4.4** are for capturing all the possible combinations of two different hierarchical *conditioned* derived relations between a pair of roles. Example 4.3.2 illustrates the application of rules to determine derived relations for the hierarchy in Fig. 4.9.

Table 4.3
The inference rules for derived hierarchical relations

Rule	**Case**	**Inference Rule**	
***R*1**	(***Monotype hierarchy***)		
		$(x\langle f\rangle y) \wedge (y\langle f\rangle z) \rightarrow (x\langle f\rangle z)$ *for all* $\langle f\rangle \in \{\geq^t, \succcurlyeq^t, \succsim^t\}$	
***R*2**	(***Hybrid hierarchy with unconditioned relations***)		
	1	$(x\langle f_1\rangle y) \wedge (y\langle f_2\rangle z) \rightarrow (x\geq^t z)$ *for all* $\langle f_1\rangle, \langle f_2\rangle \in \{\geq^t, \succsim^t\}$, *such that* $\langle f_1\rangle \neq \langle f_2\rangle$	
	2	$(x\succsim^t y) \wedge (y\succcurlyeq^t z) \rightarrow (x\succcurlyeq^t z)$;	
	3	$(x\succcurlyeq^t y) \wedge (y\langle f\rangle z) \rightarrow (x[\{y\}]\langle f\rangle z)$ *for* $\langle f\rangle \in \{\geq^t, \succsim^t\}$	
***R*3**	(***Hybrid hierarchy with one unconditioned derived relation***)		
	1	*for* $\langle f\rangle \in \{\geq^t, \succcurlyeq^t, \succsim^t\}$ *such that* $\langle f_1\rangle \neq \langle f_2\rangle$;	
		a.	$(x\,[A](B)\geq^t y) \wedge (y\geq^t z) \rightarrow (x\,[A\cup C]\geq^t z)$, where $C=\{y\}$ *if* $\lvert B\rvert > 0$, else $C=\varnothing$;
		b.	$(x\,[A](B)\geq^t y) \wedge (y\succsim^t z) \rightarrow (x\,[A\cup C]\,(C)\geq^t z)$ where $C=\{y\}$ *if* $\lvert B\rvert > 0$, else $C=\varnothing$
	2	*for* $\langle f\rangle \in \{\geq^t, \succcurlyeq^t, \succsim^t\}$ *such that* $\langle f_1\rangle \neq \langle f_2\rangle$	
		a.	$(x\,[A](B)\succsim^t y) \wedge (y\geq^t z) \rightarrow (x\,[A\cup C]\geq^t z)$ where $C=\{y\}$ *if* $\lvert B\rvert > 0$, else $C=\varnothing$
		b.	$(x\,[A](B)\succsim^t y) \wedge (y\succsim^t z) \rightarrow (x\,[A]\succsim^t z)$;
	3	*for* $\langle f\rangle \in \{\geq^t, \succsim^t\}$ $(x\,[A]\,(B)\langle f\rangle y) \wedge (y\succcurlyeq^t z) \rightarrow (x\succcurlyeq^t z)$	
***R*4**	(***Hierarchy with multiple paths between two roles; subscripts indicate the path number***)		
	1	$(x\langle f\rangle y)_1 \wedge (x\langle f\rangle y)_2 \rightarrow (x\langle f\rangle y)$ *for all* $\langle f\rangle \in \{\geq^t, \succcurlyeq^t, \succsim^t\}$	
	2	$(x\langle f_1\rangle y)_1 \wedge (x\langle f_2\rangle y)_2 \rightarrow (x \succsim^t y)$ *for all* $\langle f_1\rangle, \langle f_2\rangle \in \{\geq^t, \succcurlyeq^t, \succsim^t\}$ *such that* $\langle f_1\rangle \neq \langle f_2\rangle$	
	3	*for all* $\langle f\rangle, \langle f_1\rangle, \langle f_2\rangle \in \{\geq^t, \succsim^t\}$ *such that* $\langle f_1\rangle \neq \langle f_2\rangle$	
		a	$(x\,[A]\,(B)\langle f\rangle y)_1 \wedge (x\langle f\rangle y)_2 \rightarrow (x\langle f\rangle\, y)$
		b.	$(x\,[A]\,(B)\langle f\rangle y)_1 \wedge (x\succcurlyeq^t y)_2 \rightarrow x\,[A]\,(\text{ISen}(y))\langle f\rangle y)$
		c.	$(x\,[A]\,(B)\langle f_1\rangle y)_1 \wedge (x\langle f_2\rangle y)_2 \rightarrow (x\succsim^t y)$
	4	*for all* $\langle f\rangle, \langle f_1\rangle, \langle f_2\rangle \in \{\geq^t, \succsim^t\}$ *such that* $\langle f_1\rangle \neq \langle f_2\rangle$	
		a.	$(x\,[A_1]\,(B_1)\langle f\rangle y)_1 \wedge (x\,[A_2]\,(B_2)\langle f\rangle y)_2 \rightarrow (x[A_1\cup A_2](B_1\cup B_2)\langle f\rangle\, y)$
		b.	$(x\,[A_1]\,(B_1)\langle f_1\rangle y)_1 \wedge (x\,[A_2]\,(B_2)\langle f_2\rangle y)_2 \rightarrow (x[A_1\cup A_2](A\cup B_1\cup B_2)\geq^t y)$ *s.t.* $A= A_1$ if $\langle f_1\rangle=\succsim^t$ else $A=A_2$

Table 4.4

Application of inference rules over the hierarchy of Fig. 4.9

Rule applied		***Derive relations***
R1		(PD $\geq^t$ N), (HD $\succcurlyeq t$ N)
R2	1	(ED $\succsim t$ ND) $\wedge$ (ND $\geq^t$ N) implies(ED $\geq^t$ N)
	2	(HD $\succsim t$ SD) $\wedge$ (SD$\succcurlyeq t$ DD) implies (HD $\succcurlyeq t$ DD)
	3	(SD $\succcurlyeq t$ DD) $\wedge$ (DD $\geq^t$ N) implies (SD[{DD}] $\geq^t$ N)
		(HD $\succcurlyeq t$ ED) $\wedge$ (ED $\succsim t$ ND) implies (HD[{ED}] $\succsim t$ ND)
		(HD $\succcurlyeq t$ DD) $\wedge$ (DD $\succsim t$ N) implies (HD[{DD}] $\succsim t$ ND)
R3	2a	(HD[{ED}} $\succsim t$ DD) $\wedge$ (DD $\geq^t$ N) implies (HD[{ED}] $\geq^t$ N)
		(HD[{ED}} $\succsim t$ ND) $\wedge$ (ND $\geq^t$ N) implies (HD[{ED}] $\geq^t$ N)
R4	1	(ED $\geq^t$ N) (one through DD, another through ND)
	2	(ED $\geq^t$ N) $\wedge$ (ED $\succcurlyeq t$ N) implies (ED $\succsim t$ N)
	3b	(HD[{ED}] $\geq^t$ N) $\wedge$ (HD $\succcurlyeq t$ N) implies (HD[{ED}]({ED}) $\geq^t$ N) (which is same as (HD[{ED}] $\succsim t$ N)
	4a	(HD[{DD}] $\geq^t$ N) $\wedge$ (HD[{ND}] $\succsim t$ N) implies (HD[{DD, ND}] $\geq^t$ N)
	4b	(HD[{DD, ND}} $\geq^t$ N) $\wedge$ (HD[{ED}] $\succsim t$ N) implies (HD[{DD, ND, ED}] $\geq^t$ N)

`Example 4.3.2:` Applications of the rules over the hierarchy of Fig. 4.9 is illustrated in the Table 4.4.

4.3.2 Soundness and Completeness of the Inference Rules

In this section, we show that the set of inference rules introduced above is *sound* and *complete*, using the notion of *authorization consistent hierarchies*, which is defined below. In the definition, we use predicate `can_activate` (*u, r, t, H*) to mean that *u* can activate role *r* using *role-activation* semantics in role hierarchy *H* at time *t*. Similarly, we use the predicate `can_be_acquired` (*p, r, t, H*) to mean that permission *p* can be acquired through role *r* at time *t* using *permission-inheritance* semantics in hierarchy *H*. Let UAH(*H*) and PAH(*H*) be sets of all the user-role and role-permission assignments related to roles in Roles(*H*).

Definition 4.3.1 (*Authorization consistent hierarchies*): *Let H_1 and H_2 be two hierarchies such that* Roles(H_1) = Roles(H_2), UAH(H_1) = UAH(H_2) *and* PAH(H_1) =

PAH(H_2). *Then, we say that* H_1 *and* H_2 *are* authorization consistent (*written as* $H_1 \approx H_2$) *if for all* $r \in$ Roles(H_1), *the following conditions hold*:

1. $\forall u \in$ `Users`, ***can_activate*** (u, r, t, H_1) iff ***can_activate*** (u, r, t, H_2),
2. $\forall p \in$ `Permissions`, ***can_be_acquired*** (p, r, t, H_1) iff ***can_be_acquired*** (p, r, t, H_2).

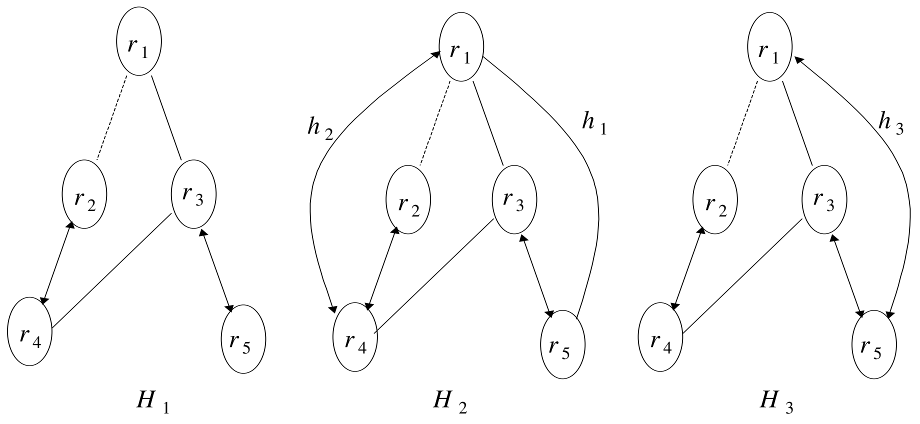


Fig. 4.10. Example of authorization consistent hierarchies; $H_1 \approx H_2$, $H_1 \not\approx H_3$ and $H_2 \not\approx H_3$

Here, we note that the two hierarchies considered have the same role set, user-role assignments and role-permission assignments. Condition (1) implies that if a user u can activate a role r in Roles(H_1) under hierarchy H_1, then s/he can activate it even if H_1 is replaced by H_2 (and vice versa). Similarly, the second condition says that the set of permissions that can be acquired through a role under H_1 is also the same set of permissions that can be acquired through that role in H_2 for any given user. This signifies that if two hierarchies are *authorization consistent* then a user assigned to a role can activate exactly the same set of roles and acquire the same set of permissions under the two hierarchies. This means the permission-inheritance and role-activation semantics in the two hierarchies are the same even if the sets of hierarchical relations in the two hierarchies are different. Fig. 4.10 depicts an example of the notion of *authorization consistency*. Here, the hierarchy relation h_1 in H_2 can be inferred from the hierarchical relations ($r_1 \geq^t r_3$) and ($r_3 \succsim^t r_5$), whereas, h_2 can be inferred from the two hierarchical paths from role r_1 and r_4. Hence, H_1 adds no new access capability compared to H_1. However, h_3 in H_3 is not inferred from the hierarchical relations ($r_1 \geq^t r_3$) and ($r_3 \succsim^t r_5$). In H_3, a user assigned to r_3 can activate r_5 also, which is not possible in H_1 or H_2. Hence, $H_1 \not\approx H_3$, and

$H_2 \not\approx H_3$. We use this notion of *authorization consistency* between two hierarchies to show that the set of rules presented above is *sound*; i.e., each new derived relation that can be deduced from a given set of hierarchical relations using the rules produces the same inheritance and activation semantics that is already present in the original hierarchy. Within a hierarchy H, we use h_{xz} to represent (x<f>z) for <f>$\in \{\geq^t, \succcurlyeq^t, \succsim^t\}$ or $x[A](B)$<f>z for <f>$\in \{\geq^t, \succsim^t\}$, where $x, z \in$ Roles(H) and A, $B \subseteq$ Roles(H). The following theorem formally states this result.

Theorem 4.5 (Soundness of rules *R*1-*R*4): *Given a role hierarchy H, if a new hierarchical relation h_{xz} is derived from hierarchical relations in H as per rules* ***R*1-*R*4,** *and H' = H $\cup$ {h_{xz}}, then H and H'* are authorization consistent, i.e. *H $\approx$ H'.*

The theorem implies that the new relations derived using the rules do not allow a user to inherit more (or less) permissions than was allowed to him before the derived relation is added. Similarly, the new derived relation does not allow a user to be able to activate more (or less) number of roles than was allowed before the derived relation is introduced. Next, we present the *completeness* theorem for the rules ***R*1-*R*4**. We write H[***R*1-*R*4**] $\models h_{x,z}$ to indicate that the relations in H can logically derive relation $h_{x,z}$ using rules ***R*1-*R*4**.

Theorem 4.6 (Completeness of rules *R*1-*R*4): *Given a role hierarchy H, rules* ***R*1-*R*4** *are complete; That is, if* $\neg$ H[***R*1-*R*4**] $\models h_{x,z}$, *for any pair of roles x, z* $\in$ Roles(H), *then H* $\not\approx$ *H* $\cup$ {$h_{x,z}$}, *i.e., the hierarchies H* and *H' = H* $\cup$ {$h_{x,z}$} *are not* authorization consistent.

The theorem indicates that if a relation, say <f>, between any two roles, say x and z, of Roles(H) cannot be derived from the hierarchical relations in H, then any role hierarchy containing such a relation is not *authorization consistent* with H. In other words, we can take every pair of roles (x, z) of Roles(H) and every possible hierarchical relation between them, including *conditioned* derived relations, and extend H by adding it to get H'. If we get $H \approx H'$, the theorem implies that the rules ***R*1-*R*4** are able to derive it. Hence, this shows that the rules are *complete*. Using the transitivity of the hierarchical relations and considering all the cases of the rules, we can easily construct the proofs. The proofs for both the theorems are provided in appendix B.

4.4. Hierarchy Transformations

In an organization, roles evolve with time, affecting the existing role hierarchies. New roles may need to be added and old ones deleted or modified. Permission sets of existing roles or their temporal properties may need to be altered. Making such changes may require restructuring the hierarchies to avoid undesirable situations. In this section, we analyze transformations of a role hierarchy when a role is added, modified, or deleted that best maintain the permission inheritance and role activation semantics of the original hierarchy.

4.4.1 Role Addition

Typically, a new role is added to an existing hierarchy to distribute a unique set of new permissions among the already existing roles in the hierarchy. Before we add a new role to a hierarchy, we need to properly define its permissions and identify the existing sets of roles that can be its seniors and juniors. Furthermore, we need to account for other existing temporal, separation-of-duty and activation constraints in the hierarchy.

Let r_n be the new role to be added in the original hierarchy H_o. Suppose r_n is to be added between roles s and j, and $s{<}f{>}j \in H_o$. By adding the new role, assume we obtain the new hierarchy H_n. Then, it is easy to see that $H_n = H_o/(s{<}f{>}j) \cup \{(s{<}f_1{>}r_n), (r_n{<}f_2{>}j)\}$ for some hierarchy relations $<f_1>$ and $<f_2>$.

Table 4.5 lists various criteria for hierarchy transformations. Criteria ***C*1** states that the roles of the original hierarchy H_o can be activated by a user in the new hierarchy H_n if and only if the user can activate it in H_o. Similarly, criteria ***C*2** states that the permissions associated with the roles of the original hierarchy H_o can be acquired by a user in the new hierarchy H_n if and only if the user can acquire it in H_o. It may also be possible that the addition of a new role actually results in a *conditioned derived relation* between roles s and j, when originally $s \geq^t j$ or $s \succsim^t j$. Semantically, it means that in H_n, instead of acquiring the permissions of j by activating role s, which is originally allowed in H_o, now a user u assigned to s or its senior needs to activate the new role to acquire j's permission. In such a case, the "*can be acquired*" semantics is not completely lost as u still does not have to explicitly activate j to acquire its permission. However, the original "*can be acquired*" semantics also is not entirely retained. Hence, we say that such a scenario results in a *restricted* transformation. We represent such a scenario as criteria $\boldsymbol{C2^r}$ indicating that criteria ***C*2** has been satisfied in a restricted sense.

Table 4.5

Criteria for hierarchy transformations

Criteria		
1	**$C1$**:	$\forall u \in \texttt{Users}, r \in \text{Roles}(H_o)$ $\texttt{can\_activate}\,(u,r,t,H_o) \leftrightarrow \texttt{can\_activate}\,(u,r,t,H_n)$
2	**$C2$**	$\forall p \in P(\text{Roles}(H_o), t), r \in \text{Roles}(H_o)$, $\texttt{can\_be\_acquired}\,(p,r,t,H_o) \leftrightarrow \texttt{can\_be\_acquired}\,(p,r,t,H_n)$.
	$C2^r$	$\forall LH = (\{x_1, x_2, \ldots, x_i\ s, j\}, [f_{LH}]), r \in \{x_1, x_2, \ldots, x_i, s\}$, $(r\langle f\rangle j) \in H_o \leftrightarrow (r[r_n]\langle f\rangle j) \in H_n$, where $\{x_1, x_2, \ldots, x_i\ s, j\} \subseteq \text{Roles}(H_o)$, $[f_{LH}] \subseteq \{\geq^t, \succcurlyeq^t, \succsim^t\}$ and $\langle f\rangle \in \{\geq^t, \succsim^t\}$.

Table 4.6

Scenarios for hierarchy transformations

Scenarios for role addition	
***S*1**	No extra constraint is added with respect to the new role r_n;
***S*2**	A *permission-centric* activation constraint is added for the new role r_n
***S*3**	A *user-centric* activation constraint is added for the new role r_n;
***S*4**	(s, r_n) is considered to be in DSoD
***S*5**	(r_n, j) is considered to be in DSoD

When a new role is added, various new constraints related to the new role may need to be added as well. It is important to note that the above criteria may not be satisfied if we introduce new constraints along with the new role. We consider the five scenarios ($S1$ through $S5$), shown in Table 4.6 to describe the addition of constraints related to the new role and describe with regards to them various transformations that satisfy criteria $C1$ and $C2$ or $C1$ and $C2^r$, as shown in Table 4.7. Here, √ indicates that the transformation satisfies the indicated criteria under the given scenario and × indicates otherwise. Note that the static separation of duty (SSoD) constraint between hierarchically related roles is not appropriate [Gav98]. However, an A-hierarchy allows dynamic SoD (DSoD) to be defined on a role [Jos02]. Hence, we only consider DSoD between roles as a scenario.

Table 4.7

Transformation with criteria satisfied for different scenarios

		$(s\langle f\rangle j) \in H_o$	$(s\langle f_1\rangle r_n), (r_n\langle f_2\rangle j) \in H_n$	Criteria Satisfied	***S*1**	***S* 2**	***S* 3**	***S* 4**	***S* 5**
a	*i*	$s \succcurlyeq^t j$	$s \succcurlyeq^t r_n,\ r_n \succcurlyeq^t j$	***C*1, *C*2**	√	√	×	√	√
	ii		$s \succsim^t r_n,\ r_n \succcurlyeq^t j$		√	×	√	×	√
	iii		$s\langle f\rangle r_n,\ r_n \geq^t j$, *for any* $\langle f\rangle$		×	×	×	×	×
b	*i*	$s \geq^t j$	$s \succcurlyeq^t r_n,\ r_n \geq^t j$	***C*1, *C*2**^{r}	√	√	×	√	×
	ii		$(s \geq^t r_n,\ r_n \geq^t j);(s \succsim^t r_n, r_n \geq^t j);(s \geq^t r_n,\ r_n \succsim^t j)$	***C*1, *C*2**	√	×	√	×	×
	iii		$(s \succcurlyeq^t r_n, r_n \geq^t j)$; $(s \succsim r_n,\ r_n \succsim j)$; $(s\langle f\rangle r_n,\ r_n \succcurlyeq^t j,)$ *for any* $\langle f\rangle$	***C*1, *C*2**	×	×	×	×	×
c	*i*	$s \succsim^t j$	$s \succcurlyeq^t r_n,\ r_n \succsim^t j$	***C*1, *C*2**^{r}	√	√	×	√	×
	ii		$s \succsim^t r_n,\ r_n \succsim^t j$	***C*1, *C*2**	√	×	√	×	×
	iii		$(s\langle f\rangle r_n,\ r_n \succsim^t j);(s\langle f\rangle r_{n},,\ r_n \geq^t j,)$ *or* $(s\langle f\rangle r_{n},,\ r_n \succcurlyeq^t j)$, *for any* $\langle f\rangle$		×	×	×	×	×

Various transformation cases for role addition indicated in Table 4.7, which are depicted in Fig. 4.11, can be easily explained by applying the inference rules to infer the derived rules between s and j in H_o. Note that DSoD constraints are allowed among roles that are only A-hierarchically related. Similarly, permission-centric activation constraints are appropriate when A-hierarchy is used whereas the user-centric activation constraint is appropriate in an I or IA-hierarchy.

Fig. 4.11(a) depicts the addition of role r_n when $(s \succcurlyeq^t j)$ for case (a) of Table 4.7. As Fig. 4.11(a) shows, only cases (*i*) and (*ii*) satisfy $C1$ and $C2$ under some scenarios. Case (*i*) allows defining a *permission-centric* activation time constraint on role r_n ($S2$) because of the new relation $(s \succcurlyeq^t r_n)$. Defining a *DSoD* constraint between roles s and r, and r and j ($S4$ and $S5$) is allowed by the A-relations between them. Case (*ii*) allows a user-centric activation-time constraint on role r_n because of the relation $(s \succsim^t r_n)$, $(r_n \succcurlyeq^t j)$. Cases depicted in (*iii*) do not retain original hierarchical properties, not even when no constraint is added for the new role ($S1$). The main reason is the introduction of an I-relation that removes the original activation semantics.

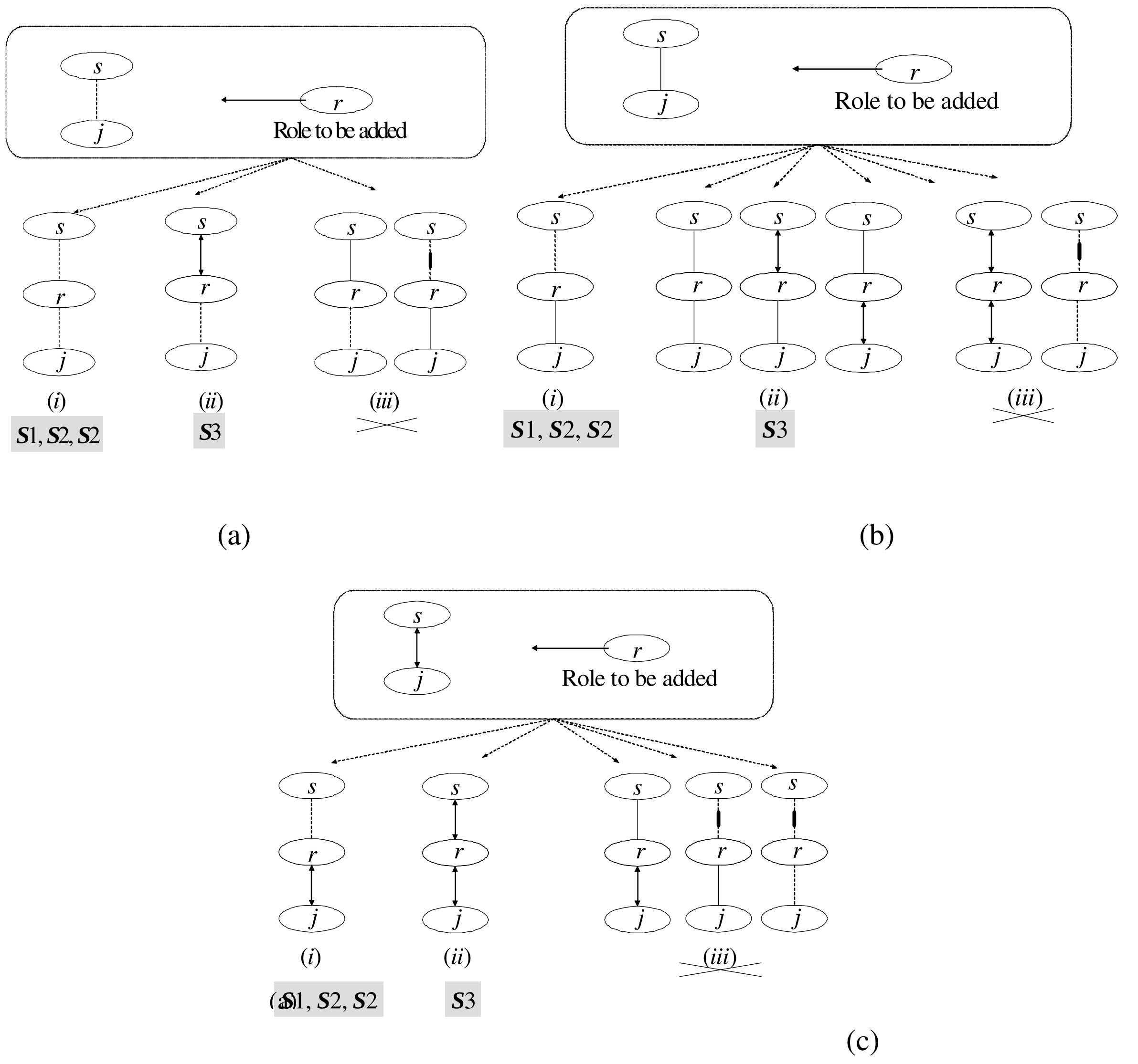


Fig. 4.11. Addition of a new role r between roles s and j

Fig. 4.11(b) depicts the addition of role r_n when ($s \geq^t j$) is in H_o. Because of the new relation ($s \succcurlyeq^t r_n$) for case (i), we see that $\mathbf{C2}^r$ is satisfied for some scenarios as s and j are related by an I-relation in H_o only. Furthermore, ($s \succcurlyeq^t r_n$) allows the *permission-centric* activation constraint on r_n and DSoD constraints on s and r_n (S 4 and S5). Cases depicted in Fig. 4.11(b)(ii) maintain the original derived relation ($s \geq^t j$). These choices allow defining a *user-centric* activation time constraint on role r_n (S3). Cases depicted in Fig. 4.11(b)(iii) either introduce an A-relation between s and j, which is not present in the original hierarchy, or makes them hierarchically unrelated, removing the original I-relation between s and j. Hence, such transformations do not satisfy any criteria.

Fig. 4.11(c) depicts the addition of role r_n when $(s \succsim^t r_n)$ is in H_o. Choice (*i*) introduces an *A*-relation between s and r_n, removing the *I*-relation between *s* and *j*; thus, only $C2^r$ instead of *C*2 can be satisfied for some scenarios. Choice (*ii*) retains the original relation between *s* and *j* and hence is the most accurate transformation. However, there is more restriction on choice (*ii*) in terms of what constraints can be defined for r_n than in choice (*i*). In the choices depicted in (*iii*), the *I*-relation between *s* and *j* is completely removed, hence no transformation is possible.

4.4.2 Role Deletion

When a role is deleted from a hierarchy, the crucial issue is what to do with the permissions associated with it and the users assigned to it. Generally, we can expect to require that the permissions be retained in the system, and thus making them available through other roles in the hierarchy. This requires redistributing the permissions associated with the deleted role to other roles in the hierarchy, and reassigning the users originally assigned to the deleted role. We identify the following three approaches for a deletion of a role from a hierarchy: (1) the *first approach* is to reassign the permissions of the deleted role to its immediate seniors; (2) the *second approach* is to reassign the permissions of the deleted roles to its immediate juniors; and (3) the *third approach* is to reassign the permissions of the deleted role to each of the senior roles through which the permissions of the deleted role can be acquired within the original hierarchy. One key problem with these approaches is the reassignment of the users who were originally assigned to the deleted role. Any reassignment will result in a privilege escalation of some users assigned to roles in the hierarchy. The third approach is ad-hoc and defeats the purpose of a hierarchy structure. In practice, this approach may be applicable when the whole hierarchy needs to be restructured. We do not discuss the third approach further.

Let H_o be the original hierarchy and H_n the new hierarchy obtained by deleting role *r*. Furthermore, let U_r and P_r be the sets of users and permissions explicitly assigned to role *r*. For each immediate junior *j* of *r*, let U_j be the set of users assigned to *j*. Let *s* be an immediate senior of *r*. Table 4.8 depicts different cases of transformations for the *first* and the *second* approaches that attempt to meet the criteria *C*1 and *C*2 introduced earlier.

As shown in the table, privilege escalation of users in U_r occurs in the *first approach* as they are assigned to senior roles, whereas that of users in U_j occurs in the *second approach*, as the permissions associated with the senior role are assigned to role *j*.

The table further shows what the appropriate transformations are for different sets of relations between s and r, and r and j in H_o.

Table 4.8
Deletion of a role using approaches 2 and 3

	The First Approach		***The Second Approach***	
For $r\langle f\rangle j\in H_o$, $\langle f\rangle$ is	*For* $s\langle h\rangle r\in H_o$, $\langle h\rangle$ is	*For* $s\langle f\rangle j\in H_n$, $\langle f\rangle$ is (for appropriate transformation)	*For* $s\langle f\rangle r\in H_o$, $\langle h\rangle$ is	*For* $s\langle f\rangle j\in H_n$, $\langle f\rangle$ is (for appropriate transformation)
A-hierarchy ($\succcurlyeq^t$)	I-hierarchy ($\geq^t$)	*no relation*	I-hierarchy ($\geq^t$)	*none appropriate*
	A-hierarchy ($\succcurlyeq^t$)	A-hierarchy ($\succcurlyeq^t$)	A-hierarchy ($\succcurlyeq^t$)	A-hierarchy ($\succcurlyeq^t$)
	IA-hierarchy ($\succsim^t$)	A-hierarchy ($\succcurlyeq^t$)	IA-hierarchy ($\succsim^t$)	A-hierarchy ($\succcurlyeq^t$) (*restrictive*)
I-hierarchy ($\geq^t$)	any	I-hierarchy ($\geq^t$)	any	I-hierarchy ($\geq^t$)
IA-hierarchy ($\succsim^t$)	*for any* $\langle h\rangle$	$\langle h\rangle$	*for any* $\langle h\rangle$	$\langle h\rangle$
Reassignments$\rightarrow$	U_r and P_r are assigned to role s		U_r and P_r are assigned to role j	
Result of reassignment $\rightarrow$	Privilege escalation for users in U_r		Privilege escalation for users in U_j	

Fig. 4.12 depicts various transformations when $(r\succcurlyeq^t j)\in H_o$ under the *first approach*. Note that s may be related to its immediate senior by any of the three hierarchical relations. To show the overall picture, we include roles x, y, and z as seniors of s with respect to I, A and IA-relations, respectively. Let $\langle h\rangle$ be the original relation between s and r. When $\langle h\rangle$ is an I-hierarchy, s and j are not hierarchically related, as s does not inherit j's permissions. Neither is any user assigned to s or its seniors able to activate j in H_o. Hence, case (*i*) in Fig. 4.12(a) (i.e., "*no relation*" between s and j) retains the original derived relation between s and j, also indicated in the table. The choices (*ii*), (*iii*) and (*iv*) in Fig. 4.12(a) result in undesirable situations as each one makes something possible that was not originally possible. Similarly, when $\langle h\rangle$ is an A or IA-hierarchy, s and j have a derived relation $(s\succcurlyeq^t j)$. Hence, as shown in figures 4.12(b) and 4.12(c), after the deletion of role r, we can introduce the direct relation $(s\geq^t j)$ or $(s\succcurlyeq^t j)$. We note that after the deletion of role r, if we have $(s\succsim^t j)$, it makes the inheritance of j's permissions by s possible, something that was not originally allowed.

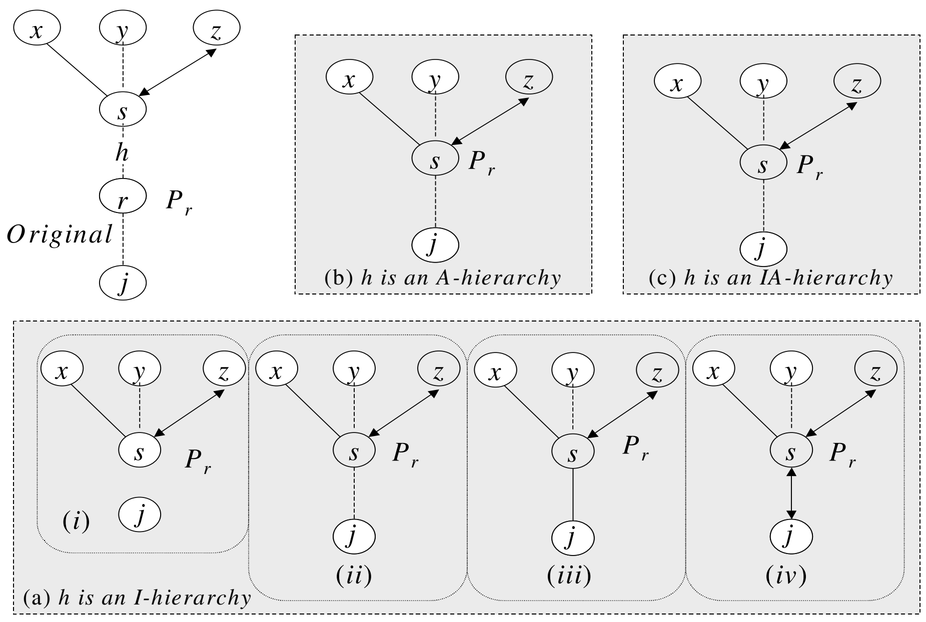


Fig. 4.12. Deletion of role r *when* $(r \succcurlyeq^t j)$

The cases for $(r \geq^t j)$ or $(r \succsim^t j)$ in H_o can be similarly explained. When $(r \geq^t j) \in H_o$, for all relations between s and r, the resulting relation between s and j will be $(s \geq^t j)$ as shown in the table. It is straightforward to see that it is so when h is an *I*-relation. If <h> is an *IA*-relation, then $(s \geq^t j)$ is the derived relation in H_o; hence after the transformation, the relation is maintained. However, if <h> is an *A*-relation, then the original relation between s and j would be $(s\{r\} \geq^t j)$. If in the transformed hierarchy we use relation $(s \geq^t j)$, then users who can activate s still cannot activate j, but still can acquire j's permission, now by activating s in place of the deleted role r. Hence, the semantics about a user *not being able to activate it but acquire its permission by activating some senior role* is still present in the hierarchy with the new relation $(s \geq^t j)$. It is, however, to be noted that this transformation affects the original relations between j and role s or those above it. The change is in terms of what needs to be activated to acquire j's permission.

Various cases for the *second approach* can be similarly explained. The key difference is when $(r \succcurlyeq^t j) \in H_o$. Here, if $(s \geq^t r) \in H_o$, no hierarchical relation between s and j can be derived in H_o; hence, we cannot have any relation between s and j. However, "*no relation*" between s and j means that the permission set P_r, now assigned to j, cannot be used by any user who can activate s. Note that in H_o, a user who can activate s can also activate r and hence acquire P_r. For this case, therefore, there is no appropriate transformation. Similarly, if $(s \succsim^t r) \in H_o$, the only possible new relation is $(s \succcurlyeq^t j)$. However, it is somewhat *restrictive* in the sense that, in H_o, a user, u, who can activate s need not explicitly activate j to acquire its permission. But in H_n, u needs to activate j to acquire its permission.

4.4.3 Role Partition

Sometimes it is essential that an existing role be simply partitioned to change the semantics of the hierarchy. In particular, partitioning may indicate the requirement for separating the role's permissions into different subsets. We identify the following three ways to partition a role: (1) *vertical partitioning*: here a role is partitioned into a set of new roles that form a linear path with the permission set of the old role distributed among the new roles; (2) *horizontal partitioning*: here the role's permission set is partitioned into a number of disjoint sets, each of which is assigned to a new role; the new roles do not have any hierarchical relations between them; and (3) *hybrid partitioning*: here both vertical and horizontal partitioning are applied on the role resulting in an arbitrary hierarchy over the new roles. Fig. 4.13 illustrates these partitions.

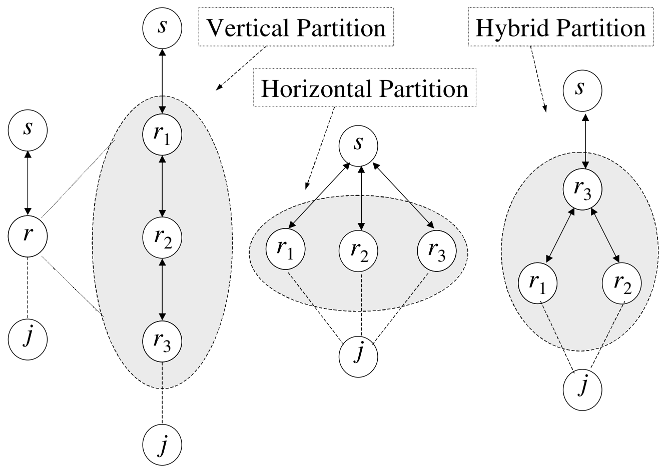


Fig. 4.13. Partitioning a role r into three roles r_1, r_2 and r_3

In each case, the set of new roles replaces the partitioned role in the hierarchy. Once a role is partitioned, it is possible that an administrator completely redefine the hierarchical relationships in the part of the hierarchy above the partitioned role. Such a case requires offline redesign of the system. However, it may be required that the original hierarchical semantics as defined by criteria $C1$ and $C2$ (see Table 4.5) do not change.

Table 4.9 lays out various transformation characteristics of the three approaches. Here, role r of the original hierarchy H_o is partitioned into a set of new roles $RP = \{x_1, x_2, ..., x_n\}$. As usual, let s and j represent a senior and a junior of r.

Table 4.9

Transformation characteristics for different approaches to role partitioning

Role $r \in$ Roles(H_o) *is partitioned into* $RP = \{x_1, x_2, ..., x_n\} \subset$ Roles(H_n)		***Vertical Partition***		***Horizontal Partition***	***Hybrid Partition***
1	Hierarchy characteristics	$L = (RP, <f>)$ (i.e. forms linear path)		No pair is hierarchically related	$H = (RP, [f]) = \{LH_1, LH_2, ..., LH_n\}$ for $n >$ 1(i.e. a hierarchy which i s not a linear path) such that Roles$(LH_i) \subset RP$ Condtion*:* *if* $s<f>j$ *is a derived relation in* H_o *then at least one linear path* LH_i *must allow deriving relation* $s<h>j$ *in* H_o.
		if	*then*		
		$(s \geq^t r)$, $(r<h>j) \in H_o$, or $(s \succsim^t r)$, $(r \geq^t j) \in H_o$	$<f> \in \{\geq^t, \succsim^t\}$		
		$(s \succsim^t r)$, $(r \succsim^t j) \in H_o$ or $(s \succsim^t r)$, $(r \succcurlyeq^t j) \in H_o$	$<f> = \succsim^t$		
		$(s \succcurlyeq^t r)$, $(r<h>j) \in H_o$	$<f> \in \{\succcurlyeq^t, \succsim^t\}$		
2	Reassignment of U_r	For all $u \in U_r$, u is assigned to x_1		For all $x \in RP$, $u \in U_r$, u is assigned to x	For all $x \in \{S_{LH1}, S_{LH2}, ..., S_{LHn}\}$, $u \in U_r$, u is assigned to x
3	Relation with s where $(s<f>r) \in H_o$	$s<f>x_1$		For all $x \in RP$, $s<f>x$	For all $x \in \{S_{LH1}, S_{LH2}, ..., S_{LHn}\}$, $s<f>x$
4	Relation with r where $(r<f>j) \in H_o$	$x_n<f>j$		For all $x \in RP$, $x<f>j$	For all $x \in \{J_{LH1}, J_{LH2}, ..., J_{LHn}\}$, $x<f>j$

Row 1 shows various hierarchy characteristics associated with the roles in RP. As already indicated above, in *vertical partitioning*, the new roles form a linear path. The linear path can be a monotype in order to retain original hierarchy semantics. As shown in the table, if originally $(s \geq^t r)$, $(r<f>j) \in H_o$, or $(s \succsim^t r)$, $(r \geq^t j) \in H_o$, then in the new hierarchy H_n, the monotype hierarchy over the roles in RP should be of type $\geq^t$ or $\succsim^t$. This is necessary to retain the original derived relation $(s \geq^t j)$ in the transformed hierarchy. If $(s \succsim^t r)$, $(r \succsim^t j) \in H_o$ or $(s \succsim^t r)$, $(r \succcurlyeq^t j) \in H_o$, then the new linear path over the roles in RP

should be of *type* $\succcurlyeq^t$ *or* $\succsim^t$. Similarly, if $(s \succcurlyeq^t r)$, $(r \langle h \rangle j) \in H_o$, then the new linear path over the roles in *RP* should be of type $\succsim^t$.

The original semantics as defined by criteria $C1$ and $C2$ are ensured in the vertical partitioning by these transformations and by the new relations defined in rows 3 and 4. For *horizontal partitioning*, the roles in *RP* are not hierarchically related. For hybrid partitioning, the roles in *RP* form multiple linear paths. The condition for the hybrid partitioning states that at least one linear path must allow inferring the derived relation $s\langle f\rangle j$ of H_n. For the linear path that maintains the original derived relation $s\langle f\rangle j$, we can use the transformations outlined for vertical partitioning in the *if-then* columns.

Entries in row 2 indicate the reassignments of the users in U_r originally assigned to role r, to new role(s) in *RP*. The reassignments shown here are defined on the basis that the original access capabilities of the users are to be retained, although privilege escalation for some user may result from the process. In practice, this may not be the actual case, and the relations among roles in the partition shown in row 1 may need to be accordingly adjusted. Rows 3 and 4 indicate how the roles s and j are related to the new roles in the partition. For a *vertical partitioning* approach, the original relation between s and j is used between s and x_1 (x_n and j) as indicated. Note that x_1 and x_n are the senior-most and the junior-most roles of the new linear path created by the roles in *RP*. In case of *horizontal partitioning*, s and j are made senior and junior of each of the roles in *RP*. The case for *hybrid partitioning* is similar to that of the horizontal partitioning except that the role s is made senior to the senior-most roles of each of the linear paths formed over the roles in *RP*, whereas j is made the junior of each of junior-most roles of these linear paths.

As indicated above, the need for such partitioning is primarily to restructure or redistribute permission sets in a hierarchy. Another reason for doing such partitioning may be because of the temporal properties. For example, a role may need to be vertically partitioned to arrange the temporal properties in such a way that the intervals associated with a senior role contain the interval associated with the junior roles. Similarly, a horizontal partition may need to be done to create roles with distinct nonoverlapping intervals. Furthermore, a hybrid partitioning may be needed to properly structure very complex temporal properties. An analysis of such partitioning based on temporal properties will be considered in detail in a slightly different context in Chapter 7.

4.5 Conclusions

In this chapter, we have presented an analysis of hybrid temporal role hierarchies for the GTRBAC model. We have introduced the notion of a *uniquely activable set* of a hierarchy that identifies access capabilities of a user assigned to a role in a hierarchy in a single session. The formal results we have presented allow determining uniquely activable sets for hybrid temporal role hierarchies and provide a basis for controlling privilege distribution to the users by restricting activable sets associated with the roles they are assigned to. The results related to the *AC-equivalence* between different role hierarchies also show that, in cases where the principle of least privilege is not a concern, a monotype hierarchy may be used. Furthermore, as an *A*-hierarchy does not allow direct permission-inheritance, the results show that the *A*-hierarchy provides the most needed flexibility. In particular, an *A*-hierarchy allows DSoD constraints to be defined on hierarchically related roles. Furthermore, the inherit-all-permission semantics of *I*-hierarchy as well as *IA*-hierarchy has several pitfalls in terms of their ability to handle many organizational control principles [Mof98].

We have also introduced a set of inference rules which can be employed to infer hierarchical relationships between pairs of roles that are not directly related. We have formally showed that the set of inference rules is *sound* and *complete*. In a complex hybrid hierarchy, these rules provide a formal basis for analyzing the permission acquisition and role activation semantics. We have also introduced the notion of *conditioned derived relation* which augments the three hierarchies (*I*, *A* and *IA*-hierarchies) and facilitates capturing much fine-grained derived permission acquisition and activation semantics within a hierarchy.

