DISSERTATION

ACCESS CONTROL MODELS FOR PERVASIVE COMPUTING ENVIRONMENTS

Submitted by

Manachai Toahchoodee

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2010

COLORADO STATE UNIVERSITY

April 28, 2010

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY MANACHAI TOAHCHOODEE ENTITLED ACCESS CONTROL MODELS FOR PERVASIVE COMPUTING ENVIRONMENTS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate work

Ross M. McConnell

Indrajit Ray

Stephen Hayne

Advisor: Indrakshi Ray

Department Chair: L. Darrell Whitley

ABSTRACT OF DISSERTATION

ACCESS CONTROL MODELS FOR PERVASIVE COMPUTING ENVIRONMENTS

With the growing advancement of pervasive computing technologies, we are moving towards an era where context information will be necessary for access control. Traditional access control models like Mandatory Access Control (MAC), Discretionary Access Control (DAC), and Role-Based Access Control (RBAC) do not work well in this scenario for several reasons. First, unlike traditional applications, pervasive computing applications usually do not have well-defined security perimeter–the entities an application will interact with or the resources that will be accessed may not be known in advance. Second, these applications are also dynamic in nature–the accessing entities may change, resources requiring protection may be created or modified, and an entity's access to resources may change during the course of the application, which make the resources protection during application execution extremely challenging. Third, pervasive computing applications use the knowledge of surrounding physical spaces to provide services; security policies designed for such applications must therefore use contextual information. Thus, new access control models and technologies are needed for pervasive computing applications.

In this dissertation, we propose two types of access control models for pervasive computing environments; one determine the accessibility based on the spatio-temporal constraints, and the other determine the accesibility based on the trustworthiness of the entities. The different features of access control models may interact in subtle ways resulting in conflicts. Consequently, it is important to analyze and understand these models before they are widely deployed. The other contribution of this dissertation is to verify the correctness of the model. The results obtained by analyzing the access control models will enable the users of the model to make

informed decisions. Toward this end, we propose automated verification techniques for our access control models.

Manachai Toahchoodee
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Summer 2010

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my advisor, Professor Indrakshi Ray for her constructive criticism, invaluable guidance, patiently support, and continually encouragement, which guided me through each step of my graduate studies at Colorado State University. Her insightful advice and inspiring vision contributed to my personal life in the USA, my publications, and especially this dissertation. I always consider myself fortunate to be accepted as her advisee and I hope someday I would be able to to mentor, inspire, and provide such valuable guidance to my students.

I would like to thank Professor Indrajit Ray for his continuous help during my PhD study and for teaching me the knowledge in computer security especially in the area of the access control model which constructs the backbone of this dissertation. His invaluable comments after the presentation of my preliminary results immensely help me to prepare my final presentation. I would like to thank Professor Ross M. McConnell for his continuous kind support, his unique way of teaching me about the algorithm, and his contribution in the graph algorithm used for detecting the conflict in the STARBACD model. I would like to thank Professor Stephen Hayne for his enthusiastic help, diverse suggestions, and constructive comments during the presentation of my preliminary results, which crucially help me to improve the final version of this dissertation. I also want to thank Professor Charles Anderson for his useful advice which helps me in preparation for the defense exam. My thanks also to Professor Yashwant Malaiya, Professor Robert France, and all faculty members in Computer Science department, for helping me on subject knowledge and my professional growth.

I would like to thank Sharon Van Gorder, Carol Calliham, Kim Judith, Wayne Trzyna, and all Computer Science department staffs, for granting me a favor all the time. Thanks to all my friends and colleagues here, Geri Georg, Xing Xie, Ramadan Abdunabi, Rinku Dewri,

HyunChul Joh, and Elliott Forney, for all their encouragement and support. My special thanks to all my friends at the Thai Student Association, for all their support and for being my family here.

To my parents Suith Toahchoodee and Yupha Sae-Khow, I thank you for giving me your appreciation during my study here. To all my brothers and sisters, Ekachai, Sumitra, Supattra and Suvimol, I thank you for your inspiration and encouragement. Your unique way of support is what keep me going through all the tough times here. I also would like to thank my in-laws, relatives, and friends for their support.

Last, but not least, I would like to express my deepest love and gratitude to my wife Supparat for her unconditional support. This accomplishment would not be possible without your love, patience, and sacrifice. Thank you for believing in me right from the beginning of this journey. I love you.

DEDICATION

This dissertation is dedicated to my parents,

to my brother Ekachai,

to my sisters Sumitra, Supattra and Suvimol,

and to my wife Supparat.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Mark Weiser [88] has given the quote regarding the definition of pervasive computing as "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.".

1.1 Ubiquitous or Pervasive Computing

What is pervasive computing? Below are some definitions we can get from the Internet.

- "The trend towards an information environment in which users have access to ICTs throughout the environment. This trend is particularly associated with the growth of wireless technologies that allow users to access online information and services remotely and synchronize data between different computers."
  (`http://www.parliament.vic.gov.au/sarc/E-Democracy/Final_Report/Glossary.htm`)

- "Inexpensive microprocessors embedded in everyday objects and environments. Characterized by being numerous, casually accessible, often invisible computing devices, frequently mobile or embedded in the environment and connected to an increasingly ubiquitous network structure."
  (`http://framework.v2.nl/archive/archive/node/text/default.xslt/nodenr-156647`)

- "The use of a computing infrastructure that supports information appliances from which users can access a broad range of network-based services, including Internet-based e-commerce services. Pervasive computing thus provides users with the ability to access and take action on information conveniently."
  (`http://www-03.ibm.com/ibm/history/reference/glossary_p.html`)

- "Ubiquitous computing (ubicomp, or sometimes ubiqcomp) integrates computation into the environment, rather than having computers which are distinct objects. Another term for ubiquitous computing is pervasive computing. Promoters of this idea hope that embedding computation into the environment would enable people to move around and interact with computers more naturally than they currently do." (`http://en.wikipedia.org/wiki/Pervasive_Computing`)

In summary, pervasive computing is a technology that relies on the computing and communication capability. This technology communicates with the user in such a way that the user merely recognizes its existence.

1.1.1 Pervasive Computing Model

The technology necessary to build a pervasive computing environment fall into four broad areas [70]: devices, networking, middleware, and applications. Figure 1.1 [70] illustrates their relationships.

Devices

Pervasive computing environment consists of various device types interacting with each other to serve common purposes (See Figure 1.2 [81] for the example of pervasive devices). These devices include traditional input devices, such as mice or keyboards, and output devices, such as speakers or light-emitting diodes; wireless mobile devices, such as pagers, personal digital assistants, cell phones, palmtops, and so on; and smart devices, such as intelligent appliances, floor tiles with embedded sensors, and biosensors. All cooperating together under the middleware which mediates interactions among them. In theory, pervasive computing should