We have also addressed the issue of hierarchy transformation with respect to role addition, deletion, and partitioning. These transformations essentially form the basis for policy evolution in an organization. It is to be noted that transformations that retain original hierarchical semantics in a hybrid hierarchy need to be based on what type of additional role constraints exist or will be added in the hierarchy.

The results presented in this chapter provide a formal basis for developing administrative tools for the management of GTRBAC systems. Such security administrative functions are crucial for a well-planned, timely control of unauthorized accesses as well as for distributing least access capabilities to users in order to allow them to carry out their activities and at the same time minimize damage that may be caused by misuse of privileges.

5. CARDINALITY, DEPENDENCY AND SEPARATION OF DUTY CONSTRAINTS

Cardinality and SoD constraints are crucial for securing many applications in a commercial environment. Many researchers have highlighted the importance and use of cardinality and SoD constraints in RBAC models. However, no one has addressed the time-based cardinality and SoD constraints. Use of a particular constraint for a period of time or duration is important for emerging applications as access requirements frequently change with time. Dependency constraints are relevant to role based systems as roles often embody organizational functions that may be inter-dependent. For instance, a doctor in training may be allowed to work *only if* some senior doctor who can supervise him is also on duty. Some aspect of dependency constraints, such as history based access control, operational SoD, etc., have been mentioned in general access control literature [Sim97], but they have not been adequately addressed for general RBAC systems. Such dependency constraints have been well-recognized in workflow systems where workflow tasks have inherent dependencies.

In this chapter, we focus on these constraints within the GTRBAC modeling framework. The key contributions of this chapter are as follows:

- We introduce a generic framework for expressing a wide range of time-based cardinality constraints with the help of GTRBAC status predicates, a function to evaluate these predicates, and a projection operator that extracts a set of elements from the evaluation of the predicates.
- We develop an elaborate trigger expression that can capture complex dependencies among events and conditions. In particular, we define CFD constraints that can be used to express stricter control flow dependencies. Furthermore, we show that the trigger framework and the CFD constraint expressions can be easily extended to provide an elaborate time-based RBAC model for context-based access control.
- We identify a large set of possible SoD constraints using the GTRBAC status predicates. These SoDs subsume the SoDs that have been identified in the RBAC literature, and at the same time provide a much finer modeling capability. In

particular, extended with a temporal dimension, these SoDs can provide various forms of semantics generating a richer set of fine grained SoD constraints.

5.1 Generalized Cardinality Constraint Expression

In this section we develop a framework for expressing cardinality constraints with respect to all states of GTRBAC systems including role states, assignments states as well as user sessions. First, we revisit the status predicates introduced in Chapter 3.

Table 5.1.
Various status predicates

Predicate(s_t)	**Evaluation Domain**(*DOM*)	**Semantics**
P:permission set, R:role set, U:user set, S:set of sessions, T:time instants, $r \in R$, $p \in P$, $u \in U$, $s \in S$, $t \in T$		
`enabled`(r, t)	$R \times T$	*r is enabled at time t*
`u_assigned`(u, r, t)	$U \times R \times T$	*u is assigned to r at time t*
`p_assigned`(p, r, t)	$P \times R \times T$	*p is assigned to r at t*
`can_activate` (u, r, t)	$U \times R \times T$	*u can activate r at t*
`can_acquire` (u, p, t)	$U \times P \times T$	*u can acquire p at t*
`r_can_acquire` (u,p, r, t)	$U \times P \times R \times T$	*u can acquire p through r at t*
`can_be_acquired`(p, r,t)	$P \times R \times T$	*p can be acquired through r at t*
`active(`r, t)	$R \times T$	*r is active in at t*
`u_active(`u, r, t)	$U \times R \times T$	*r is active in u's session at t*
`us_active(`u, r, s, t)	$U \times R \times S \times T$	*r is active in u's session s at t*
`acquires`(u, p, t)	$U \times P \times T$	*u acquires p a timet t*
`r_acquires`(u, p, r, t)	$U \times P \times R \times T$	*u acquires p through r at t*
`s_acquires`(u, p, s, t)	$U \times P \times S \times T$	*u acquires p in session s at t*
`rs_acquires`(u, p, r, s, t)	$U \times P \times R \times S \times T$	*u acquires p through r in session s at t*

Table 5.1 lists all the possible GTRBAC status predicates. The non-temporal counterparts of each predicate can be simply obtained by removing the time parameter. A non-temporal predicate $\boldsymbol{s}$ simply indicates that its corresponding temporal predicate $\boldsymbol{s}_t$ applies at all times, i.e., $\boldsymbol{s} \rightarrow \forall t,\ \boldsymbol{s}_t$. Inversely, $\boldsymbol{s}_t$ means that status predicate $\boldsymbol{s}$ holds at time t. The second column of Table 5.1 specifies the evaluation domain for the predicates in the first column. The third column describes the semantics of the predicate. The axioms, as introduced in Chapter 4, capture the key relationships among various predicates in Table 5.2 and provide the basis for defining precisely the permission-acquisition and role-activation semantics of a GTRBAC system.

Table 5.2.

Relations among predicates

1	`can_acquire` $(u,\ p, t) \leftrightarrow \exists\, r \in R,$ `r_can_acquire` (u, p, r, t)
2	`active` $(r, t) \leftrightarrow \exists\, u \in U,$ `u_active` (u, r, t)
3	`u_active` $(u,\ r, t) \leftrightarrow \exists\, s \in S,$ `us_active` (u, r, s, t)
4	`acquires` $(u,\ p, t) \leftrightarrow \exists\, r \in R,$ `r_acquires` (u, p, r, t)
5	`acquires` $(u,\ p, t) \leftrightarrow \exists\, s \in S,$ `s_acquires` (u, p, s, t)
6	`acquires` $(u,\ p, r, t) \leftrightarrow \exists\, s \in S,$ `rs_acquires` $(u, p, r,\ s, t)$
7	`acquires` $(u,\ p, s, t) \leftrightarrow \exists\, r \in R,$ `rs_acquires` $(u, p, r,\ s, t)$

5.1.1 Predicate Evaluation Function and Projection Operator

Next, we formulate a general framework for expressing cardinality constraints by using the status predicate. This allows us to define cardinality constraints with respect to any state of a GTRBAC system. For this purpose, we use a *predicate evaluation* function and a *projection operator* over the result of the *evaluation* function to extract information about users, roles, permissions, sessions, and time. The predicate evaluation function eval returns the subset of the evaluation domain of the predicate specified as its argument. The projection operation $\Pi_{\pi 1,\ \pi 2,\ \ldots,\ \pi m}$ projects over the specified set of elements from the evaluated domain. $\Pi_{\pi 1,\ \pi 2,\ \ldots,\ \pi m}$ is similar to the projection operator in relational calculus. The two functions are defined as follows:

Definition 5.1.1(eval, Π_i): *Let s_t(alist) be a status predicate, where alist is a list of arguments a_1, …, a_i, …, a_n associated with* domains D_1, …, D_i, …, D_n, *respectively*

($\forall j \in \{1, ..n\}$, $D_j \in \{R, P, U, S, T\}$) . *If DOM is the evaluation domain of $s_t(alist)$, then we define evaluation function* eval *and projection operator* $\Pi_{\pi 1, \pi 2, .., \pi m}$ *as follows:*

- eval($s_t(alist)$)=$\{(x_1, \ldots, x_i, \ldots, x_n) \mid ((x_1, \ldots, x_i, \ldots, x_n) \in DOM) \wedge s_t(x_1, \ldots, x_i, \ldots, x_n)\}$
- $\Pi_{\pi 1, \pi 2, .., \pi m}$eval($s_t(alist)$)=$\{(x_{\pi 1}, x_{\pi 2} .., x_{\pi m}) \mid \{\pi_1, \pi_2, \ldots, \pi_m\} \subseteq \{1, 2, \ldots, n\}; \forall x_{\pi i} \in D_{\pi i},$; *and for all pairs* $(x_1, x_2\ldots , x_n)$, $(y_1, y_2\ldots , y_n) \in$ eval($s_t(a_1, \ldots a_{i-1}, a_i, a_{i+1},\ldots, a_n)$), $x_j = y_j$ *for all* $j \in \{1, 2, \ldots, n\}/\{\pi_1, \pi_2, \ldots, \pi_m\}$;
  moreover, for all such j we replace the argument by its constant value in quotes; i.e., we denote a constant value $x \in D$ by "*x*" *in the argument list*}.

Evaluation function eval returns the subset of the evaluation domain corresponding to the predicate that it evaluates. For instance, eval (*enabled*(*r, t*)) is a subset of domain ($R \times T$). Similarly, $\Pi_{\pi 1, \pi 2, .., \pi m}$ allows us to project the evaluation of a predicate over a particular argument indexed by *i*. For instance, $\Pi_1$eval(*enabled*(*r, "t"*)) returns the set of all roles that are enabled at time "*t*". Similarly, $\Pi_2$eval(*enabled*(*"r", t*)) returns the set of all time instants at which role "*r*" is enabled. Let us denote the set of all projection functions over the predicates defined in Table 5.2 as Π. Note that we can also have evaluation of the negation of the predicates of Table 5.2, for instance, $\Pi_1$eval ($\neg$*enabled*(*r, "t"*)). Π^{-1} denotes the set of projection operators over negated predicates. Based on these projection operators and the original set of set elements Orig=$\{R, U, P, S, T\}$, we build a framework for expressing an exhaustive set of cardinality constraints as follows. Let OP $\in \{\cup, \cap, /\}$ be a set operation; then we have a generic set function f as follows:

1. $f \in (\Pi \cup \Pi^{-1})$;
2. $f = (f$ OP $X)$, where $X \subseteq E \in$ Orig;
3. $f = (f_1$ OP $f_2)$, where f_1 and f_2 are generic set functions.

We can express a cardinality constraint as ($|f|$ cop n), where $|f|$ is the number of elements in set f, cop $\in \{=, \neq, <, >, \geq, \leq\}$ is a comparator operator, and n is a positive number. Some examples of the cardinality constraint expressions are shown in Table 5.3. It is to be noted that while projection operators in Π make sense in a general context (as shown in Table 5.3), those in Π^{-1} may not have a clear meaning. Therefore, care should be taken in constructing cardinality constraints based on them. For example, the function $\Pi_2$eval($\neg$*u_active*(*"u", r, "t"*)) refers to a set of roles that are not active in any of user *u*'s sessions at time *t*. Hence, $|\Pi_2$eval($\neg$*u_active*(*"u", r, "t"*))$| \leq n$ states that the number of roles *not* active in any of user *u*'s sessions at time *t* cannot be more than *n*.

However, it is not clear whether it is *n* out of those that *u* can activate or out of those in *R*. Depending upon the application, a distinction may need to be made a priori. For instance, we can say that, "*by default, out of those that u can activate*". Furthermore, we note that some cardinality constraints of type $C = (|\Pi_{\pi 1,\ \pi 2,\ \ldots,\ \pi m}\mathsf{eval}(s_t(plist))|\ \mathsf{cop}\ n)$ may not have direct application in a general RBAC framework. For example, $\Pi_1\mathsf{eval}(\mathtt{s\_active}(u, \text{"}r\text{"}, \text{"}s\text{"}, \text{"}t\text{"}))$ (set of users that have activated *r* in session *s* at time *t*) associates multiple users with the same session. Such cases may be useful if we consider a collaborative system where a session is created with multiple active users.

Table 5.3.
Examples of cardinality constraints

1	$\|\Pi_1\mathsf{eval}(\mathtt{enabled}(r, \text{"}t\text{"})\| \geq n$	Number of roles enabled at time "*t*" cannot be less than *n*
2	$\|\Pi_1\mathsf{eval}(\neg\mathtt{enabled}(r, \text{"}t\text{"})\| \leq n$	Number of roles disabled at time "*t*" cannot be more than *n*.
3	$\|\Pi_2\mathsf{eval}(\mathtt{u\_assigned}(\text{"}u\text{"}, r, \text{"}t\text{"}))\| \leq n$	Number of roles assigned to "*u*" at time "*t*" cannot be more than *n*
4	$\|\Pi_2\mathsf{eval}(\mathtt{can\_activate}(\text{"}u\text{"}, r, \text{"}t\text{"}))\| \leq n)$	Set of roles that *u* can activate at time *t* cannot be more than *n*.
5	$(Daytime, \|\Pi_1\mathsf{eval}(\mathtt{u\_assignedSet}(u, \text{"Nurse"}, t)\| \leq n)$	Number of users assigned to Nurse role in *Daytime* cannot exceed *n*

5.1.2 Time Based Cardinality Constraints

Periodicity and duration constraints on a cardinality constraint $C = (|f|\ \mathsf{cop}\ n)$ can be simply defined using the GTRBAC temporal framework as (*I*, *P*, *C*), which indicates that the cardinality constraint is valid for each instant in intervals defined by (*I, P*), and as ([*I*, *P*,| *D*], D_x *C*), with D_x indicating the duration in which the cardinality constraint is valid. However, the different semantics may need to be attached to its interpretation. To illustrate these possible interpretations, we first present the cardinality constraint in a single interval, say τ:

Definition 5.1.2 (**Interval-constraint on Cardinality**): *Let C* $=(|\Pi_{\pi 1,\ \pi 2,\ \ldots,\ \pi m}\mathsf{eval}(s_t(plist))|\mathsf{cop}\ n)$*be a cardinality constraint and τ an interval. Then, C can be time-constrained within τ in the following ways*:

1. Weak form: (τ, C_W) *is said to be a weak form of time-based cardinality constraint if the following is satisfied:*

$$\forall t \in \tau, i \in \{1, 2, .., m\},\ (\pi_i = t) \wedge (|\Pi_{\pi 1, \pi 2, .., \pi m}\mathsf{eval}(s_t(plist))|\mathsf{cop}\ n)$$

2. Strong form: (τ, C_S) *is said to be a strong form of time-based cardinality constraint if the following is satisfied*:

$$(\sum_{t \in \tau} |\Pi_{\pi 1, \pi 2, .., \pi m}\mathsf{eval}(s_t(plist))|)\ \mathsf{cop}\ n$$

Note that the difference between the two forms is that, in the *weak* form, the cardinality constraint is defined with respect to each instant in the specified interval. For example, if the cardinality constraint is ((9am-9pm), $(|\Pi_{u=\text{"Smith"}}\ \mathsf{eval}(\texttt{u\_assigned}(u, r, t)| \le 5)_W)$, then it implies that at each time unit between 9am and 9pm, the user Smith is assigned to not more than five roles. At different time units, Smith may be assigned different roles. In contrast, the strong form, ((9am-9pm), $(|\Pi_{u=\text{"Smith"}}\ \mathsf{eval}(\texttt{u\_assigned}(u, r, t)| \le 5)_S)$ would mean that, between 9am and 9pm, the total number of roles assigned to Smith, simultaneously or otherwise, should be less than or equal to five.

Next, we extend these forms to provide differing semantics for periodicity constraints.

Definition 5.1.3 (**Periodicity constraint on Cardinality**): *Let C* $=(|\Pi_{\pi 1, \pi 2, .., \pi m}\mathsf{eval}(s_t(plist))|\mathsf{cop}\ n)$*be a cardinality constraint and* (*I, P*) *be a periodicity expression. Then, C can be time-constrained in* (*I, P*) *in the following ways*:

1. Weak form: (*I, P*, C_W) *is said to be a weak form of time-based cardinality constraint if the following is satisfied:*

$$\forall \tau \in \Pi(I, P), (\tau, P, C_W)$$

Alternately, (*I, P*, C_W) *can be expressed as*:

$$\forall t \in Sol(I, P),\ i \in \{1, 2, .., m\},\ (\pi_i = t) \wedge (|\Pi_{\pi 1, \pi 2, .., \pi m}\mathsf{eval}(s_t(plist))|\mathsf{cop}\ n)$$

2. Strong form: (*I, P*, C_S) *is said to be a strong form of time-based cardinality constraint if the following is satisfied*:

$$\forall \tau \in \Pi(I, P), (\tau, P, C_S)$$

3. Extended Strong form: (I, P, C_{EW}) *is said to be a strong form of time-based cardinality constraint if the following is satisfied*:

$$\left(\sum_{t \in Sol(I,P)} |\Pi_{\pi 1, \pi 2, .., \pi m} \text{eval}(s_t(plist))| \right) \text{ cop } n$$

Note that the *weak* periodicity constraint implies that in each of the recurring intervals, the weak interval-constraint semantics applies. Similarly, the *strong* periodicity constraint implies that for each recurring interval of the periodic expression, the interval-constraint applies. It is also intuitive to extend the strong interval-constraint on cardinality to all the time instants of the periodic expression. We refer to this as the *extended strong* form.

5.2 Extended Triggers and Control Flow Dependency Constraints

Another set of constraints that are often needed in the commercial systems is that of dependencies between roles and other events associated with RBAC entities. GTRBAC provides a trigger mechanism that can be used to express some dependency constraints. However, there are much stricter forms of dependency constraints known as control flow dependency (CFD) constraints, which are needed in various applications. In this section, we extend the original GTRBAC triggers and define the CFD constraints using extended triggers.

5.2.1 Extended GTRBAC Triggers

The basic trigger expression of GTRBAC is of the form: (E_1 ,…, E_m, C_1 ,…, C_k $\rightarrow$ *pr:E* `after` Δt), where E_i is an event and C_i is a status condition. Semantically, it means that the prioritized event *pr:E* with priority *pr* can take place Δt time units after the trigger fires. The definition, however, is limiting in the following ways: (1) it only allows scenarios in which all the antecedent events E_1,…, and E_m occur at the same time and all the conditions C_1 ,…, and C_k hold; it does not allow capturing history information in which events are spread in the temporal dimension; (2) it does not allow specifying temporal intervals in which the occurrence of an event E_i can take place, or a condition

C_i is satisfied; (3) it is possible that in some cases a condition C_i must be valid for a specified duration before triggering the event E; such a requirement is also not captured by the current triggers; and (4) the current trigger considers that $E \neq s$`:activate` r `for` u; this needlessly prevents specifying any preconditions for activation events. In some cases, an activation request may need to be granted only if certain conditions have been satisfied. We define the extended trigger form, which is temporally more expressive than the current GTRBAC triggers and does not have the above limitations, as follows:

Definition 5.2.1 (**Extended Triggers**): *The extended trigger expression has the following form*:

$$(E_1 \texttt{ in } \pi_1) \text{ op}_1 \ldots \text{op}_{m-1} (E_m \texttt{ in } \pi_m) \text{ op}_m \ (C_1 \texttt{ in } \tau_1 \texttt{ for } d_1) \text{ op}_{m+1} \ldots \text{op}_{m+n-1} (C_n \texttt{ in } \tau_n \texttt{ for } d_n) \rightarrow pr{:}E \texttt{ after } \Delta t \texttt{ for } \Delta d, \textit{ where}$$

- *E_is are simple event expressions or run time requests; and C_is are GTRBAC status expressions,*
- *$pr{:}E$ is a prioritized event expression with priority pr.*
- *If ($E = s$`:activate` r `for` u) is an activation request at time $t_a \geq (t + \Delta t)$ then* `u_active(`*u, r, s, t) is true in the interval (t_a, $t_a + \Delta d$), provided that the trigger fires at time t,*
- *the trigger is fired if τ_i (π_i) is an interval such that there exists a $t \in \tau_i$ (π_i) at which C_i (E_i) becomes valid, and C_i remains valid for duration d_i; we simply write "C_i* `in` *τ_i" to mean that there exists a $t \in \tau_i$ at which C_i is valid for some duration; we write "C_i* `at` *t" ("E_i* `at` *t") instead of "C_i* `in` *τ_i" ("E_i* `in` *t") when τ_i (π_i) = (t, t); we write "C_i* `for` *d_i" to mean that C_i is valid for some duration d_i.*
- *Δt is the duration between the firing of the trigger and the occurrence of the event E, and Δd is the duration for which the event E remains valid. If not specified, $\Delta t = 0$, and $\Delta d = \propto$ $\text{op}_i \in \{\vee, \wedge\}$ and $\wedge$ has precedence over $\vee$. For simplicity, we use "," to denote the $\wedge$ operator and "|" to denote the $\vee$ operator.*

We note that the old trigger form cannot be used to specify the temporal information such as "E_i `in` π_i" or "C_i `in` τ_i". The earlier form is actually a special case of the extended form, in which all the antecedent events and conditions are associated with the same time instant. That is, for any t,

E_1 at t,.., E_m at t, C_1 at t,.., C_k at $t \rightarrow pr{:}E$ after Δt, (*a*)

The duration information Δd associated with the triggered event E in the extended trigger simplifies specification but does not increase the expressive power over the earlier form. The following trigger:

E_1 at t,…, E_m at t, C_1 at t,…, C_k at $t \rightarrow$ $pr{:}E$ after Δt for Δd (*b*)

is semantically equivalent to the combination of the following two old triggers

1. E_1 ,…, E_m, C_1 ,…, C_k $\rightarrow$ $pr{:}E$ after Δt,
2. $E \rightarrow$ pr':Conf(E) after Δd, where Conf(E) is the conflicting event of E and $pr' \geq pr$.

We note that the triggers of form (*a*) (one with "at t_n" phrase) can represent the extended form (one with "in π_m" phrase); however, it is easy to see that the extended form achieves compaction in expression over the form (*a*). For instance, the extended trigger form without the "for d_1" part can only be represented by using multiple triggers of form (*a*), each with a permutation of time instants from π_1, π_2,…, π_m, τ_1, τ_2,…, τ_n. Similar compaction is achieved by the use of the two logical operators.

Note that triggers allow GTRBAC events and status conditions only. However, it can easily be extended to include other events and conditions. For instance, condition C_i can be any predicate expression that evaluates contextual information that affects access control decisions. Consider the following trigger:

$(Location(x) = \text{"}EmergencyRoom\text{"}) \mid (situation\,() = \text{"}LifeThreatening\text{"}) \rightarrow$
$pr{:}E$ enable EmergencyDoctor

Here, if the room, indicated by variable x, is *EmergencyRoom*, or the current *situation* is *LifeThreatening* then the *EmergencyDoctor* role is enabled, thus capturing environmental context. Similarly, we can allow event E to be any system related event. With a predefined set of predicates to capture static as well as dynamic environmental conditions and events, the extended GTRBAC trigger framework can easily provide a very elaborate support for context-based access control.

5.2.2 Control Flow Dependency Constraints

Control flow dependency (CFD) constraints often occur in task-oriented systems and are stricter forms of dependency constraints than those that can be expressed using GTRBAC triggers. The following example illustrates such CFD constraints.

`Example 5.2.1`: Consider the following requirements: (1) a junior employee of an office is allowed to activate the Junior_Employee role in the system *only if* his manager has activated the Manager role; (2) whenever a system administrator makes some changes in the system, the activation of the SysAdmin role that he uses *must* enable the SysAudit role so that another user can activate the SysAudit role and log those changes. The SysAudit role may need to be activated by the user within the next τ minutes; (3) everyday, if both the roles SysAdmin and SysAudit are activated, then the SysAdmin role must be activated before the SysAudit role.

The three requirements imply (1) *pre-condition*, (2) *post-condition* and (3) *precedence* constraints. Next, we show that GTRBAC does not adequately model these constraints, but we can semantically define CFDs in terms of these triggers.

Pre-condition Constraints

A *pre-condition* constraint between two events essentially implies that an event can occur *only if* the other event has already occurred and/or the required conditions have already become true, as in the first case above. The following trigger closely resembles the pre-condition constraint (1):

`s:activate` Manager for John $\rightarrow$ `enable` Junior_Employee
(Assume that John is the manager)

However, the "*only if*" semantics of the *pre-condition* constraint requires that there be no other events that will enable the Junior_Employee role; i.e., the Junior_Employee role is not enabled if John does not activate the Manager role. This means the above trigger can enforce the *pre-condition* constraint only if we also enforce the additional restriction that no other constraint or trigger allows the enabling of the

Junior_Employee role. However, GTRBAC's trigger mechanism currently does not imply such an additional restriction; hence, it falls short in providing support for the *pre-condition* constraint. For instance, in addition to the above trigger, assume that we also have the following periodicity constraint:

Everday between 9am and 6pm, `enable` Junior_Employee

The presence of this *periodicity* constraint does not allow the above trigger to enforce the *pre-condition* constraint as it allows the role to be enabled even if the Manager role is not enabled.

Post-condition Constraints

A *post-condition* constraint between two events essentially implies that *if* an event occurs or a condition is satisfied, then the other event also *must* occur, as indicated in the second case in the example above. Here, *if* the SysAdmin role is enabled then the SysAudit role *must* also be enabled; otherwise, it may incur certain security risks. However, the activation of the SysAudit role may also be triggered by other events in the system. In essence, the *post-condition* constraint will not be enforced if there are some other triggers or constraints that do not allow the SysAudit role to be enabled even though the SysAdmin role has been enabled. Thus, it is easy to see that the following trigger:

`enable` SysAdmin $\rightarrow$ `enable` SysAudit

enforces the *post-condition* constraint *only if* the system additionally makes sure that there are no other constraints or triggers that prohibit enabling of the SysAudit role when this trigger fires; this cannot be expressed using GTRBAC triggers.

Precedence Constraints

A *precedence* constraint is said to exist between two events if there is a condition that *if the two events occur, then one must always precede the other*, as shown in requirement (3). Another real world scenario in which such a *precedence* semantics applies is a pair of tasks involving *authorizing a check* and *cashing* it. It is easy to see that such *precedence* semantics is not enforced by triggers alone.

Next, we formalize the syntax and semantics of the CFD constraints in GTRBAC using triggers. In the definitions we will use (t_s, t_e), such that (t_s, t_e) is in an interval of (I, P), or (t_s, t_e) is some duration D. For $(t_s, t_e) = D$, t_s is the time instant when D starts and is non-deterministic. A constraint $c = (D, C)$ needs to be enabled by a trigger or a runtime event. Assume that T is the set of all GTRBAC constraints and `Causes`(c, pr:E, t) is a predicate that evaluates to *true* if there is a constraint c in T which causes event pr:E to fire at time t. Furthermore, we use Y to denote the left hand side of a trigger expression, *i.e.*,

$$Y = E_1 \texttt{ in } \pi_1, \ldots, E_m \texttt{ in } \pi_m, C_1 \texttt{ in } \tau_1 \texttt{ for } d_1, \ldots, C_n \texttt{ in } \tau_n \texttt{ for } d_n$$

The following precedence rule is applied in a GTRBAC system - if there are conflicting pairs of events (e.g., `assign` and `deassign`, `activate` and `deactivate`, etc.) then the negative event takes precedence (e.g., `deassign` takes precedence over `assign`) if the priority of the two events are the same; otherwise the higher priority event takes precedence.

Definition 5.2.2 (*Pre-condition constraint*): *The pre-condition constraint is expressed as* ($[I, P|D,]$ pre, Y, pr:E `after` Δt `for` Δd). *Semantically, to say that* ($[I, P|D,]$ pre, Y, pr:E `after` Δt `for` Δd) $\in T$ *is equivalent to saying that:*

1. ($Y \rightarrow pr$:E `after` Δt `for` $\Delta d)_t \in T$ *is an* extended-trigger, *and*
2. $\neg\exists\, c \in T$ *s.t.* $(\forall t_x \in (t + \Delta t,\ t + \Delta t + \Delta d)$ *and* $pr' \geq pr$, `Causes`(c, pr': E, t_x)) *is true for* $pr' \geq pr$.

Definition 5.2.3 (*Post-condition constraint*): *The post-condition constraint is expressed as* ($[I, P|D,]$ post, Y, pr:E `after` Δt). *Semantically, to say that* ($[I, P|D,]$ post, Y, pr:E `after` Δt `for` Δd) *is in* T *is equivalent to saying that*:

1. ($Y \rightarrow pr$:E `after` Δt `for` $\Delta d)_t \in T$ *is an* extended-trigger;
2. $\neg\exists\, c \in T$ *s.t.* $\forall\, t_x \in (t + \Delta t,\ t + \Delta t + \Delta d)$, `Causes`($c$, pr': Conf(E), t_x) *is true for* $pr' \geq pr$.

Note that condition (2) in each definition ensures that the additional conditions required for the two CFDs, as discussed earlier, are enforced. Next, we define a precedence constraint, which relates two events. We can also define a CFD with "*if and only if*" by combining the above two constraints. Next we define a precedence constraint that essentially relates two events.

Definition 5.2.4 (*Precedence constraint*): *Let* $pr_1{:}E_1$ and $pr_2{:}E_2$ *be prioritized events such that* ([I, $P|D$,] prec, $pr_1{:}E_1$, $pr_2{:}E_2$ `after` Δt), i.e. $pr_2{:}E_2$ is precedent on $pr_1{:}E_1$. *Then, for all t such that* $t \in (t_s, t_e)$:

$$([I,\ P|D,]\ \text{precedence}, pr_1{:}E_1, pr_2{:}E_2) \rightarrow$$
$$(\textit{for each pair } c_1, c_2 \in T,$$
$$\texttt{Causes}(c_1, pr{:}E_1, t_1) \wedge \texttt{Causes}(c_2, pr{:}E_2, t_2) \rightarrow\ (t_e \le t_1 + \Delta t \le t_2 \le t_s))$$

The safety notion introduced in Chapter 3 identifies scenarios that have ambiguous execution semantics, for example, the existence of a cycling dependency among events through triggers. The safety checking algorithm can be easily extended to identify the violation of the CFD constraint by introducing extra checks to ensure that additional restrictions are enforced for the CFDs. Furthermore, it is easy to see that by using triggers, we can easily express all the 13 temporal relations between a pair of GTRBAC events [All83]. Moreover, using CFD constraints, we can define stronger forms of the temporal relations.

5.3 Time-Based Separation of Duty Constraints

Separation of Duty (SoD) policies have been found to be very crucial for securing commercial applications. Role-based systems are particularly very beneficial for expressing and enforcing such policies. Various SoDs have been identified in the literature. However, all earlier research focus on SoDs in a non-temporal environment.

The triggers and CFD constraints introduced in the previous section can also be used to define SoD constraints that are based on access history, such as the history-dependent SoD, order-dependent SoDs, object-based SoDs, which are identified in [Ahn00, Sim97]. In this section, we define various SoD constraints that cannot be captured by such CFD constraints, some of which correspond to those already identified in the literature. These SoDs will be defined with respect to GTRBAC status predicates introduced earlier.

Table 5.4

Enabling time and assignment SoDs

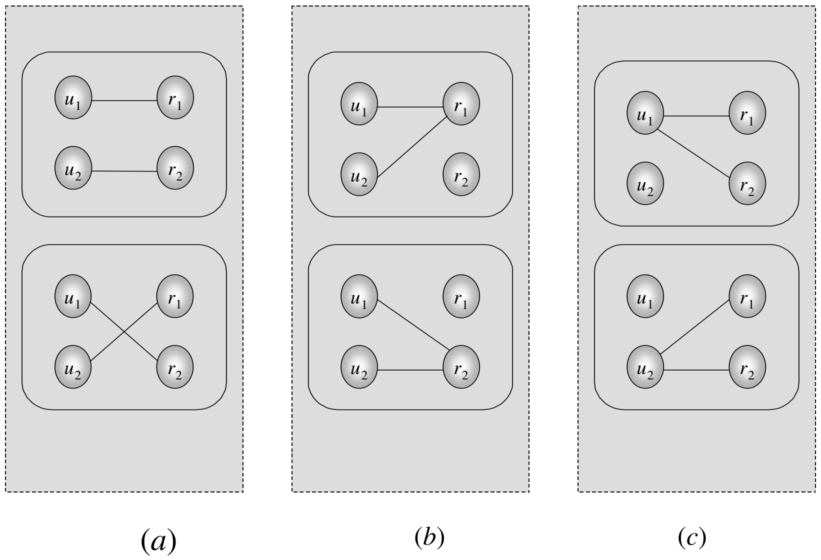
SoD Type	Expression (*SoD*)	Semantics $\forall u, u_1, u_2 \in \mathsf{U}, \forall r, r_1, r_2 \in \mathsf{R}, \forall p, p_1, p_2 \in \mathsf{R}, , (u_1 \neq u_2), (r_1 \neq r_2)$ and $(p_1 \neq p_2)$ *the following holds*:
Enabling/Disabling SoD		
EN-SoD	$(I, P, \mathsf{EN}, \mathsf{R})$	$SoD \wedge \mathtt{enabled}(r) \rightarrow \neg\mathtt{enabled}(r_2)$
	No two roles in R can be *enabled* at the same time	
(DIS-SoD)	$(I, P, \mathsf{DIS}, \mathsf{R})$	$SoD \wedge \mathtt{disabled}(r_1) \rightarrow \neg\mathtt{disabled}(r_2)$
	No two roles in R can be *disabled* at the same time	
User-Role assignment/de-assignment SoDs		
UAS-SoD$_1$	$(I, P, \mathsf{UAS}_1, \mathsf{U}, \mathsf{R})$	$SoD \wedge \mathtt{u\_assigned}(u, r_1) \rightarrow \neg\mathtt{u\_assigned}(u, r_2)$
	No two roles in R can be *assigned* to a user in U at the same time	
UAS-SoD$_2$	$(I, P, \mathsf{UAS}_1, \mathsf{U}, \mathsf{R})$	$\forall r \in R,\ SoD \wedge \mathtt{u\_assigned}(u_1, r, t) \rightarrow \neg\mathtt{u\_assigned}(u_2, r, t)$
	No two users in U can be *assigned* to a role in R at the same time	
UAS-SoD$_3$	$(I, P, \mathsf{UAS}_3, \mathsf{U}, \mathsf{R})$	$SoD \wedge \mathtt{u\_assigned}(u_1, r_1, t) \rightarrow \neg\mathtt{u\_assigned}(u_2, r_2, t)$
	Different users in U cannot be *assigned* different roles in R at the same time	
UAS-SoD$_4$	$(I, P, \mathsf{UAS}_4, \mathsf{U}, \mathsf{R})$	$SoD \leftrightarrow$ UAS-SoD$_2$ $\wedge$ UAS-SoD$_3$
	Roles in R can be assigned to only one of the users in U at the same time	
UAS-SoD$_5$	$(I, P, \mathsf{UAS}_5, \mathsf{U}, \mathsf{R})$	$SoD \leftrightarrow$ UAS-SoD$_1$ $\wedge$ UAS-SoD$_3$
	Users in U can be *assigned* only one of the roles in R at the same time	
UAS-SoD$_6$	$(I, P, \mathsf{UAS}_6, \mathsf{U}, \mathsf{R})$	$SoD \leftrightarrow$ UAS-SoD$_1$ $\wedge$ UAS-SoD$_2$
	A role in R can be assigned to only one user in U (and vice versa) at the same time	
Role-Permission assignment/de-assignment SoDs		
PAS-SoD$_1$	$(I, P, \mathsf{PAS}_1, \mathsf{P}, \mathsf{R})$	$\forall p \in P,\ SoD \wedge \mathtt{p\_assigned}(p, r_1, t) \rightarrow \neg\mathtt{p\_assigned}(p, r_2, t)$
	No two roles in R can be *assigned* a permission in P at the same time	
PAS-SoD$_2$	$(I, P, \mathsf{PAS}_2, \mathsf{P}, \mathsf{R})$	$\forall r \in R,\ SoD \wedge \mathtt{p\_assigned}(p_1, r, t) \rightarrow \neg\mathtt{p\_assigned}(p_2, r, t)$
	No two permissions in P can be *assigned* to a *role* in R at the same time	
PAS-SoD$_3$	$(I, P, \mathsf{PAS}_3, \mathsf{P}, \mathsf{R})$	$\forall p_1, p_2 \in P,\ SoD \wedge \mathtt{p\_assigned}(p_1, r_1, t) \rightarrow \neg\mathtt{p\_assigned}(p_2\ r_2, t)$
	Different permissions in P cannot be *assigned* to different roles in R at the same time	
PAS-SoD$_4$	$(I, P, \mathsf{PAS}_4, \mathsf{P}, \mathsf{R})$	$SoD \leftrightarrow$ PAS-SoD$_2$ $\wedge$ PAS-SoD$_3$
	Roles in R can be assigned only one of the permissions in P at the same time	
PAS-SoD$_5$	$(I, P, \mathsf{PAS}_5, \mathsf{P}, \mathsf{R})$	$SoD \leftrightarrow$ PAS-SoD$_1$ $\wedge$ PAS-SoD$_3$
	Permissions in P can be assigned to only one of the roles in R at the same time	
PAS-SoD$_6$	$(I, P, \mathsf{PAS}_6, \mathsf{P}, \mathsf{R})$	$SoD \leftrightarrow$ PAS-SoD$_1$ $\wedge$ PAS-SoD$_2$
	Permissions in P can be assigned to only one of the roles in R at the same time	

5.3.1 Enabling Time SoD Constraints (predicates: `enabled`/`disabled`)

The SoD constraints related to enabling and disabling events are shown in Table 5.4. *EN-SoD* indicates that roles from a given role set cannot be *enabled* at the same time. If there are role enabling events that attempt to enable more than one role at the same time, then the enforcement mechanism must use some criteria to enable only one of the roles. SoD *DIS-SoD* is defined with respect to the role-disabling event. The difference between them is that *EN-SoD* does not allow all the roles to be enabled at the same time but allows them to be disabled at the same time, whereas *DIS-SoD* allows all the roles to be enabled at the same time but does not allow them to be disabled at the same time. Role enabling SoDs (*EN-SoD*) are cases where limiting access is a primary concern. Similarly, role disabling SoD (*DIS-SoD*) is more useful in cases where availability is the key concern. For example, in a hospital, a requirement may state that "*Both the* Nurse *and* Doctor *roles cannot be disabled at the same time*". These SoDs can be expressed in the form of cardinality constraints introduced earlier; e.g., *EN-SoD* can be expressed as $|\Pi_1 \text{eval}(\textit{enabled}(r, t)) \cap \text{R}| \leq 1$. Similarly, other SoDs defined below can also be expressed in this form; however, we use uniformly the implication rule to provide the semantics of these SoDs.

5.3.2 Assignment Time SoD Constraints(predicates: `u_assigned, p_assigned`)

Table 5.4 defines various user-role and role-permission assignment time constraints. *UAS-SoD*$_1$ indicates that multiple roles from R cannot be *assigned* to a user in U at the same time. Accordingly, the roles from R can be assigned to any user not included in U. In other words, this implies that the role set R has conflicting semantics only with respect to the user set U. Allowing specification of such a set of conflicting roles *with respect to* a particular user set provides the benefit of expressing fine-grained SoD constraints. *UAS-SoD*$_2$ states that different users of U cannot be assigned to a role in R; i.e., the users in U are conflicting with respect to role set R. Fig. 5.1 depicts various assignment combinations that are not allowed by the user-role assignment constraints for $\text{U} = \{u_1, u_2\}$ and $\text{R} = \{r_1, r_2\}$. Here, a line from u to r indicates that u is assigned to r. In general, set U can be expected to be the set `Users`. In Fig. 5.1, *UAS-SoD*$_1$ does not allow assignment combinations depicted in (*c*), whereas, *UAS-SoD*$_2$ does not allow assignment combinations shown in (*b*).



SoD	Doesn't Allow	SoD	Doesn't Allow
UAS-SoD$_1$	(*c*)	*UAS-SoD*$_4$	(*a*), (*b*)
UAS-SoD$_2$	(*b*)	*UAS-SoD*$_5$	(*a*), (*c*)
UAS-SoD$_3$	(*a*)	*UAS-SoD*$_6$	(*b*), (*c*)

Fig. 5.1. User-assignment SoDs with U=$\{u_1,u_2\}$ and R=$\{r_1,r_2\}$

UAS-SoD$_3$ states that different users from set R cannot be assigned to different roles. Here, U and R have conflicting semantics with respect to each other. Note that the notion of conflict here is slightly different from that of *UAS-SoD*$_1$ and *UAS-SoD*$_2$. However, this constraint allows a single user from U to be assigned to multiple roles of R, and a single role from R to be assigned to multiple users. *UAS-SoD*$_3$ does not allow the assignment scenario depicted in (*a*). In a real world scenario, U of *UAS-SoD*$_3$ may refer to a set of employees who are related. The assignment of any two of these employees to different roles will allow them to commit fraud. If, for instance, one employee is assigned to role AuthorizationManager (that authorizes checks) and another is assigned to role CashingClerk (for cashing authorized checks), they can easily commit fraud. Another practical scenario in which this constraint can be applicable is when a set of roles represents subtasks of a bigger task, with the constraint that the different users of U cannot carry out different subtasks.

UAS-SoD$_4$, *UAS-SoD*$_5$ and *UAS-SoD*$_6$ can be derived as combinations of earlier SoDs, as shown in Table 5.1. We note that, although *UAS-SoD*$_3$ allows defining constraints such as all the subtask roles which need be assigned to the same user, it also allows the assignment scenario (*b*), which may not be relevant with regards to such a requirement. *UAS-SoD*$_4$, for instance, omits the possibility of assigning all the users to the same subtask role rendering the overall task un-accomplishable. As shown in the figure, *UAS-SoD*$_5$ prevents the set of assignments of the type shown in (*a*) and (*c*) – i.e., it allows multiple users to be assigned to only one of the roles, such as those in Fig. 5.1(b). That is, as soon as one of the roles, say role *r*, is assigned to a user, then none of the users can be assigned to any other roles; however, role *r* can be assigned to any number of users. An example of the application of *UAS-SoD*$_5$ is the assignment of a given set of consultants (set U) to the same consultancy duty (the assignment of all the users to the role ConsultantOfCompanyA).

The role-permission assignments have semantics similar to that of the user-role assignments. Note that, here, we are using the notion of conflicting permission, for example in *PAS-SoD*$_2$.

5.3.3 Activation Time SoD Constraints (predicate: `active`)

Activation time SoD constraints are listed in Table 5.5. *ACT-SoD*$_1$ implies that activation of conflicting roles at the same time in the same session or different sessions by a user in U is not allowed. Fig. 5.2 depicts the scenarios for U = $\{u_1, u_2\}$ and R = $\{r_1, r_2\}$ when both the roles are active. Here, s: $u_1(r_1, r_2)$ indicates that roles r_1 and r_2 are active in u_1's session *s*. *ACT-SoD*$_1$ does not allow activation combinations depicted in figures 5.2(*b*) and 5.2(*c*). *ACT-SoD*$_2$ does not allow activation of a role by conflicting users at the same time. Similarly, *ACT-SoD*$_3$ does not allow conflicting roles to be active in different users' sessions, as depicted in Fig. 5.2(*a*).

ACT-SoD$_4$ does not allow the scenario in Fig. 5.2(*c*); i.e. it prevents activation of the conflicting roles in the same session simultaneously. *ACT-SoD*$_5$ does not allow scenarios depicted in 5.2(*b*); i.e. it prevents the activation of the conflicting roles in the different sessions of the same user simultaneously. *ACT-SoD*$_6$ and *ACT-SoD*$_7$ are combinations of the earlier SoDs, as indicated in the table.

Fig. 5.3 illustrates the usefulness of SoD constraints *ACT-SoD*$_1$ - *ACT-SoD*$_5$. In Fig. 5.3(i), roles r_1 and r_2 have a common set of permissions. Now suppose we allow users u_1 and u_2, assigned to roles r_1 and r_2, respectively, to activate the respective roles at the same time. As the *read* permission on the object O_x is available to both the roles, the information that each role writes to object O_x is visible to the other. Hence, O_x opens up the information flow channel between the two users. Common permissions like these may occur explicitly, in a non-hierarchical case, or implicitly through an *I*-hierarchy relation such as in the one shown in Fig. 5.3(*i*). In a non-hierarchical case, declaring the two roles as conflicting and applying *ACT-SoD*$_3$ is a straightforward solution if such information flow needs to be contained.

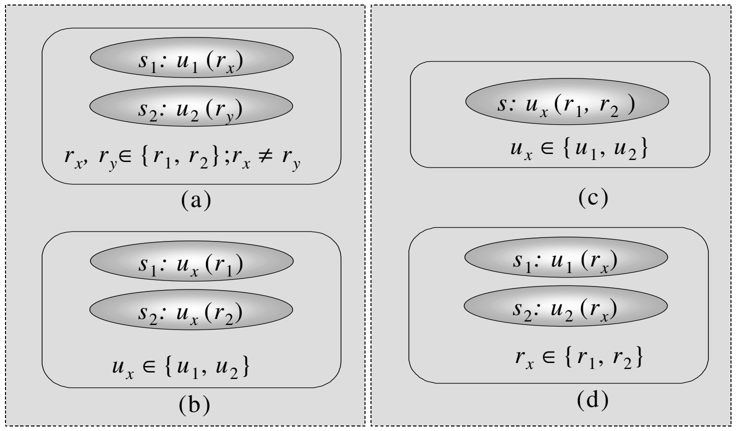
Table 5.5

Activation time SoDs

Type	***SoD***	**Semantics** $\forall u, u_1, u_2 \in \mathsf{U}, \forall s, p_1, p_2 \in \mathsf{R}, \forall r, r_1, r_2 \in \mathsf{R}, \forall t \in Sol(I, P)$, $(u_1 \neq u_2)$, $(r_1 \neq r_2)$ and $(p_1 \neq p_2)$, *the following holds*:
ACT-SoD$_1$	$(I, P$, ACT-SoD$_1$, U, R)	$\forall u \in U, SoD \wedge \mathtt{u\_active}(u, r_1, t) \rightarrow \neg\mathtt{u\_active}(u, r_2, t)$
	No two roles in R can be in *active* state in session(s) of a user in U at the same time	
ACT-SoD$_2$	$(I, P$, ACT-SoD$_2$, U, R)	$\forall u \in U, SoD \wedge \mathtt{u\_active}(u_1, r, t) \rightarrow \neg\mathtt{u\_active}(u_2, r, t)$
	No two users in U can have a role in R *active* at the same time	
ACT-SoD$_3$	$(I, P$, ACT-SoD$_3$ U, R)	$\forall u_1, u_2 \in U, SoD \wedge \mathtt{u\_active}(u_1, r_1, t) \rightarrow \neg\mathtt{u\_active}(u_2, r_2, t)$
	No two users in U can have two different roles in R *active* at the same time	
ACT-SoD$_4$	$(I, P$, ACT-SoD$_4$, U, R)	$\forall u \in U, \forall s \in S, SoD \wedge \mathtt{u\_active}(u, r_1, s, t) \rightarrow \neg\mathtt{u\_active}(u, r_2, s, t)$
	Two roles in R cannot be in *active* state at the same time in a single session of a user in U	
ACT-SoD$_5$	$(I, P$, ACT-SoD$_5$, U, R)	$\forall u \in U, \forall s_1, s_2 \in S, SoD \wedge \mathtt{u\_active}(u, r_1, s_1, t) \rightarrow \neg\mathtt{u\_active}(u, r_2, s_2, t)$
	No two sessions of a user in U can have two roles in R *active* at the same time	
ACT-SoD$_6$	$(I, P$, ACT-SoD$_6$, U, R)	$SoD \leftrightarrow$ ACT-SoD$_2 \wedge$ ACT-SoD$_4$
	Roles in R can be *active* in a session(s) of only one of the users in U at the same time	
ACT-SoD$_7$	$(I, P$, ACT-SoD$_7$, U, R)	$SoD \leftrightarrow$ ACT-SoD$_5 \wedge$ ACT-SoD$_6$
	Roles in R can be *active* in a single session of only one of the users U at the same time	

Kuhn [Kuh99] indicates that roles that have common permissions cannot form a conflicting pair. We believe that such semantics is too restrictive. Moreover, with that semantics, we indirectly impose mutual exclusion on the permission sets of the two roles.

This may not be what is required in practice. For example, there may be situations where only the private permissions of a pair of roles are conflicting, but the roles may have a common set of permissions.



SoD	**Does not allow**	**SoD**	**Does not allow**
ACT-SoD$_1$	(*b*), (*c*)	*ACT-SoD*$_5$	(*b*)
ACT-SoD$_2$	(*d*)	*ACT-SoD*$_6$	(*a*), (*d*)
ACT-SoD$_3$	(*a*)	*ACT-SoD*$_7$	(*a*), (*b*) , (*d*)
ACT-SoD$_4$	(*c*)		

Fig. 5.2. Activation time SoDs for $U = \{u_1, u_2\}$ and $R = \{r_1, r_2\}$

In such scenarios, conflicting roles imply a conflicting set of private permissions only. For example, an Employee role in general can be used to group the basic set of permissions available to all the employees of an organization. We may have two roles such as AuthorizationManger and CashingClerk, which are both senior to Employee but are considered to be conflicting; however, conflicting semantics is obviously limited to their private permissions rather than the common permissions inherited from Employee. Kuhn's strict mutual exclusion semantics necessitates partitioning even such basic roles in order to enforce mutual exclusion over the total sets of permissions associated with the two roles. However, sometimes in such a scenario, common permissions may create information flow when the private permissions of the two roles conflict. In Fig. 5.3(*i*), for example, when user u_1 activates role r_1, and u_2 activates role r_2 at the same time, they can exchange information contained in O_{1i} and O_{2j} to each other. *ACT-SoD*$_3$ prevents such possibilities.

ACT-SoD$_1$ can be used in cases where a user needs to be restricted from acquiring permissions that give him/her enough power to carry out some activities. For example, Fig. 5.3(*b*) shows two roles that contain permissions for the subtasks of a bigger task. If we want that the same user not carry out the two subtasks, then we can employ the *ACT-SoD*$_1$ constraint. Furthermore, the roles may be organized as an *A*-hierarchy, where role *r* represents the actual task role and is the senior of roles r_1 and r_2 that represent sub-tasks 1 and 2. If users from U are assigned to *r* and the *ACT-SoD*$_1$ is defined with respect to R = $\{r_1, r_2\}$, then the task can only be performed by two different users of U working at the same time.

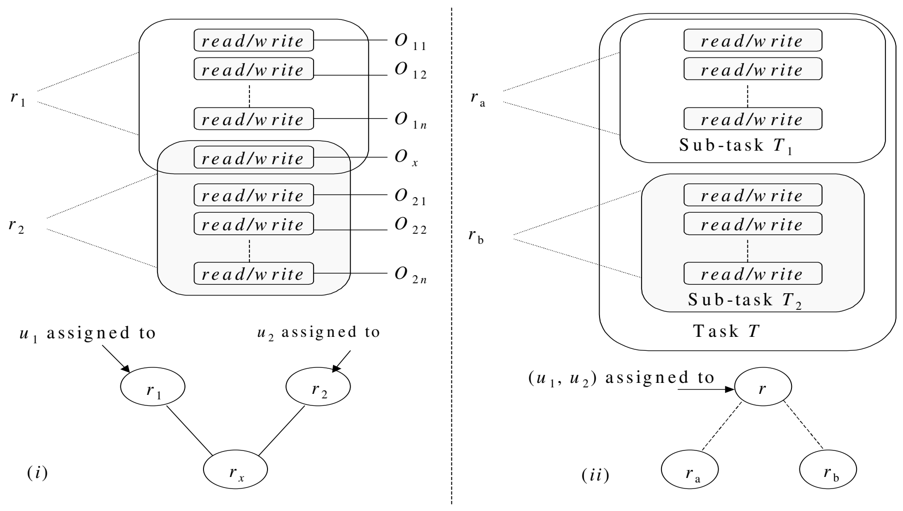


Fig. 5.3. Session time SoD examples

ACT-SoD$_2$ can be used to enforce the requirement that a particular task can be performed by only one person at a time by assuming the task role. *ACT-SoD*$_4$ limits the access capability of a user by not allowing the conflicting roles to be active in a single user session. Its usefulness comes from the fact that a session in RBAC system is semantically the same as a subject in traditional access control models (DAC, MAC, etc.) [San94]. Similarly, *ACT-SoD*$_5$ prevents a user from simultaneously acting as two subjects.

5.3.4 Possibilistic Activation SoD Constraints (predicates: `can_activate`)

We also define SoDs based on the `can_activate` predicate, as shown in Table 5.6. *CACT-SoD*$_1$ prevents all possible activation of conflicting roles by users in U.

Table 5.6
Possibilistic role activation SoDs

Possibilistic Activation (`can_activate`) SoDs		
Type	**SoD**	**Semantics** $\forall u, u_1, u_2 \in$ U, $\forall r, r_1, r_2 \in$ R, $\forall t \in Sol(I, P)$, $(u_1 \neq u_2)$ and $(r_1 \neq r_2)$, *the following holds*:
CACT-SoD$_1$	$(I, P,$ CACT-SoD$_1$, U, R$)$	$\forall u \in U$, $SoD \wedge$ `can_activate`$(u, r_1, t) \rightarrow \neg$`can_activate`$(u, r_2, t)$
	No two roles in R can be activated by a user in U at the same time	
CACT-SoD$_2$	$(I, P,$ CACT-SoD$_1$, U, R$)$	$\forall u_1, u_2 \in U$, $SoD \wedge$ `can_activate`$(u_1, r_1, t) \rightarrow \neg$`can_activate`$(u_2, r_2, t)$
	No two users in U can activate two roles in R at the same time	
CACT-SoD$_3$	$(I, P,$ CACT-SoD$_3$, U, R$)$	$SoD \leftrightarrow$ CACT-SoD$_1$ $\wedge$ CACT-SoD$_2$
	Users in U *can activate* only one of the roles in R at the same time	
CACT-SoD$_4$	$(I, P,$ CACT-SoD$_4$, U, R$)$	$\forall u \in U$, $SoD \wedge$ `can_activate`$(u, r_1, s, t) \rightarrow \neg$`can_activate`$(u, r_2, s, t)$
	No two roles in R can be activated by a user in U in a single session *s* at the same time	
	Users in U *can activate* only one of the roles in R in a single session *s* at the same time	

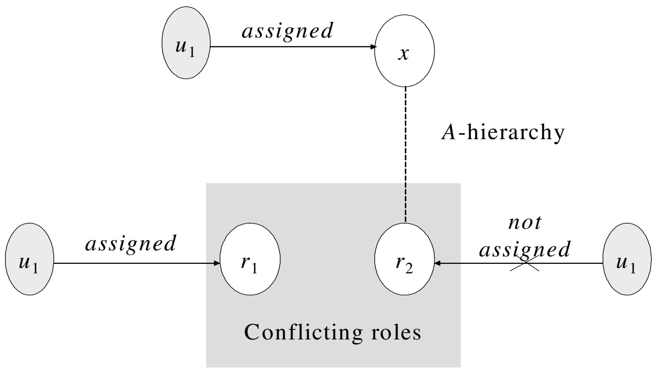


Fig. 5.4. Implication of possibilistic activation SoDs in presence of A-hierarchy

Note that the purpose of *UAS-SoD*$_1$ is essentially to prevent activation of conflicting roles by a user by not allowing explicit assignments to conflicting roles in the first place. For example, when U = $\{u_1, u_2\}$ and R = $\{r_1, r_2\}$, we can prevent the possibility of activation of both the roles by explicitly denying assignments to conflicting roles using *UAS-SoD*$_1$. Let's assume that because of this constraint u_1 is assigned to r_1 but not r_2. Now, assume that there is a role x such that x is senior to r_2 with respect to an A-hierarchy; i.e., any user assigned to x can also activate role r_2, as depicted in Fig. 5.4. Now, if we allow the assignment of u_1 to x, the purpose of preventing u_1 from activating both r_1 and r_2 at the same time is not fulfilled. This is because the A-hierarchy between s and r_2 makes the predicate `can_activate`(u_1, r_2, t) true, hence allowing u_1 to activate r_2 even when u_1 is already assigned to r_1. Therefore, when we have role hierarchies, implicit assignment may be possible through the use of the `can_activate`(u, r, t) predicate. The use of this predicate may make it possible for a user to activate conflicting roles even if the constraint *UAS-SoD*$_1$ is already employed. *CACT-SoD*$_1$ prevents such scenarios; i.e., it prevents both implicit and explicit assignments of a user to conflicting roles. Furthermore, *CACT-SoD*$_2$ is an activation-time counterpart of *UAS-SoD*$_3$, and *CACT-SoD*$_3$ is the activation-time counterpart of *UAS-SoD*$_5$. *CACT-SoD*$_4$ is a session specific counterpart of *CACT-SoD*$_1$.

Note that one way to prevent the scenarios depicted in Fig. 5.4 is to consider that r_1 is in conflict with all the roles hierarchically superior to r_2. However, this approach is very restrictive, and makes the task of properly designing a role hierarchy very difficult.

5.3.5 Possibilistic Permission Acquisition SoD Constraints
(predicates: `can_acquire, can_be_acquired`)

Table 5.7 lists the possibilistic permission acquisition SoDs. *CACQ-SoD*$_1$ prevents the acquisition of permissions through the conflicting roles that will not be caught by *PAS-SoD*$_3$, similar to the way *CACT-SoD*$_1$ prevents the activation of conflicting roles not prevented by *ACT-SoD*$_1$. That is, constraint *CACQ-SoD*$_1$ employs the "*can acquire*" semantics and hence captures both explicit and implicit role-permission assignments. Note that *PAS-SoD*$_3$ can prevent the acquisition of permissions through the conflicting roles by a user by restricting explicit role-permission assignment. However, permission acquisition may also be allowed through the implicit role-permission assignment because of some hierarchical relations. For example, let us consider $\mathsf{P} = \{p_1, p_2\}$ and $\mathsf{R} = \{r_1, r_2\}$. Suppose we have the SoD constraint *PAS-SoD*$_3$; then the same permission in P cannot be assigned to the two roles. Provided there are no hierarchies in the system, the effect (and hence the purpose) of this SoD constraint is that the same permission is not acquired through two roles even if a user is allowed to activate them both. Now, assume there is a role *x* such that r_2 is the senior of *x* with respect to an *I*-hierarchy, as shown in Fig. 5.5. Suppose we allow the assignment of p_1 to *x*. Furthermore, suppose we have the following assignment: p_1 is assigned to r_1, and hence p_1 is not assigned to r_2 by virtue of the constraint *PAS-SoD*$_2$. But, as p_1 is also assigned to *x*, and ($r_2 \geq^t x$), p_1 is also implicitly assigned to r_2, the SoD constraint *PAS-SoD*$_2$ does not prevent p_1 being acquired through role r_2 using hierarchy semantics. *CACQ-SoD*$_1$ prevents such permission-acquisitions through implicit assignments. *CACQ-SoD*$_2$ is to *CACQ-SoD*$_1$ the way *UAS-SoD*$_3$ was to *UAS-SoD*$_1$.

CACQ-SoD$_3$ allows the acquisition of permissions in R by users through only one of the conflicting roles, whereas *CACQ-SoD*$_4$ does not allow different users to acquire different permissions through the conflicting roles. Similar to the *CACQ-SoD*$_1$ constraint, *CACQ-SoD*$_5$ prevents the acquisition of permissions which is allowed by both explicit and implicit assignments. *CACQ-SoD*$_5$ prevents conflicting users from acquiring a permission of *P* through the same role, as in Fig. 5.4, or through different roles, as in Fig. 5.5. *CACQ-SoD*$_6$, on the other hand, does not allow two separate permissions to be acquired by conflicting users neither through the same role (as in (c)) nor the different roles (as in (d)). *CACQ-SoD*$_7$ prevents conflicting users from acquiring a permission in P through the same role in *R* at the same time and hence does not allow case (a). Similarly, the table in Fig. 5.7 shows the cases depicted in Fig. 5.6 that are not allowed by *CACQ-SoD*$_8$, *CACQ-SoD*$_3$ and *User-SoD*$_4$. Various combinations of these SoDs define the SoDs from *CACQ-SoD*$_7$ to *CACQ-SoD*$_{13}$.

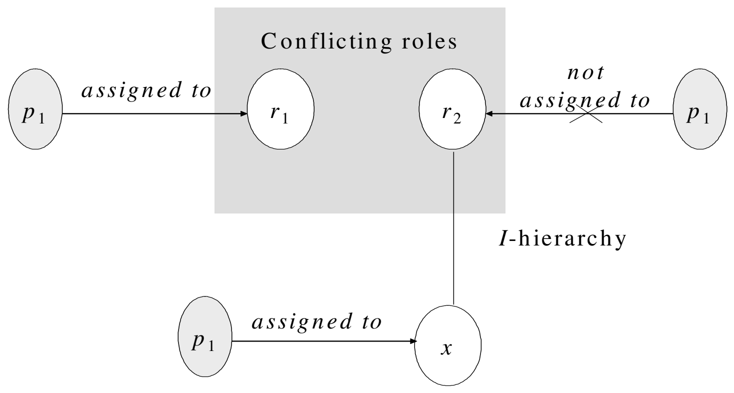


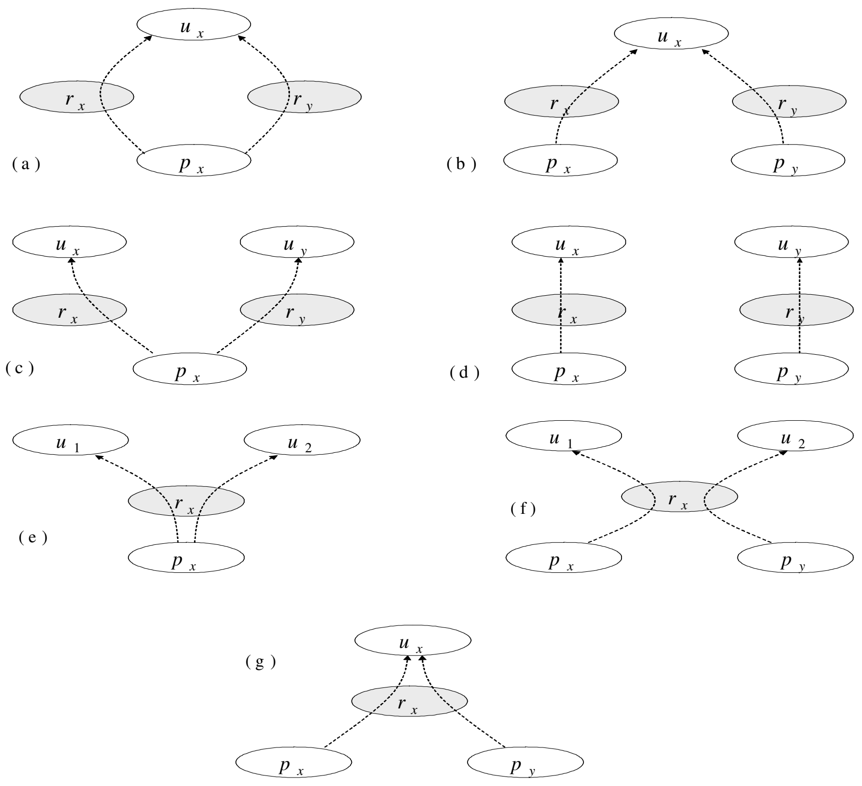
Fig. 5.5. Implication of permission acquisition SoDs in presence of I-hierarchy

Table 5.7.

Possibilistic permission acquisition SoDs

Possibilistic User-Permission Acquisition (`can_be_acquired` and `can_acquire`) SoDs		
Type	**SoD**	**Semantics** $\forall u, u_1, u_2 \in$ U, $\forall r, r_1, r_2 \in$ R, $\forall p, p_1, p_2 \in$ P, $\forall t \in Sol(I, P)$, $(u_1 \neq u_2)$, $(r_1 \neq r_2)$ and $(p_1 \neq p_2)$ *the following holds*:
CACQ_ SoD$_1$	$(I, P,$ CACQ$_1$, U, P, R)	$SoD \wedge \texttt{can\_acquire}(u_2, r_1, t) \rightarrow \neg\texttt{can\_acquire}(u,p\ r_2, t)$
	A permission in P cannot be acquired by a user in U through different roles in R at the same time	
CACQ_ SoD$_2$	$(I, P,$ CACQ$_2$, U, P, R)	$\forall u \in U, \forall p_1, p_2 \in P, SoD \wedge \texttt{can\_acquire}(u, p_1, r_1, t) \rightarrow \neg\texttt{can\_acquire}(u, p_2, r_2, t)$
	No two permissions in P can be acquired by a user in U through roles in R at the same time	
CACQ_ SoD$_3$	$(I, P,$ CACQ$_3$, U, P, R)	$\forall u_1,u_2 \in U, \forall p \in P, SoD \wedge \texttt{r\_can\_acquire}(u_1,p, r_1,t) \rightarrow \neg\texttt{can\_acquire}(u_2,p, r_2,t)$
	No two users in U can acquire a permission in P through different roles at the same time	
CACQ_ SoD$_4$	$(I, P,$ CACQ$_4$, U, P, R)	$\forall u_1,u_2 \in U, \forall p_1,p_2 \in P, SoD \wedge \texttt{r\_can\_acquire}(u_1,p_1, r_1,t) \rightarrow \neg\texttt{r\_can\_acquire}(u_2,p_2, r_2,t)$
	No two users in U can acquire different permissions in U through two roles at the same time	
CACQ_ SoD$_5$	$(I, P,$ CACQ$_5$, U, P)	$\forall p, SoD \wedge \texttt{can\_acquire}(u_1,p,t) \rightarrow \neg\texttt{can\_acquire}(u_2, p, t)$
	No two users in C *can acquire* a *permission* in P at the same time.	
CACQ_ SoD$_6$	$(I, P,$ CACQ$_6$, U, P)	$\forall p_1, p_2 \in P, SoD \wedge \texttt{can\_acquire}(u_1, p_1, t) \rightarrow \neg\texttt{can\_acquire}(u_2, p_2, t)$
	No two permissions in P can be *acquired* by the different users in U at the same time	
CACQ_ SoD$_7$	$(I, P,$ CACQ$_7$, U, P, R)	$\forall p \in P, \forall r \in R, SoD \wedge \texttt{can\_acquire}(u_1, p, r, t) \rightarrow \neg\texttt{acquires}(u_2, p, r, t)$
	A permission in P cannot be *acquired* by different users in U through the same role in R at the same time.	
CACQ_ SoD$_8$	$(I, P,$ CACQ$_8$, U, P, R)	$\forall p_1, p_2 \in P, \forall r \in R, SoD \wedge \texttt{can\_acquire}(u_1, p_1, r, t) \rightarrow \neg\texttt{acquires}(u_2, p_2, r, t)$
	No two permissions in *P* can be *acquired* by two users through the same role in *R* at the same time.	
CACQ_ SoD$_9$	$(I, P,$ CACQ$_9$, R, P)	$\forall r \in R, SoD \wedge \texttt{can\_be\_acquired}(p_1, r, t) \rightarrow \neg\texttt{can\_be\_acquired}(p_2, r, t)$
	No two permissions in P can be *acquired* through a *role* in R at the same time.	
CACQ_	$(I, P,$ CACQ$_{10}$, R, P)	$\forall r_1,r_2 \in R, SoD \wedge \texttt{can\_be\_acquired}(p_1, r_1, t) \rightarrow \neg\texttt{can\_be\_acquired}(p_2, r_2, t)$

SoD$_{10}$	Different *permissions* in P cannot be *acquired* through different roles in R at the same time.	
CACQ_ SoD$_{11}$	(I, P, CACQ$_{11}$, U, P)	$\forall u \in U$, $SoD \wedge$ `can_acquire`$(u, p_1, t) \rightarrow \neg$`can_acquire`$(u, p_2, t)$
	A user in U cannot acquire different *permissions* in P at the same time.	
CACQ_ SoD$_{12}$	(I, P, CACQ$_{12}$, U, P)	$\forall u_1, u_2 \in U$, $SoD \wedge$ `can_acquire`$(u_1, p_1, t) \rightarrow \neg$`can_acquire`$(u_2, p_2, t)$
	No two users in U can acquire different *permissions* in P at the same time.	
CACQ_ SoD$_{13}$	(I, P, CACQ$_{13}$, U, P, R)	$\forall r \in R$, $\forall u$, $SoD \wedge$ `can_acquire` $(u, p_1, r, t) \rightarrow \neg$ `can_acquire` (u, p_2, r, t)
	Permissions in P cannot be *acquired* by a *user* in U through a role in R at the same time.	



SoD	Does not allow	SoD	Does not allow	SoD	Does not allow
CACQ-SoD$_1$	(*i*)	*CACQ-SoD*$_6$	(*vi*)	*CACQ-SoD*$_{11}$	(*ii*), (*vii*)
CACQ-SoD$_2$	(*ii*)	*CACQ-SoD*$_7$	(*iii*), (*v*)	*CACQ-SoD*$_{12}$	(*iv*), (*vi*)
CACQ-SoD$_3$	(*iii*)	*CACQ-SoD*$_8$	(*iv*), (*vi*)	*CACQ-SoD*$_{13}$	(*vii*)
CACQ-SoD$_4$	(*iv*)	*CACQ-SoD*$_9$	(*vi*), (*vii*)		
CACQ-SoD$_5$	(*v*)	*CACQ-SoD*$_{10}$	(*ii*), (*iv*)		

Fig. 5.6. Possibilistic permission acquisition for $U = \{u_1, u_2\}$, $P = \{p_1, p_2\}$ and $R = \{r_1, r_2\}$

Table 5.8.
Comparison with SoDs proposed in the literature

SH: Simon-Zurko's SoDs [Sim97]; AH: Ahn's SoDs [Ahn00].			**GTRBAC (non-temporal forms)**
1	**SZ**	**Strong *SSoD*** (*no user can be assigned to conflicting roles*)	*UAS-SoD*$_1$
	AH	**SSoD-CR** (*no user should be (implicitly and explicitly) assigned to conflicting roles, i.e., no user can-activate conflicting roles*)	*CACT-SoD*$_1$
2	**AH**	**SSoD-CP** (*a user cannot acquire conflicting permissions*)	*CACTs-SoD*$_9$
3	**AH**	***Variation of* 2** (2 + *conflicting permissions cannot be acquired through a role*)	*CACTs-SoD*$_9$ ∧ *CACQ-SoD*$_{4.1}$
4	**AH**	***Variation of* 1** (1 + *conflicting permissions cannot be acquired through a role + conflicting permissions cannot be assigned to a role*	*CACT-SoD*$_1$ ∧ *CACQ-SoD*$_{13}$ ∧ *PAS-SoD*$_2$
5	**AH**	**SSoD-CU** (1` + *conflicting users cannot be assigned to a role*)	*CACT-SoD*$_1$ ∧ *CACQ-SoD*$_1$ ∧ *UAS-SoD*$_2$
6	**AH**	***Variation*: (4) ∧ (5)**	**(4) ∧ (5)** above
7	**SZ**	*Simple* DSoD	*ACT-SoD*$_1$
	AH	***User-based* DSoD** (*Conflicting roles cannot be active at the same time for a user*)	
8	**AH**	***User-based* DSoD *with* CU** (*Conflicting roles cannot be active at the same time for a user*)	*Same as 7 but U is also a conflicting set*
9	**AH**	***Session-based* DSoD** (*Conflicting roles cannot be active at the same in the same user session*)	*ACT-SoD*$_4$
10	**AH**	***Session-based* DSoD with CR** (*Conflicting roles cannot be active at the same in the same user session*) *Only difference from 9 is that it has conflicting set of users*	*Same as 9 but R is also a conflicting set*
11	**SZ**	***Object-based* DSoD** (*no user may act upon a target that that user has previously acted upon*)	*Can be rephrased as: if a user acquires a permission then he cannot acquire it again. Post-condition constraint can be used here.*
12	**SZ**	***Operational* DSoD** (*no user may assume a set of roles that have capability for a complete business job*)	*Task oriented: if the task can be represented by at least two roles (sub-tasks) then it can be easily represented using UAS-SoD*$_1$ *or ACT-SoD*$_1$

13	**SZ**	***History-based* DSoD** (*no user is allowed to perform all the actions in a business task in the same target or collection of targets)*	*Comment similar to 12 can be made here, too.*
14	**SZ**	***Order-dependent* SoD** (*The roles must perform their actions in a particular order*)	*It can be expressed as a sequence of precedence constraints*
15	**SZ**	***Order-independent* SoD** (*Order does not matter as long as both happen*)	*Triggers $x \rightarrow y$ after Δt , $y \rightarrow x$ after Δt can be used to enforce this.*

5.3.6 Comparison with other SoD Constraints

Table 5.9 shows the correspondence between the major SoDs identified in the literature and the ones proposed here. First, we note that our SoDs take into account time, a factor which has not been considered earlier. Secondly, we can express all the SoDs in [Ahn00] with our constraint expressions or their combinations. The table also shows how our SoDs correspond to those proposed in [Sim97]. We note that the SoDs in rows 10 through 15 are more task oriented. However, with the help of the triggers and dependency constraints along with some transformation of the problem to map into RBAC domain, our framework can easily express them. Since previously identified SoDs are non-temporal, they correspond to the special case of the time-constrained SoDs proposed here, where $(I, P) = all$ and any occurrence of `U`, `R` or `P` in GTRBAC SoDs refer to the complete sets `Users`, `Roles` and `Permissions`. Furthermore, by using the GTRBAC status predicates, several new SoDs have been identified.

5.3.7 Various Interpretations of Time-based SoD Constraints

In the earlier section, we introduced periodicity constraints on SoDs. Note that these SoDs apply for each time instant in (I, P). However, we can allow other interpretations of these SoDs in the temporal dimension. Before presenting the different interpretations of a periodicity constraint (I, P, SoD), we first observe that for a single interval, say π, the constraint expression (π, SoD) can be interpreted in two ways, as defined for *weak* and *strong* forms. To illustrate these interpretations, we use the UAS-SoD_1, as shown in Table 5.9. For all other SoDs, the various forms of temporal constraints may be similarly defined.

The *weak form* (π, SoD_W) implies that within the specified interval there does not exist a time instant in which conflicting roles are assigned to the same user. (π, SoD_W)

does not, however, restrict conflicting roles being assigned to the same user at different time instants. The *strong* form (π, SoD_S) implies that within the specified interval, if there is an instant in which a role, say r, is assigned to a user, then at no other instants in π, the user can be assigned to a role that conflicts with r. By using these two forms, we obtain three semantic interpretations of the periodicity constraint (I, P, SoD) listed in the table. The *weak* form (I, P, SoD_W) implies that at each time instant in (I, P), a user should not be assigned to conflicting roles. (I, P SoD_W), however, allows a user to be assigned to two conflicting roles at different time instants. Note, this is the form of the SoDs that we presented in the earlier section.

Table 5.9.
Time-based SoD constraints

Interval constraint on SoD		**The following condition holds:** $\forall u \in U$, $\forall r_1, r_2 \in R$ *such that* $r_1 \neq r_2$
Weak	(π, UAS$_1$, U,R) as (π, SoD_X)	$\forall t \in \pi$, $SoD_W \wedge \texttt{u\_assigned}(r_1, u, t) \wedge \rightarrow \neg \texttt{u\_assigned}(r_2, u, t)$
Strong		$\exists\, t \in \pi$, $SoD_S \wedge \texttt{u\_assigned}(r_1, u, t) \rightarrow \neg\, \exists t \in \pi$, $\texttt{u\_assigned}(r_2, u, t)$
Periodicity constraints on SSoD		**The following condition holds**
Weak	(I, P, UAS$_1$, U,R) as (I, P, SoD_X)	$\forall t \in Sol(I, P)$, $SoD_W \wedge \texttt{u\_assigned}(r_1, u, t) \rightarrow \neg \texttt{u\_assigned}(r_2, u, t)$ *Furthermore, we see that* (I, P, $SoD_W \leftrightarrow \forall \pi \in \prod(I, P)$, ($\pi$,$SoD_W$)
Strong		(I, P, SoD_S) $\leftrightarrow \forall \pi \in \prod(I, P)$, ($\pi$, SoD_S)
Extended Strong		$\exists\, t \in Sol(I, P)$, $SoD_{ES} \wedge \texttt{u\_assigned}(r_1, u, t) \wedge \rightarrow \neg (\exists\, t \in Sol(I, P)$, $\texttt{u\_assigned}(r_2, u, t))$

The *strong* form (I, P, SoD_S) implies that for each recurring interval in (I, P), the *strong* form of interval constraint (π, SoD_S) applies. The *extended strong* form (I, P, SoD_{ES}) implies that there do not exist two or more time instants in (I, P) for which a user is assigned to conflicting roles. The *weak*, *strong* and *extended strong* forms also exist for duration constraints of the form ([I, $P|D$], D_x, SoD). The *weak*, *strong* and *extended* strong forms also exist for periodicity and duration constraints of the forms (I, P, $SSoD$) and ([I, $P|D$], D_x $DSoD$) on DSoD constraints.

5.4 Conclusion

We have presented constraints for the GTRBAC model including cardinality constraints, control flow dependency constraints, and separation of duty constraints. We used an evaluation function and a projection operator associated with a set of GTRBAC status predicates to build an elaborate framework for expressing cardinality constraints. GTRBAC's trigger has been extended so that more complex time-based past information can be captured. A set of control flow dependency constraints have been introduced using the trigger framework to enforce much stricter dependency constraints than those that can be expressed using triggers. We also showed that by generalizing to system events and conditions, the triggers and CFD framework provides an elaborate model for capturing context-based access requirements. Our approach to separation of duty constraints is based on the fact that the notion of conflict between elements in a set is often associated with another set. This allows us to consider SoDs that are of much finer-granularity. We have shown that the separation duty constraints identified in the literature can be easily expressed by a subset of our separation duty constraint expressions.

6. MINIMALITY OF GTRBAC CONSTRAINTS AND DESIGN ISSUES

An open issue for any model with a rich constraint language is its expressiveness and minimality. That is, it is important to determine whether the set of constraints the model contains is minimal. If the constraint model is not minimal, then a crucial issue is to determine whether the non-minimal model provides any practical benefits over the minimal model. It is possible that a model may not be minimal, and yet it is more advantageous to have all the constraints so that the model is more flexible and provides better practical benefits in terms of complexity and usability than the minimal model. Issues concerning expressive power and minimality for RBAC constraint languages is very relevant given the large variety of such languages that have been recently proposed [Cra03, Neu03].

In this chapter, we present an analysis framework for addressing the issue of the expressiveness and minimality of constraint languages for RBAC. We cast our analysis in the framework of the GTRBAC model, as this model has a very rich constraint language. It is easy to see that GTRBAC's rich set of constraints is not minimal. We thus show that there exists a minimal model that has a subset of constraint types defined in the GTRBAC model and yet has the same expressive power as the GTRBAC model. We show that the sets of different constraint types can be used to generate a family of GTRBAC models having the same expressive power.

An important issue, as mentioned above, is to determine if having a non-minimal set of constraints in the GTRBAC model is at all beneficial. In particular, we show that the GTRBAC model, although it is not minimal, has advantages in terms of complexity of specification and usability. Usability of the model is informally expressed as manageability and user convenience in policy specification. It also refers to maintaining a clear semantics among the constraints. For example, as we show in this dissertation, time based user-role assignment policies may be represented by temporal constraints on role enabling (the role may be newly created) in place of a timing constraint on the user-role assignment; however, by doing so, the semantics may be lost; i.e., we will be representing the fact that a user is scheduled to assume a particular role in a given interval of time *by* a semantics that the particular role is enabled at that interval of time. Based on our analysis, we provide a set of design guidelines aimed towards improving efficient and convenient use of various constraints to represent time-based RBAC policies.

To the best of our knowledge, there exists no work similar to ours that addresses the issue of the minimality of an RBAC model versus its complexity and usability.

6.1 Activity-Equivalent Family of GTRBAC Models

As shown in earlier chapters, the GTRBAC model allows specification of a large set of time-based constraints. A pertinent question is whether such an exhaustive set of temporal constraints is desirable at all, or if there is a minimal set of constraint types that have the same expressive power as the set containing all the constraint types introduced in this dissertation. In this section, we show formally that the set of GTRBAC constraint types is not minimal. By introducing the notion of *activity-equivalence* or *a-equivalence*, we show that there exists a minimal set of constraint types that have an expressive power equivalent to the set of all the GTRBAC constraint types. However, we show through an extensive analysis that even though such a minimal set exists, the set of GTRBAC constraints provides better alternatives for representing access constraints. Such alternatives allow one to favor user convenience and lower complexity of representation over the use of the minimal set of constraints. Furthermore, the large set of constraints in the GTRBAC model makes it flexible and allows an appropriate choice of specification, enhancing the usability of the model.

6.1.1 Minimality of GTRBAC

Given a GTRBAC system, note that we call the set containing all the constraints its Temporal Constraint and Activation Base (*TCAB*). A *TCAB* Γ can be represented as (C_{URp}, C_{Rp}, C_{PRp}, C_{URd}, C_{Rd}, C_{PRd}, C^{a}_{dr}, C^{a}_{dur}, C^{a}_{mr}, C^{a}_{mur}, C^{a}_{nr}, C^{a}_{nur}, C^{a}_{nmr}, C^{a}_{nmur}, C_{tr}, C_{c}) (see Table 3.1, Chapter 3). In the discussion below, we use a shorter version, such as $T = (C_{Rp}, C_{URp})$, when only C_{Rp} and C_{URp} are nonempty sets of constraints. The behavior of a GTRBAC system depends on T, the set of users `Users`, the set of roles `Roles`, the set of permissions `Permissions`, and the role hierarchy *RH*. Therefore, we can use the tuple (T, `Users`, `Roles`, `Permissions`, *RH*) to indicate a GTRBAC configuration. We will also use the notation ($u \overset{Cf}{\underset{t}{\Rightarrow}} p$) to read "*u acquires permission p at time t under* *configuration Cf'*. Next, we define the notion of *a-equivalence* between two GTRBAC configurations.

Definition 6.1.1 (**Activity-equivalence or a-equivalence**): *Given a GTRBAC system with two configurations* Cf_1 = (T_1, `Users`, `Roles`$_1$, `Permissions`, RH_1) and Cf_2 = (T_2, `Users`, `Roles`$_2$, `Permissions`, RH_2), *the configurations* Cf_1 *and* Cf_2 *are said to be a-equivalent* (*written as* $Cf_1 \approx Cf_2$) *if, for all pairs* (u, p) *such that* $u \in$ `Users`, $p \in$ `Permissions`, *the following condition holds*: $(u \overset{Cf_1}{\underset{t}{\Rightarrow}} p)$ *iff* $(u \overset{Cf_2}{\underset{t}{\Rightarrow}} p)$. *Furthermore, if* $Cf_1 \approx Cf_x$ *and* $Cf_x \approx Cf_2$, *then* $Cf_1 \approx Cf_2$ (*transitivity*).

The *a-equivalence* between two configurations of a GTRBAC system indicates that a user can perform the same accesses under the two configurations. Hence, by replacing configuration Cf_1 by Cf_2, we do not change the accesses that are allowed for each individual user. It is to be noted that *a-equivalence* does not necessarily imply policy equivalence as we consider the same set of users and permission. Policy equivalence would mean that at all times the two configurations follow the same rule. Our goal here is to show different configurations of roles and constraints allowing the same set of permissions being acquired by the same set of users, and analyze the complexities of these configurations.

Next, we show that the set of GTRBAC constraint types is not minimal; i.e., some constraint types can be removed without reducing the expressive power of the GTRBAC constraints system. For example, the temporal constraints on assignments can be expressed by using temporal constraints on roles. Using *a-equivalence* between GTRBAC configurations, we will show that there is a minimal representation that uses only periodicity and duration constraints on roles, and the *per-role* activation constraints. However, we will still need the *default* assignments that simply assign users or permissions to roles without specifying any temporal restriction. Although *default* assignments can be considered as a special case of periodicity constraints, we will consider it a special constraint type (non-temporal constraint) represented by C_d. Next we introduce two algorithms that can be used to generate an *a-equivalent* configuration of a given GTRBAC configuration with the temporal constraints on user-role and role-permission assignments removed.

`TransformPR` shown in Fig. 6.1 takes in a GTRBAC configuration and produces an *a-equivalent* configuration with all the temporal constraints on role-permission assignments removed. Similarly, the algorithm `TransformUR` shown in Fig. 6.2 produces a new configuration that is *a-equivalent* to the input configuration Cf_{in}, with all user-role assignments and *per-user-role* activation constraints removed.

```
Algorithm TransformPR
Input :$Cf_{in}$; Output : $Cf_{out}$
1. $Cf_{out}$ ={$T'$, Users, Roles′, Permissions, $RH'$}= $Cf_{in}$={$T$, Users, Roles, Permissions,
   $RH$};
2. FOR each $c = (X, pr$:assign/deassign $p$ to $r) \in T$, where $X = \{(I, P), ([(I, P)|, D_x], D)\}$ DO
3.     Create a unique role $r_i$;
4.     Replace all occurrences of {$X$, $pr$:assign/deassign $p$ to $r$} by
           {$X$, $pr$:enable/disable $r_i$} in $T'$
5      Add default assignment assign/deassign $p$ to $r_i$ to $T'$
6.     FOR each trigger $TR \in T'$, where $TR$ = "$E_1, \ldots, E_n, C_1, \ldots, C_k \rightarrow pr{:}E_{n+1}$ after $\Delta t$" DO
7.       Replace $TR$ by $TR'$ = "$E'_1, \ldots, E'_n, C'_1, \ldots, C'_k \rightarrow pr{:}E'_{n+1}$ after $\Delta t$",
           such that,($i$ =1 to $n$+1, $j$ = 1 to $k$) &
8.         IF ($E_i$== "assign/deassign $p$ to $r$") THEN $E'_i$ = "enable/disable $r_i$" ;
9.         ELSE $E'_i = E_i$ ;
10.        IF ($C_j$ == "assigned/deassigned $p$ to $r$" ) THEN
                     $C'_j$ = "enabled/disabled $r_i$" ;
12.        ELSE $C'_j = C_j$;
13.      ENDFOR
14.      Roles' = Roles' ∪ {$r_i$};
15.      FOR each role $r_j \in$ Roles such that {$r \geq_s r_j$} DO
16.            $RH' = RH' \cup \{r_i \geq_s r_j\}$; $RH' = RH' - \{r \geq_s r_j\}$
17.      ENDFOR
18.      $RH' = RH' \cup \{r \geq_s r_i\}$;              // Note: all are strongly restricted I-hierarchy
19. ENDFOR
20. RETURN $C_{out}$;
```

Fig. 6.1 Algorithm `TransformPR`

The following two lemmas formally show that the transformation done by each algorithm is correct. To maintain readability, we include the proofs of these lemmas and the other formal statements presented later in this chapter in appendix C.

Lemma 6.1 (Correctness of `TransformPR`): *Given an input configuration* Cf_{in}, *algorithm* `TransformPR` *produces* C_{out} *such that there are no temporal role-permission assignments in* Cf_{out}, *and* $Cf_{\text{in}} \approx Cf_{\text{out}}$.

```
Algorithm TransformUR
Input : $Cf_{in}$; Output : $Cf_{out}$
1.  $Cf_{out}$ = $Cf_{in}$ (i.e., {$T'$, Users, Roles′, Permissions, $RH'$}={$T$, Users, Roles,
    Permissions, $RH$}); $S = \emptyset$;
2.  FOR each $c$ = ($X$, $pr$:assign/deassign $u$ to $r$) $\in T$, where $X = \{(I, P), ([(I, P)|, D_x], D)\}$ DO
3.    Create a unique role $r_i$;  $S = S \cup (u, r, r_i)$      // function $getSu_i(S, u, r)$ used in line 24 returns $r_i$
4.    Replace all occurrences of {$X$, $pr$:assign/deassign $u$ to $r$} by
          {$X$, $pr$:enable/disable $r_i$} in $T'$
5.    Add default assignment "assign/deassign $u$ to $r_i$ to $T'$ "
6.    FOR each trigger $TR \in T'$, where $TR$ = "$E_1, \ldots, E_n, C_1, \ldots, C_k \rightarrow pr{:}E_{n+1}$ after $\Delta t$" DO
7.      Replace $TR$ by $TR'$ where $TR'$ == "$E'_1, \ldots, E'_n, C'_1, \ldots, C'_k \rightarrow pr{:}E'_{n+1}$ after $\Delta t$" such that
8.        IF ($E_i$=="assign/deassign $u$ to $r$") THEN $E'_i$ := "enable/disable $r_i$" ;
9.        ELSE $E'_i := E_i$ ;
10.           IF ($C_j$="assigned/deassigned $u$ to $r$") THEN $C'_j$:="enabled/disabled $r_i$";
12.           ELSE $C'_j := C_j$;
13      ENDFOR
14.     Roles' = Roles' $\cup \{r_i\}$;
15.     FOR each role $r_j \in$ Roles such that $\{r_j \succcurlyeq_s r\}$ DO
16.           $RH' = RH' \cup \{r_j \succcurlyeq_s r_i\}$; $RH' = RH' - \{r_j \succcurlyeq_s r\}$;        // Strongly restricted A-hierarchy
17.     ENDFOR
18.     $RH' = RH' \cup \{r_i \succcurlyeq_s r\}$;
19. ENDFOR
20. // Handle all the per-role-activation constraints
21. FOR  each pair ($u$, $r$) such that there is an activation constraint ($X$, $Y_u$, $u$, active$_{UY}$ $r$) $\in T'$
22.       where $X \in \{(I, P), D\}$, $Y_u \in \{D_{uactive}, D_{umax}, N_{uactive}, D_{umax}\}$and
23.           active$_{UY}$ = {active$_{UR\_total}$, active$_{UR\_max}$ ,  active$_{UR\_n}$, active$_{UR\_con}$} DO
24.   IF ($r_i$:=$getSu_i(S, u, r)$ == NIL) THEN Create a unique role $r_i$,  // $getSu_i(S, u, r)$= NIL implies there
25.   FOR each $c$ = ($X$, $Y_u$, $u$, active$_{UY}$ $r$) $\in T'$  DO                 // was no u, r assignment in line 2
26.       Let $c'$ =($X$, $Y_u$, active$_{UY}$ $r_i$);
27.       Replace $c$ in $T'$ by $c'$ where $c'$ =($X$, $Y_u$, active$_{UY}$ $r_i$);      //Note that old c will not be in T'
28.       Replace all occurrences of "enable $c$" by "enable $c'$ "
29      ENDFOR
30      IF  ($r_i$ was created in Line 24) THEN
31.       Role' = Role' $\cup \{r_i\}$;
32.       FOR each role $r_j \in$ Roles such that $\{r_j \succsim_s r\}$DO
33.           $RH' = RH' \cup \{r_j \succsim_s r_i\}$; $RH' = RH' - \{r_j \succsim_s r\}$;
34.       ENDFOR
35.       $RH' = RH' \cup \{r_i \succcurlyeq r\}$;                   // Note: all are strongly restricted IA-hierarchy
36.       Replace per-role activation constraint by (0, active$_{R\_n}$ $r$) in $T'$
37. ENDFOR
38. RETURN $Cf_{out}$;
```

Figu. 6.2 Algorithm TransformUR

Lemma 6.2 (Correctness of `TransformUR`): *Given an input configuration Cf_{in}, algorithm* `TransformUR` *produces Cf_{out} such that there are no temporal user-role assignments and per-user-role activation constraints in Cf_{out}, and $Cf_{\text{in}} \approx Cf_{\text{out}}$.*

We use the following notion of *minimal constraint set* (MCS) to express the fact that there is an *a-equivalent* configuration that has the minimum number of constraint types.

Definition 6.1.2 (Minimal Constraint Set): *Let $MCS(T)$ be the set of constraint types in TCAB T, and $CS = \{Cf_1, Cf_2, .. Cf_n\}$ be an a-equivalent set of configurations such that $Cf_i = (T_i,$ `Users`, `Roles`$_i$, `Permissions`, $RH_i)$ for $i = 1, 2, \ldots, n$. We say that $MCS(T_i)$ is the* minimal constraint set (MCS) *of CS for $i \in \{1, 2, \ldots, n\}$, if there exists no other configuration $Cf_j = (T_j,$ `Users`, `Roles`$_j$, `Permissions` , $RH_j)$, such that $i \neq j$ and $MCS(T_j) \subset MCS(T_i)$.*

The definition implies that a minimal constraint set is the one that has the least number of temporal constraint types. Note that the role set and hierarchy may be altered to reduce the number of constraint types. Next, we present the minimality result for a GTRBAC system, which is expressed by the following theorem.

Theorem 6.1 (Minimality of GTRBAC): *Let Cf_1 be a* GTRBAC *configuration, such that $\{C_d, C_{Rp}, C_{Rd}, C^a{}_r, C_{tr}, C_c\} \subseteq MCS(T_1)$; there exists a* GTRBAC *configuration Cf_2 such that*:

a. $Cf_1 \approx Cf_2$, and
b. *$MCS(T_2) = \{C_d, C_{Rp}, C_{Rd}, C^a{}_r, C_{tr}, C_c\}$, (note that $C^a{}_r$ represents per-role activation constraint types), and*
c. *$MCS(T_2)$ is a minimal constraint set,*

Theorem 6.1 shows that the set of GTRBAC constraints is not minimal, because a set of default assignments, periodic and duration constraints on role enabling (disabling), and *per-role* activation constraints and triggers can be used to represent any access policy that GTRBAC constraints can represent. We can see from the transformation algorithms that replacing temporal constraints on assignments with temporal constraints on roles, in general, increases the number of roles and the complexity of role hierarchy, which may not be desirable. This is because algorithms `transformPR` and `transformUR` create a new role for each temporal assignment that they replace. This may not be very intuitive and efficient as it means there will be as many new roles as there are temporal assignments. This results in a worst case where a role is created for each user (or permission) in the system. A more intuitive and practical approach would be to create a least number of roles such that the enabling/disabling intervals for them are non-overlapping. For example, if there is a Doctor role and each of the *n* users are assigned to it for either day time or night time (or both), then, instead of creating *n* new roles, we can simply create DayDoctor and NightDoctor roles and assign all the *n* users to one or the other (or both). Thus, to create such temporally non-overlapping roles, we must first divide *n* periodic expressions into temporally non-overlapping set of periodic expressions such as, *Daytime* and *Nighttime*. We next provide formal definitions and algorithms to generate such a disjoint set of temporal roles by generating disjoint periodicity expressions associated with temporal assignment constraints.

6.1.2 Operations on Periodicity Expressions

In this section, we first introduce the formal notions of *containment, equivalence, overlapping,* and *disjunction* operations between a pair of periodic expressions. Note that an arbitrary set of intervals can be represented by a periodic expression. This is possible because each such expression can be formulated, at the worst as a periodic expression that lists every starting point and the smallest calendar.

Definition 6.1.2 (Containment/Equivalence/Overlapping/Disjunction of periodic Expressions) : *Let $PE_1=(I_1, P_1)$ and $PE_2=(I_2, P_2)$ be two periodic expressions, then*

1. *PE_1 is said to be* contained *in PE_2 (written as $PE_1 \subseteq PE_2$), if the following conditions hold*

   $$\textit{for all } t,\ (t \in Sol(I_1, P_1) \rightarrow t \in Sol(I_2, P_2)) \wedge$$
   $$\exists t\ (t \in Sol(I_2, P_2) \rightarrow t \notin Sol(I_1, P_1))$$

2. *PE_1 and PE_2 are said to be* equivalent *(written as: $PE_1 = PE_2$) if*

   $$\textit{for all } t,\ (t \in Sol(I_1, P_1) \leftrightarrow t \in Sol(I_2, P_2));$$

3. *PE_1 and PE_2 are said to be* overlapped *(written as $PE_1 \otimes PE_2$) if the following condition holds*:

   $\exists\ t_1, t_2$ such that

   - $(t_1, t_2) \in \Pi(P_1)$, i.e. *t_1, t_2 are end points of an interval* in P_1, *and*
   - $\exists\ t_a, t_b$,

- $t_1 < t_a < t_b < t_2$, *and*
- $(t_a, t_b \in Sol(I_1, P_1) \rightarrow (t_a \in Sol(I_2, P_2) \wedge t_b \notin Sol(I_2, P_2)) \vee (t_b \in Sol(I_2, P_2) \wedge t_a \notin Sol(I_2, P_2)))$;

4. *PE_1 and PE_2 are said to be* disjoint (*written as* $PE_1 \lozenge PE_2$), *if, for all t_1, t_2 such that* $(t_1 \in Sol(I_1, P_1) \wedge t_2 \in Sol(I_2, P_2)$, *the following condition holds*:

$$((t_x \in Sol(I_y, P_y) \rightarrow (t_x \in ESol(I_2, P_2) \wedge (t_x \in ESol(I_1, P_1))), \textit{for } (x, y) \in \{(1, 2), (2, 1)\}$$

where $ESol(I, P)$ is the set of end-points of intervals in (I, P) such that if $t \in ESol(I, P)$ then $t \in Sol(I, P)$.

A set of periodic expressions is said to be *disjoint* if the periodic expressions are pair-wise disjoint, else it is said to be *non-disjoint*. Similarly, a set of periodic expressions is said to be *equivalent* if all the period expressions are equivalent to each other.

Fig. 6.3 shows some examples of these relations. Note that the fourth part of the definition implies that if only the endpoints of intervals of two periodic expressions are common, then they are considered disjoint. Ideally, we want to compute a disjoint set of periodic expressions that is minimal.

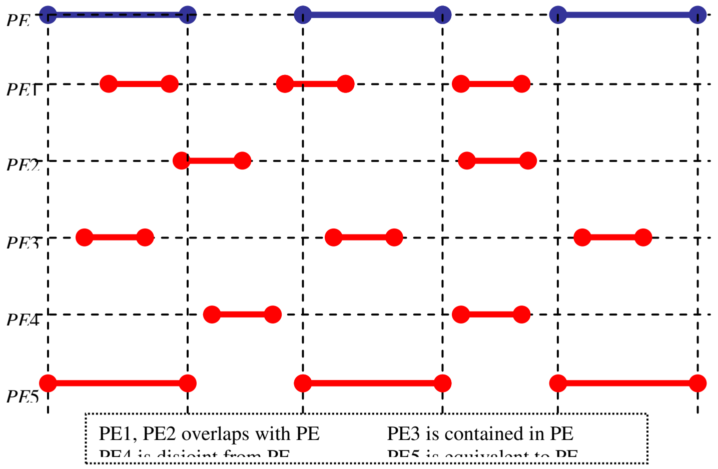


Fig. 6.3. Temporal relations between a pair of periodic expressions

The next definition expresses the notion of *minimal disjoint set* (MDS) over a set of periodic expressions.

Definition 6.2.4 (Minimal Disjoint Set): *Let $PE = \{PE_1, PE_2, \ldots, PE_n\}$ be a set of arbitrary periodic expressions. The* minimal disjoint set (MDS) over *PE is the least set of* disjoint *periodic expressions, MDS_{PE}, defined as*:

MDS_{PE} = $\min_m\{PE'_i \mid 1 \leq i \leq m\}$, *such that the following conditions hold,*

1. $PE'_1 \cup PE'_2 \cup \ldots \cup PE'_m = PE_1 \cup PE_2 \cup \ldots \cup PE_n$,
2. *for all* $1 \leq i \leq m$, $1 \leq j \leq n$ either $PE'_i \subseteq PE_j$ or $PE'_i \cap PE_j = \emptyset$.

In the definition, the first condition says that the MDS contains a disjoint set of periodic expressions containing all time instants that are contained in all the original set of periodic expressions PE_is. The second condition ensures that each PE'_i contains time instants that entirely belong to a PE_j. Associated with MDS, we define *minimal subset* (MS) *of a periodic expression over* a MDS as follows:

Definition 6.1.5 (Minimal subset (MS) for a periodic expression over a MDS): *Let* $MDS_{PE} = \min_m\{PE'_i \mid 1 \leq i \leq m\}$ *be a minimal disjoint set over periodic expressions* $PE = \{PE_1, PE_2, \ldots, PE_n\}$; *the* minimal subset (MS) for a periodic expression $PE_j \in PE$ over the MDS_{PE} *is the set* $MS_{PEj}(MDS_{PE}) = \{PE'_{\pi 1}, PE'_{\pi 2}, \ldots, PE'_{\pi k}\} \subseteq MDS_{PE}$, $1 \leq k \leq m$ *such that,*

- $\{\pi 1, \pi 2, \ldots, \pi k\} \subseteq \{1, 2, \ldots, m\}$,
- *for each* $t \in Sol(PE_j)$, *there is exactly one* $i \in \{\pi 1, \pi 2, \ldots, \pi k\}$ *such that* ($t \in Sol(PE'_j)$), and
- $|\{\pi 1, \pi 2, \ldots, \pi k\}|$ or *k is minimum.*

We see that MS of a periodic expression PE_i of PE is a subset of MDS_{PE} that collectively contains all the time instants of PE_i. Before presenting an example for MDS and MS, we first show some formal properties related to the computation of MDS and MS. We write ${}^{i}MDS_{PE}$ to mean MDS of the first *i* periodic expressions of *PE.*

Lemma 6.3 (MDS for two expressions): Let (PE_1, PE_2) be a pair of non-equivalent and non-disjoint periodic expressions; The following holds*:*

a. *if* ($PE_i \subseteq PE_j$) *then, for* $(i, j) \in \{(1, 2), (2, 1)\}$, *there exist periodic expressions* PE_x, PE_y *such that* $MDS_{PE} = \{PE_x, PE_y\}$. *Furthermore*, $PE_x = PE_i$ and $PE_y = PE_j - PE_i$.
b. *if* ($PE_i \otimes PE_j$) *then for* $(i, j) \in \{(1, 2), (2, 1)\}$, *there exist periodic expressions* PE_x, PE_y, PE_z *such that* $MDS_{PE} = \{PE_x, PE_y, PE_z\}$. *Furthermore,* $PE_x = PE_i \cap PE_j$, $PE_y = PE_j - PE_x$ *and* $PE_z = PE_i - PE_x$.

```
Algorithm PairMDS
Input: $PE_1$, $PE_2$
Output: MDS of $PE_1$, $PE_2$
IF ($PE_1 = PE_2$) THEN RETURN{$PE_1$};
IF ($PE_1 \Diamond PE_2$) THEN RETURN {$PE_1$, $PE_2$};
IF ($PE_1 \subseteq PE_2$) THEN  // as per Lemma 6.3(a)
    $PE_x = PE_1$;
    $PE_y = PE_2 - PE_x$;
    RETURN {$PE_x$, $PE_y$};
IF ($PE_2 \subseteq PE_1$) THEN  // as per Lemma 6.3(a)
    $PE_x = PE_2$;
    $PE_y = PE_1 - PE_x$;
    RETURN {$PE_y$, $PE_x$};
IF ($PE_1 \otimes PE_2$) THEN // as per Lemma 6.3(b)
    $PE_x = PE_1 \cap PE_2$;
    $PE_y = PE_2 - PE_x$;
    $PE_z = PE_1 - PE_x$;
    RETURN {$PE_x$, $PE_y$, $PE_z$}
END
```

```
Algorithm ComputeMDS
Input: $PE_1$, $PE_2$, …, $PE_n$
Output: MDS of $PE_1$, $PE_2$, …, $PE_n$
// Assume that $PE = \{PE_1, PE_2, …, PE_n\}$
$S = \varnothing$; $MDS = \varnothing$;
IF $|PE| = 1$ THEN RETURN $PE$;
IF $|PE| = 2$ THEN RETURN PairMDS($PE_1$, $PE_2$);
IF $|PE| > 2$ THEN
    $MDS$ = ComputeMDS($PE_1$, $PE_2$, …, $PE_{n-1}$);
    Let $MDS$ computed be ($PE'_1$, $PE'_2$, …, $PE'_{m1}$);
    FOR $i$ = 1 to $m1$ DO
        $PairMDS$ = PairMDS($PE_i$, $PE_n$);
        IF $|PairMDS| = 1$ THEN
            return $MDS$;
        IF $|PairMDS| = 2$ THEN
            Let $PairMDS$ computed be ($PE'_x$, $PE'_y$);
             $S = S \cup \{PE'_x\}$;
        ELSEIF  $|PairMDS| = 3$ THEN
            Let $PairMDS$ be ($PE'_x$, $PE'_y$, $PE'_z$);
             $S = S \cup \{PE'_x, PE'_z\}$;
    ENDFOR
    Let S computed be ($PE''_1$, $PE''_2$, …, $PE''_{m2}$);
    $PE''_{m2+1} = PE_n - (PE''_1 \cup PE'' \cup … \cup PE''_{m2})$;
    IF ($PE''_{m2+1} = \varnothing$) THEN
        $MDS = (PE''_1, PE''_2, …, PE''_{m2}, PE''_{m2+1})$;
    ELSE
        $MDS = (PE''_1, PE''_2, …, PE''_{m2})$;
    RETURN $MDS$
END
```

Fig. 6.4 Algorithms PairMDS and ComputeMDS

Figure 6.4 depicts the algorithms used to compute the MDS. Algorithm `PairMDS` computes MDS for a pair of periodic expressions. We note that when the two expressions are equivalent, the MDS contains a single periodic expression, which can be either of the original expressions. Similarly, when the expressions are disjoint, the MDS contains both the periodic expressions. Algorithm `computeMDS` repeatedly calls `PairMDS` and recursively builds the MDS by first finding the MDSs of smaller sizes. It uses the inductive technique used to prove Lemma 6.4. The following formal results show that `computeMDS` computes the MDS of a set of periodic expressions.

Lemma 6.4 (MDS for *n* periodic expressions): *Given a non-equivalent and non-disjoint set of periodic expressions* $PE = \{PE_1, PE_2, ..., PE_n\}$, *there exist periodic expressions* $PE'_1, PE'_2, ..., PE'_m$ *such that* $MDS_{PE} = \{PE'_1, PE'_2, ..., PE'_m\}$.

Theorem 6.2 (MDS using `computeMDS`)*:* *Given an arbitrary set of periodic expressions* $PE = \{ PE'_1, PE'_2, ..., PE'_n\}$, *there exist periodic expressions* $PE'_1, PE'_2, ..., PE'_m$, *such that*

a. $MDS_{PE} = \{PE'_1, PE'_2, ..., PE'_m\}$ *and*

b. *For PE as input, algorithm* `computeMDS` *produces* MDS_{PE}

Theorem 6.2 shows that we can construct a MDS of an arbitrary set of periodic expressions. As we will show later, this will help us in finding a minimum set of roles corresponding to a set of periodic expressions such that they are minimal and disjoint in terms of their enabling intervals. We also derive the following two corollaries:

Corollary 6.2.1 (Bounds for size of MDS): *Given a set of periodic expressions* $PE = \{PE_1, PE_2, ..., PE_n\}$, *the algorithm* computeMDS *produces* $MDS_{PE} = \{PE'_1, PE'_2, ..., PE'_m\}$ *such that if* $s_n = |MDS_{PE}|$ *then* $1 \leq s_n \leq (2^n - 1)$.

Corollary 6.2.2 (Bounds for size of MS): *Given a set of periodic expressions* $PE = \{PE_1, PE_2, ..., PE_n\}$ *and* $MDS_{PE} = \{PE'_1, PE'_2, ..., PE'_m\}$ *produced by algorithm* computeMDS, *if* $p_n = |MS_{PE1}| + |MS_{PE2}| + ... + |MS_{PEn}|$, *then* $n \leq p_n \leq n2^{n-1}$.

We illustrate the notion of MDS and MS, and the computation of MDS by algorithms `computeMDS` and `pairMDS` with the following example.

`Example 6.2.1`: To simplify notation, we consider the *Daytime* of the days listed for a periodic expression. For example, if PE = {Sun}, we mean the interval (9am, 9pm) or daytime of a Sunday. Let PE_A = {Sun, Mon, Tue, Wed, Thu, Fri}, PE_B = {Sun, Tue}, PE_C = {Sun, Tue, Thu, Fri}, PE_D = {Sun, Mon, Tue, Wed, Sat}, PE_E = {Thu, Fri}. The following steps illustrate the computation of $MDS_{\{PE\text{A}, PE\text{B}, PE\text{C}, PE\text{D}, PE\text{E}\}}$ using algorithm `computeMDS`.

1. $MDS_{\{PE\text{A}, PE\text{B}\}}$ = {PE'_1, PE'_2} = {{Sun, Tue}, {Mon, Wed, Thu, Fri}}(as $PE_B \subseteq PE_A$)

2. $MDS_{\{PE\text{A}, PE\text{B}, PE\text{C}\}}$ = MDS of { PE'_1, PE'_2, PE_C} = MDS of {{Sun, Tue}, {Mon, Wed, Thu, Fri}, {Sun, Tue, Thu, Fri}}

   Here,

   - MDS of (PE'_1, PE_C} = {PE'_{x1} ={Sun, Tue}, PE'_{y1} = {Thu, Fri}(as $PE'_1 \subseteq PE_C$)},
   - MDS of {PE'_2, PE_C}= {PE'_{x2} ={Thu, Fri}, PE'_{y2} ={Sun, Tue}, PE'_{z2}= {Mon, Wed}}(as $PE'_2 \otimes PE_C$)
   - $S = \{PE'_{x1}, PE'_{x2}, PE'_{z2}\}$
   - $PE'_{x1} \cup PE'_{x2} \cup PE'_{z2}$ = {Sun, Mon, Tue, Wed, Thu, Fri}.
   - $PE''_4 = PE_C - (PE'_{x1} \cup PE'_{x2} \cup PE'_{z2}) = \varnothing$

   Therefore, $MDS_{\{PE\text{A}, PE\text{B}, PE\text{C}\}}$ = {PE''_1, PE''_2, PE''_3} = {{Sun, Tue}, {Thu, Fri}, {Mon, Wed}}

3. $MDS_{\{PE\text{A}, PE\text{B}, PE\text{C}, PE\text{D}\}}$ = MDS of {PE''_1, PE''_2, PE''_3, PE_{D}}

   = MDS of {{Sun, Tues}, {Thu, Fri}, {Mon, Wed}, {Sun, Mon, Tue, Wed, Sat}}
   Here,

   - MDS of {PE''_1, PE_D} = {PE'_{x3} ={Sun, Tue}, PE'_{y3} = {Mon, Wed, Sat} (as $PE''_1 \subseteq PE_D$),
   - MDS of {PE''_2, PE_D} = {PE'_{x4} = {Thu, Fri}, PE'_{y4} = {Sun, Mon, Tue, Wed, Sat}} (as $PE''_2 \lozenge PE_D$),
   - MDS of {PE''_3, PE_D} = {PE'_{x5} = {Mon, Wed}, PE'_{y5} = {Sun, Tue, Sat}} (as $PE''_3 \subseteq PE_D$),
   - S = {PE'_{x3}, PE'_{x4}, PE'_{x5}}

- $PE'_{x3} \cup PE'_{x4} \cup PE'_{x5}$ = {Sun, Mon, Tue, Wed, Thu, Fri},
- $PE'''_4 = PE_D - (PE'_{x3} \cup PE'_{x4} \cup PE'_{x5})$ = {Sat};

Therefore, $MDS_{\{PEA, PEB, PEC, PED\}} = \{PE'''_1, PE'''_2, PE'''_3, PE'''_4\}$ ={{Sun, Tue}, {Thu, Fri}, {Mon, Wed}, {Sat}}

4. $MDS_{\{PEA, PEB, PEC, PED, PEE\}}$ = MDS of $\{PE''_1, PE''_2, PE''_3, PE'''_4, PE_E\}$

   = MDS of {{Sun, Tue}, {Thu, Fri}, {Mon, Wed}, {Sat}, {Thu, Fri}}

   Since $PE_E = PE''_2$, $MDS_{\{PEA, PEB, PEC, PED, PEE\}}$ = $\mathrm{MDS}_{\{PEA, PEB, PEC, PED\}}$

   = $\{PE'''_1, PE'''_2, PE'''_3, PE'''_4\}$ = {{Sun, Tue}, {Thu, Fri}, {Mon, Wed}, {Sat}}

Also, we see that,

1. $MS_{PEA}(MDS_{\{PEA, PEB, PEC, PED, PEE\}}) = \{PE'''_1, PE'''_2, PE'''_3\}$.
2. $MS_{PEB}(MDS_{\{PEA, PEB, PEC, PED, PEE\}}) = \{PE'''_1\}$
3. $MS_{PEC}(MDS_{\{PEA, PEB, PEC, PED, PEE\}}) = \{PE'''_1, PE'''_2\}$.
4. $MS_{PED}(MDS_{\{PEA, PEB, PEC, PED, PEE\}}) = \{PE'''_1, PE'''_3, PE'''_4\}$.
5. $MS_{PEE}(MDS_{\{PEA, PEB, PEC, PED, PEE\}}) = \{PE'''_2\}$.

```
Algorithm TransformMDS
Input  :$Cf_{in}$
Output : $Cf_{out}$
1.  $Cf_{out}$ = {$T'$, Users, Roles′, Permissions, $RH'$}
          = $Cf_{in}$={$T$, Users,Roles, Permissions, $RH$};
2.  FOR each $r \in$ Roles DO
3.      Let $PE = \{PE_1, PE_2 \ldots, PE_n\}$ and $U = \{u_1, u_2 \ldots, u_n\}$ be such that ($PE_i$, assign $r$ to $u_i$) $\in T'$;
4.      Compute $MDS$ of $PE$;  Let the computed $MDS = \{PE'_1, PE'_2 \ldots, PE'_n\}$;
5.      FOR $i$ = 1 to $n$ DO
6.          Compute $MS_{PEi}$ for $PE_i$
7.      ENDFOR
8.      FOR each $PE'_i \in MDS$ DO
9.          Create a unique role $r_i$;
10.         FOR all $u_k \in U$ such that $PE'_i \in MS_{PEk}$ DO
11.             Add default assignment (assign $r_i$ to $u_k$) in $T'$.
12.             Add constraint ($PE'_i$, enable $r_i$ ) in $T'$.
13.             Remove constraint ($PE_i$, assign $r$ to $u_i$)  from $T'$;
14.             Roles' = Roles' $\cup \{r_i\}$;
15.             $RH' = RH' \cup \{r_i \succcurlyeq_s r\}$;          // Note: Strongly restricted A-hierarchy
16.         ENDFOR
17.     ENDFOR
18. //ENDFOR
```

Fig. 6.5. Algorithm TransformMDS

We next present an algorithm that produces an a-equivalent configuration for a given GTRBAC system by removing the temporal constraints on user-role assignments. The following theorem establishes its correctness:

Theorem 6.3 (Correctness of `TransformMDS`**):** *Given an input configuration* Cf_{in} *with only periodicity constraints on user-role assignments, algorithm* `TransformMDS` *produces a configuration* Cf_{out} *such that the following holds:*

1. $Cf_{in} \approx Cf_{out}$*, and*
2. Cf_{out} *has no periodicity user-role assignment constraints.*

Here, we have considered only the presence of the periodicity constraints on *user-role* assignment. If we allow the presence of *per-role* constraints, algorithm `TransformMDS` can be extended easily to handle it by introducing *per-role* constraints on the newly created roles.

6.2 Complexity of Specification and Design Issues

The complexity of a GTRBAC system may have different components. Foremost among them is the number of roles. Typically, we do not want an unmanageable number of roles in a system. Another component is the number of temporal constraints. Then we have the complexity incurred by a hierarchy. Finally, we have the *default assignments* with no timing constraint. In *default assignments*, the only check needed is the membership check, for example, to determine whether a particular user is assigned to a role or not. Thus, we can expect temporal assignments to introduce additional complexity compared to an RBAC system without temporal constraints because it involves, besides checking for membership, ensuring the temporal validity of a membership. To simplify our discussion on trade-offs and complexity issues, we first develop a family of GTRBAC models that have equivalent expressive power, based on the results in the previous section, and we then investigate the potential benefits of a model at a higher level of family hierarchy over those at the lower level. For our analysis of complexity of policy specification, we use the complexity parameters notation shown in Table 6.1. Note that we have left H to just indicate that some hierarchy processing overhead is present.

The *minimality* result in the previous section shows that the minimal model of GTRBAC system is the one that includes the following components: *per-role* activation constraint, *periodicity* and *duration* constraints for role-enabling/disabling, constraint enabling, and triggers, as shown in Table 6.2. Figure 6.6 shows the *minimal model* as $GTRBAC_0$ at level 0.

Table 6.1
Complexity parameters and notation used

Complexity parameter		**Notation Description**
Role	R	*n.R indicates n roles* **(Note**: *we write 1.R simply as R)*
Default (simple) assignment	S	*n.S indicates n default assignments*
Enabling time constraints on	T_r	*n.*T_r *indicates n periodicity/duration constraints on (n) roles*
Temporal constraints on assignments	T_{ur} , T_{rp}	*n.*T_{ur} *(n.*T_{rp}*) indicates n periodicity/duration constraints on (n) user- role (role-permissions) assignment*
Activation time constraints on	A_{ur} , A_r	*n.*A_{ur} *(n.*A_r*) indicates n per-user-role (per-role) activation time*
Hierarchy	H	*n.H indicates ln hierarchical relations*

Table 6.2.
GTRBAC Family of models

Level	**Model**	**Constraint Set**
2	$GTRBAC_2$	$T = T_{1,A} \cup T_{1,U} \cup T_{1,P}$
1	$GTRBAC_{1,P}$	$T_{1,P} = T_0 \cup \{C_{PRp}, C_{PRd}\}$
	$GTRBAC_{1,U}$	$T_{1,U} = T_0 \cup \{C_{Urp}, C_{Urd}\}$
	$GTRBAC_{1,A}$	$T_{1,A} = T_0 \cup \{C^a_{dur}, C^a_{mur}, C^a_{nur}, C^a_{nmur}\}$
0	$GTRBAC_0$ *Minimal*	$T_0 = \{C_d, C_{Rp}, C_{Rd}, C^a_r, C_{tr}, C_c\}$

At level 1, we have three different models, each of which adds a new type of constraints to the constraint set of $GTRBAC_0$. $GTRBAC_{1,A}$ represents the model having all the temporal constraints of $GTRBAC_0$ plus the *per-user-role* activation constraints. Similarly, $GTRBAC_{1,U}$ represents the model having all the temporal constraints of $GTRBAC_0$ plus the *user-role* assignment constraints, whereas, $GTRBAC_{1,P}$ represents the model having all the temporal constraints of $GTRBAC_0$ plus the *role-permission* assignment constraints. At level 2, we have the overall $GTRBAC_2$ model that contains all the temporal constraints. We note that we can have other models between level 1 and level 2 that represent models representing the pairs of models at level 1 models. However, for our analysis, we adopt this simpler hierarchy. We also keep in mind that, according to the results in the previous section, all the models in Fig. 6.6 have the same expressive power; i.e, these models can be used to generate *a-equivalent* configurations.

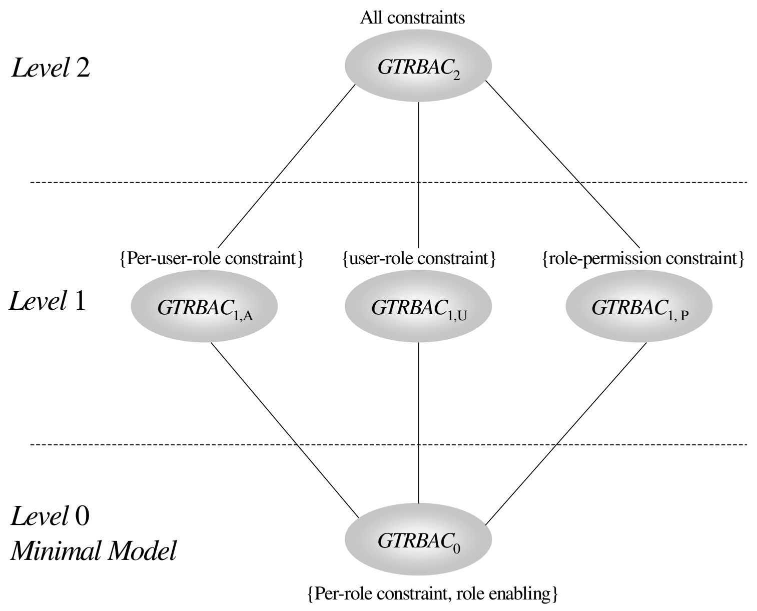


Fig. 6.6. GTRBAC family of models

Next, we show through analysis that it is advantageous to use a model at a higher level in terms of *user-convenience*, *clarity of semantics,* and *complexity of representation*. Our analysis will focus on the advantages and disadvantages of using a Level 1 model compared against that of the Level 0, the *minimal model*.

6.2.1 Role Enabling *vs.* Role Assignment Constraints

We have shown in an earlier section that all temporal constraints on *user-role* and *role-permission* assignments can be transformed into the temporal constraints on roles. However, such a transformation may result in a large number of roles and/or produce inconvenient or complex access control structures. In this section, we look at various design alternatives for choosing constraints on role enablings and assignments. We do this by comparing the complexity of representation using a Level 1 model against those of various representations using the *minimal model* for expressing the same set of access requirements.

As we can see, in `TransformUR` (see Fig. 6.2), the transformation from temporal constraints on *user-role* assignments to the temporal constraints on roles is similar to the transformation from temporal constraints on *role-permission* assignments to the temporal constraints on roles in `TransformUR` (see Fig. 6.2), except for the difference in hierarchy relation. That is, in the first case, the newly created roles are made senior of the original role, whereas in the second case, the original role is made the senior of the new roles. Because of this similarity, we will focus mainly on the *user-role* assignments, as similar results can be obtained for the *role-permission* assignments. Also, algorithm `TransformUR` transforms both the *periodicity* and *duration* constraints in a similar way; i.e., each such constraint is replaced by a new role. Hence, the complexity analysis we apply for *periodicity* constraints will apply for the duration constraints as well. We will, therefore, focus on the *periodicity* constraints and point out important considerations related to *duration* constraints whenever they apply.

A temporal constraint on *user-role* assignment states that the user can activate a role in the specified periods or for a specified duration, provided the role is enabled. Instead of using a temporal constraint on *user-role* assignment (the user is still assigned to the role using default assignment), we enforce the desired access control by using temporal constraints on role enabling. Next, we will present the complexity issues related to the representations of a set of access requirements using $GTRBAC_0$ and $GTRBAC_{1,U}$ models. For our purpose, we use the following example:

Example 6.3.1: Let us assume that there is a DayDoctor role in a hospital. Five doctors *A, B, C, D*, and *E* are assigned to this role in the periods given by the periodic expressions PE_A, PE_B, PE_C, PE_D, and PE_E of Example 6.2.1. We assume that we have the $GTRBAC_{1,U}$ representation of these constraints (hence, there are no activation constraints). We will also look at two different representations using the $GTRBAC_0$ model, which we will denote as the $GTRBAC_0^1$ and $GTRBAC_0^2$ representations.

***$GTRBAC_{1,U}$* representation**: For each doctor, a periodicity constraint on his assignment to the DayDoctor role is specified using periodic expressions shown in Fig. 6.7(a). For example, for doctor *A*, PE_A is the periodic expression used – i.e., there is a constraint (PE_A, assign DayDoctor to *A*) in *T*. Similarly, assignment constraints for the remaining doctors with the respective periodic expressions are specified.

***$GTRBAC_0^1$* representation**: In this alternative, we use algorithm TransformUR with the above $GTRBAC_{1,U}$ representation as the input. Accordingly, a role is created for each constraint, and a *default* assignment and a periodicity constraint on the new role are added. For instance, for a constraint (PE_A, assign DayDoctor to A), a role, say r_A, is created and a new constraint (PE_A, enable r_A) is added, whereas the constraint (PE_A, assign DayDoctor to A) is replaced by *default* assignment (assign r_A to A). Similarly, all other temporal assignments are replaced. This is depicted in Fig. 6.7(b).

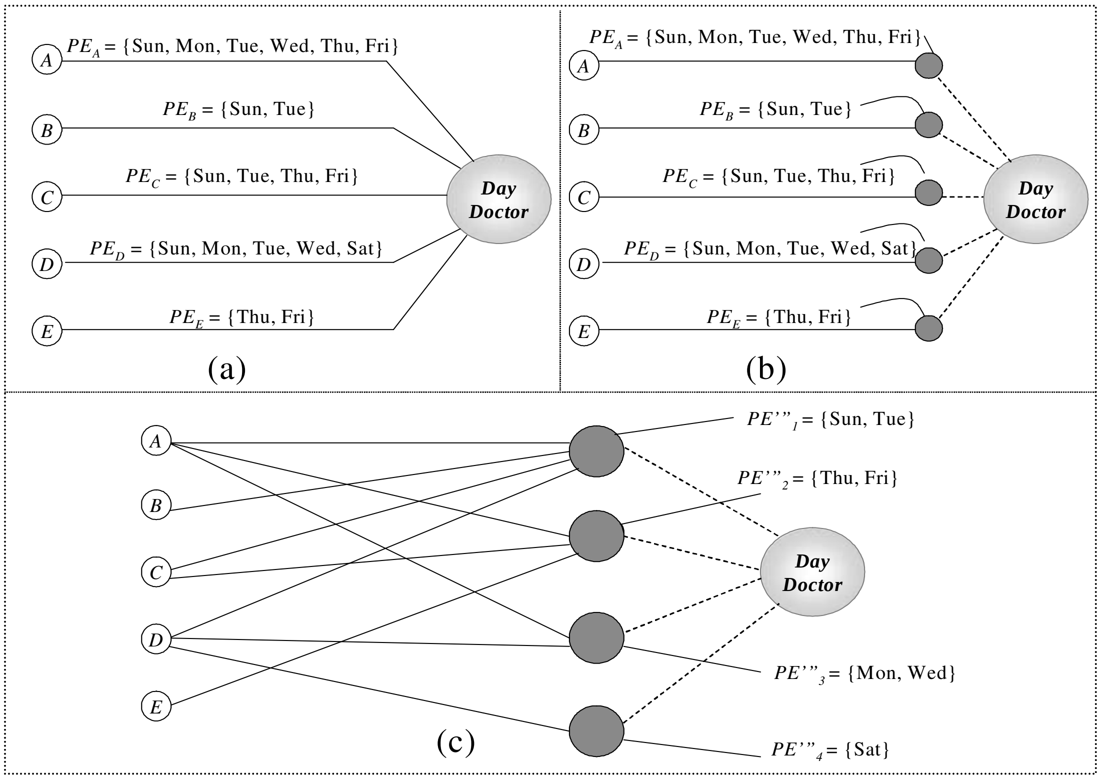


Fig. 6.7. Access requirements of Example 6.2.1 using (a) $GTRBAC_{1,\,U}$ representation (b) $GTRBAC_0^1$ representation and (c) $GTRBAC_0^2$ representation

$GTRBAC_0^2$ representation: This alternative uses the *minimal disjoint* set approach using algorithm `TransformMDS` (see Fig. 6.5). The result is as shown in Figure 6.7(c). From Example 6.2.1, we know that $MDS_{\{PEA,\ PEB,\ PEC,\ PED,\ PEE\ \}}$ = {PE'''_1, PE'''_2, PE'''_3, PE'''_4}= {{Sun, Tue}, {Thu, Fri}, {Mon, Wed}, {Sat}}. A role is created for each periodic expression of $MDS_{\{PEA,\ PEB,\ PEC,\ PED,\ PEE\ \}}$. As $|MDS_{\{PEA,\ PEB,\ PEC,\ PED,\ PEE\ \}}|$ = 4, four new roles are created, and a *periodicity* constraint is added for each new role. The ith new role is associated with the ith periodic expression of $MDS_{\{PEA,\ PEB,\ PEC,\ PED,\ PEE\ \}}$. Each doctor is assigned to a set of new roles that corresponds to the periodic expressions that constitutes *MS* of the periodic expression associated with him; e.g., since $MS_{PEC}(MDS_{\{PEA, PEB, PEC, PED, PEE\}}) = \{PE'''_1, PE'''_2\}$, doctor C is assigned to the new roles that correspond to periodic expressions PE'''_1 and PE'''_2.

In the complexity expressions we will neglect original role and any activation constraints associated with it, as they remain the same in all the representations. We can see that for the $GTRBAC_{1,\,U}$ representation, the complexity is: $n.T_{UR}$. The following theorem establishes formally the complexities of the alternative representations using the $GTRBAC_0$ model.

Theorem 6.4 (Complexity expressions for $GTRBAC_0^1$ and $GTRBAC_0^2$ representations): *Let n be the number of users assigned to a role r, and let PE =* $\{PE_1, PE_2 \ldots, PE_n\}$ *be the set of the periodic expressions in the user-role assignment constraints corresponding to n users assigned to r; i.e., there is a* (PE_i, assign r to u_i) *for each i* = 1 *to n; Then, the general complexity expressions for the alternative representations* $GTRBAC_0^1$ *and* $GTRBAC_0^2$ *are as follows*:

1. $GTRBAC_0^1$ *representation*: $n.S + n.T_R + n.R + n.H,$
2. $GTRBAC_0^2$ *representation*: $p_n.S + s_n.T_R + s_n.R + s_n.H;$

where $p_n = |MS_{PE1}(MDS_{PE})| + |MS_{PE2}(MDS_{PE})| + \ldots. + |MS_{PEn}(MDS_{PE})|$, *and* $s_n = |MDS_{PE}|$.

Based on this, we get the following complexities for each representation of `Example 4.3`, which is shown in Figure 6.7.

$GTRBAC_{1,\,U}$ representation: $5.T_{UR}$.

$GTRBAC_0^1$ representation: $5.S + 5.T_R + 5.R + 5.H$

$GTRBAC_0^2$ representation: $10.S + 4.T_R + 4.R + 4.H$

(using algorithm `TransformUR`)

We see that, for the above example, the $GTRBAC_{1,\,U}$ representation is the best in terms of complexity – it has the least number of roles, no hierarchy overhead, and no default assignments; furthermore, it is simple and intuitive to use and hence very convenient. The main difference between the $GTRBAC_0^1$ and $GTRBAC_0^2$ representations is that the latter always produces roles that are temporally disjoint. The $GTRBAC_0^1$ representation associates one role for each user for whom there is a temporal assignment constraint. However, the $GTRBAC_{1,U}$ representation may not be the best for all cases as we show below.

It can be seen that the complexities of the $GTRBAC_{1,U}$ and $GTRBAC_0^1$ representations each remain the same for a given n, irrespective of how periodic expressions are pair-wise related. The complexity of the $GTRBAC_0^2$ representation, for a given n, depends on *MS* and *MDS* of *PE*. The following corollary states the effect of MS and MDS on the complexity of the $GTRBAC_0^2$ representations.

Corollary 6.4.1(Complexity cases for $GTRBAC_0^2$ representations): *Let n be the number of users assigned to a role r, and let PE =* $\{PE_1, PE_2 \ldots, PE_n\}$ *be the set of the periodic expressions in the user-role assignment constraints corresponding to n users; i.e., there is a* (PE_i, assign r to u_i) *for each* $i = 1$ *to n; then:*

1. *if* $PE_i \neq PE_j$, *for all i, j pairs such that* $1 \leq i, j \leq n$ (*i.e., they are pair-wise disjoint*), *then the following holds true*:

   complexity of $GTRBAC_0^2$ = *complexity of* $GTRBAC_0^1$

   In other words, the complexity of $GTRBAC_0^2 = n.S + n.T_R + n.R + n.H$
2. *if* $PE_i = PE_j$, for all *i, j pairs such that* $1 \leq i, j \leq n$ (i.e., *they are pair-wise equivalent*), *then the following holds true*:

   the complexity of $GTRBAC_0^2 = n.S + T_R + R + H$.
3. *the worst case for* $GTRBAC_0^2$ *is:* $n2^n.S + 2^n.T_R + 2^n.R + 2^n.H$.

The first part of the corollary shows that when all the periodic expressions associated with the *user-role* assignments are disjoint, the $GTRBAC_0^2$ representation is the same as the $GTRBAC_0^1$ representation. When $PE_i = PE_j$, for all $i, j = 1$ to n and n is large, $GTRBAC_0^2$ is substantially better than the $GTRBAC_{1,U}$ representation**,** based on the fact that *temporal constraints* incur more processing cost than *default* assignments. The hierarchy overhead introduced by an extra role can be expected to be negligible to a membership check associated with a *default* assignment for a large *n*. Furthermore, the new role created can be combined with the original role, if that does not introduce extra complicacies, thus removing the *hierarchy overhead*.

However, the worst case for the $GTRBAC_0^2$ representation, as indicated by the third part of corollary 6.4.1, is $O(2^n)$ in the number of new roles created, temporal constraints on roles, and new hierarchical relations, and $O(n2^n)$ in the number of default assignments.

Based on the above observation, we can summarize the following design guidelines.

1. The $GTRBAC_{1,\ U}$ representation is preferable to $GTRBAC_0^{\ 1}$ representations as its complexity in terms of the number of roles, the number of temporal constraints, and/or the number of hierarchical relations created, is always better.
2. The $GTRBAC_{1,\ U}$ and $GTRBAC_0^{\ 1}$ representations may result in using temporal constraints that can be avoided because of some common periodic expressions. For example, there may be a large number of doctors who need to use the role DayDoctor at *daytime*, making *daytime* a common period for many users. Using the $GTRBAC_0^{\ 1}$ representations in such cases also results in the same temporal periodicity constraints on different roles, as algorithm TransformUR does not attempt to reduce constraints based on common periodicity expressions. The $GTRBAC_0^{\ 2}$ is a good solution in all such cases where some user-role assignments have common periodic expressions. If all the periodic expressions are equivalent, then it produces a single role and all users are assigned to that role, as indicated by the results in the second part of corollary 6.4.1. Theorem 6.4 and corollary 6.4.1 show that $GTRBAC_0^{\ 2}$ is advantageous when the *MS* set of each periodic expression is very small (the smallest case is when it has one member, as indicated by the second part of the corollary; i.e., when all the periodic expressions are equivalent). Furthermore, we want a small *MCS* set, as it determines the number of new roles created.

   Similarly, if all the periodic expressions are pair-wise disjoint, then the $GTRBAC_0^{\ 2}$ representation becomes equivalent to the $GTRBAC_0^{\ 1}$ representation as shown by the first part of corollary 6.4.1.
3. The $GTRBAC_{1,\ U}$ representation is very flexible with respect to access specification since it supports temporal constraints on *user-role* assignments, in addition to the constraints on role enabling. For example, we can have the following constraints:

   ([*Mon*, *Wed*, *Fri*], `assign` John `to` DayDoctor)

   ([*Tue*, *Thur*], `assign` John `to` NightDocotor).

   ([10am, 3pm], `assign` Greg `to` DayDoctor).

   By using the above constraints, we can keep the roles that have static temporal enabling times fixed in the system and express individual user requirements using periodicity constraints. Here, DayDoctor and NightDoctor roles are more or less fixed in the system and, as illustrated, users are assigned to it as required. Furthermore, these are semantically much clearer than the $GTRBAC_0^{\ 1}$ and $GTRBAC_0^{\ 2}$ forms with only role enabling time temporal constraints.
4. Note that if there are *per-user-role* activation constraints, using the $GTRBAC_0^{\ 2}$ representations may not be advantageous. For example, in the example above (Fig.

6.7(c)), each user is assigned to multiple new roles. In such a case, if there had been a *per-user-role* constraint for each user, we would have needed to take extra steps during its transformed representation. Here, we note that algorithm `TransformMDS` creates an A_s-hierarchy (*strongly restricted activation* hierarchy) between the new roles and the original role. So if we leave the *per-user-role* constraints untouched; i.e., in the transformed representation, if the *per-user-role* is still specified in terms of the original role, then the new representation is still valid, as the users assigned to the new role will have to explicitly activate the new role. However, it is neither intuitive nor convenient to keep track, as the users are only implicitly assigned to the original role. Therefore, in the presence of *per-user-role* activation constraints, $GTRBAC_0^1$ *and* $GTRBAC_{1,U}$ provide more intuitive and convenient representations than $GTRBAC_0^2$.

5. Unlike periodicity constraints, duration constraints are somewhat inflexible in terms of being replaced (for example, replacing *user-role* assignment by role enabling). As duration constraints have non-deterministic start times, such constraints depend on some other events. Such dependencies often have some application semantics, and even though it may be possible to replace a duration constraint on *user-role* assignment, as in the case of periodicity constraints, care must be taken to ensure that the dependency semantics is not hindered. We illustrate this with an example:

`Example 6.3.2:` Consider Manager and Employee roles in an office and assume that the Employee role is enabled on weekdays from 9am to 5pm, whereas the Manager role is enabled everyday. At other times, the Employee role is enabled only if Mr. Smith, the manager who is also the owner, has activated his Manager role. This can be expressed using the following trigger:

`activate` Manager `for` Smith → `enable` Employee (t_1)

Suppose Smith wants to allow John, an employee in his office, to work on Saturday and Sunday when he is also working, for at most four hours. Then he can do that by adding the following constraints:

([Sat], 4 hours, `assign` John `to` Employee) (c_1)
`activate` Manager `for` Smith → `assign` John `to` Employee (t_2)
`de-activate` Manager `for` Smith → `disable` Employee (t_3)

When Smith activates the Manager role on Saturday, it enables Employee using trigger t_1 and assigns John to the Employee role using trigger t_2. Because of the constraint c_1 active at the time, the assignment gets restricted to four hours during which John can work.

In this case, if we try to use the duration constraint on the Employee role instead, the implicit dependency between the activation of the Manager role and allowing John to work is lost.

6. We note that the transformation such as in $GTRBAC_0^2$ is not possible for *user-role* assignment with duration constraints. Although there may be common duration values associated with different *user-role* assignments, there is an inherent dependency semantics associated with each duration constraint that relates it to a trigger or a constraint enabling expression.
7. Except for the discussion presented in guideline 4, all apply to *role-permission* assignments too.

Thus, we can see, except for some cases, where $GTRBAC_0^2$ is better in terms of complexity of representations. $GTRBAC_{1,\text{U}}$ gives the best representational form, in terms of complexity, user convenience, and semantic clarity.

6.2.2 Per-role *vs.* Per-user-role Activation Constraints

In this section, we compare the use of the $GTRBAC_0$ and $GTRBAC_{1,\text{A}}$ models to express the same set of activation constraints. For simplicity, we assume that $GTRBAC_{1,\text{A}}$ has only *total active duration* constraints in addition to constraints in $GTRBAC_0$. The same kinds of analysis apply to other activation constraints. In the complexity expressions, we use A_{UR} to mean *per-user-role* activation constraint, and A_R to mean *per-role* activation constraints as shown in Table 6.1. In addition, we will not include the original role and any of its associated *per-role* constraints in the complexity expressions. For the discussions that follow, we use the following example:

Example 6.3.3: Let *A, B, C, D* and *E* be the users subscribing 100, 100, 100, 250, 50 hours of active time per week respectively from a Video Library. A straightforward representation of these constraints using the $GTRBAC_{1,\text{A}}$ model is shown in Fig. 6.8(a) (we will refer to this as the $GTRBAC_{1,\text{A}}^s$ representation). To represent these constraints using $GTRBAC_0$, we can use the part of algorithm `TransformUR` that removes *per-user-role* activation constraints (or we can simply assume that there are no temporal assignment constraints and run the `TransformUR` on this configuration). Such a representation, later referred to as the *$GTRBAC_0^s$* representation, is shown in Fig. 6.8 (b).

From the example, it is clear that the straightforward representation of a set of *n* *per-user-role* constraints for *n* users assigned to a role (a *per-role* constraint on the role may or may not be present) using the two models incurs the following costs:

(*i*) *$GTRBAC_{1,A}^s$* representation: $n.A_{UR}$

(*ii*) *$GTRBAC_0^s$* representation: $n.A_R + n.R + n.H$
(*using algorithm* `TransformUR`)

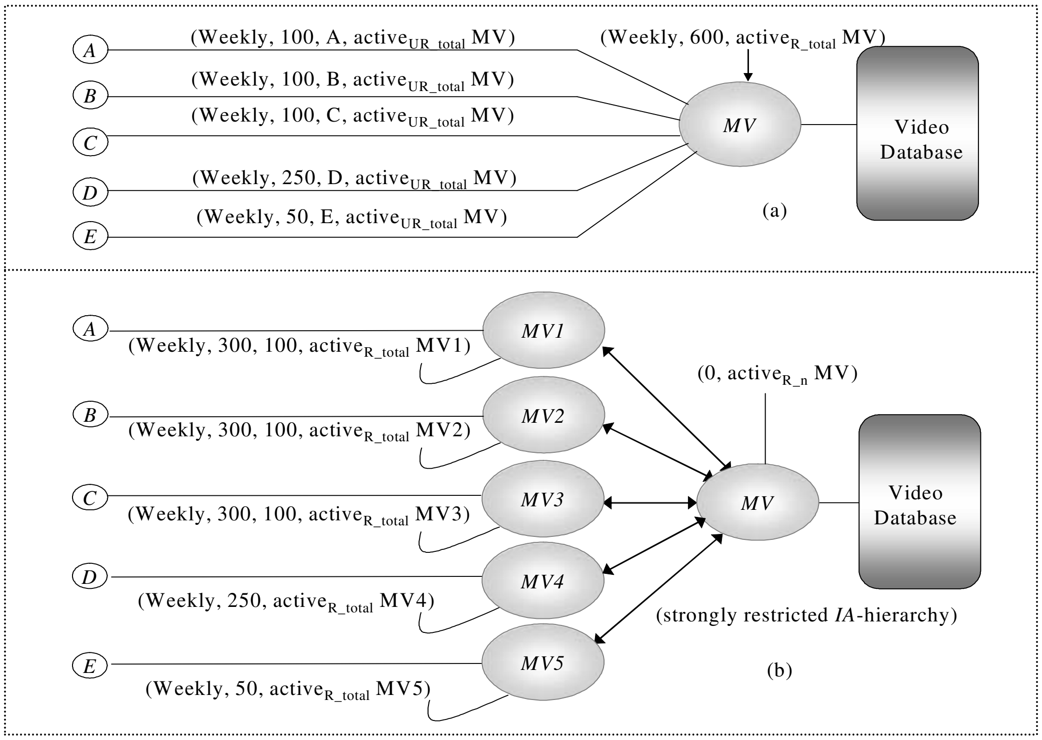

A
B
C
D
E
(Weekly, 100, A, active$_{UR\_total}$ MV)
(Weekly, 100, B, active$_{UR\_total}$ MV)
(Weekly, 100, C, active$_{UR\_total}$ MV)
(Weekly, 250, D, active$_{UR\_total}$ MV)
(Weekly, 50, E, active$_{UR\_total}$ MV)
(Weekly, 600, active$_{R\_total}$ MV)
MV
Video Database
(a)
(Weekly, 300, 100, active$_{R\_total}$ MV1)
(Weekly, 300, 100, active$_{R\_total}$ MV2)
(Weekly, 300, 100, active$_{R\_total}$ MV3)
(Weekly, 250, active$_{R\_total}$ MV4)
(Weekly, 50, active$_{R\_total}$ MV5)
MV1
MV2
MV3
MV4
MV5
(0, active$_{R\_n}$ MV)
MV
Video Database
(strongly restricted IA-hierarchy)
(b)

Fig. 6.8. Requirements of Example 6.3.3 using (a) *$GTRBAC_{1,A}^s$* representation (b) *$GTRBAC_0^s$* representation (algorithm `TransformUR`) on a *$GTRBAC_{1,A}^s$* configuration

Note that we did not include the original role and any *per-role* constraints on it, as they will always remain the same. We can see that between the two cases illustrated above, the *$GTRBAC_{1,A}^s$* model gives a better representation in terms of the reduced number of roles. The total number of activation constraints is the same in both. However, we want to know if these give the best representations. We observe that in Fig. 6.8(a), the users *A*, *B,* and *C* have the same *per-user-role* access requirements and hence, can possibly be expressed as one *per-role* constraint. Similarly, we see that in

Fig. 6.8(b), *MV1*, *MV2* and *MV3* have the same *per-role* constraint values, which can possibly be combined. The following theorem formally shows that such reduction in complexity can be achieved, when there are duration constraint values that are common.

Theorem 6.5 (Complexity expression for *$GTRBAC_0$* and *$GTRBAC_1$* representations): *Let n be the number of users assigned to role r, $D = \{d_1, d_2,, \ldots d_n \mid d_i$ be the total active duration that the i^{th} user is allowed over role r$\}$, $D_m = \{d'_1, d'_2,, \ldots d'_m\} \subseteq D$ be the set of distinct elements of D, and $C_m(d)$ be the number of times d occurs in D; then the complexities of the following two representations are as follows:*

1. *$GTRBAC_{1,A}$ representation*: $(n_x - n_y).A_{UR} + n_y.A_R + c.(b.n_y + 1).R + c.H$
2. *$GTRBAC_0$ representation*: $n_x.A_R + n_x.R + n_x.H$

where,

- $n_x = |D_m|$ *and* $n_y = |D'|$ *, such that*
    - *i.* $D' \subseteq D_m$ *,and*
    - *ii. if* $d \in D'$ *then* $C_m(d) > 1$
- $b = 1$ *if* $(n > n_x)$; $b = 0$ *otherwise*,
- $c = 1$ *if* $(n > n_x > 0)$; $c = 0$ *otherwise*.

The complexities of the previously mentioned representations of the constraints as shown in figures 6.8(a) and 6.8(b) can be easily derived by forcing each element in *D* to be considered as unique. Note that the values of some d_is in *D* may be equal. In that case, *D*' is non-empty and contains those elements of *D* that occur more than once.

$n_x = |D_m| = n$, $n_y = 0$, $b = 0$ *and* $c = 0$

and hence, the complexities are as follows:

$GTRBAC_{1,A}^s$ representation:

$= (n_x - n_y).A_{UR} + n_y.A_R + c.(b.n_y + 1).R + c.H = n.A_{UR}$ (*same as* (*i*))

$GTRBAC_0^s$ representation:

$= n_x.A_R + n_x.R + H = n.A_R + n.R + n.H$ (*same as* (*ii*))

Thus, for Example 6.3.3, we have the following complexities, as given by Theorem 6.5 (the constraints are as shown in Figure 6.8):

$GTRBAC_{1,A}^s$ representation: $5.A_{UR}$

$GTRBAC_0^s$ representation: $5.A_R + 5.R + 5.H$

Here, we see that, in the $GTRBAC_0^s$ representation, there are mainly temporal constraints for the five new roles and one for the old role. In the $GTRBAC_{1,A}^s$ representation, there is just one role with one per-role constraint (the original role and hence not included) but there are five *per-user-role* and one *per-role* constraints.

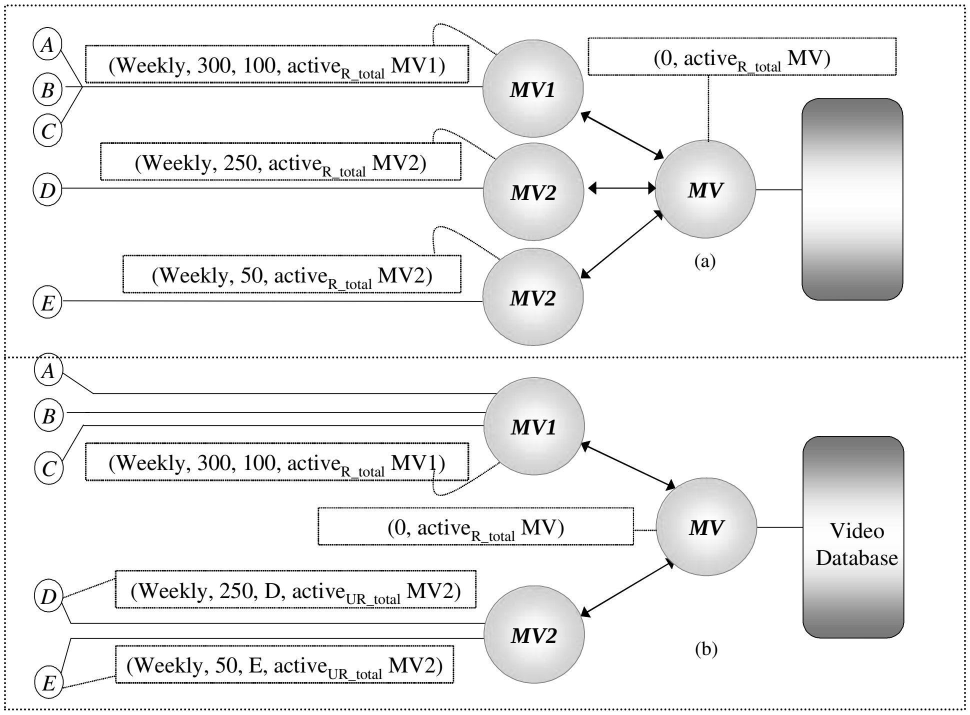

A
B
C
(Weekly, 300, 100, active$_{R\_total}$ MV1)
MV1
(0, active$_{R\_total}$ MV)
(Weekly, 250, active$_{R\_total}$ MV2)
D
MV2
MV
(Weekly, 50, active$_{R\_total}$ MV2)
E
MV2
(a)
A
B
MV1
C
(Weekly, 300, 100, active$_{R\_total}$ MV1)
(0, active$_{R\_total}$ MV)
MV
Video Database
D
(Weekly, 250, D, active$_{UR\_total}$ MV2)
MV2
(b)
E
(Weekly, 50, E, active$_{UR\_total}$ MV2)

Fig. 6.9. Constraints of `Example 6.3.3` (a) using $GTRBAC_0$ representation (b) using $GTRBAC_{1,A}$ representation

Fig. 6.9 illustrates the general constraint design that combines common total active duration constraints as used in Theorem 6.5. Here, we get n_x=3 as D_m={50, 100, 250}, n_y=1 as D'={100}, b = 1 and c = 1. Therefore, the complexities are:

$GTRBAC_0$ representation: $= n_x.A_R + n_x.R + H = 3.A_R + 3.R + 3.H$

$GTRBAC_{1,A}$ representation: $= (n_x - n_y).A_{UR} + n_y.A_R + c.(b.n_y + 1).R + c.H$

$= 2.A_{UR} + 1.A_R + 2.R + H$

We can summarize the following guidelines based on the above observation:

1. If there are many users having a common active duration requirement, then using a role and a constraint that specify both the *total* and *default* duration constraint minimizes both the number of roles and the number of temporal constraints, as shown by Theorem 6.5.

2. If the expected requirements for active durations for individual users vary substantially from user to user, the $GTRBAC_{1,A}$ representation is preferable.
3. If flexibility is needed, using *per-user-role* constraints (and hence the $GTRBAC_{1,A}$ representation) is better. For example, if the users *A, B, C, D* and *E* request a different active duration every week, then the use of *per user-role* constraints is more appropriate.
4. In some cases, a hybrid approach utilizing constraints on both *per role* and *per user-role* will give a more efficient representation, as shown by Figure 6.9(b). This is the $GTRBAC_{1,A}$ representation as per Theorem 6.5.

Thus, we see that the $GTRBAC_{1,A}$ representation has distinct advantages over the $GTRBAC_0$ representation.

6.3 Conclusions

In this chapter, we have addressed the issue of the expressiveness of the GTRBAC model. As our major contribution, we showed through exhaustive analysis of the minimality of the GTRBAC model that a comprehensive set of GTRBAC constraints can provide distinct advantages over minimal GTRBAC model in terms of user convenience and the complexity of constraint representation. This is practically a significant result as it shows that although the GTRBAC model is not minimal, its constraints set provides constraint designers with flexibility and intuitive choices over various constraint expressions as well as much better and less complex representations in certain cases. Based on these results, we outlined some design guidelines that can assist constraint designers in choosing more convenient and less complex constraint expressions.

7. X-GTRBAC – AN XML BASED GTRBAC POLICY SPECIFICATION LANGUAGE

In this chapter, we present X-GTRBAC, an XML based GTRBAC policy specification language. The X-GTRBAC language design and implementation is a joint work in [Bha03] and therefore, only an overview of the language and system architecture is provided in this dissertation. The implementation details can be found in [Bha03].

We first motivate the choice of using XML as the policy specification language. We then present an overview of the features of X-GTRBAC. In particular, the language has been extended with the capability to support credential-based role assignment. Such a capability allows X-GTRBAC to be used in a generic web-based environment where the users are not known *a priori*. Furthermore, X-GTRBAC also includes some features to allow role mapping between multiple security domains. While there are many challenging technical issues related to secure interoperability in a multidomain environment that are yet to be adequately addressed [Gon96, Hos92a, Hos92b, Jos01a, Jon95, Kuh95, Tar97a, Tar97b, Vas94, Vuo01], such a provision in X-GTRBAC allows, at least at a specification level, support for specifying metapolicies that govern the permissible interdomain access mapping between a pair of domains.

7.1. Motivation for an XML Based Policy Specification Language

The eXtensible Markup Language (XML) [Ber01b, URLa], which evolved from a simple subset of SGML [Iso86], is considered as the most promising technology for information interchange across heterogeneous, distributed domains [Vuo01]. XML can be considered as a meta-language that allows users to design their own markup language, using some agreed-upon vocabulary for application-specific purposes. XML offers this capability by providing an extensible set of markup tags for creating custom documents, as well as a set of related technologies for their interpretation.

XML documents have logical as well as physical structures [Ber99a, Ber01b, Vuo01]. An XML document is physically composed of *entities*, which may include other *entities* and *attributes*. Each document has a *root* or *document entity*. Each XML document is logically composed of declarations, elements, comments, character references, and processing instructions. These logical components are all indicated in a document by explicit markup. Additionally, elements may contain attributes.

```
<enterprise>
 <depts>
  <engineering>
    <engg_manager job_id= “EM”>
      <name>John</name>
      <level>5</level>
      <…> .. </..>
    </engg_manager>
    <product_engineer job_id=”PE”>
      <name>Paul</name>
      <shift>1</shift>
      <…> .. </..>
    </product_engineer>
  </engineering>
 </depts>
</enterprise>
```

Fig. 7.1. An XML instance document

```
<xs:schema>
 <xs:element name =“enterprise”>
  <xs:complexType>
   <xs:element name = “depts”>
    <xs:complexType>
     <xs:element name = “engineering”>
      <xs:complexType>
       <xs:element name = “engg_manager”>
        <xs:complexType>
         <xs:attribute name =“job_id” type=”xs:string”/>
         <xs:element name = “name” type=”xs:string”/>
         <xs:element name = “level” type=”xs:string”/>
       <xs:/complexType>
      <xs:/element>
      <xs:element name = “product_engineer”>
       <xs:complexType>
        <xs:attribute name =“job_id” type=”xs:string”/>
        <xs:element name = “name” type=”xs:string”/>
        <xs:element name = “shift” type=”xs:string”/>
       …….
       <xs:/complexType>
      <xs:/element>
     <xs:/complexType>
    <xs:/element>
    ……
  <xs:/complexType>
 <xs:/element>
<xs:schema>
```

Fig. 7.2. An XML schema for document in Fig. 7.1

The syntactic structure of an XML document instance is defined by an associated XML schema, which itself is an XML document [URLc]. The instance document is said to conform to its associated schema. Fig. 7.1 illustrates an XML instance document that conforms to the XML schema shown in Fig. 7.2. Essentially, the structure of XML tags in the instance document is as per the schema definition. Note, each element may contain its own set of child elements. For instance, the second line in Fig. 7.2 indicates that `enterprise` is the root element of the document that contains element `depts` as its child element. Elements can thus form a hierarchy. The `engg_manager` and `design_manager` elements have an attribute named job_id. The tags are usually chosen to be meaningful within the context of the application that the XML document is part of. Such an extensible naming feature of XML allows capturing the application specific needs of an organization to create customized documents. The detailed specifications of XML and XML Schema can be found at [URLa, URLc].

The use of XML for expressing GTRBAC policies has many advantages. In particular, we are motivated by the fact that XML is user-friendly, extensible, and widely supported by all the main platform and tool vendors [URLd]. Hence, the XML policy specification language will have a wider applicability. In particular, an XML-based policy specification language can be used in any web-enabled e-commerce applications. Another motivation for using XML as the language of choice for specifying access control policies is the heterogeneity of collaborating entities, within a large distributed enterprise environment, that enable high level information system services. The functional entities within an enterprise, connected through multiple media, and each comprised of heterogeneous information systems that are linked together by the EC technology, require a common policy specification language to efficiently express and enforce the enterprise level access control policy. As XML provides a uniform, vendor-neutral representation of enterprise data, and allows a mechanism for interchange, sharing and dissemination of information content across heterogeneous systems [Bar97, Jos01a, Bis98], an enterprise can benefit significantly from an XML-based policy specification language.

7.2 Overview of X-GTRBAC Language Features

In this section, we present the features of the proposed X-GTRBAC language. X-GTRBAC has been designed to be used in a web-based environment or a large enterprise setting. The key features that X-GTRBAC provides are discussed below.

Specification of GTRBAC Elements: X-GTRBAC allows specification of all the elements of the GTRBAC model. These elements include user, role and permission definitions, specification of hierarchies and separation of duty constraints, periodicity and duration expressions, and triggers.

Specification of Credentials Based Role Assignment: X-GTRBAC allows the specification of dynamic assignment of users to roles based on the credentials that a user presents. In an RBAC model, users are assigned memberships to roles. However, in emerging web-based applications, the pool of users is not known in advance. While in such cases, other access control models have limitations [Jos01b, San94], RBAC provides roles as the basis for capturing the access control requirements of an application. However, the policy also should include how unknown users in the open Internet environment may be assigned to roles so that services provided by the application are available to legitimate, yet unidentified users. Furthermore, in a large enterprise, a considerable amount of administrative effort needs to be expended in the administration of the assignment of tens of thousands of users to hundreds of roles. Credential based dynamic assignments of users to roles allows efficient administration of access control policies in such environments by defining rules on credential attributes for assigning roles to strangers. X-GTRBAC allows an administrator to define credentials that can be assigned to users and later used to resolve the assignment of roles when access requests are made.

Specification of Content and Context based policy : X-GTRBAC provides some level of support for content-based access control, in particular when access control policies are defined for XML documents themselves. For protected XML documents, protection granularity can be at the schema level, an instance document level, or an element level. In a generic case, we believe, the abstraction provided by a permission as an authorized operation on an object, can be used to capture generic content based access control. In such a case, the content-based access control framework described in [Jos02] can be used. We have not addressed this issue in this dissertation.

Although we have not formally addressed the issue of general context-based extension of the G-TRBAC model, it has been noted in Chapter 5 that context based access control policies can be easily defined by allowing triggers to include events and conditions that are external to the GTRBAC system. X-GTRBAC allows capturing such context information through a trigger on constraints on assignments to provide context-based dynamic access control support. In addition to triggers, X-GTRBAC allows specifying pre-conditions for each state of a role enabling, role activation or role assignment in order to allow authorizations to be controlled based on dynamic context information. While triggers and control flow dependency constraints can capture such issues, we include the facility for the specification of such preconditions in X-GTRBAC to allow simplified policy specification and flexibility. Policy designers may choose either triggers or such pre-conditions, or a combination of them, to better express the policy requirements. The notion of pre-condition for roles provides good support for credential and context-based policies, as these pre-conditions can be defined on credential attributes as well as the contextual information of an application environment. For each role, we define the following three types of preconditions

I. *Role enabling/disabling precondition*: This precondition needs to be satisfied for a role to be enabled.
II. *Role assignment/deassignment precondition*: This precondition needs to be satisfied before a user can be assigned/deassigned to a role.
III. *Role activation/deactivation precondition*: This precondition needs to be satisfied before an authorized user can activate a role.

Specification of Policy Mapping Between Different Security Domains: X-GTRBAC also provides some support for expressing mapping between roles of GTRBAC policies belonging to different domains. Multidomain environments have manifested in various forms of emerging systems. Those particularly becoming prominent include *Web-services* and *Grid-based* systems [Azz02, Pea02]. *Web Services* are typically employed in B2B applications where a service provider may need to expose specific information to a client website, or an automated transaction may need to be carried out between two e-commerce applications based on pre-specified rules. *Grid-based* systems, on the other hand, are emerging as a promising technology that can span an environment with the size and scope of the Internet with heterogeneous computing systems geographically distributed across multiple administrative domains [Azz02]. In general a multidomain environment can be characterized as *loosely* coupled or *tightly* coupled systems. In a loosely coupled environment, the interoperation needs are transient and dynamically determined, whereas in the tightly coupled or federated environment, the policies from multiple systems are integrated to enable higher goals. X-GTRBAC can be used to support the specification of interdomain role mapping in such environments.

7.3. X-GTRBAC Syntax

In this section, we describe the X-GTRBAC syntax using a BNF-like grammar that we refer to as X-Grammar. The X-Grammar supports the tagging notation of XML and allows expressing attributes within element tags. The non-terminals are represented as <!-- "non_terminal_name"> XML tags, and terminals as standard XML tags. We use the optional tags by placing them within square brackets "[ ]". Group portions of a production are included in curly brackets "{}", with a subscript to indicate the repeat count. The default count, if the subscript is not specified, is one. A "*" and a "+" indicates a count of "zero or more" and "one or more" respectively, whereas a "-" indicate a range. Alternative elements are separated by "|"s in a production. Any data placed in parenthesis "( )" is not part of the terminal symbol, and shall be supplied by the security administrator. The X-Grammar has been used to simplify expressing production rules for the X-GTRBAC language constructs.

```
<!-- Policy Definition--> ::=
<Policy [policy_id = "(value)"]>
  <PolicyName> (name)
  </PolicyName>
  [<!--XCredType Definition Sheet>]
  [<!--XTemporalConstraint Definition Sheet>]
  <!-- XML User Sheet>
  <!-- XML Role Sheet>
  <!-- XML Permission Sheet>
  <!-- XML User-Role Assignment>
  <!-- XML Role-Permission Assignment>
  [<!-- XSoD Definition Sheet>]
  [<!-- XHierachy Definition Sheet>]
  [<!-- Local Policy Definitions-->]
  [<!-- Policy Relationship Definitions>]
</Policy>
```

Fig. 7.3. X-GTRBAC policy sheet

Policy Document: The XML syntax for general policy definition (<!-- Policy Definition-->) is shown below. The key policy component definitions include the XML Role Sheet (XRS), the XML User Sheet (XUS), the XML Permissions Sheet, (XPS), the XML User-Role Assignment Sheet (XURAS), and the XML Permission-Role Assignment Sheet (XPRAS). Moreover, a policy can include multiple constituent policies, thus facilitating the specification of policies for multidomain environments.

Each constituent policy may be a local policy of a federated system or a policy of a partner domain in a loosely coupled environment. Local policy definitions are included or simply referred to by using the *id*s of the local policies. If local policies are defined, then the set of relationships between the global policy and each of the local policies needs to be defined. The relationship definition will include mapping specification between the global entities and the local entities.

```
<!-- XCredType Defintion Sheet > ::=
<XCredType xCrType_id = (id)>
  {<!-- Definitions of Credential Types>}+
</XCredType>

<!-- Definitions of Credential Types> ::=
<XCredType [xctd_id = (id) ] >
  {<!-- Credential Type Definition>}+
</XCredType>

<!-- Credential Type Definition> ::=
<CredType cred_type_id = (id)
       type_name= (type name) >
  <AttributeList>
    {<!-- Attribute Definition>}+
  </AttributeList>
</CredType >

<!-- Attribute Definition> ::=
<Attribute>
  <AttributeName
         usage = "mand | opt"
         type = (type)>
   (name)
  </AttributeName>
</Attribute>
```

```
<!-- XML User Sheet > ::=
<XUS xus_id = (id)>
  {<!-- User Definitions>}+
</XUS>

<!-- User Definitions > ::=
<Users>
  {<!-- User Definition>}+
</Users>

 <!-- User Definition>   ::=
<User user_id = (id)>
   <UserName> (name) </UserName>
   {<!--CredType>}+
   <MaxRoles>  (number) </MaxRoles>
</User>

<!-- CredType>  ::=
<CredType cred_type_id = (id)
        type_name= (type name) >
  <!-- Credential Expression>
</CredType>

<!-- Credential Expression>      ::=
<CredExpr>
  {<(attribute name)> (attribute value)
  </(attribute name)>}+
</CredExpr>
```

Fig. 7.4 XUS syntax

Credential Type Definition and XML User Sheets (XUS): X-GTRBAC uses the notion of credentials proposed in [Ber99a]. Credential type definition specifies the attribute list associated with a credential type. The value of each attribute is assumed to be of string type. Each attribute of a credential type may be defined as mand, to indicate that it is mandatory, or as opt, to indicate that it is optional. Consider the following user credential based on a general credential expression of the form (`cred_type_id`, cred_expr), where cred_type_id is a unique credential type identifier and cred_expr is a set of attribute-value pairs.

(Nurse, {(credtypeid, "C100", mand), (uname, "John", mand), (age, 30, opt), (level, 5, mand)})

X-GTRBAC allows the definition of new credential types to group users based on their credentials. The schema for the credential type definition (XCredType) is shown in Fig. 7.4. The credential information in XCredType sheet allows adding vocabulary to express the credentials needed by the users. Users and their credentials are expressed in XUS. Fig. 7.4 shows the grammar for XUS. User definition may simply define user name and user id, or additionally specify the assigned credentials that the user may carry. The MaxRoles tag indicates the maximum number of roles that a user can be assigned to.

XML Permission Sheet (XPS): Permissions are specified in X-RBAC using the syntax for XPS shown in Fig. 7.5. The permissions for a given system are defined in terms of *objects* and associated *operations* such as *read*, *write*, *delete*, *modify*, etc. Permissions are defined by security administrators. The set of permissions for a system is expressed in the form of an XML document that we refer to as the XML Permission Sheet (XPS). The grammar for XPS is shown in Fig. 7.5.

```
<!-- XML Permission Sheet> ::=
<XPS [xps_id = (id) ]>
  {<!-- Permission Definition>}+
</XPS>
```

```
<!-- Permission Definition> ::=
<Permission perm_id = id
          [prop= (prop op)] >
  <Object type= (type name)  id= (id)/>
  <Operation>  (access op)
</Operation>
</Permission>
```

Fig. 7.5 XPS syntax

The "perm_id" uniquely identifies a permission. Each object also has a unique id and an associated type attribute. When the resources in the system are modeled as XML, the hierarchical structure of an XML document is used to capture the physical object hierarchy described. An object hierarchy could be composed of either documents, or document elements (in case of XML documents). A permission can, hence, be allowed on such XML schema, document instances and on each element in the element hierarchy of the document. When XML documents are to be protected, an optional propagation option, given by the "prop" attribute, can be specified which indicates whether or not the authorization of privileges propagates down the object hierarchy. X-GTRBAC allows propagation options "`no_prop`", "`first_`level" and "`cascade`" [Ber01b], with default being "`no_prop`", meaning that no propagation of authorized privilege is to be allowed down the element hierarchy.

XML Role Sheets (XRS): Role definitions are provided in an XRS as shown in Figure 7.4. For each role, a set of role attributes is specified. Preconditions are defined on these attributes. Such attributes may refer to credential attributes that users present as well as contextual information of the system. As mentioned earlier, each role may have associated with it preconditions for its *enabling, assignment* and *activation* that are separately defined using the `<EnabCondition>`, and `<ActivCondition>` tags in Fig. 7.6.

```
<!-- XML Role Sheet> ::=
<XRS [xrs_id = (id) ]>
  {<!-- Role Definition>}+
</XRS>

<!-- Role Definition> ::=
<Role role_id = (id)
      role_name = (role name)>
  [<!--Attributes>]
  [<!--{En|Dis}abling Constraint>]
  [<!--[De]Activation Constraint>]
  [<Junior HType = "I|A|IA"
          PConst = (TempExID)>
   (name)
   </Junior>]
  [<Senior HType = "I|A|IA"
          PConst = (TempExID)>
   (name)
   </Senior>]
  [<Cardinality> (number)
</Cardinality>]
</Role>
```

```
<!-- {En|Dis}abling Constraint>  ::=
  <{En|Dis}abConstraint
      [op = {AND|OR|NOT}]>
    {<!--{En|Dis}abling Condition>}+
</{En|Dis}abConstraint>

<!-- {En|Dis}abling Condition> ::=
<{En|Dis}abCondition
    [{pt_expr_id=(id)  |
      d_expr_id=(id)}]  >
  [<!-- Logical Expression>]
<{En|Dis}abCondition>

<!--[De]Activation Constraint> ::=
  <[De]ActivConstraint
          [op = {AND|OR|NOT}]>
  {<!--[De]ActivationCondition>}+
</[De]ActivConstraint>

<!--[De]Activation Condition> ::=
  <[De]ActivCondition
          [d_expr_id=(id)]>
    [<!-- Logical Expression>]

</[De]ActivCondition >
```

Fig. 7.6 XRS syntax

For enabling/disabling preconditions, we can use a temporal expression, which can be defined separately, as a condition. Semantically, it means that the role is enabled if the current time instant is contained in the periodic time expression. The language allows specifying additional logical predicates to be used to express context based conditions using the generic syntax for the logical expressions. Periodic time and logical expressions are shown in Fig. 7.7 and discussed below. Note that we may allow any complex logical expression using this syntactic framework. A role definition may specify hierarchy relations by specifying its juniors using the `<Junior>` and `<Junior>` tags and specifying the type `HType`. Furthermore, the hierarchy relation may be associated with a temporal expression to time-constrain the validity of the relation and to express role cardinality using the `<MaxUsers>` tag.

```
<!-- Definitions of Temporal Constraints>
::=
<XTempConstDef [xtcd_id = (id) ]>
    {<!—Interval Expression>}*
    {<!-- Periodic Time Expression>}*
    {<!-- Duration Expression>}*
</XTempConstDef>

<!—Interval Expression ::=
<IntervalExpr i_expr_id = id)>
    <begin> (date)</begin>
    <end> (date)</end>
<IntervalExpr>

<!-- Periodic Time Expression> ::=
 <PeriodicTimeExpr pt_expr_id = (id)
  [d_expr_id = (id) ]  [i_expr_id = (id) ] >
  <!-- Start Time Expression>
</PeriodicTimeExpr>

<!-- Duration Expression> ::=
<DurationExpr d_expr_id = (id)>
  <cal>{Years|Months|Weeks|Days}</cal>
  <len> (number)</len>
</DurationExpr>

<!-- Start Time Expression> ::=
<StartTimeExpr [pt_id_ref =(pt_id)]>
   [<Year>{all|odd|even} /<Year>]
  [<!--MonthSet>]
  [<!--WeekSet>]
  [<!--DaySet>]
</StartTimeExpr>

<!--MonthSet>  ::=
<MonthSet>   {<Month>{1|..|12}</Month>}1-
12
</MonthSet >
```

```
<!--WeekSet> ::=
  <WeekSet>
    {<Week>{1|..|4}</Week>}1-4
</WeekSet >

<!--DaySet> ::=
<DaySet>
  {<Day>{1|..|7}</Day>}1-7
</DaySet >
<
!—Temporal Expression> ::=
<!—Interval Expression>|
<!-- Periodic Time Expression>|
<!-- Duration Expression>|

<!-- Logical Expression>::=
<LogicalExpr [op ={AND|OR|NOT}]>
  {<!-- Predicate>}+
</LogicalExpr>

<!-- Predicate> ::=
<Predicate>
   [ {<Operator> {gt|lt|eq|neq}
  </Operator>
      [<FuncParam>(functionname)
       </FuncParam>]
       {<NameParam

  type=(role|user|attribute)]>
             (parameter name)
        </NameParam>
       }+

  <ValueParam>(value)</ValueParam>
     }
     |
    < !--LogicalExpression> ]
</Predicate>
```

Fig. 7.7 Schema for temporal and logical expression

Temporal Constraint and General Logical Expressions: X-GTRBAC allows defining temporal expressions separately so that they can be associated with role enabling, assignments and other constraints. Temporal constraint expressions are defined in the `XTempConstDef` sheet, presented in Fig. 7.7.

The "Attributes" tag of the role contains a list of role attributes that may be parameters of the context conditions, or credential a,ttributes which need to be dynamically evaluated for any role enabling/disabling or activation/deactivation. The context conditions may be based on parameters such as system load, location information etc., or on status expressions such as "whether role R has been enabled by user U". The "`(En|Dis)abling Constraint`" and "`[De]Activation Constraint`" tags contain a set of conditions, where each condition is composed of possibly multiple logical expressions for specification of the respective constraints based on both temporal and non-temporal context-dependent parameters. The constraint tag has an optional `op-code` attribute to specify the logical operators applied on its child elements. An `op-code` of (i) "AND" implies that all constituent expressions must be true for the constraint to be true, (ii) "OR" implies that at least one expression must be true for the constraint to be true, and (iii) "NOT" implies that none of the expressions must be true for the constraint to be true. The op-code defaults to "AND" if none is specified.

Each condition tag may contain a "`pt_expr_id`" or "`d_expr_id`" attribute that refers to a periodic-time or a duration expression respectively. These expressions are the XML representation of the periodic-time expression framework provided in the GTRBAC model, and bind the corresponding condition with the respective periodic expression. We give an XML representation for each of the start-time, interval, and duration expressions that together constitute the periodic-time expression. Following the notion of "calendars" used in the GTRBAC model, the start time expression consists of "calendar sets", where each calendar is a unit of time, e.g. years, months, weeks, etc. As an example, an event that occurs at the start of the second week of every first and eighth month of every odd year would be represented by using "{odd}" as the Year set, "{1,8}" as the Month Set, and "{2}" as the Week Set. The optional "`pt_id_ref`" attribute indicates start time with reference to the provided periodic-time expression id. If it is supplied, then the start time is the same as that of the referenced periodic time. Note that a "`pt_id_ref`" is provided only when the calendar sets are not provided, and vice versa. Any new start time is always explicitly defined using new calendar sets. An interval is given by a (`begin_date`, `end_date`) pair, and a duration is specified as a (calendar, calendar_length) pair. The semantics of the periodic time expression thus dictate that the associated event can only occur if the start time expression is satisfied by the time of request, and if such time falls within the interval specified by the interval expression. The duration of the event, if it occurs, would be governed by the duration expression.

The "`Logical Expression`" tag contains a set of predicates, where each predicate may contain a context-condition expressed in terms of role attributes, or embed within itself another logical expression. Hence, the structure allows evaluation of nested conditions expressed by multiple logical expressions. The predicates are composed of context-based parameters, where the "`NameParam`" tag contains the name of the parameter to be evaluated, and the "`ValueParam`" tag contains its value that is to be checked according to the given "`Operator`". For instance, any attribute supplied as part of a user credential expression may be compared for a pre-requisite value needed for certain role assignment or activation by supplying the attribute name as "`NameParam`", the required values as "`ValueParam`", and the comparison operator as "`Operator`". The "`FuncParam`" is an optional tag which is useful if the parameters in question can only be evaluated through a system review function, expressed as the status expressions of the GTRBAC model. Multiple parameter names may be passed to functions that evaluate multiple parameters, with the distinction among parameter types made with the "type" attribute. As an example of predicates, we might evaluate status expressions for a role by supplying a status condition such as "`active` *r* `for` *u*" as "FuncParam", the role name and the user id as two instances of "`NameParam`", and the value of either "`True`" or "`False`" as the "`ValueParam`". In such situations where a boolean output is returned, only the "`eq`" operator is useful for comparison. The "`Logical Expression`" tag also has an optional `op-code` attribute that determines the evaluation logic of the predicates. On the similar lines as the constraint tag, an op-code of (i) "AND" implies that all constituent predicates must be true for the logical expression to be true, (ii) "OR" implies that at least one predicate must be true for the logical expression to be true, and (iii) "NOT" implies that none of the predicates must be true for the logical expression to be true. The op-code defaults to "AND" if none is specified. The grammar for logical expression specification is shown in Fig. 7.7.

Trigger: Fig. 7.8 shows the syntax of the GTRBAC trigger expression. As the predicates within a logical expression can include both temporal and non-temporal context-based parameters, they allow for the specification of context-based triggers in our

X-GTRBAC framework. This set of triggers is supplied in a separate XTrigDef sheet. The grammar for the XTrigDef sheet is shown in Fig. 7.8.

The "Head" tag of the trigger has an attribute that indicates the target role or the permission on which the trigger action is performed. An optional "user_id" attribute is also supplied for triggers that need to perform the action with respect to certain individual users. The triggering constraint in the "Body" tag is semantically similar to the constraints discussed above, and is evaluated in an analogous manner. The action associated with the trigger is performed if the constraint evaluates to true.

```
<!--Triggers Specification>::=
<XTrigDef [xtd_id = (id) ]>
  {<!-- Trigger>}*
</XTrigDef>

<!--Trigger>  ::=
<Trigger [trig_id = (id) ]>
      <Head {role_name = (name) | perm_id = (id) }
         [user_id = (id)]
           action = {enable | disable |
                  assign | deassign | deactivate} >
   </Head>
   </Body> <!--Triggering Constraint> </Body>
</Trigger>
```

```
<!--Triggering Constraint>  ::=
  <TrigConstraint
      [op =
{AND|OR|NOT}]>
    {<!--Triggering Condition>}+
</TrigConstraint>

<!—Triggering Condition>
<TrigCondition>
   [<!-- Logical Expression>]
<TrigCondition>
```

Fig. 7.8 Trigger syntax

Separation of Duty Expression: The separation of duty constraint specification simply uses a predefined set of SoD types to identify the type of SoD as shown in Fig. 7.9. For each SoD a set of roles, users, or permissions may need to be specified. If any of these sets is not specified, and the specified SoD type requires one as discussed in Chapter 5, then the entire set of the users, roles, or permissions will be considered.

```
<!-- Separation of Duty Definitions> ::=
<XSoDDef [xsod_id = (id) ]>
   {<!—SoD Definition>}*
</XSoDDef>
```

```
<!—SoD Definition>
<XSoD [xsod_id = (id)
       [xsod_type = (id)]>
   [<!—UserSet >]
   [<!—RoleSet >]
   [<!—PermissionSet >]
   [<!—Temporal Expression >]
</XSoD>
```

Fig. 7.9 Separation of duty expression

XURAS: The schema for XURAS is shown in Fig. 7.10. Each "`UserRoleAssignment`" (URA) tag has an associated "`role_name`" attribute, and contains a set of "`AssignUsers`" tags containing the set of users who are to be considered for potential assignment to the specified role. Each such user is identified by the "`user_ id`" attribute of the corresponding "`AssignUser`" tag. This tag also contains the assignment constraint for this particular user. The assignment constraint has a "`cred_type`" attribute that specifies the credential type that the user must possess in order to be considered for a potential role assignment. The remaining part of the constraint is semantically similar to the constraints discussed above and is evaluated in an analogous manner. The user is assigned to the specified role if the constraint evaluates to true. Similar logic applies to de-assignment of users from roles. Note that a special user with `user_id` = "any" is recognized by the system as an unknown user, who may be required to supply additional assignment conditions in order to be assigned to a particular role. If no explicit conditions are specified, then any user could be assigned the particular role, which usually is the "guest" role in most enterprise applications.

```
<!-- XML User-role Assignment Sheet>  ::=
<XURAS [xuras_id = (id) ]>
  {<!-- User-role Assignment>}+
</XURAS>

<!-- User-to-role Assignment> ::=
<URA ura_id=(id) role_name=(name)>
<[De]AssignUsers>
    {< !--[De]Assign User>}+
</[De]AssignUsers>
</URA>

<!--[De]Assign User >    ::=
<[De]AssignUser
    user_id=(id)>
 <!--[De]Assign User Constraint>
</[De]AssignUser>

<!--[De]Assign User Constraint> ::=
<[De]AssignUserConstraint
  [op = {AND|OR|NOT|XOR}]>
  <!--[De] Assign User Condition>
</[De]AssignUserConstraint>
```

```
<!--[De]Assign User Condition>  ::=
<[De]AssignUserCondition
  cred_type="type_name"
  [{pt_expr_id=(id) |
    d_expr_id=(id)}] >
   [<!-- Logical Expression>]
</[De]AssignUserCondition>

<!-- XML Permission-role Assignment Sheet>
::=
<XPRAS [xpras_id = (id) ]>
  {<!-- Permission-role Assignment>}+
</XPRAS>

<!-- Permission-role Assignment>       ::=
<PRA pra_id=(id) role_name=(name)>
<[De]AssignPermissions>
    {< !--[De]Assign Permission>}+
</[De]AssignPermissions>
</PRA>
<!--[De]Assign Permission >    ::=
<[De]AssignPermission
    [{pt_expr_id=(id) |
    d_expr_id=(id)}]
   {<PermId>(id)</PermId>}+
</[De]AssignPermission>
```

Fig. 7.10 User-role and role-permission assignments

XPRAS: The grammar for XPRAS is shown in Fig. 7.10. Each "`PermissionRoleAssignment`" (PRA) tag has an associated "`role_name`" attribute, and contains a set of "`AssignPermission`" tags containing the set of permissions that are to be potentially assigned to the specified role. Each such permission is identified by a "`PermId`" tag within the corresponding "`AssignPermission`" tag. Note that the permissions would typically be subject to periodic-time or duration constraints, and hence we allow the option of specification of periodic-time or duration constraint expression for the permission assignment. The permission is assigned to the specified role if the constraint evaluates to true. Similar logic applies to de-assignment of permissions from roles.

```
<!—Local Policy  Definitions -->
<LocalPolicies>
  <!—(Local) Policy Definition -->
</LocalPolicies>

<!—Policy Relationship  Definitions -->
<PolicyRelationships
  [prs_id = (id)] [pt_id =  (id) -->
    {<!-- Definition of Policy Relation>}+
</PolicyRelationships>

<!-- Definition of Policy Relation>::=
<PolicyRelation pr_id = (id)
                        [pt_id = (id)]>
  <GlobalToLocalMapping
          [gMap_id =  (id)] >
    {<!-- Role Mapping>}+
  </GlobalToLocalRoleMapping>
</PolicyRelation>
```

```
<!-- Role Mapping>::=
<RoleMapping>
   <MappedRole [r_id = (id)]> (name)
     {<!—Roles Mapped To>}+
   </MappedTo>}+
</RoleMapping>

<!-- Roles Mapped To>::=
<MappedTo>
  <Role [r_id=(id)] [policy_id = (id)]>
    (name)
   </Role>
   <!-- Mapping Condition>
</MappedTo>
```

Fig. 7.11 Schema for metapolicy specification

Specification of Metapolicy: Within a policy definition, we can include local policy definitions using the XML syntax depicted in Fig. 7.11. Note that each policy may itself be a global policy over a set of local domains. Thus a hierarchy of policy may be specified in which a multidomain environment may become a component of a larger multidomain environment and so on. A relevant principle for mediation policies is the following scoping rule:

Scoping rule: *If a policy P becomes a local policy of a higher level policy, then P's local policy definitions and the policy relations are not known to the higher level policy.*

Such a rule says that, within a global policy definition, only the entities of its local policies, and not those of constituent domains of these local policies, are visible. This abstraction simplifies the metapolicy construction. However, if the higher level policy management must oversee the policy of the overall federation, then this rule may need to be relaxed.

With local policies included, we need to define the relationships among their policy entities with the global entities. The XML syntax for defining policy relationships is shown in Fig. 7.11. Each global role may be mapped to a number of local roles, which may belong to the same or different local domains. For each mapping, a condition can be specified. We require that the local roles that a global role can be mapped to are included in the local policy definitions.

Example of metapolicy specification: The following example illustrates the specification of the mapping relationship depicted in Fig. 7.12:

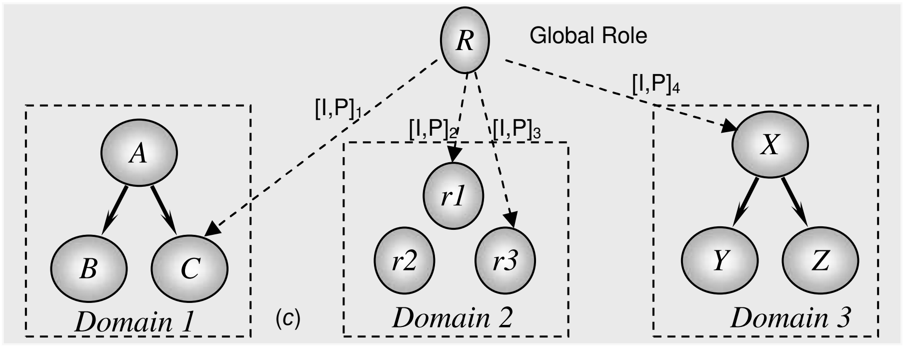


Fig. 7.12. Metapolicy example

Fig. 7.12 shows the mapping of the global role *R* to the local roles *C* in *Domain* 1, r_1 and r_2 in *Domain* 2, and *X* in *Domain* 3. Next, we illustrate the possible use of such a mapping *healthcare* systems.

Consider a healthcare federated system of three hospitals which allow cross-appointment of a doctor. In such a case, *R* can represent a FederatedDoctor which is mapped to local doctor roles, say *C* = DayDoctor in *hospital* 1, r_1 = DayDoctor and r_2 = EmergencyDoctor in *hospital* 2, and *X* = SupervisorDoctor in *hospital* 3. Furthermore, assume that the global-to-local role mappings are valid in the intervals defined as follows:

$[I, P]_1$ = {*Mondays*, *Wednesdays*}, $[I, P]_2$ = {*Tuesdays* and *Thursdays*}, $[I, P]_3$ = {*Fridays*} and $[I, P]_4$ = *Weekends*. With these mappings, a user, say Dr. Smith, who needs to be cross-appointed to different hospitals at different times, for instance, can be assigned to the FederatedDoctor role between 9*am* and 6*pm* on *Mondays* through *Saturdays*. This means that during the daytime between 9*am* and 6*pm*, Dr. Smith can assume:

- DayDoctor role in *hospital* 1 on *Mondays* and *Wednesdays*, and in *hospital* 2 on *Tuesdays* and *Thursdays*,
- EmergencyDoctor role in *hospital* 2 on *Fridays*, and
- SupervisorDoctor role on *Saturdays* in *hospital* 3.

```
<PolicyRelation pr_id = PR1>
  <GlobalToLocalMapping [gMap_id = "GL1"]>
    <RoleMapping>
      <MappedRole> FederatedDoctor
      </MappedRole>
      <MappedTo>
         <Role policy_id = "Policy1"> DayDoctor </Role>
         <MappingCondition>
            <PeriodicTime pt_id = "PTa" />
         </MappingCondition>
      </MappedTo>
      <MappedTo>
         <Role policy_id = "Policy2"> DayDoctor </Role>
         <MappingCondition>
           <PeriodicTime pt_id = "PTb"/>
          </MappingCondition>
      </MappedTo>
      <MappedTo>
         <Role policy_id = "Policy2"> EmergencyDoctor </Role>
         <MappingCondition>
           <PeriodicTime pt_id = "PTc" />
          </MappingCondition>
      </MappedTo>
      <MappedTo>
         <Role policy_id =  "Policy3"> SupervisorDoctor </Role>
         <MappingCondition>
            <PeriodicTime pt_id = "PTd" />
        </MappingCondition>
      </MappedTo>
    </RoleMapping>
  </GlobalToLocalRoleMapping>
</PolicyRelation>
```

Fig. 7.13. X-GTRBAC policy specification for metapolicy of Fig. 7.12.

The global-to-local role mapping component can be specified using X-GTRBAC as shown in Fig. 7.13. Note that in this case the policy of the domains 1 2 and 3 will be expressed as local domains. The global role FederatedDoctor will be defined at the top level of the policy definition sheet.

7.4 X-GTRBAC System Architecture

In this section, we present the system architecture of X-GTRBAC. We briefly provide an overview of the system components and technologies.

The X-GTRBAC framework allows the access policies to be specified and enforced through a Java-based GUI-enabled application. The application code can be readily integrated into a Web browser by an application-to-applet transformation mechanism provided by Java.

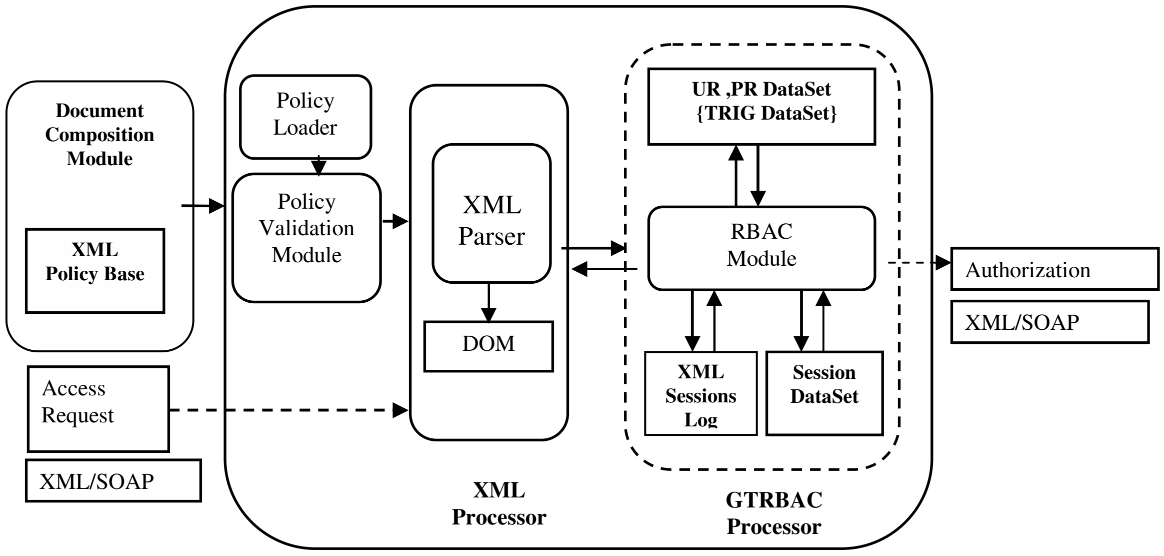


Fig. 14. X-GTRBAC architecture

The overall system architecture is depicted in Fig. 14. Information about security policy is contained in the XML Policy Base. A document composition module external to X-GTRBAC facilitates the composition of the policy components discussed earlier. The policy sheets from the XML Policy Base are then loaded into the X-GTRBAC Module by the security administrator. As shown in the Fig. 14, the two main sub-systems of the X-GTRBAC Module are the XML Processor and the GTRBAC Processor. The XML processor is implemented in Java using a Java API for XML Processing (JAXP). The

Document Object Model (DOM) instances of the parsed XML documents representing the policies are forwarded to the GTRBAC Processor. The GTRBAC Module then enforces the policy accordingly. Since X-GTRBAC can act both as a stand-alone and web-deployable application, it may be invoked from either the local system, or remotely through an XML-aware browser. Hence, the X-GTRBAC Module seamlessly interfaces with an external client across distributed domains over an interconnect network (i.e. LAN, WAN etc.). The client may submit an access request through any standard XML-based Web services messaging protocol, like SOAP [URLe]. Similarly, the access authorization is returned via the same protocol.

7.4.1 XML Processor

The XML Processor consists of the XML Parser and the DOM tree structure of the XML policy documents. A Policy Loader loads the policy sheets for a given policy from the policy base. A Policy Validation Module is used to validate the policy sheets in terms of existence checking and type conformance. In other words, all users, roles, and permissions referenced in the XURAS, XPRAS and `XTrigDef` sheet should be defined in the corresponding XUS, XRS and XPS respectively. Further, all the referenced data must exist in the corresponding definition files. This implies that (i) the credential types associated with the users in XUS must conform to the type definitions in the `XCredTypeDef` sheet, (ii) the separation of duty constraint sets referenced in the XRS must be present in a `XSoDDef` sheet, and (iii) the periodic-time, start-time, interval, and duration expressions referenced in XRS must be defined in a `XTempConstDef` sheet. Currently, this validation support is provided by the Apache Xalan XSLT engine built into JAXP. The DOM tree representations of validated policy documents are generated and passed on to the GTRBAC Processor. A GUI facility is provided to display the instance of the DOM tree.

7.4.2 GTRBAC Processor

The GTRBAC Processor contains the GTRBAC Module and associated data items generated by the GTRBAC Module. It performs the policy administration and enforcement tasks.

The GTRBAC Module provides functionality to parse the DOM tree structures supplied by the XML Processor, and retrieves the relevant information into its internal data structures. It may be noted that for all the users assigned to roles, the actual role activation occurs when the users actually log into the system and request a roles. The notion of role assignment in this context is of static type; i.e. it implies that the user has been declared as assignable to the associated role based on already supplied credential information. A dynamic role assignment for an unknown user based on his/her credentials supplied at the time of login is possible. These static and dynamic policy assignments, together with the role activation and enabling rules and triggers information, create the complete internal representation of the XML Policy Base within the GTRBAC Processor for enforcement of the policy. A collection of these policy information items are referred as UserRole (UR) datasets, PermissionRole (PR) datasets, and TRIG dataset. A facility is provided to display the UR, PR and TRIG datasets via the X-GTRBAC GUI.

```
<xas [xas_id= (id)]>
 <login login_id= (id)>
   [<!--CredType>]
 </login>
 ……….
 <xar xar_id= (id)>
   {<Object type= (type name)
    id= (id)/>}+
</xar>
<xas>
```

```
<xss [xss_id= (id)]>
 <session>
  <session_id> (id) </session_id>
  <user_id> (user id) </user_id>
  <role_name> (role name) </role_name>
  <domain> (domain name) </domain>
  <login_time> (time) </login_time>
  <login_date> (date) </login_date>
  <duration> (duration) </duration>
  <active> {Yes|No} </active>
 </session>
</xss>
```

Fig. 15. XAS and XSS sheets

The information from the internal data structures is then used by the GTRBAC Module to enforce the policy and manage user sessions. The initial login into the system creates a default session for the user with a pre-specified “minimal” set of roles activated based on the supplied user credentials. The initial login can be the “user_id” from the XUS provided it is a known user, or a “user_id” of “any”, as discussed above. In addition to the default set of activated roles, more roles can also be activated if the user’s credentials permits. Any triggers associated with role activation or other events are handled by the GTRBAC Module based on the information from the TRIG dataset. Access to resources is requested in the form of an XML Access Request (XAR) that specifies the “object type” and “object id” of the requested resource. An XAR can be submitted locally or remotely as an assertion in SOAP or through a similar XML-based messaging protocol. This access request is then evaluated based on the currently activated roles for this user. Only those resources may be accessed during a session for which the activated set of roles has the respective permissions. Both the login information and XARs for a user are stored in an XML Access Sheet (XAS). The session-related information is contained in the Sessions Dataset within the GTRBAC processor. This information is extracted from an activity log maintained for every user by the GTRBAC module which we refer to as an XML Sessions Sheet (XSS). A session parameter is included in the XSS to record the domain from which the user is generating the access request. In addition to the domain of the requesting user, the XSS also contains the attributes such as "login_time", "login_date", and "duration" of the session. These attributes are used to capture the activity profile of the user. Such information is constantly updated into the Sessions DataSet, where it can be dynamically processed, and incorporated into the access decisions. This feature is useful in certain situations where context information may be an important decision parameter, as discussed in Section 3.2. The grammar for a typical XAS and XSS is shown in Fig. 15.

7.5 Conclusions

In this chapter, we have presented an XML-based specification language for expressing GTRBAC policies and an implementation architecture. X-GTRBAC provides compact representation of access control policies and allows context-aware access control and metapolicy features. We have emphasized the separation of language schemas to provide efficient specification of definitions of RBAC elements, user-to-role and permission-to-role assignments, hierarchical and separation of duty constraints, and an elaborate set of temporal constraint expressions.

8. CONCLUSIONS AND FUTURE WORK

In this chapter, we summarize the contributions of this dissertation and discuss future research directions.

8.1 Research Contributions

Following are the main contributions of this dissertation:

1. We have proposed a generalized temporal role based access control model that can handle a comprehensive set of temporal constraints. The model allows temporal constraints on role enablings and role activations. Various temporal restrictions can be specified on the user-role and role-permission assignments. We use a notion of safeness to generate a safe execution model for a GTRBAC system.
2. We have identified various types of temporal hierarchies that capture permission-inheritance and role-activation semantics based on the temporal properties of hierarchically related roles. In case, different types of hierarchical relations are allowed to coexist in a system, complex permission-inheritance and role-activation semantics are needed. In such a case, it becomes difficult to determine the permissions that can be acquired and the roles than can be activated by a user assigned to a role in the hierarchy. We have presented formal analysis and a set of inference rules related to such role hierarchies that characterize roles that a user can activate. The detailed formal analysis presented in this dissertation can be used in security administration tools to efficiently administer access control policies. Furthermore, we have presented an analysis mechanism for managing evolution of a role hierarchy.
3. We have presented a comprehensive set of constraints that can be used to capture complex access control requirements. A generic framework for expressing a wide range of time-based cardinality constraints associated with GTRBAC states has been presented. We have developed a trigger expression that can capture complex dependencies among events and conditions. In particular, we have defined a set of

control flow dependency (CFD) constraints that can be used to express stricter access control requirements typical in workflow applications. We have presented a comprehensive set of time-based SoD constraints that subsume the SoDs that have been identified in the earlier work on the RBAC models. Furthermore, these constraints provide basis for formally expressing many of the earlier identified SoDs, including history based, and order independent SoDs.

4. We have presented an analysis of GTRBAC's expressiveness and generated a set of GTRBAC family of models with equivalent expressive power. We have shown that the set of constraints in the GTRBAC model, although not minimal, provides better flexibility in expressing access control policies with less complexity. Various design guidelines have been provided to allow specification of access policies in a simplified manner.

   We believe that our approach for analyzing the expressiveness and reducing specification complexity for the GTRBAC model has a broader significance. This is because the proposed analytical approach is generally applicable to any specification model that uses more than the minimal specification constructs needed. Such analysis can be used to derive design guidelines that can be used to generate specification model with reduced complexity. With the rapidly increasing complexity of information systems, the current emphasis is on developing more complex models capturing the semantics and/or requirements of such systems. Analysis similar to that presented in this dissertation need to be carried out to study tradeoff between complexity and usability such models.
5. Finally, we have presented X-GTRBAC, an XML-based specification for GTRBAC. The proposed X-GTRBAC framework allows specification of GTRBAC policies as well as specification of credential and context based dynamic access control policies and metapolicies. The framework is particularly beneficial because of the growing importance of XML and its widespread use in emerging large scale enterprise applications that span multiple security domains.

8.2 Future Work

The GTRBAC model can provide a foundation for pursuing several challenging research problems that are becoming relevant in the context of emerging large scale applications. Below, we summarize several directions in which our work can be pursued.

1. An immediate extension of our work will be to extend SQL language for specifying temporal constraints of the proposed GTRBAC model. Such an extension would allow using GTRBAC policy framework in database applications.
2. Another possible research direction is to develop tools to support security policy administration using various analysis techniques we have presented in this dissertation. Such tools can significantly reduce the administration efforts in large enterprise environments.
3. One key issue is the verifiability of a policy specification in terms of its safety and liveness. The GTRBAC execution model provides a restricted notion of safeness that only guarantees that certain ambiguous situation does not occur, as discussed in Chapter 3. As pointed out in Chapter 2, the general notion of safety is to ensure that something bad does not occur. Similarly, to verify liveness of a specification model is to ensure that something desirable eventually happens. One future direction is to address the safety and liveness characteristic of the GTRBAC model.
4. In the dissertation, we have referred to WFMSs as motivation for the work presented. Although roles and temporal constraints capture requirements of workflow tasks in organizational context, research can be done to reconcile RBAC models with WFMSs so that complex WFMSs can benefit from the use of RBAC models. Several concepts regarding restricted hierarchies, SoD constraints as well as temporal constraints presented in this dissertation can be used to facilitate requirement specification of WFMSs. X-GTRBAC can be extended to fully address the unique requirement of WFMSs.
5. As mentioned earlier, RBAC provides a promising approach for addressing the complex issues of access control in multidomain environments because of its policy-neutral nature and the flexibility that it provides in expressing a wide range of access control policies. Accordingly, the GTRBAC model can serve as a foundation for addressing the complex access control needs of dynamic multidomain environments. Various types of hierarchies and constraints introduced in this dissertation can provide support for handling complex scenarios encountered during integration of multiple policies. Emerging multidomain environments such as Web Services based application environments and Grids pose several challenges in terms of access control and policy representation. X-GTRBAC framework can be integrated with other security functionalities such as authentication and cryptography to generate holistic solutions to the problems of such multidomain environments.

REFERENCES

REFERENCES

[Aba93] M. Abadi, M. Burrows, B. W. Lampson, G. Plotkin, “A Caculus for Access Control in Distributed Systems”, *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 4, September 1993, pages 706-734.

[Ahn00] G. Ahn, R. Sandhu, “Role-Based Authorization Constraints Specification,” *ACM Transactions on Information and System Security*, 3(4), November 2000.

[All83] J. F. Allen, “Maintaining Knowledge about Temporal Intervals”, *Communications of the ACM*, Vol. 26, No. 11, November 1983, pages 832-843.

[Atl96b] V. Atluri and W-K. Huang, “An Authorization Model for Workflows”, *Proceedings of the Fifth European Symposium on Research in Computer Security*, Rome, Italy, and *Lecture Notes in Computer Science*, No. 1146, Springer-Verlag, September, 1996, pages 44-64.

[Amm92] Ammann, P.E. and Sandhu, R.S. "The Extended Schematic Protection Model.", Journal of Computer Security, Volume 1, Numbers 3 and 4, 1992, pages 335-383.

[And01] R. Anderson, “Security Engineering: A Guide to Building Dependable Distributed Systems,” John Wiley & Sons Inc., 2001.

[Atl99] V. Atluri editor. *Proc. of the Fourth ACM Workshop on Role-Based Access Control*, Fairfax (VA), 1999.

[Atl02] V. Atluri, A. Gal, “An Authorization Model for Temporal and Derived Data: Securing Information Portals,” *ACM Transactions on Information and System Security*, 5(1), February, 2002, pages 62 – 94.

[Att93] P. C. Attie, M. P. Singh, A. Sheth, M. Rusinkiewicz, “Specifying and Enforcing Intertask Dependeices”, *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, 1993, pages 134-145.

[Azz02] F. Azzedin, M. Maheswaran, “Towards Trust-Aware Resource Mangement”, *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid* (CCGrid’02), 2002.

[Bac02] J. Bacon, K. Moody, W. Yao, "A Model of OASIS Role-based Access Control and its Support for Active Security*", ACM Transactions on Information and System Security,* Volume 5 , Issue 4 , November 2002.

[Bar97] J. Barkley, A. Cincotta, D. Ferraiolo, S. Gavrila, and D.R. Kuhn. Role Based Access Control for the World Wide Web. In *Proceedings of 20th National Information System Security Conference*, NIST/NSA, 1997.

[Bel76] D. E. Bell and L. J. LaPadula, "Secure Computer System: Unified Exposition and Multics Interpretation," MTR-2997, MITRE Corp., Bedford, MA, March, 1976. Available as NTIS AD A023 588.

[Ber98] E. Bertino, C. Bettini, E. Ferrari, P. Samarati, "An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning," *ACM Transactions on Database Systems*, 23(3), September 1998, pages 231-285.

[Ber99a] E. Bertino, S. Castano, E. Ferrari, M. Mesiti, "Controlled Access and Dissemination of XML Documents", *Workshop On Web Information And Data Management*, November 1999.

[Ber99] E. Bertino, E. Ferrari, V. Atluri, "The Specification and Enforcement of Authorization Constraints in Workflow Management Systems," *ACM Transactions on Information and System Security*, 2(1), February 1999, pages 65-104.

[Ber01] E. Bertino, P. A. Bonatti, E. Ferrari, "TRBAC: A Temporal Role-based Access Control Model," *ACM Transactions on Information and System Security*, 4(3), August 2001, pages 191-233.

[Ber01] E. Bertino, S. Castano, E. Ferrari, "Securing XML Documents with Author X", *IEEE Internet Computing*, May-June, 2001.

[Bew89] D. F.C. Bewer, M. J. Nash, "The Chinese Wall Security Policy," *In Proceedings of the Symposium on Security and Privacy*, IEEE Computer Society, May 1989, pages 206-214.

[Bha03] R. Bhatti, "X-GTRBAC: An XML-based Policy Specification Framework and Architecture for Enterprise-Wide Access Control," Master's Thesis, School of Electrical and Computer Engineering, Purdue University, 2003.

[Bib77] K. J. Biba, "Integrity Considerations for Secure Computer Sytems," Technical Report ESD{TR{76-372, The MITRE Corporation, HQ Electronic Systems Division, Hanscom AFB, MA, April 1977.

[Bis98] J. Biskup, U. Flegel, Y. Karabulut, "Secure Mediation: Requirements and Design," In *Proceedings of 12th Annual IFIP WG 11.3 Working Conference on Database Security*, Chalkidiki, Greece, July 1998.

[Cla87] D. D. Clark, D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies", *IEEE Symposium on Security and Privacy*, 1987, pages 184-194.

[Cor90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, "Introduction to Algorithms," MIT Press, 1990.

[Cra03] J. Crampton, "Specifying and enforcing constraints in role-based access control," *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies*, Como, Italy, June 02 - 03, 2003, pages 43-50.

[Den76] D. Denning, "A Lattice Model of Security Information Flow", *Communications of ACM*, Vol. 19, 1976, pages 236-243.

[Ede99] J. Eder, E. Panagos, Michael Robinovich, "Time Constraints in Workflow Systems", *Proc. of 11th Int. Conf. on Adv. Inf. Systems Engineering* (CAiSE 99), Heidelberg, Germany, 1999.

[Fer93] D. F. Ferraiolo, D. M. Gilbert, N. Lynch, "An Examination of Federal and Commercial Access Control Policy Needs," In *Proceedings of NISTNCSC National Computer Security Conference*, Baltimore, MD, September 20-23, 1993, pages 107-116.

[Fer99] D. F. Ferraiolo, J. F. Barkley, D. R. Kuhn, "A Role-Based Access Control Model and Reference Implementation within a Corporate Intranet," *ACM Transaction on Information and System Security*, Vol. 2, No. 1, February, 1999, pages 34-64.

[Fer01] D, F. Ferraiolo , R. Sandhu , S. Gavrila, D. Richard Kuhn, R. Chandramouli, "Proposed NIST standard for role-based access control," *ACM Transactions on Information and System Security*, 4(3), August 2001, pages 224 - 274.

[Gao02] "Critical Infrastructure Protection: Significant Homeland Security Challenges Need to Be Addressed," GAO-02-918T July 9, 2002.

[Gav98] S. I. Gavrila , J. F. Barkley, "Formal Specification for Role Based Access Control User/role and Role/role Relationship Management," *Proceedings of the third ACM workshop on Role-based Access control*, Fairfax, Virginia, United States, October 22-23, 1998, pages 81-90.

[Gar96] S. Garfinkel, E. H. Spafford, "Practical UNIX & Internet Security", *O'Reilly & Associates, Inc.*, 2nd Edition, April 1996.

[Gar97] S. Garfinkel, E. H. Spafford, "Web Security & Commerce", *O'Reilly & Associates*, Inc., Sebastapol, CA, 1997.

[Gho98] A. K. Ghosh, "E-Commerce Security: No Silver Bullet", *Proceedings of the Twelfth IFIP WG 11.3 Working Conference on Database Security*, Greece, July 14-17, 1998.

[Giu95] L. Giuri, "A New Model for Role-based Access Control," In *Proceedings of 11th Annual Computer Security Application Conference*, New Orleans, LA, December 11-15 1995, pages 249-255.

[Giu97] L. Giuri. Role-based Access Control: A natural approach. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*. ACM, 1997.

[Gli98] Gligor, V.D., S.I. Gavrila, and D. Ferraiolo, "On the formal definition of separation-of-duty policies and their composition," *Proceedings 1998 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, Calif., May 1998), IEEE Computer Society, pages 172- 183.

[Gon96] L. Gong, X. Qian, "Computational Issues in Secure Interoperation," *IEEE Transaction on Software and Engineering*, Vol. 22, No. 1, January 1996.

[Gra72] G. Graham and P. Denning, "Protection -- principles and practice," *In Proc. Spring Joint Computer Conference*. AFIPS Press, 1972.

[Gra91] J. Gray, "Toward a Mathematical Foundation for Information Flow Security," *In Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society Press, 1991.

[Har76] M. H. Harrison, W. L. Ruzzo, J. D. Ullman, "Protection in Operating Systems", *Communications of the ACM*, Vol. 19, No. 8, 1976, pages 461-471.

[Hos92] H. H. Hosmer, "Metapolicies I", *ACM SIGSAC Data Management Workshop*, San Antonio, Texas, December, 1991, *ACM SIGSAC Review* 1992.

[Hos92] H. H. Hosmer, "Metapolicies II," *In Proceedings of the 15th NISTNCSC National Computer Security Conference*, NISTNCSC, United States Government Printing Office:1992-625-512:60546, 1992, pages 369-378.

[Iso86] Standard Generalized Markup Language (SGML), ISO 8879. Information Processing -- Text and Office Systems - Standard Generalized Markup Language (SGML), 1986

[Jaj97] S. Jajodia, P. Samarati, V. S. Subrahmanian, E. Bertino, "A Unified Framework for Enforcing Multiple Access Control Policies," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1997, pages 474-485.

[Jon95] D. Jonscher, K.R. Dittrich, "Argos – A Configurable Access Control System for Interoperable Environments" *Proceedings of the IFIP WG 11.3 Ninth Annual Working Conference on Database Security*, Rensselaerville, NY, August 1995.

[Jos01a] J. B. D. Joshi, A. Ghafoor, W. Aref, E. H. Spafford. Digital Government Security Infrastructure Design Challenges. *IEEE Computer*, Vol. 34, No. 2, February 2001, pages 66-72.

[Jos01b] J. B. D. Joshi, W. G. Aref, A. Ghafoor and E. H. Spafford. Security Models for Web-based Applications. *Communications of the ACM*, 44, 2 (Feb. 2001), pages 38-72.

[Jos02] J. B. D. Joshi, K. Li, H. Fahmi, B. Shafiq, A. Ghafoor, "A Model for Secure Multimedia Document Database System in a Distributed Environment", *IEEE Transactions on Multimedia: Special Issue of on Multimedia Databases,* Vol. 4, No. 2, June*,* 2002.pages 215-234.

[Kan99] S. Kandala and R. Sandhu. Extending the BFA Workflow Authorization Model to Express Weighted Voting. In *Research Advances in Database and Information Systems Security*, pages 145-159, Kluwer Academic Publishers, 1999.

[Ker02] A. Kern, "Advanced Features for Enterprise-Wide Role-Based Access Control," *Annual Computer Security Applications Conference*, 2002

[Kuh95] W. E. Kuhnhauser, M. K. Ostrowski, "A Formal Framework to Support Multiple Security Policies", *Proceedings of the 7th Canadian Computer Security Symposium*, Ottawa, Canada, May 1995.

[Kun99] D. R. Kuhn, "Mutual Exclusion of Roles as a Means of Implementing Separation of Duties in a Role-based Access Control System," *ACM Transactions on Information and System Security*, 2(2), 1999, pages 177-228.

[Lam71] B. Lampson, "Protection," *In the Princeton Symposium on Information Sciences and Systems*, March 1971. *Reprinted in ACM Operating Systems Review*, 8(1) (1974).

[Lam73] B. Lampson, "A note on the Confinement Problem," *Communications of the ACM*, 16(10), October 1973, pages 613-615.

[Lam77] L. Lamport, "Time, Cocks, and the Ordering of Event in a Distributed System," *Communications of the ACM*, 21(7), 1977, pages 558-565.

[Mcl90] J. McLean, "Security Models and Information Flow," *In Proceedings 1990 IEEE Symposium on Security and Privacy*, Oakland, CA, 1990, pages 180—187.

[Mcl94] J. McLean, "Security Models," In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

[Mof98] J. D. Moffet, "Control Principles and Role Hierarchies," In *Proceedings of 3rd ACM Workshop on Role-Based Access Control*, November 1998.

[Mof99] J. D. Moffet. E. C. Lupu, "The Uses of Role Hierarchies in Access Control," In *Proceedings of 4th ACM Workshop on Role-Based Access Control*, October, 1999.

[Nie92] M. Niezette and J. Stevenne, "An Efficient Symbolic Representation of Periodic time," In *Proc.First International Conference on Information and Knowledge Management*, 1992.

[Neu03] G. Neumann, M. Strembeck, "An Approach to Engineer and Enforce Context Constraints in an RBAC Environment", *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies*, Como, Italy, June 02-03, 2003, pages 65–79.

[Nya93] M. Nyanchama, S. L. Osborn, "Role-Based Security, Object-Oriented Databases and Separation of Duty", *SIGMOD Rec.* 22, 4, December 1993, pages 45-51.

[Nya94] M. Nyanchama, S.L. Osborn, "Access Rights Administration in Role-Based Security Systems," *Database Security VIII: Status & Prospects*, Biskup, Morgenstern and Landwehr, eds. North-Holland, 1994, pages 37-56.

[Nya95] M. Nyanchama, S. L. Osborn, "Modeling Mandatory Access Control in Role-Based Security Systems," *Database Security IX: Status and Prospects*, Spooner, Demurjian and Dobson, eds. Chapman & Hall, Aug. 1995, pages 129-144.

[Nya99a] M. Nyanchama and S. Osborn. The Role Graph Model and Conflict of Interest. *ACM Transactions on Information and System Security*, 2(1), 1999, pages 3-33.

[Osb97] S. L. Osborn, "Mandatory Access Control and Role-Based Access Control Revisited", *Proceedings of Second ACM Workshop on Role-Based Access Control*, November 1997.

[Osb00a] S. Osborn editor. *Proc. of the Fifth ACM Workshop on Role-Based Access Control*, Berlin, Germany, July 2000.

[Osb00b] S. L. Osborn, R. Sandhu, 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, February 2000.

[Pea02] L. Pearlman, V. Welch, Ian Foster, Carl Kesselman, S. Tuecke, "A Community Authorization Service for Group Collaboration," *2002 IEEE Workshop on Policies for Distributed Systems and Networks*.

[Pow00] R. Power, ""Tangled Web": Tales of Digital Crime from the Shadows of Cyberspace," Que/Macmillan Publishing, Aug. 31, 2000.

[San88] R. Sandhu, "The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes," Journal of the ACM, 35(2), 1988, pages 404-432.

[San91] R. Sandhu, "Separation of Duties in Computerized Information Systems", In *Database Security IV: Status and Prospects*. Elsevier North-Holland, Inc., New York, 1991, pages 179-189.

[San92] R.S. Sandhu, "The Typed Access Matrix Model," *In Proceedings IEEE Computer Society Symposium on Research In Security and Privacy*, Oakland, CA, May 1992, pages 122-136.

[San92b] R. Sandhu, "Lattice-based Enforcement of Chinese Walls," *Computers and Security*, 11(8), December 1992, pages 753—763.

[San93] R. Sandhu, "Lattice-Based Access Control Models", *IEEE Computer*, Vol. 26, No. 11, 1993.

[San94] R. Sandhu, P. Samarati, "Access Control: Principles and Practice", IEEE Computer, Sept 1994, pp 40-48.

[San95] R. Sandhu, editor. *Proc. of the First ACM Workshop on Role-Based Access Control*, Fairfax (VA), 1995.

[San96a] R. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman, "Role-Based Access Control Models," *IEEE Computer* 29(2), IEEE Press, 1996, pages 38-47.

[San96b] R. Sandhu, "Access Control: The Neglected Frontier (Invited Paper)," *In Proceedings of the First Australasian Conference on Information Security and Privacy*. Wolongong, Australia, June 23-26, 1996.

[San96c] R. Sandhu, "Role Hierarchies and Constraints for Lattice-based Access Controls," In E. Bertino, H. Kurth, G. Martella, and E. Montolivo Eds., *Computer Security - Esorics'96*, LNCS N. 1146, Rome, Italy, 1996, pages 65-79.

[San96d] R. Sandhu, "Role Hierarchies and Constraints for Lattice-Based Access Controls", *Proceeding Fourth European Symposium on Research in Computer Security*, Rome, Italy: Springer-Verlag, Published as *Lectures Notes in Computer Science*, Computer Security- *ESORICS96*.

[San97] R. Sandhu, editor. *Proc. of the 2nd ACM Workshop on Role-Based Access Control*, Fairfax (VA), 1997.

[San98a] R. Sandhu editor. *Proc. of the 3rd ACM Workshop on Role-Based Access Control*, Fairfax (VA), 1998.

[San98b] R. Sandhu, "Role-based Access Control," *Advances in Computers*, vol. 46, Academic Press, 1998.

[Sch96] B. Schneier, ""Applied Cryptography: Protocols, Algorithms, and Source Code in C," John Wiley & Sons, Inc., 1996.

[Sim97] R. Simon, M.E. Zurko, "Separation of Duty in Role-based Environments," In *Proc. 10th IEEE Computer Security Foundations Workshop*, June 1997.

[Tar97] Z. Tari, G. Fernandez, "Security Enforcement in the DOK Federated Database System", *Database Security X: Status and Prospects*, P. Samarati, R. Sandhu (eds), Chapman & Hall, 1997, pages 23-42.

[Tar97] Z. Tari, S. Chan, "A Role-Based Access Control for Intranet Security," *IEEE, Internet Computing*, Sept-Oct, 1997, pages 24-34.

[Tid98] J. Tidswell, J. Potter. A Dynamically Typed Access Control Model. In Proceedings of the Third Australasian Conference on Information Security and Privacy, July 1998.

[Tho97] R. K. Thomas, R.S. Sandhu, "Task-based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-oriented Authorization Management," *Proceedings of the IFIP WG11.3 Workshop on Database Security*, Lake Tahoe, California, August 11-13, 1997.

[Thu01] B. M. Thuraisingham, C. Clifton, A. Gupta, E. Bertino, E. Ferrari, "Directions for Web and E-Commerce Applications Security," *10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises* (WETICE 2001), 20-22 June 2001, Cambridge, MA, USA. IEEE Computer Society 2001.

[Woo93] T. Y. C. Woo, S. S. Lam, "Authorizations in Distributed Systems: A new Approach", *Journal of Computer Security*, 2(2, 3), 1993, pages 107-136.

[Vas94] J. V'asquez-G'omez, "Multidomain Security," *Computers & Security*, 13(2), 1994, pages 161-184.

[Vuo01] N. N. Vuong, G. S. Smith, Y. Deng, "Managing Security Policies in a Distributed Environment Using eXtensible Markup Language (XML)", *Symposium on Applied Computing*, March 2001.

[URLa] "eXtensible Markup Language (XML) 1.0 (Second)," W3C Recommendation 6 October 2000, http://www.w3.org/TR/REC-xml

[URLb] "XML Path Language (XPath) 2.0" Working Draft 16 August 2002, http://www.w3.org/TR/xpath20/

[URLc] W3C XML Schema, http://www.w3.org/XML/Schema

[URLd] XACML 1.0 Specification, http://xml.coverpages.org/ni2003-02-11-a.html

[URLe] Simple Object Access Protocol (SOAP), 1.1 http://www.w3.org/TR/SOAP/

APPENDIX A
Proofs of Theorems of Chapter 3

Proof of Theorem 3.1: **Proof of part 1**

By definition 3.3.5, the `Caused`(*t, EV, ST, Γ, RQ*) contains only those events that are caused at time *t* and satisfy all the constraints in *T*. Hence, to prove that the status update done by `computeST`(*t*) satisfies all the constraints in *Γ*, we need to prove that the update done is with respect to each of the non-blocked events in the `Caused`(*t, EV, ST, CT, Γ, RQ*). This is because the non-blocked events of `Caused`(*t, EV, ST, Γ, RQ*) are the only events that actually happen at time *t*. Since by definition 3.3.6, *EV*(*t*) = `Caused`(*t, EV, ST, Γ, RQ*), we can proceed by showing that each of the events of `Nonblocked`(*EV*(*t*) is considered by the algorithm. In addition, the algorithm needs to ensure that all the valid activation constraints are also satisfied by the updated information. We will proceed step by step.

Step 1: We discuss with respect to each of the cases listed in this step.

User-role role-permission deassignments: In this step, all the non-blocked deassignment events of *EV*(*t*) are considered. Since the presence of a *u-snapshot* associated with user *u* in U_r corresponding to the role *r* indicates that the role *r* has been assigned to user *u*, the removal of the *u-snapshot* corresponding to the user to whom the role *r* is deassigned correctly updates the effect of the deassignment event. Since *ut* contains all the activation status of a particular user associated with a role, the removal of the *u-snapshot ut* completely removes all such information. Similarly, permissions that are deassigned from a role are removed from the permission list P_r associated with the role. The next earliest change to these new values of U_r and P_r occur at time *t*+1. We also note that no activation constraints affect deassignments. Hence, the changes are according to condition (1).

Role-permission assignment: Here, we simply add the permission *p* to the set P_r associated with the role *r* if "`assign` *p* `to` *r*" is a non-blocked element in *EV*(*t*). The presence of *p* in set P_r associated with the role *r* indicates that *p* is assigned to *r* and hence this update correctly establishes the status of the assignment until it is changed by a deassignment at time *t*+1 or after.

Role-permission assignment: Here, we simply add a *u-snapshot*, say *ut*, associated with user *u* to the set U_r that is associated with the role *r* if "`assign` *r* `to` *u*" is a non-blocked element in *EV*(*t*). The presence of *ut* in set U_r associated with the role *r* indicates that *u* is assigned to *r* and hence this update correctly establishes the status of the assignment. Furthermore, the set of sessions and their durations are currently empty. The remaining parameters are set to initial default values of ∞. Thus, the updates reflect that an assignment event occurs and is in accord with condition (1).

Deactivation of roles: This step simply updates the effect of deactivation events. For each deactivation event, there is a session and the duration associated. Thus for a (*s*:`deactivate` *r* `for` *u*), the associated session *s* and its associated duration *d* are removed from the *u-snapshot* of *u* associated with role *r*. Note that session *s* may still be present in which other roles are still active. In such a case, the session *s* will be present in the *u-snapshot* of *u* associated with the other roles. Thus, removal of (*s, d*) is in accord with condition (1).

Step 2: In this step, all non-blocked events that disable roles are considered. Since a role *r* is disabled at *t*, its status is changed to `disabled`, which remains so until it is changed later, as there are no statements below step 2 that change *rt.status*; the only change that can occur to this parameter afterwards is at time (*t*+1), which occurs if the "`enable` *r*" event is non-blocked at time (*t*+1). As a role is disabled, all valid *per-role* activation constraints whose validity is not restricted by (*I, P*) or a duration *D* (and hence they are valid for each enabled duration of the role), must be reset to default values. This is done by each of the next set of IF statements. Furthermore, each *per-user-role* constraints must also be considered and corresponding associated parameter values reset. This is done by the FOR loop which considers each of the users assigned to the role being disabled. We note that, for *per-role* activation constraints of type (*I, P, C*) and (*D, C*), the updating of the corresponding parameter values is dictated by (*I, P*) and *D* respectively as is done in step 3 and is not affected by the disabling of a role. Hence, step 2 does the required update in accord with condition (1).

Step 3: The fact that a constraint of form (*I, P, C*) or (*D, C*) which was in *CT*(*t*-1) but is not in *CT*(*t*) implies that these constraints are not in effect anymore (disabled). This step considers all such *per-role* activation constraints and resets the values of the corresponding parameters such as d_{ra}, n_{ra}, etc., to ∞. These parameter values can only be altered at *t*+1 or later; hence, the update is in accord with condition (1).

Step 4: This step handles the effect of the enabling of a role *r*. First the status is updated to `enabled`. The first IF condition checks if a constraint of type *C1* in one of the forms (*I, P, C*), (*D, C*), or (*C*), is present. If it is, then d_{ra} is updated to the minimum of D_{active} in *C1* or the current value of d_{ra}. If $d_{ra} < \infty$ then it implies that some valid constraint is restricting the role activation time and is left unchanged. If $d_{ra} = \infty$, then it means no constraint of type *C1* has been active earlier. Thus, the update essentially conforms to the semantics and hence is in accord with condition (1). Similar arguments apply to the remaining IF statements of step 4.

Step 5: In this step, all non-blocked activation events are considered. Since a new activation of a role is being added, if there is a *per-role* cardinality constraint active at this time ($rt.n_{ra}<\infty$), then $rt.n_{ra}$ must be decremented, as required. The new value will be checked to control role activation events in the next run of `computeCauseSet` at (t+1). Note that we do not need to check if $rt.n_{ra}$=0, because the reason the activation event is in `nonblocked`($EV(t)$) is because no cardinality constraint has blocked the activation event as per definition 3.3.5. Hence, the simple decrement operation is adequate. A similar argument applies to the *per-user-role* cardinality constraint (the associated parameter is $ut.n_{ua}$).

Each of the IF statements that follows updates the user parameter based on the type of *per-user-role* or *per-role* constraint. The first IF statement checks to see if a *per-user-role* total activation constraint is active at time *t*. If it is active, then the corresponding user parameter d_{ua} is updated as specified. If the constraint form is (*C1*) then this value will be restricted for the duration d_{ua} or until the time the corresponding role *r* is disabled (in which case, step 2 will reset this value when it is updated next). The ELSE part considers the *per-role* constraint that applies to the activation of *r* by *u*. In that case, the default value specified is assigned to d_{ua}. We note that this default value, if not explicitly specified, is equal to the value specified for the role, for example, D_{active} in this case). Hence, the IF statement updates the user parameter as required. Similar arguments apply to the rest of the IF statements. The remaining duration for the activation of *r* by *u* is then added to sets S_u and D_u, so that they can be decremented at each time instant until they become 0. Hence, step 5 updates the required parameters in accord with condition (1).

Step 6: In this step, for each enabled role, the duration of each session is decremented. Similarly, the total active duration of the role is also adjusted. The decrement value represents the total value that will be decremented at the end of the interval (t, t+1). The else part simply decrements the value of d_{ra} by one. This is necessary because there may be a *per–role* constraint of type (*I, P, C*) or (*D, C*) on the role which is valid even when the role is disabled. Similarly, each user values are also decremented. Thus, step 6 updates all the duration values as required.

Hence, it follows that the condition of (1) is satisfied by $ST(t)$ produced by the algorithm.

Proof of part 2 and 3

We look at the complexity of each step and sum them up to get the overall complexity. The FOR loop of step 1 checks each of the events in that is non-blocked. At most, there are $(n_R.n_U)$ user-role deassignments, and $(n_R.n_P)$ role-permission assignments. Similarly, in the worst case, all user-role activations, i.e., $(n_R.n_{Sm})$, need to be deactivated. Thus, the worst case for step 1 is $O(n_R.(n_U + n_P + n_{Sm})$

Step 2 handles role-disabling events. At worst, all roles need to be disabled. The inner loop of step 2 repeats all users assigned to each role. Thus, the worst case for step 2 is $(n_R.n_U)$.

Step 3 checks for the *per-role* activation constraints active at the time. Since, there are n_R roles, the maximum number of such constraints is $4n_R$ as there are four types of per-role constraints, hence we have $O(n_R)$. The FOR loop of step 4 repeats at most n_R times. Step 5 is bounded by the maximum number of sessions allowed in the system; hence the worst case is $(n_R.n_{Sm})$. Step 6 repeats for each role, and the worst case occurs when the else part is executed, giving the worst case of $(n_R.n_U)$. Because n_R, n_U, n_P and n_{Sm} are each finite, we see that each step terminates. Hence the algorithm `computeST` terminates. Thus the complexity of the algorithm can be expressed as:

$O(n_R.(n_U + n_P + n_{Sm}))$.

APPENDIX B
Proofs of Theorems of Chapter 4

Proof of Theorem 4.1: Let u be assigned to S_H. If H is an I-hierarchy, u can only activate the seniormost role. Hence, $UAS(H, t) = \{\{S_H\}\}$. Similarly, if H is an A-hierarchy, u can activate all the combination of roles in the hierarchy. Furthermore, for each combination of roles, u acquires a unique set of permissions. Therefore, $UAS(H, t) = 2^X/\varnothing$ (we exclude the empty set). If H is an IA-hierarchy, if u can activates the seniormost role, he permissions of all the junior roles as well. Hence, $\{S_H\}$is an element of $UAS(H, t)$ and no other element can include S_H, as per Definition 4.2.1. Same argument applies to the seniormost roles of each of the sub-hierarchy of role S_H. Note that the each element of a sub-hierarchy of S_H can occur in combination with each element of another sub-hierarchy of S_H. Furthermore, elements of the UAS of different sub-hierarchies can be activated together to get different permission sets. Thus, all combinations of a set containing one element from the UAS of each sub-hierarchy can form an element with unique permission set. Thus, $UAS(H, t) = \{\{x_1\}\} \cup \{2^Z \mid Z = \bigcup_i z_i \text{ ; } z_i \in \text{UAS}(SubH_i\,(H))\}/\varnothing$. Here, each Z is a set of elements that contains elements belonging to one UAS element of each of the sub-hierarchies of role S_H.

Proof of Lemma 4.1: Let u be assigned to S_{Lm}. By definition 4.2.2, $S_{Lm} = S_{L1}$, $J_{Lm} = J_{L2}$ and $J_{L1} = S_{L2}$.

Case 1: First, consider $(<f_1>, <f_2>) = (\geq^t, <f_2>)$ s.t. $<f_2> \in \{\gtrsim^t, \succcurlyeq^t\}$. As the first hierarchy is an I-hierarchy, $UAS(L_1, t) = \{S_{L1}\} = \{x_1\}$ by theorem 4.1 and hence, none of the roles in L_2 can be activated by u. Therefore, we get $UAS(Lm, t) = UAS(L_1, t)$. Similarly, let $(<f_1>, <f_2>) = (<f_1>, \geq^t)$ where $<f_1> \in \{\gtrsim^t, \succcurlyeq^t\}$. As L_2 is an I-hierarchy, u cannot activate any junior roles. But u acquires all the permissions of L_2 when s/he activates S_{L2}, as L_2 is an I-hierarchy. Because of $<f_1>$, u can activate J_{L1} ($= S_{L2}$) and acquire all the permissions of L_2. Hence, $UAS(Lm, t) = UAS(L_1, t)$. Therefore, if $\geq^t \in \{\gtrsim^t, \succcurlyeq^t\}$, then $UAS(Lm, t) = UAS(L_1, t)$.

Case 2: Consider $(<f_1>, <f_2>) = (\succcurlyeq^t, \gtrsim^t)$. Here, first we note that $UAS(Lm, t)$ must contain the following:

1. all the activable elements of L_1, i.e. all elements of $UAS(L_1, t)$,
2. all the activable elements of L_2, i.e. all elements of $UAS(L_2, t)$, and
3. all the possible combination of activable elements in L_1 and L_2.

From Theorem 4.1, $UAS(L_1, t) = 2^{X1}/\varnothing$ and $UAS(L_2, t) = \{\{J_{L1}\}, , \ldots, \{x_n\}\}$. We need to show that $UAS(Lm, t) = UAS(L_{1U}, t) \cup UAS(L_2, t) \cup (UAS(L_{1U}, t) \otimes UAS(L_2, t))$ exactly contains the elements mentioned in 1 through 3. We see that $UAS(L_1, t) = UAS(L_{1U}, t) \cup (UAS(L_{1U}, t) \otimes \{\{J_{L1}\}\})$. But as $J_{L1} = S_{L2}$, $J_{L1} \in UAS(L_2, t)$. Therefore, $(UAS(L_{1U}, t) \otimes \{\{J_{L1}\}\}) \subseteq (UAS(L_{1U}, t) \otimes UAS(L_2, t))$. Hence, elements of both $UAS(L_1, t)$ and $UAS(L_2, t)$ are in $UAS(Lm, t)$. We further note that as there are no common roles in L_{1U} and L_2; hence, $UAS(L_{1U}, t)$ and $UAS(L_2, t)$ are disjoint. It is easy to see that $(UAS(L_{1U}, t) \otimes UAS(L_2, t))$ consists of all the combinations of the activable sets of $UAS(L_{1U}, t)$, and $UAS(L_2, t)$. $(UAS(L_{1U}, t) \otimes UAS(L_2, t))$ is disjoint from $UAS(L_{1U}, t)$ and $UAS(L_2, t)$ as each of its role sets contains roles from the elements of both $UAS(L_{1U}, t)$ and $UAS(L_2, t)$. (Hence, cardinality computation is simply $|UAS(Lm, t)| = |UAS(L_{1U}, t)| \cup |UAS(L_2, t)| \cup |(UAS(L_{1U}, t) \otimes UAS(L_2, t))|$).

Case 3: The case for $(<f_1>, <f_2>) = (\succsim^t, \succcurlyeq^t)$ is similar to that of case 2.

Proof of Theorem 4.2: Let u be assigned to S_{Lm}. By definition 4.2.2, $S_{Lm} = S_{L1}$, $J_{Lm} = J_{LM2}$ and $J_{L1} = S_{L2}$.

Case 1 ($<f_1> = \geq^t$): As L_1 is an *I*-hierarchy, by reasoning similar to the case for $(<f_1>, <f_2>) = (\geq^t, <f_x>)$ in the proof for Lemma 4.1, the result follows immediately.

Case 2($<f_1> = \succcurlyeq^t$): As L_1 is an *A*-hierarchy, if $<f_x> = \geq^t$ then no roles below S_{Lx} can be in the set $UAS(Lm, t)$; hence, $UAS(Lm, t) = UAS(L_1, t)$. If $<f_x> = \succsim^t$, then the case is similar to that of Lemma 4.1.

Case 3: As L_1 is an *IA*-hierarchy, if $<f_x> = \geq^t$ then no roles below S_{L2} can be in the set $UAS(Lm, t)$; hence $UAS(Lm, t) = UAS(L_1, t)$. If $<f_x> = \succsim^t$ then the case is similar to that of Lemma 4.1 except for the fact that the LM_2 is not necessarily an *A*-hierarchy. Hence, by following similar reasoning, the result follows.

Proof of Theorem 4.3: Here, $UAS(H_1)$ has two parts, namely S_1 and S_2. S_1 constitutes the subset of $UAS(H, t)$ which are either from LM_1 or from H_1. Hence $S_1 = (UAS(LM_1, t) \cup UAS(H_1, t))$. Similarly, S_2 constitutes the subset of $UAS(H, t)$ resulting from the combination of $UAS(LM_1, t)$ and $UAS(H_1, t)$. Here we note that there may be common elements. At the least, the senior-most role is common to both. Therefore, $S_2 = (UAS(LM_1, t)/B \otimes UAS(H_1, t)/B)$ contains all combinations of the possible elements of the components LM_1 and H_1. Here B represent elements of $UAS(LM_1, t)$ and $UAS(H_1, t)$ that have common elements. However, $I = S_1 \cup S_2$ may not still be the required activable set. This is because in both LM_1 and H_1, the same two roles may have a hierarchical relation (direct or derived) showing alternative relations between the roles. The result of the two alternative relations is that we may have a new derived relation (as discussed in Section 4.3). For example in LM_1, x and y may be related by an A-relation and hence appear together in an element of the $UAS(LM_1, t)$, whereas in H_1, x and y may be related by an I-relation (or an IA-relation). As a result the derived relation between x and y becomes an IA-relation in H. Thus, x and y should not appear together in an element of $UAS(H, t)$. C determines exactly those elements in $UAS(H, t)$ that are IA-related (directly or derived); hence, $UAS(H, t) = I/C$.

Proof of Theorem 4.4: We note that H_1 and H_2 have some pair-wise related roles only differing in the hierarchical relation. Hence the hierarchy structure is the same and H_2 is a monotype whereas H_1 can be monotype or hybrid type. We prove this case-wise. Let $X = \text{Role}(H_2)$.

Case 1 (*H_1 is monotype*): Assume that H_2 is an I-hierarchy, Obviously, $P_{max}(H_2, t) = P(S_{H2}, t) = P(X, t)$. We have three cases for H_1. If H_1 is also I-hierarchy then by they are obviously the same hierarchy. Let H_1 be an A-hierarchy. Then a user assigned to S_{H1} can activate all the roles at once in a session. Hence, $P_{max}(H_1, t) = P(X, t) = P_{max}(H_2, t)$. Now, let H_1 be an IA-hierarchy. Thus, by activating role S_{H2}, a user can acquire all the permissions of roles in X; i.e. $P_{max}(H_1, t) = P(S_{H1}, t) = P(X, t) = P_{max}(H_2, t)$. Thus, all monotype hierarchies are AC-equivalent.

Case 2 (*H_1 is a hybrid type*): Here it is possible that H_1 has a linear component $LM_i = (LM', L_x, LM_{mid}, L_y, LM'')$. In such a case, a user assigned to the senior-most role cannot acquire the permissions associated with roles in $(X_y \cup X'')$ (considering $L_y = (X_y, <f_y>)$ and $L'' = (X'', <f''>)$) as f_x is an I-relation. Thus, a user assigned to the senior-most role can acquire the permission set $P(X, t)$ in H_2, whereas he can acquire only the permission set $P(X/(X_y \cup X''), t)$ in H_1. However, if such a component is not presents then by Theorem 4.1 and 4.2, the user acquires the permission set $P(X, t)$ in both H_1 and H_2. Hence, we get the result.

Proof of Theorem 4.5 (Soundness of rules *R*1-*R*4): We prove this by taking all the possible cases of h that can be derived from each rule. Let us assume that user u is assigned only to the senior role x.

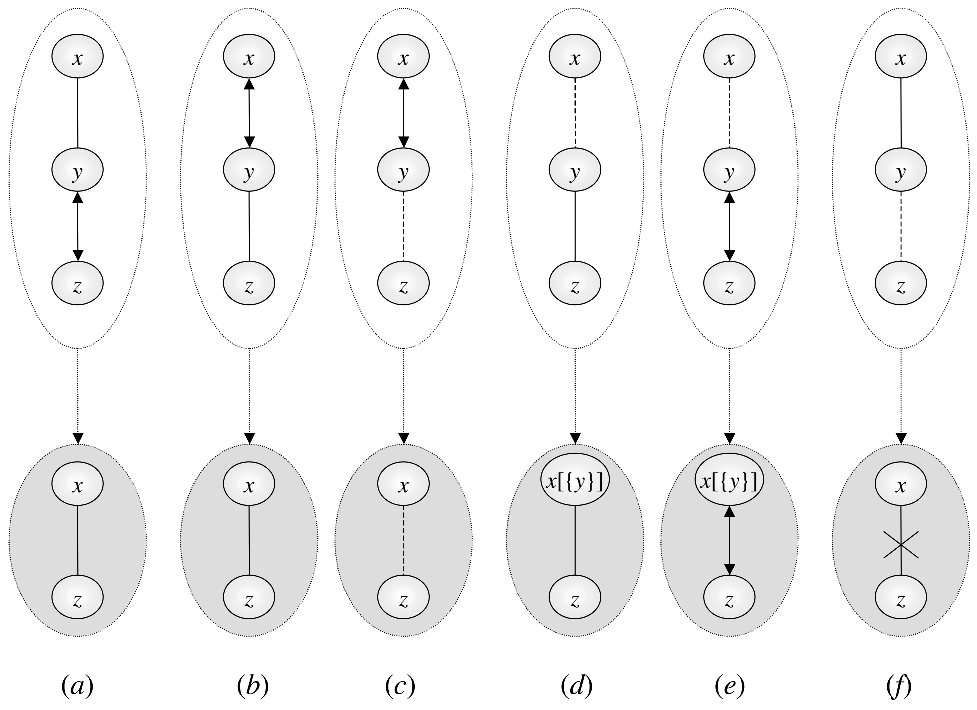


Fig. B.1. Derived hierarchical relation for two consecutive types (rule ***R*2**)

***Case 1*: *h* is derived from rule *R*1**: Here, (x<f>z) because (x<f>y) and (y<f>z). Let us consider <f> = $\geq^t$. Assume that permission p can be acquired through role z at time t; i.e., `can_be_acquired`(p, z, t) holds. As $y \geq^t z$, p can be acquired through role y at time t. Again, as $x \geq^t y$, p can also be acquired through role x at time t. Similarly, $x \geq^t z$ also indicates that p can also be acquired through role x at time t. Hence, the result follows. The cases for the A-hierarchy and IA-hierarchy can be shown similarly.

***Case 2*: *h* is derived from rule *R*2**: Fig. B.1 depicts all the six possible combinations of (x<f_1>y) and (x<f_2>z). Figures 4.14(a)- 4.14(e) correspond to the cases ***R*2.1**(i), ***R*2.1**(ii), ***R*2.3**, ***R*2.3**(i) and ***R*2.3**(ii) respectively. Rule ***R*2.1**(i) is straightforward. As both hierarchical relations allow *permission inheritance*, role x can inherit all the permissions of role z. However, u cannot activate role y and hence he cannot activate role z. Thus, roles x and z are related by an I-relation only. The rules ***R*2.1**(ii) and ***R*2.2** can be shown in a similar way.

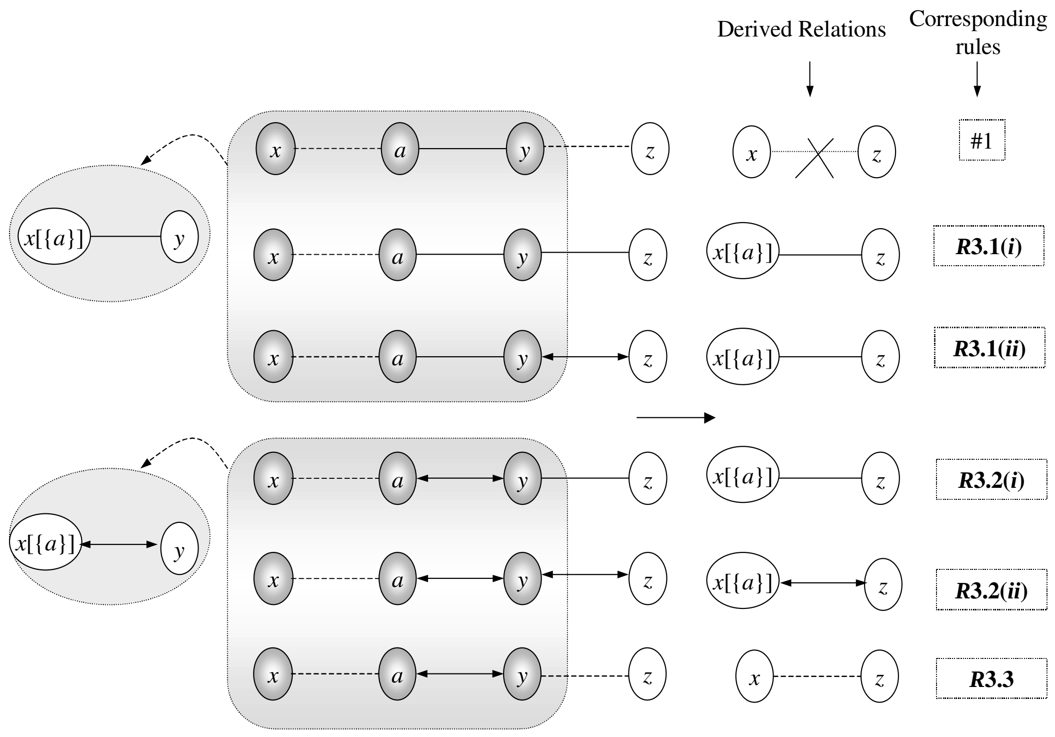


Fig. B.2. Derived relations in a general linear hierarchyusing rules ***R***3 where set *B* is empty

In rule ***R*2.3**(*i*) (refer to Fig. B.1(*d*)), *x* and *y* are related by an *A*-relation; hence, *u* can also activate role *y*. However, as *y* and *z* are related by an *I*-relation, *u* cannot activate role *z*. *u* can still acquire the permissions of role *z* without activating it, but to do that s/he has to activate role *y*. Hence, we get a *conditioned* derived relation $x[\{y\}]\geq^t z$, as per definition 4.3.1. In rule ***R*2.3**(*ii*) (refer to Fig. B.1(*e*)), *u* can activate *z*. However, *u* can also acquire *z*'s permissions without activating it, but to do that s/he has to activate role *y*. Hence, we again get, as per definition 4.3.1, a *conditioned* derived relation $x[\{y\}](\{y\})\succsim^t z$. We note that the combination shown in Fig. B.1(*f*) does not derive any relation between *x* and *z*. Here, the *I*-relation between *x* and *y* prohibits the activation of role *y* and hence that of role *z* by *u*. The *A*-relation between *y* and *z* prohibits the inheritance of *z*'s permissions through *y* and hence, *u* cannot directly inherit *z*'s permissions. Thus, *u* can neither activate *z* nor inherit its permissions. Hence, *x* and *z* are not hierarchically related. Thus, Fig. B.1 shows all the possible cases in which a hierarchical relation (either direct or *unconditioned* derived) of one type follows another type (direct or derived). Thus, ***R*2** captures completely all such cases.

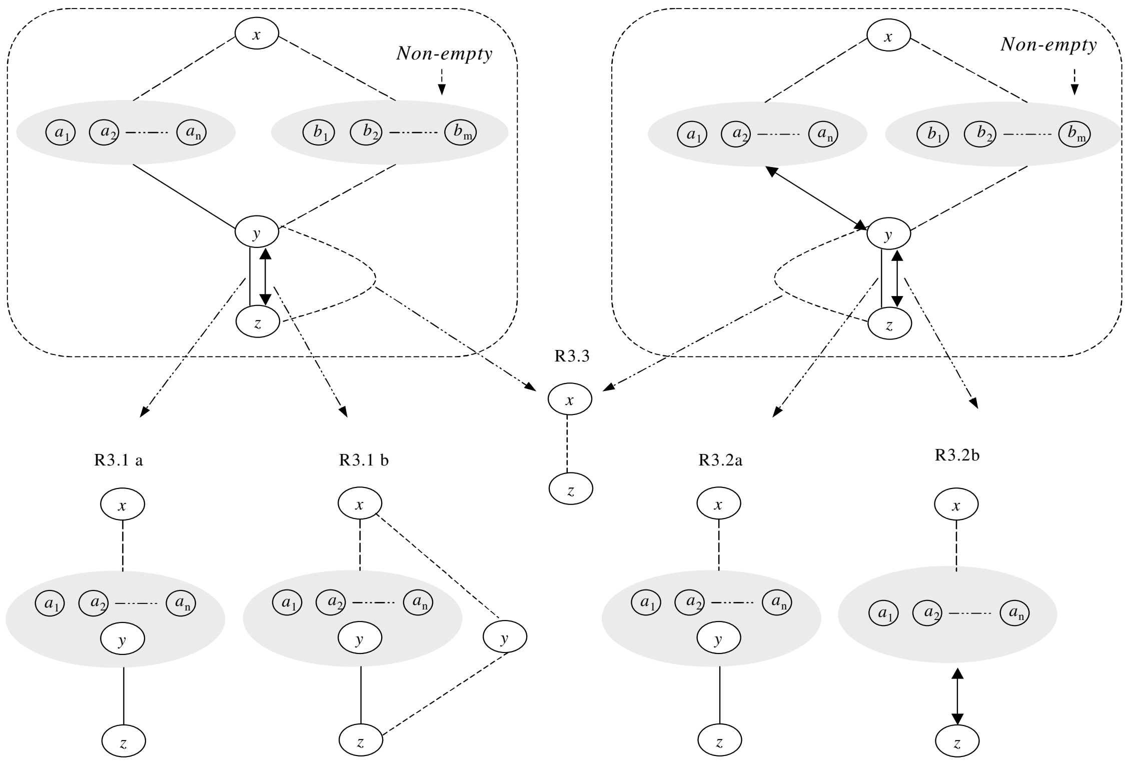


Fig. B.3. Derived relations in a general linear hierarchy using rules ***R***3 with set *B* is non-empty

***Case 3*: *h* is derived from rule *R*3**: ***R*3** deals with cases in which a *conditioned* derived relation is immediately followed by an *unconditioned* relation in a hierarchical path. Fig. B.2 illustrates graphically all such possible combinations for $A = \{a\}$ and $B = \varnothing$. The combination labeled #1 that corresponds to $(x\,[A]\geq^t y)\wedge(y \succcurlyeq^t z)$ does not derive any new relation. This is because $(a\geq^t y)$ is followed by $(y \succcurlyeq^t z)$ (see Figure 4.14(*f*)). The remaining combinations correspond to the five cases of ***R*3** in the Fig. B.2.

In ***R*3.1(*i*)**, corresponding to rule ***R*3.1.*a***, the *conditioned derived* relation between *x* and *y* is an *I*-relation. As the relation between *y* and *z* is also an *I*-relation, *u* can still acquire the permissions of *z* through the activation of a role in [*A*] (role *a* in Fig. B.2) as *z*'s permissions can be acquired through *y*. Hence, the result is the *conditioned derived* relation $x[\{a\}]\geq^t z$. Similarly, in rule ***R*3.1(*ii*)**, corresponding to rule ***R*3.1.*b***, the presence of a more restrictive *conditioned I*-relation before the *IA*-hierarchy between *y* and *z* prohibits *u* to activate *z*. But, because of the *IA*-hierarchy, *z*'s permissions can be inherited through *y* and hence through the activation of a role in *A*. Thus, the *conditioned derived* relation $x[\{a\}]\geq^t z$ holds. In rule ***R*3.2(*i*)**, corresponding to rule ***R*3.2.*a***, we see that a user assigned to x can acquire y's permissions by activating a role in A. But the I-relation between y and z allows z's permissions to be inherited through y and hence by the activation of a role in A. However, the same I-relation between y and z prohibits her/him to activate z. Hence, the result is a *conditioned* I-relation $x[\{a\}]\geq^t z$. In rule ***R*3.2(*ii*)**, corresponding to rule ***R*3.1.*a***, because of IA-relation between y and z, u can activate z. At the same time, u can also acquire through y all of z's permissions by simply activating a role in A. Hence, the inferred rule is $x\,[A]\succsim^t z$.

In rule ***R*3.3**, corresponding to rule ***R*3.3**, u can activate y because of the IA-relation. The A-relation between y and z allows u to activate z also. However, because of the A-relation between y and z, u cannot acquire z's permissions without activating z. Hence, the result is ($x\succcurlyeq^t z$). Note that it is not a *conditioned* derived relation. Again, we note that ***R*3** captures rules for deriving any new derived relations from all possible combinations of a *conditioned derived* relation followed by an unconditioned relation.

Fig. B.3 shows the same cases when set B is non-empty. The proof is similar to that for the above case, except that whenever the new derived relation between x and z is an A or IA-hierarchy, then the set is non-empty.

***Case 4*: *h* is derived from rule *R*4**: The first rule ***R*4.1** is a trivial case and is straightforward. In rule ***R*4.2**, we consider those alternate paths that have different *unconditioned* relations. When we have ($x\geq^t y$) and ($x\succcurlyeq^t y$), u can directly inherit y's permissions through the first relation and at the same time u can activate y using the second relation, hence ($x\succsim^t y$). Similarly, when one of the two relations is ($x\succsim^t y$), u can inherit y's permissions directly as well as activate y. Thus, no matter what the other relation is, we have the derived relation ($x\succsim^t y$). In rule ***R*4.3**, we have various cases in which one relation is a *conditioned derived* relation and the other is an *unconditioned* relation, as depicted in Fig. B.4. In ***R*4.3a**, both the hierarchical relations are of the same type. If the relation is I-hierarchy then the *unconditioned* relation allows direct inheritance of role y's permissions through role x; hence, the resulting derived relation is an *unconditioned* I-relation. Similarly, when the relation is an IA-hierarchy, y's permissions are inherited by simply activating role x using the *unconditioned* relation and hence the derived relation is not *conditioned*. In rule ***R*4.3b**, the first relation is a *conditioned* one. When <*f*> is an I-relation, as in Fig. B.4, then it means that through that hierarchical path u can acquire y's permission by activating a role in A. Furthermore, x and y also is related by A-hierarchy through sets B and C. Note that B and/or C may contain role x itself. The activation path is simply the union of the activation paths of the two paths. The case where <*f*> is an IA-relation is similar.

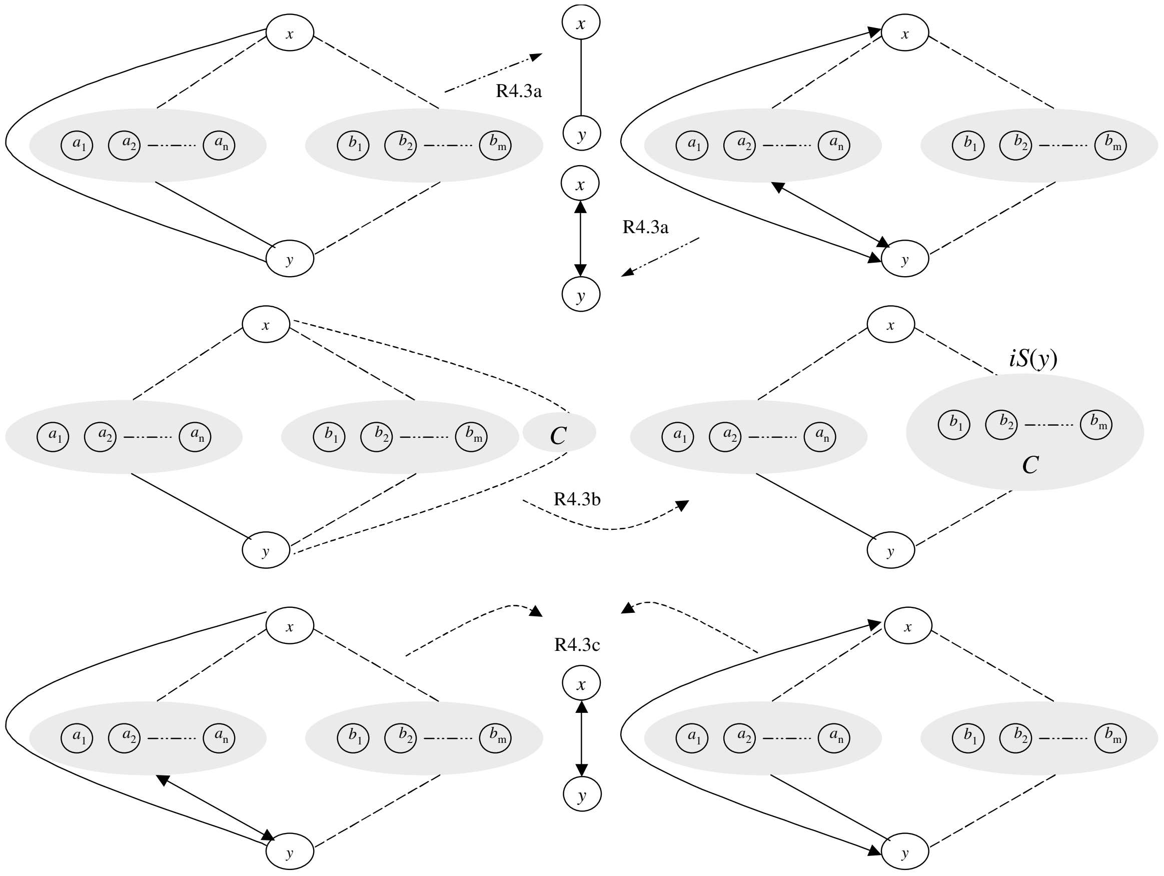


Fig. B.4 Derived relations for rules ***R***4.3

In rule ***R*4.3c**, we have *I* and *IA*-relations, as shown in Fig. B.4. In the first case, the first path allows an *A*-relation between *x* and *y* through the conditioned relation, either through roles in *A* or *B*, whereas the alternate path allows the *I*-relation. Hence, the derived relation is an *IA*-relation. In the second case, the *IA*-path allows both the semantics, hence the result is the *IA*-relation between *x* and *y* ***R*4.4** captures cases where the both alternate paths are *conditioned* (Fig. B.5). Rule ***R*4.4a** deals with the cases where both the alternate derived relations are the same. In rule ***R*3.4b**, both the alternates have *conditioned* derived relations but are of different types: *I* and *IA*-hierarchy. In the first case, there is a *conditioned I*-hierarchy portion on the first hierarchical path through A_1 and a conditioned *IA*-hierarchy portion on the second hierarchical path through A_2. As the path through A_2 provides an *IA*-relation, this means A_2 provides both *I* and *A*-relations. Now, by taking the union of the *I* and *A*-paths separately, we get the result. The second case is similar to the first one. Thus, the inference rules covers all possible combinations. Hence, the result follows.

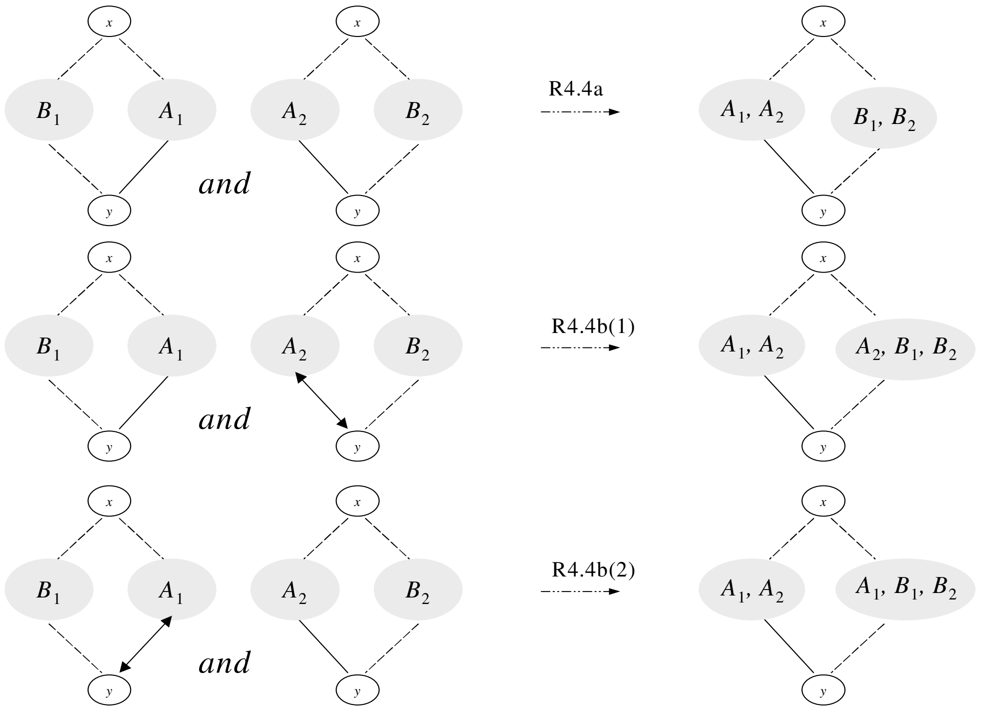


Fig. B.5. Derived relations for rules ***R***4.4

Before we prove Theorem 4.6, *we first prove the following lemmas:*

Lemma 4.6.1 (Completeness of rule *R*1 in monotype linear hierarchy): *Given a monotype linear hierarchy L, rule* **R1** *is* complete *with respect to L*; *that is, if for any pair of roles* $x, z \in$ Roles(L) $\neg$ L[***R*1**] $\models h_{x,z}$, *then* $L \not\approx L \cup \{h_{x,z}\}$; *i.e., the hierarchies L* and $L' = L \cup \{h_{x,z}\}$ *are not* authorization consistent.

Lemma 4.6.2 (Completeness of rules *R*1-*R*3 in a hybrid linear hierarchy): *Given a hybrid linear hierarchy Lh, rules* **R1-R3** *are* complete *with respect to Lh*; *that is, if for any pair of roles* $x, z \in$ Roles(Lh), $\neg$ Lh[***R*1-*R*3**] $\models h_{x,z}$, *then* $Lh \not\approx Lh \cup \{h_{x,z}\}$, *i.e., the hierarchies Lm* and $Lm' = Lm \cup \{h_{x,z}\}$ *are not* authorization consistent.

Proof of Lemma 4.6.1: Assume otherwise, i.e., there exists a relation $h_{x,z} = x\langle f\rangle z$, such that $\neg$ L[***R*1**] $\models h_{x,z}$ but $L \approx L'$. As $\neg L$[***R*1**] $\models h_{x,z}$, it implies that there exists no y such that $(x\langle f\rangle y)\wedge(y\langle f\rangle z)$ holds under L. It is easy to see that $(x\langle f\rangle y)\wedge(y\langle f\rangle z)$ cannot hold *iff* the following hold under L:

1. for all y such that $(x\langle f\rangle y)$, we have $\neg(y\langle f\rangle z)$, or
2. for all y' such that $(y'\langle f\rangle z)$, we have $\neg(x\langle f\rangle y')$,

Let us first consider $\langle f\rangle \in \geq^t$, i.e. $h_{x,z} = x\geq^t z$. Note that there is no direct hierarchical relation between x and z as $h_{x,z}$ is not in L. The first condition above indicates that x can inherit permissions from y because of the relation ($x \geq^t y$), but because $\neg(y \langle f \rangle z)$ also holds for all such y, x cannot inherit from z. Similarly, the second condition indicates that none of the roles that are senior to z is a junior of x. Therefore, x cannot inherit z's permissions. Hence, it follows that u cannot acquire permissions from z under hierarchy L. But as $h_{x,z}$ is in L', u can acquire z's permissions under L'. Hence, it follows that $L \not\approx L'$, contradicting our assumption. Thus, if $\neg L[\boldsymbol{R1}] \models h_{x,z}$, then L and L' are not authorization consistent.

Proof Lemma 4.6.2: As Lh is a *hybrid* linear hierarchy, by definition 4.4 we can write $Lh = \{L_1, L_2, \ldots, L_n\}$, where each L_i is a monotype linear hierarchy. Let L_i be $x_{i(1)} \langle f_i \rangle x_{i(2)} \langle f_i \rangle \ldots \langle f_i \rangle x_{i(|Li|)}$. Then by Lemma 4.6.1, we get the derived relations $x_{i(\pi i)} \langle f_i \rangle x_{i(\eta i)}$ for $1 \leq \pi_i \leq (|L_i|-2)$ and $3 \leq \eta_i \leq |L_i|$. For each linear component, the derived set is complete as per Lemma 4.6.1. But, we know that for $i = 2$ to n, $S_{Li} = J_{L(i-1)}$ by definition 4.4. To show completeness of ***R*1-*R*3** with respect to Lh, we show that any derived relation between a role in L_1 with that of a role in L_j for $2 \leq j \leq n$ can be inferred from ***R*1-*R*3**. To do that, we use induction on the following hierarchical chain from an arbitrary role $x_{1(\pi 1)}$ in L_1 to an arbitrary role $x_{n(\eta n)}$ in L_n:

$$Lh_c = \underline{x_{1(\pi 1)} \langle f_1 \rangle x_{1(|L1|)}} \, \overline{\langle f_2 \rangle x_{2\,(|L2|)} \ldots.} \, \underline{x_{(n-1)\,(|L(n-1)|)} \langle f_n \rangle x_{n\,(\eta i)}}$$

Note that each hierarchical relation in this chain is a relation derived in each component using ***R*1**.

Basis: Consider $n = 2$, i.e., $Lh_c = x_{1(\pi 1)} \langle f_1 \rangle x_{1(|L1|)} \langle f_2 \rangle x_{2(\eta 2)}$. In the proof for soundness of ***R*2** (Fig. B.1), we showed that ***R*2** captures all possible combinations of $\langle f_1 \rangle$ and $\langle f_2 \rangle$. Hence, as ***R*2** is sound, by employing an argument similar to the proof of Lemma 4.6.1, it follows that ***R*2** is complete with respect to Lh_c. Note that as $x_{1(|L1|)} \langle f_2 \rangle x_{2(|L2|)}$ by ***R*1**. Rules ***R*1-*R*3** is complete for $Lh_c = x_{1(\pi 1)} \langle f_1 \rangle x_{1(|L1|)} \langle f_2 \rangle x_{2(|L2|)}$

Induction Hypothesis: Assume that rules ***R*1-*R*3** are complete for $Lh_c = x_{1(\pi 1)} \langle f_1 \rangle x_{1(|L1|)} \langle f_2 \rangle x_{2\,(|L2|)} \ldots. x_{(n-1)\,(|L(n-1)|)}$. Now we need to show that they are complete for $Lh_c = x_{1(\pi 1)} \langle f_1 \rangle x_{1(|L1|)} \langle f_2 \rangle x_{2\,(|L2|)} \ldots. x_{(n-1)\,(|L(n-1)|)} \langle f_n \rangle x_{n\,(\eta n)}$. As ***R*1-*R*3** are complete $Lh_c = x_{1(\pi 1)} \langle f_1 \rangle x_{1(|L1|)} \langle f_2 \rangle x_{2\,(|L2|)} \ldots. x_{(n-1)\,(|L(n-1)|)}$, we can deduce a derived relation between $x_{1(\pi 1)}$ and $x_{(n-1)(|L(n-1)|)}$, which is either

1. $(x_{1(\pi 1)} \langle f \rangle x_{(n-1)(|L(n-1)|)})$ where $\langle f \rangle \in \{\geq^t, \succcurlyeq^t, \succsim^t\}$, or
2. $(x_{1(\pi 1)}[A](B) \langle f \rangle x_{(n-1)(|L(n-1)|)})$ where $\langle f \rangle \in \{\geq^t, \succsim^t\}$ and $A, B \subseteq \{x_{1|L1|}, x_{2|L2|}, \ldots., x_{(n-1)|L(n-1)|}\}$.

Assume that the derived relation is *unconditioned*, i.e., it is $(x_{1(\pi 1)} \langle f \rangle x_{(n-1)(|L(n-1)|)})$. But as pointed out in the proof for Soundness of ***R*1** – ***R*4**, rule ***R*2** can be employed over $(x_{11} \langle f \rangle x_{(n-1)|L(n-1)})$ and $(x_{(n-1)|L(n-1)|} \langle f_n \rangle x_{n|Ln|})$ to derive any relation between $x_{1(\pi 1)}$ and $x_{n|Ln|}$ (i.e., as indicated earlier, all combinations of relations $\langle f \rangle$ and $\langle f_n \rangle$ are captured by ***R*2**). As already argued in the induction basis, ***R*2** is complete for such cases.

Now suppose the derived relation is $(x_{1(\pi 1)}[A](B)\langle f \rangle x_{(n-1)|L(n-1)|})$. We know that $\langle f \rangle$ can only be $\geq^t$ or $\gtrsim^t$. But as mentioned in the proof of the Soundness Theorem, ***R*3** completely captures all the combinations where an *unconditioned* relation follows a conditioned derived relation. This argument can be easily extended to any arbitrary pair of roles. Hence, using an argument similar to that in the proof of Lemma 4.6.1, it follows that ***R*1-*R*3** is complete with respect to a (derived) *hybrid* linear hierarchy Lh_c. Using the same technique we can easily prove that a relation between any role of L_i with that of L_j $1 \leq i < j$ is completely determined by rules ***R*1-*R*3**. Hence, ***R*1-*R*3** is complete with respect to the *hybrid* linear hierarchy *Lh.*

Theorem 4.6 (Completeness of rules *R*1-*R*4): We prove this by considering various cases of *H*.

***Case* 1: *H* is a linear hierarchy**: If it is a monotype linear hierarchy, we can see that only ***R*1** applies as all other rules involve more than one hierarchy type or more than one relation between the same pair of roles. Hence from Lemma 4.6.1, it follows that ***R*1-*R*4** is complete with respect to *H*. If *H* is a *hybrid* linear hierarchy, then that means only one relation can exit between a pair of roles; i.e., there are no alternative hierarchical paths between the roles. Thus, only rules ***R*1** through ***R*3** apply. Hence, from Lemma 4.6.2, it follows that ***R*1-*R*4** is complete with respect to *H*.

***Case* 2: *H* is a not a linear hierarchy**: By definition 4.2.3, we can write $H = \{LH_1, LH_2, \ldots, LH_m\}$, where each component LH_i is a linear hierarchy (*hybrid* or *monotype*). From Lemma 4.6.1 and Lemma 4.6.2, we can see that for each LH_i, rules ***R*1-*R*3** are complete. The only remaining case is the case where two or more hierarchical paths can exist between a pair of roles. Such a case can occur only in a hierarchy that is not a linear hierarchy. Now, we need to show that when multiple hierarchical paths exist between a pair of roles, rule ***R*4** provides a basis for inferring all derivable relations. First let us consider the following *n* hierarchical relations between roles *x* and *y:* $h_{xy(\pi 1)}, h_{xy(\pi 2)}, \ldots, h_{xy(\pi n)}$, which corresponds to the linear components $LH_{\pi 1}, LH_{\pi 2}, \ldots, LH_{\pi n}$ for $n \leq m$ and $\{\pi_1, \pi_2, \ldots, \pi_n\} \subseteq \{1, 2, \ldots, m\}$. It is easy to see that each of these relations between *x* and *y* can be completely derived using rules ***R*1** through ***R*3** as each is derived within a linear component. Now, we show that ***R*4** completely covers all the possible derived rules that can be inferred by using these relations between x and y. Again, we use induction.

Basis: Let $n = 2$. Then, we have only two relations between x and y deduced in components $LH_{\pi 1}$, $LH_{\pi 2}$, which are $h_{xy(\pi 1)}$, $h_{xy(\pi 2)}$. We note that $h_{xy(\pi 1)}$, $h_{xy(\pi 2)}$ can be any of the *unconditioned* relations or the *conditioned* derived relations. As noted in the proof of the Soundness Theorem, all possible combinations are captured by the rules in ***R*4**. Hence, applying an argument similar to that used in the proof for Lemma 4.6.1, it follows that ***R*4** is complete with respect to the two relations $h_{xy(\pi 1)}$, $h_{xy(\pi 2)}$ between the same pair.

Induction hypothesis: Assume that ***R*4** is complete with respect to the n-1 relations between the same pair of roles x and y.

Induction: Let $h_{xy(\pi)}$ be the relation between x and y derived from n-1 different relations between them using rule ***R*4**. Now we have two relations between x and y: $h_{xy(\pi)}$ and $h_{xy(\pi n)}$. It is easy to see that $h_{xy(\pi)}$ is one of the *unconditioned* relations because of the application of rules ***R*4.1**, ***R*4.2**, ***R*4.3a**, and ***R*4.3c,** or it is the *conditioned* derived relation with a possibly bigger set on which the relation is conditioned, as is possible because of ***R*4.3b** and ***R*4.4**. But the same rules also apply to $h_{xy(\pi)}$ and $h_{xy(\pi n)}$. This is because, the rules in ***R*4** cover all possible combinations of *conditioned* and/or *unconditioned* alternate relations. Thus, by applying an argument similar to that used to prove Lemma 4.6.1, it follows that ***R*4** is complete for all n different relations.

Hence, it follows that rules ***R*1**- ***R*4** are complete with respect to a hierarchy.

APPENDIX C
Proofs of Theorems of Chapter 6

Proof of Lemma 6.1: (**Correctness of** `TransformPR`) : Let us consider an arbitrary user u such that $(u \underset{t}{\overset{Cf_{in}}{\Rightarrow}} p)$. We need to show that $(u \underset{t}{\overset{Cf_{out}}{\Rightarrow}} p)$. Since $(u \underset{t}{\overset{Cf_{in}}{\Rightarrow}} p)$, the following conditions must be true at time t for Cf_{in}:

(1) there is a constraint $\{X,$ *pr*:`assign/deassign` p `to` $r\} \in Cf_{\text{in}}$ because of which p is assigned to role r,

(2) role r is enabled,

(3) user u is assigned to role r,

(4) there is no activation constraint that prevents the user from activating the role.

We note that algorithm `TransformPR` only replaces the constraints of types $\{X,$ *pr*:`assign/deassign` p `to` $r\}$ to produce Cf_{out}, and temporal constraints on original roles are not changed. Hence, conditions (2), (3) and (4) are still valid in Cf_{out} at time t. The FOR loop in line 2 repeats for every constraint of type $\{X,$ *pr*:`assign/deassign` p `to` $r\}$. Each constraint $\{X,$ *pr*:`assign/deassign` p `to` $r\}$ is replaced by a temporal constraint on role enabling/disabling in line 4. Thus a constraint of type $\{X,$ *pr*:`assign/deassign` p `to` $r\}$ is not in Cf_{out}. We need to show that $Cf_{\text{in}} \approx Cf_{\text{out}}$ for the following two cases:

Case 1: Let $X = (I, P)$, i.e. $\{X,$ *pr*:`assign/deassign` p `to` $r\}$ *in* (1) *is a periodicity constraint*: We note that following replacements take place in T' (initially $T' = T$) according to lines 4, 5, and 7:

(i) the replacement of all temporal role-permission assignment expressions by
  a. temporal constraint on the corresponding new role in line 4, and
  b. default assignments, as is shown in line 5.

(ii) the replacement of all occurrences of temporal role-permission assignment expressions and role-permission assignment status expressions in triggers by constraint and status expressions on the new roles as shown in line 7-12.

Because of (i) and (ii), for all triggers or constraint enabling events that cause an "`assign/deassign` p `to` r" event in Cf_{in}, the algorithm produces triggers and constraints that cause an "`enable/disable` r_i" event in Cf_{out} and vice versa. Hence, if because of $\{(I, P), pr$:`assign/deassign` p `to` $r\}$ in Cf_{in}, permission p is assigned to r at time t, then because of $\{(I, P), pr$:`enable/disable` $r_i\}$ and the default assignment $\{pr$:`assign/deassign` p `to` $r_i\}$ in Cf_{out}, p is assigned to r_i at time t and vice versa. Hence, as conditions (2), (3) and (4) are satisfied at t, a user u who is assigned to role r that is enabled at time t can inherit p through r_i, using *restricted inheritance* I_r in Cf_{out}. This inheritance allows u to acquire exactly those permissions that he can acquire in Cf_{in}. Hence, $(u \underset{t}{\overset{Cf_{out}}{\Rightarrow}} p)$ and therefore, $Cf_{\text{in}} \approx Cf_{\text{out}}$.

Case 2: (X = $([(I, P)| D], D_x)$, i.e. $c = \{X, pr$:`assign/deassign` p `to` $r\}$ *in* condition (1) *is a duration constraint*): The transformation indicated by line 4 also replaces all duration constraints of this form by the same duration constraint on the new role's enabling/disabling times. Thus, enabling/disabling of the assignment constraint in Cf_{in} done by any "`enable/disable` c" expression (independent constraint enabling expressions or in triggers) now enables the duration constraint on the new role and vice versa. Thus, since conditions (2), (3), and (4) are satisfied, user u, who is assigned to role r, which is enabled at time t, can inherit p through r_i, using *restricted inheritance* I_r in Cf_{out}. Thus, $(u \underset{t}{\overset{Cf_{out}}{\Rightarrow}} p)$. Hence, $Cf_{\text{in}} \approx Cf_{\text{out}}$.

Proof of Lemma 6.2 (**Correctness of** `TransformUR`) : We prove this by considering following cases:

Case 1: *There are no per-user-role activation constraints in T*: In this case, lines 20-37 do not apply. We also note that except for the hierarchy relations added to *RH*' with respect to the new roles, everything else is the same as that of algorithm `transformPR` if the assignment of permissions is replaced by an assignment of users. So, by arguments similar to one used to prove Lemma 6.1, we can show that the transformation of temporal constraints on user-assignments done by `transformUR` produces an *a-equivalent* configuration.

Case 2: *There are no temporal constraints on user assignments*: In this case, only lines 20-37 apply. Since S is empty, new roles will be created for all *per-user-role* activations in line 30. Each set of *per-user-role* constraints associated with user role pair (u, r) is replaced by a new role and a corresponding set of *per-role* constraints on it so that all activation constraints associated with a user-role pair applies to the corresponding new role. Since each new role is assigned to only one user, in Cf_{out}, the *per-role* constraint on it has the same effect as the *per-user-role* with the matching constraint value (total active duration, cardinality, etc.). Since a zero duration activation time constraints is added for the old role, no users can activate the role (line 36); however, because of the *strongly restricted IA*-hierarchy, permissions are inherited by the new roles. Thus, as $(u \underset{t}{\overset{Cfin}{\Rightarrow}} p)$, it follows that $(u \underset{t}{\overset{Cfout}{\Rightarrow}} p)$.

Case 3: *Both temporal user assignments and per-user-role activation constraints are present*: This case is similar to case 1, in that a new role is created for each user assignment. In addition, all *per-user-role* activation constraints are transformed into *per-role* constraints for the new role created, as indicated by line 2 and 30 (use of $getSu_i$ allows creation of one new role for a (u, r) pair). As the new roles still have only one user assigned to it, the *per-role* constraints applied to them have the same effect as the original *per-user-role* constraints. Hence, as $(u \underset{t}{\overset{Cfin}{\Rightarrow}} p)$, it follows that $(u \underset{t}{\overset{Cfout}{\Rightarrow}} p)$.

Case 4: *There are no user-role assignment and no per-user-role activation constraints*: In this case the algorithm simply returns Cf_{in} as Cf_{out} as both the FOR loops at lines 2 and 21 are not entered.

Hence, it follows that for a given input Cf_{in}, if Cf_{out} is the output produced by algorithm `TransformUR`, then Cf_{out} contains no temporal user assignments and *per-user-role* activation constraints, and $Cf_{\text{in}} \approx Cf_{\text{out}}$.

Proof of Theorem 6.1 (Minimality of GTRBAC)

Proof for (a) and (b): To prove (a) and (b), we can carry out the following two step process:

Step 1: Let C_{12}= `transformPR`(Cf_1); i.e., configuration Cf_1 is input to algorithm `transformPR`, and Cf_{12} is the new *a-equivalent* configuration returned by it.

Step 2: Let Cf_2= `transformUR`(Cf_{12}); i.e., configuration Cf_{12} is input to algorithm `transformUR`, and Cf_2 is the new *a-equivalent* configuration returned by it.

Since $Cf_1 \approx Cf_{12}$ by Lemma 6.1, and $Cf_{12} \approx Cf_2$ by Lemma 6.2, it implies that $Cf_1 \approx Cf_2$. As `transformPR` removes all temporal role-permission assignments, Cf_{12} does not have any temporal constraints on role-permission assignments.

Similarly, since `transformUR` removes all temporal user-role assignments and *per-user-role* activation constraints, Cf_{12} does not have any temporal constraints on role-permission assignments and *per-user-role* activation constraints. Hence, $MCS\ (T_2) \subseteq \{C_d, C_{Rp}, C_{Rd}, C^a{}_r, C_{tr}, C_c\}$

Proof for (c): From (b), we have $MCS(T_2) = \{C_d, C_{Rp}, C_{Rd}, C^a{}_r, C_{tr}, C_c\}$. We need to prove that $MCS(T_2)$ is minimal. We show that a constraint type from $MCS(T_2)$ can not be replaced by another constraint type of $MCS(T_2)$ to produce an *a-equivalent* configuration. We show this case-wise.

Case 1: (*Periodicity*(C_{Rp}) *vs. Duration constraints*(C_{Rd}) *on role*): Periodicity constraint specifies each time instant at which a role is enabled/disabled, whereas, duration constraint does not specify the starting/ending time at which a role is enabled/disabled. Furthermore, an event associated with a duration constraint needs to be triggered or caused by a runtime request. A *periodicity* constraint can be represented by a *duration* constraint if there is a way to enable it (the duration constraint) at a specific time instant that corresponds to the start time of the *periodic* expression. But GTRBAC does not support such specific constraint enabling unless we use a trigger in which a clock timer is allowed to trigger an "`enable` *c*" event that enables the duration constraint, which then becomes equivalent to the original periodicity constraint. However, even if we allow that, the duration constraint that is generated to enforce the periodicity constraint will allow any other trigger or run-time event to enable the role, which is not what the periodicity constraint is intended to do.

Similarly, a duration constraint cannot be specified using a periodicity constraint as it does not have deterministic start times.

Case 2: (*Duration constraint vs. Trigger*): Assume we have the following set of triggers:

$$B \rightarrow \texttt{enable}\ r \qquad (1)$$

$$\texttt{enable}\ r \rightarrow\ \texttt{disable}\ r\ \texttt{after}\ \Delta t. \qquad (2)$$

When trigger (1) fires the non-blocked event `enable` *r*, trigger (2) will allow role *r* to be enabled for a duration Δt. In effect, this is similar to the duration constraint ($D = \Delta t$, `enable` *r*). However, if we also have a periodicity constraint (I, P, `enable` *r*) in Γ of Cf_{in}, then whenever, for an instant $t \in Sol(I, P)$, the non-blocked event `enable` *r* is caused, trigger (2) will enable the duration constraint. This is semantically different from a duration constraint ($D = \Delta t$, `enable` *r*), in which only a trigger or a run-time event can cause the duration restriction for event "`enable` *r*" as specified by ($D = \Delta t$, `enable` *r*). Thus, representing the duration constraint by triggers is not possible in the GTRBAC framework.

Case 3: (*Activation vs. Non-activation constraint*): Replacement of one by the other is not possible because they refer to the different states of a role. In addition, for an enabling/disabling of a role, no user needs to be assigned to the role. An activation constraint needs to be enforced only when a user is actually using the associated role.

Hence, $MCS\ (T_2) \subseteq \{C_d, C_{Rp}, C_{Rd}, C^a_r, C_{tr}, C_c\}$ is minimal.

Proof of Lemma 4.2.1 (MDS of two periodic expressions)

a. Here, we have $PE_i \subseteq PE_j$. Hence for all $t \in Sol(PE_i)$, it is also true that $t \in Sol(PE_j)$. But since $PE_i \neq PE_j$ (non-equivalent), there exists some $t \in Sol(PE_j)$ such that $t \notin Sol(PE_i)$. Therefore, there are two groups of time instants of which one group belongs to both PE_i and PE_j, and the other group belongs to only PE_j. This implies that at least two groups of periodic expressions are needed to represent the time instants of both the periodic expressions. This is because if there is a single group for both PE_i and PE_j, then we need $PE_x = PE_i \cup PE_j$ in order to satisfy the first condition of an MDS. But then, if we consider t_1 and t_2 such that $t_1, t_2 \in Sol(PE_i)$, $t_1 \in Sol(PE_j)$ and $t_2 \notin Sol(PE_j)$, then the second condition required for an MDS is not satisfied.

   As the first group contains time instants that belong to both PE_i and PE_j, we can write the first expression to denote this group as $PE_x = PE_i \cap PE_j$, but $PE_i = PE_i \cap PE_j$; hence, $PE_x = PE_i$ as all time instants that are in $Sol(PE_i)$ are also in $Sol(PE_j)$. The second group of time instants belongs to only PE_j; hence, we can denote the second group as $PE_y = PE_j - PE_i = PE_j - PE_x$. We can see that PE_y do not contain time instants in PE_x, hence, PE_x and PE_y are disjoint.

   From the construction of PE_x and PE_y, we can see that $PE_x \cup PE_y = PE_i \cup PE_j$, which is the first condition for an MDS (Definition 6.2.4 (a)). Furthermore, since $PE_x = PE_i$, only those time instants in PE_i belong to PE_x; any time instant t not in PE_i also is not in PE_x. Similarly, since PE_x is contained in PE_j, only a proper subset of time instants in PE_j is in PE_x, and no time instant that is not in PE_j is in PE_x. Similarly, by construction, only a proper subset of time instants in PE_j is in PE_y, and no time instant that is not in PE_j is in PE_x.. Thus, PE_x and PE_y satisfy the condition (b) of Definition 6.2.4, too. Hence, $MDS_{PE} = \{PE_x, PE_y\}$.

b. Here, we have $PE_i \otimes PE_j$. Hence, as the definition of $PE_i \otimes PE_j$ implies, there are three groups of time instants. The first group belongs to both PE_i and PE_j. The second group belongs only to PE_i, whereas the last group belongs only to PE_j. As there exist some common time instants in the two periodic expressions, based on the argument presented in (a) above, its MDS must contain more that one periodic expression.

Assume that we can create an MDS that contains two disjoint periodic expressions. Since there is a group of periodic instants that belong to both PE_i and PE_j, they must be represented by a single periodic expression, otherwise we cannot get a disjoint pair as required for an MDS. So assume that $PE_x = PE_i \cap PE_j$. Now, we have two remaining groups of time instants, one that belongs only to PE_i and the other that belongs only to PE_j. If we combine the two groups to get PE_y, then $\{PE_x, PE_y\}$ can not be an MDS, because it will not satisfy the second condition (just take time instants t_1, t_2 such that t_1 belong to PE_i, and t_2 belongs to PE_j but not to PE_i, then t_1, t_2 do not satisfy the second condition).

Thus, the problem is that one group of time instants belongs to only PE_i and the other belongs to only PE_i" not PE_i. Now, if we construct $PE_y = PE_i - PE_x$ and $PE_z = PE_j - PE_x$, we get the disjoint set of periodic expressions $\{PE_x, PE_y, PE_z\}$. As in (a), it is easy to see that $\{PE_x, PE_y, PE_z\}$ satisfies the two conditions of an MDS. Hence, it follows that $\{PE_x, PE_y, PE_z\}$ is an MDS of $\{PE_i, PE_j\}$.

Proof of Lemma 6.4 (MDS of n periodic expressions)

We show this by induction on the number of periodic expressions n. Note that ${}^{i}MDS_{PE}$ represents the MDS of the first i periodic expressions of PE.

Basis: $n = 2$: That is, $PE = \{PE_1 , PE_2\}$. Then by Lemma 6.1, we have the following:

- if $PE_1 \subseteq PE_2$ then $MDS_{PE} = \{PE_x, PE_y\}$, and
- if $PE_1 \otimes PE_2$ then $MDS_{PE} = \{PE_x, PE_y, PE_z\}$.

Hypothesis: Assume that it is true for n-1; i.e. there exists ${}^{n-1}MDS_{PE} = \{PE'_1, PE'_2, \ldots, PE'_{m1}\}$ for $PE = \{PE_1, PE_2, \ldots, PE_{n-1}\}$.

We need to show that $MDS_{PE} = \{PE''_1, PE''_2, \ldots, PE''_{m2}\}$ for $PE = \{PE_1, PE_2, \ldots, PE_n\}$.

We start by writing MDS_{PE} $(\{PE_1, PE_2, \ldots, PE_n\}) = MDS_{PE}$ $(\{{}^{n-1}MDS_{PE}, PE_n\}) = MDS_{PE}$ $(\{PE'_1, PE'_2, \ldots, PE'_{m1}, PE_n\})$ (This is true because $PE'_1 \cup PE'_2 \cup \ldots \cup PE'_{m1} = PE_1 \cup PE_2 \cup \ldots \cup PE_{n-1}$). Now, we look at pair-wise relations between PE_n and PE'_i, for $1 \leq i \leq m1$. First, we note that it is possible that PE_n is equivalent to some PE'_i. A simple example is when $PE_n = PE_i \cap PE_k$ and PE'_i represents $PE_i \cap PE_k$ in ${}^{n-1}MDS_{PE}$. However, as PE is not a disjoint set, PE_n cannot be disjoint from all PE'_i , $1 \leq i \leq m1$. We look at each of the possible relations that PE_n may have with each PE_is.

Case 1: $PE_n = PE'_i$ for some i, such that $1 \leq i \leq m1$: then, MDS_{PE} $(\{PE_1, PE_2, \ldots, PE_n\}) = {}^{n-1}MDS_{PE}$ and we are done.

Case 2: $PE'_i \subseteq PE_n$ for some i, such that $1 \leq i \leq m1$: then, by Lemma 6.3(a), $MDS_{\{PE'i,\ PEn\}} = \{PE''_{xi}, PE''_{yi}\}$, where $PE''_{xi} = PE'_i$, and $PE''_{yi} = PE_n - PE''_{xi}$.

Case 3: $PE'_i \otimes PE_n$ for some i, such that $1 \leq i \leq m1$: then by Lemma 6.3(b), $MDS_{\{PE'i,\ PEn\}} = \{PE''_{xi}, PE''_{yi}, PE''_{zi}\}$, where $PE''_{xi} = PE'_i \cap PE_n$, and $PE''_{yi} = PE_n - PE''_{xi}$, *and* $PE''_{zi} = PE'_i - PE''_{xi}$.

We can see that PE_n may be related to each of the PE'_is, $1 \leq i \leq m1$ by either case 2 or case 3 (As shown above, we need not worry about case 1; the case of $PE_n \subseteq PE'_i$ can be handled easily by reversing the periodic expressions of MDS in case 2).

Now, consider that $PE'_i \subseteq PE_n$ and $PE'_j \subseteq PE_n$ for $i \neq j$. (i.e., case 2 applies to both i and j). Thus, we have $MDS_{\{PE'i,\ PEn\}} = \{PE''_{xi}, PE''_{yi}\}$ and $MDS_{\{PE'j,\ PEn\}} = \{PE''_{xj}, PE''_{yj}\}$. We see that PE''_{xi} and PE''_{xj} are disjoint as $PE''_{xi} = PE'_i$ and $PE''_{xj} = PE'_j$, and PE'_i and PE'_j belong to ${}^{n-1}MDS_{PE}$. However, we do not know how PE''_{yi} and PE''_{yj} are related; but we do know that each of them is a proper subset of PE_n.

Now consider that $PE'_i \otimes PE_n$ and $PE'_j \otimes PE_n$ for $i \neq j$. (i.e., case 3 applies to both i and j). As shown in case 3, we get: $MDS_{\{PE'i,\ PEn\}} = \{PE''_{xi}, PE''_{yi}, PE''_{zi}\}$and $MDS_{\{PE'j,\ PEn\}} = \{PE''_{xj}, PE''_{yj}, PE''_{zj}\}$. Now we know that, PE''_{xi} is a subset of PE'_i and PE''_{xj} is a subset of PE'_j. Hence, PE''_{xi} and PE''_{xj} are disjoint (as PE'_i and PE'_j are disjoint). Similarly, PE''_{zi} is a subset of PE'_i and PE''_{zj} is a subset of PE'_j and hence, PE''_{zi} and PE''_{zj} are disjoint. Again, we are left with PE''_{yi} and PE''_{yj} but we do not know how they are related. However, again, we know that each of them is a subset of PE_n.

And lastly consider that $PE'_i \subseteq PE_n$ and $PE'_j \otimes PE_n$ for $i \neq j$ (i.e., case 2 applies to the first and case 3 applies to the second; we ignore the situation in which case 3 applies to the first and case 2 applies to second, as it is a simple case of exchanging the index values). Thus, we get $MDS_{\{PE'i,\ PEn\}} = \{PE''_{xi}, PE''_{yi}\}$ and $MDS_{\{PE'j,\ PEn\}} = \{PE''_{xj}, PE''_{yj}, PE''_{zj}\}$. Similar to the reasons given above, PE''_{xi} and PE''_{xj} , and PE''_{xi} and PE''_{zj} are disjoint. Again, we are left with PE''_{yi} and PE''_{yj}, and we do not know how they are related. However, here too, we do know that they are each a subset of PE_n.

Hence, $\{MDS_{\{PE'1,\ PEn\}}, MDS_{\{PE'2,\ PEn\}}\ldots, MDS_{\{PE'm1,\ PEn\}}\} =$
$\{PE''_1, PE''_2, \ldots, PE''_{m2}, PE''_{y1}, PE''_{y2}, PE''_{ym1}\}$, where
$\{PE''_1, PE''_2, \ldots, PE''_{m2}\} =$
$\{PE''_{xi}, PE''_{xj} |$ case 2 applies both to $MDS_{\{PE'\ i,\ PEn\}}$ and $MDS_{\{PE',j,\ PEn\}}$ and $i \neq j\}$
$\cup\ \{PE''_{xi}, PE''_{xj}, PE''_{zj} |$ case 2 applies to $MDS_{\{PE'\ i,\ PEn\}}$, case 3 to $MDS_{\{PE',j,\ PEn\}}$ and $i \neq j\}$.

This implies that $(PE''_i \lozenge PE''_j)$, for all i, j pairs such that $i \neq j$, $1 \leq i, j \leq m2$. However, we cannot guarantee that $(PE''_i \lozenge PE''_{yj})$ for all i, j pairs such that i, j, $1 \leq i \leq m2$ and $1 \leq j \leq m1$. This is because each PE''_{yj} is a proper subset of PE_n and there are some PE''_i such that $(PE''_i \otimes PE_n)$. However, since the construction of each PE''_i involves breaking down time instants contained in PE_n, we can construct a periodic expression for the group of time instants in PE_n that were not contained or overlapped with any other PE'_i.

Now let $PE''_{m2+1} = PE_n - (PE''_1 \cup PE'' \cup \ldots \cup PE''_{m2})$. Then if PE''_{m2+1} is not empty, then $MDS_{PE}(\{PE_1, PE_2, \ldots, PE_n\}) = \{PE''_1, PE''_2, \ldots, PE''_{m2}, PE''_{m2+1}\}$, otherwise $MDS_{PE}(\{PE_1, PE_2, \ldots, PE_n\}) = \{PE''_1, PE''_2, \ldots, PE''_{m2}\}$.

We need to show that MDS_{PE} constructed in this way is minimal. Assume that it is not minimal. Then there is at least one periodic expression PE''_i, $1 \leq i \leq m2$ such that all time instants in PE''_i are contained in one or more of PE''_j for $i \neq j$ and $1 \leq i, j \leq m2+1$. But it can only be possible if the periodic expressions in ${}^{n-1}MDS_{PE}$ are not disjoint, as the construction above does not introduce such a non-disjoint set. Hence, it contradicts with our assumption. Therefore, the MDS_{PE} constructed above is the MDS of PE.

Proof of theorem 6.2 (MDS using `computeMDS`)

(a) We prove this by taking all the possible cases:

Case 1 - *All the n periodic expressions are equivalent*: In this case anyone of the periodic expressiosn can constitute the MDS, as each periodic expression satisfies the conditions of an MDS.

Case 2 - *The n periodic expressions are pair-wise disjoint*: In this case we can simply consider $MDS = PE$ (and thus $MS_{PEj} = \{PE_j\}$). This satisfies the conditions of a MDS.

Case 3: *The set of n periodic expressions are non-equivalent and non-disjoint*: In this case, according to Lemma 6.4, there exists an MDS.

Therefore, there exists an MDS for an arbitrary set of periodic expressions.

(b) Again, we prove this by taking all the possible cases used above:

Case 1 - *All the n periodic expressions are equivalent*: In this case, for each $n>2$, `computeMDS` recursively computes MDS of smaller size at line 6. When the recursive call reaches $n = 2$, the algorithm calls `pairMDS` to compute the MDS of $\{PE_1, PE_2\}$. As the periodic expressions are equivalent, `pairMDS` returns $\{PE_1\}$ from line 1. This is returned by `computeMDS` in line 4 for $n = 2$. This is also the value of MDS computed by `computeMDS` at line 6 for $n = 3$. So for $n = 3$, computeMDS will compute the MDS of $\{PE_1, PE_3\}$ at line 9 by using the algorithm pairMDS. But since PE_1 and PE_3 are equivalent, again $\{PE_1\}$ is returned. And thus, from line 11, $\{PE_1\}$ will be again returned. We can see, for all $n > 2$, the MDS is the same periodic expression that was returned by the invocation of the algorithm for $n = 2$. Hence, the algorithm correctly returns the MDS for a set of periodic expressions that are equivalent.

Case 2*: The n periodic expressions are pair-wise disjoint*: Since all are pair-wise disjoint, for each pair, pairMDS returns the original pair of periodic expressions. Now if ${}^{n-1}MDS_{PE} = \{PE_1, PE_2, \ldots, PE_{n-1}\}$, then after the FOR loop in line 8, S will be $\{PE_1, PE_2, \ldots, PE_{n-1}\}$ (i.e., $m2 = n$-1 in the algorithm). Hence, $PE''_{n= m2+1} = PE_n$ at line 20. Therefore, $MDS_{PE} = \{PE_1, PE_2, \ldots, PE_n\}$. Hence, it follows that the algorithm correctly returns the MDS for a set of periodic expressions that are equivalent.

Case 3: *The set of n periodic expressions are non-equivalent and non-disjoint*: We can see that lines 5 to 25 implement the inductive method used to prove Lemma 6.4. When $n > 2$, the *MDSs* of lower values are recursively computed. The FOR loop computes the pair-wise *MDS* of the new periodic expression PE_n with each of the periodic expressions PE'_j computed for the earlier value of n. Line 11 returns the earlier *MDS* if PE_n is equivalent to any one of PE'_j. In lines 14 and 17, those periodic expressions returned by pairMDS are collected in S, which constitutes time instants that belong to the periodic expressions of ${}^{n-1}MDS_{PE}$, some of which may also belong to PE_n (when PE_n is contained in or overlaps with some PE'_j). In line 20, a periodic expression is created for any time instants that do not fall in the periodic expressions of ${}^{n-1}MDS_{PE}$ but only in PE_n. The IF-ELSE statement ensures that this periodic expression is not empty. Hence the algorithm correctly computes the *MDS* of *PE*.

Proof of Corollary 6.2.1 (Bounds for size of MDS): We prove this by induction on n.

Basis: Let $n = 1$, then trivially $1 \leq s_1 \leq (2^1 - 1)$, as implied by the first IF statement of line 3 of algorithm computeMDS.

For $n = 2$, the second IF statement of algorithm computeMDS is executed and the returned set is the set returned by algorithm pairMDS for $\{PE_1, PE_2\}$. But algorithm pairMDS returns a set whose cardinality is 1, 2, or 3. Hence, $1 \leq s_2 \leq 3 = 2^2 - 1$.

Hypothesis: We assume that it is true for n-1. That is, $1 \leq s_{n-1} = |{}^{n-1}MDS_{PE}| \leq 2^{n-1} - 1$. We need to show that $1 \leq s_n = |MDS_{PE}| \leq 2^n - 1$.

We observe that a pair-wise MDS is computed for each pair (PE'_j, PE_n), $1 \leq j \leq s_{n\text{-}1} = m1$, where $PE'_j \in {}^{n\text{-}1}MDS_{PE}$. For each such pair ($PE'_j$, PE_n), algorithm pairMDS returns at most three disjoint periodic expressions $\{PE_x , PE_y , PE_z\}$. In such a case $\text{MS}_{PE'j} = \{PE_x , PE_z\}$. Thus, we see that each of the periodic expression of ${}^{n\text{-}1}MDS_{PE}$ is split into at the most two disjoint sets. Furthermore, a new set is created for the remaining time instants of PE_n. Hence, we get the following expression,

$$s_n \leq 2\, s_{n\text{-}1} + 1 = 2(2^{n\text{-}1} - 1) + 1 = 2^n - 1$$

Furthermore, when all periodic expressions are equivalent we get $s_n = 1$. Therefore, $1 \leq s_n \leq (2^n - 1)$.

Proof of Corollary 6.2.2 (Bounds for size of MS)

Basis: Let $n = 1$. Then, trivially, $p_1 = 1$ and $n \leq p_1 \leq n2^{n\text{-}1}$. Let $n = 2$. Then, when the periodic expressions are equivalent, we get $|MDS_{PE}| = 1$ and $|MS_{PEj}| = 1$ for both $j = 1$ and 2; hence, $p_2 = 2$. However, if $MDS_{PE} = \{PE_x , PE_y , PE_y\}$, then $|MDS_{PE}| = 3$ and $|MS_{PE}| = 2$ for both $j = 1$ and 2, and hence $p_2 = 4$. Thus, $2 \leq p_2 \leq 4 = 2.2^{2\text{-}1}$.

Hypothesis: Assume that for n-1, it is true, i.e., $(n\text{ -}1) \leq p_{n\text{-}1} \leq (n\text{-}1)2^{n\text{-}2}$. We need to show that $n \leq p_n \leq n2^{n\text{-}1}$.

Induction step: If all n periodic expressions are equivalent, then $|MDS_{PE}| = 1$ and $|MS_{PEj}| = 1$ for each $1 \leq j \leq s_{n\text{-}1} = m1$. Thus, $p_n = n$. Since this creates the minimum number of expressions in $|MDS_{PE}|$, we have $n \leq p_n$.

When we add the n^{th} periodic expression PE_n, each of PE'_j of ${}^{n\text{-}1}$ MDS_{PE} is split into two periodic expressions at the most. Thus, the maximum increase in p_n occurs when PE_n overlaps with each PE'_j for $1 \leq j \leq s_{n\text{-}1}$. Thus, each of the MS_{PEj} will be split into two. Furthermore, $|MS_{PEn}| = s_{n\text{-}1} + 1$, as in the worst case, PE_n overlaps with each of the periodic expressions PE'_j of ${}^{n\text{-}1}MDS_{PE}$, and there is a periodic expression that represents those instants of PE_n that are not contained in ${}^{n\text{-}1}MDS_{PE}$. Hence, we have,

$$\begin{aligned}
p_n &\leq 2p_{n\text{-}1} + (s_{n\text{-}1} + 1) \\
&= 2(2p_{n\text{-}2} + (s_{n\text{-}2} + 1)) + (s_{n\text{-}1} + 1) = 2^2 p_{n\text{-}2} + 2^1(s_{n\text{-}2} + 1) + (s_{n\text{-}1} + 1) \\
&= \ldots \\
&= 2^{n\text{-}1}p_1 + 2^{n\text{-}2}(s_1 + 1) + \ldots + 2^1(s_{n\text{-}2} + 1) + 2^0 (s_{n\text{-}1} + 1) \\
&= 2^{n\text{-}1} + 2^{n\text{-}2}(s_1 + 1) + \ldots + 2^1(s_{n\text{-}2} + 1) + 2^0(s_{n\text{-}1} + 1) \\
&\leq 2^{n\text{-}1} + 2^{n\text{-}2}(2^1 - 1 + 1) + \ldots + 2^1(2^{n\text{-}2} - 1 + 1) + 2^0(2^{n\text{-}1} - 1 + 1) \\
&= 2^{n\text{-}1} + (n\text{-}1)2^{n\text{-}1}
\end{aligned}$$

$= \quad n2^{n\text{-}1}$

Therefore, $n \leq p_n \leq n2^{n\text{-}1}$.

Proof of Theorem 6.3 (Correctness of `TransformMDS`):

We have $PE = \{PE_1, PE_2\ldots, PE_n\}$ and $MDS = \{PE'_1, PE'_2\ldots, PE'_m\}$. Furthermore, in line 6 all the required MS_{PEi} are computed. Line 9 creates a unique role for each of the expressions PE'_i which is made senior to the original role using A_n. Line 10 inside the FOR loop of line 9 ensures that each user u_k which corresponds to PE_k in the *user-role* assignment of C_{in} is assigned to this new role associated with $PE'_i \in MS_{PEk}$. This ensures that the following hold:

a. For each $t \in PE_i$, we have $t \in MS_{PEi}$ (by Definition 6.1.5 of MS)
b. For each $u_i \in U$, we see that u_i is (*default*) assigned to each new role that corresponds to expressions in MS_{PEi} by lines 11 and **FOR** loops at lines 8 and 10, and (b)
c. For each PE'_i, (PE'_i, `enable` r_i) is added to T' by line 12

Thus from (a), it follows that a u_i can activate the original role r through one of the new roles that corresponds to expressions in MS_{PEi} at $t \in MS_{PEi}$. Furthermore, the periodic expressions $\{PE_1, PE_2\ldots, PE_n\}$ and $\{PE'_1, PE'_2\ldots, PE'_m\}$ exactly cover the same time instants. The main loop in line 2 ensures that such transformation is done for each r. Hence, it follows from (a), (b), and (c) that $C_{\text{in}} \approx C_{\text{out}}$. We note that the repetition of line 13 removes all the user-role assignments. Hence, C_{out} is free of user-role assignments.

Proof of Theorem 6.4 (**Complexity expressions $GTRBAC_0^1$ and $GTRBAC_0^2$**)

(1) $GTRBAC_0^1$ representation is $n.S + n.T_R + n.R + H$: Here, $GTRBAC_0^1$ refers to the use of algorithm `TransformUR`. For each *user-role* assignment, algorithm TransformUR creates a new role (therefore total of n roles), adds a constraint for each new role (total is $n.T_R$), and adds a default assignment (hence, total is $n.S$). Furthermore, as new roles are made senior to the original role, we introduce n hierarchical relations. Hence, for n user-role assignments the complexity incurred is: $n.S + n.\ R + n.R + n.H$.

(2) $GTRBAC_0^2$ representation is $p_n.S + s_n.T_R + s_n.R + s_n.H$: It follows immediately from corollaries 6.2.1 and 6.2.2 and Theorem 4.3.

Proof of Corollary 6.4.1 (Complexity cases for $GTRBAC_0^2$ representations)

Proof of Part 1: From Theorem 4.4, the complexities for $GTRBAC_0^1$ and $GTRBAC_0^2$ are $(n.S + n.T_R + n.R + n.H)$ and $(p_n.S + s_n.T_R + s_n.R + s_n.H)$ respectively.

When $PE_i \neq PE_j$, *for all i, j pairs such that* $1 \leq i, j \leq n$, we obtain $p_n = n$ *and* $s_n = n$ as per corollaries 6.2.1 and 6.2.2. Furthermore, hierarchy overhead is also incurred. Hence, the complexity for $GTRBAC_0^2$ representation, using `TransformMDS` (Theorem 6.3) and by Theorem 6.4, is $(p_n.S + s_n.T_R + s_n.R + s_n.H = n.S + n.T_R + n.R + n.H)$ which is also the complexity of the $GTRBAC_0^1$ representations.

Proof of Part 2: When $PE_i = PE_j$, *for all i, j pairs such that* $1 \leq i, j \leq n$, we obtain $p_n = n$ *and* $s_n = 1$ as per corollaries 6.2.1 and 6.2.2. Furthermore, hierarchy overhead is also incurred. Hence, the complexity for $GTRBAC_0^2$ representation is:

$= p_n.S + s_n.T_R + s_n.R + H = n.S + T_R + R + H.$

Proof of Part 3: As shown by corollaries 6.2.1 and 6.2.2, the worst cases for p_n *and* s_n are $n2^n$ and 2^n respectively. Thus, using it in the complexity expression given by Theorem 6.4, we get

$= p_n.S + s_n.T_R + s_n.R + H = n2^n.S + 2^n.T_R + 2^n.R + 2^n.H$

Proof of Theorem 6.5 (Complexity expression $GTRBAC_0$ and $GTRBAC_{1,A}$)

Proof of 1 ($GTRBAC_{1,A}$ representation) : We prove this by cases.

Case 1: $d_i \neq d_j$, for all i, j pairs such that $1 \leq i, j \leq n$; i.e., durations are pair-wise disjoint.

Since durations are distinct from each other, we need a *per-user-role* activation constraint for each. Hence, we have the term $n.T_{UR}$ to represent n such constraints. Other than that we do not require other constraints as they fully express the required access constraints. However, the original role is still there, and if there is a *per-role* constraint on the original role, it will still be there. Thus, the complexity of representation without including the original role and *per-role* constraints on it simply is: $n.A_{UR}$.

Now we show that the expression provided by the theorem gives this same expression. Since the durations are all pair-wise disjoint, we get $D_m = D$ and therefore $n_x = n$, and $n_y = 0$. Similarly, we see that $b = 0$ and $c = 0$. Therefore the complexity is:

$= (n_x - n_y).A_{UR} + n_y.A_R + c.(b.n_y + 1)(R + H) = n_x.A_{UR} = n.A_{UR}$.

Thus, the expression holds for this case.

Case 2: $d_i = d_j$, for all i, j pairs such that $1 \leq i, j \leq n$, i.e., durations are all equal.

When all the durations are the same, then all *per-user-role* constraints can be expressed as a *per-role* constraint on a role such that the *default* value of the *per-role* constraint is that duration. Thus, we create a new role that is senior to the *original* role and specify the new *per-role* constraint. This obviously incurs some hierarchy overhead. The complexity is, thus, $T_R + R + H$.

Now, lets look at the constraint expression given by the theorem. Since all the durations are equal, there is only one distinct duration element. Hence, $n_x = 1$. Similarly, $n_y = 1$, as the same element occurs more than once in D. Values for $b = 0$ and $c = 1$. Therefore, we get:

$= (n_x - n_y).A_{UR} + n_y.A_R + c.(b.n_y + 1)(R + H) = A_R + R + H$

Thus, the expression holds for this case also.

Case 3: There is at least one i, j pair, $1 \leq i, j \leq n$ such that $d_i = d_j$, and there is at least one i, j pair, $1 \leq i, j \leq n$ such that $d_i \neq d_j$.

In this case, $D_m \subset D$. Let $D_m = \{ d'_1, d'_2,, \ldots d'_{nx}\}$, i.e., $n_x = |D_m| < n$. We know that $D' = \{ d'_{\pi\_1}, d'_{\pi\_2},, \ldots d'_{\pi\_ny}\} \subseteq D_m$, where $n_y = |D'| \geq 0$. Since each of duration d'_{π_i} is common to at least two users, we create a role r_{π_i} corresponding to each d'_{π_i}, and specify a *per-role* constraint with d'_{π_i} as the default value and $(C_m(d'_{\pi\_i}) \times d'_{\pi\_i})$ as the total *per-role* active duration value; i.e., the new *per-role* constraint is $(C_m(d'_{\pi\_i}) \times d'_{\pi\_i}, [d'_{\pi\_i}], \text{active}_{\text{R\_total}}\ r_{\pi\_i})$. Complexity incurred by this is:

$n_y.A_R + n_y.R$ (a)

Since $n_x = |D_m|$, $(n_x - n_y)$ is the number of durations that occurs only once in D and hence we can create a role r_{nx} and assign all the users associated with these durations to it and specify an associated *per-user-role* activation constraints for each user. This will incur cost as follows:

$(n_x - n_y).A_{UR} + R$ (b)

Furthermore, the roles r_{π_i}s and r_{nx} need to be made senior to the original role, thus incurring H. Hence, adding (a) and (b) and H, we get the following complexity:

$(n_x - n_y).A_{UR} + n_y.A_R + (n_y + 1). (R + H)$ (*i*)

Now we show that this is exactly the complexity given by the theorem. According to the theorem, in this case, $b = 1$, as $n > n_x$ holds true because of the strict subset relation $D_m \subset D$. Similarly, since there is at least one i, j pair, $1 \leq i, j \leq n$ such that $d_i \neq d_j$; therefore $(n > n_x > 0)$ holds true; hence $c = 1$. Therefore, the complexity expression, according to the theorem, is:

$$(n_x - n_y).A_{UR} + n_y.A_R + c.(b.n_y + 1)(R + .H) = (n_x - n_y).\mathrm{A}_{\mathrm{UR}} + n_y.A_R + (n_y + 1)(R + H)$$

which is the same as (*i*). Thus,the complexity expression holds good for case 3 too. Since the three cases cover all the possible relations among the duration values, it follows that the complexity expression for the $GTRBAC_{1,\mathrm{A}}$ representation is true.

<u>Proof of 2</u> ($GTRBAC_0$ representation)

As n_x is the number of distinct durations *D*, we simply create n_x roles and add to each role a *per-role* constraint. Such a constraint will use the duration value in *D*. For all durations which occur only once in *D*, they are used as *per-role* duration values in the corresponding new *per-role* constraint; i.e, the new constraint is (X, *d*, $\mathrm{active}_{\mathrm{R\_total}}$, *r*), where *d* occurs only once in *D*. For all *d*'s that occur more than once in *D*, the new *per-role* constraint is (X, $C_m(d) \times d$, *d*, $\mathrm{active}_{\mathrm{R\_total}}$, *r*). The new roles are senior to the original roles, thus incurring *H*. Hence, the complexity becomes: $n_x.A_R + n_x.R + H$.

VITA

James B. D. Joshi was born in Nepal in 1969. He completed his Bachelor of Engineering in Computer Science and Engineering from Motilal Nehru Regional Engineering College, Allahabad, India, in 1993. From 1993 to 1996, he served as a lecturer in the Computer Science and Engineering department in Kathmandu University, Nepal. He joined Purdue University in the fall of 1996 and obtained an M.S. in Computer Science in 1998. His research interests include information system security, distributed database systems, multimedia systems, and networking.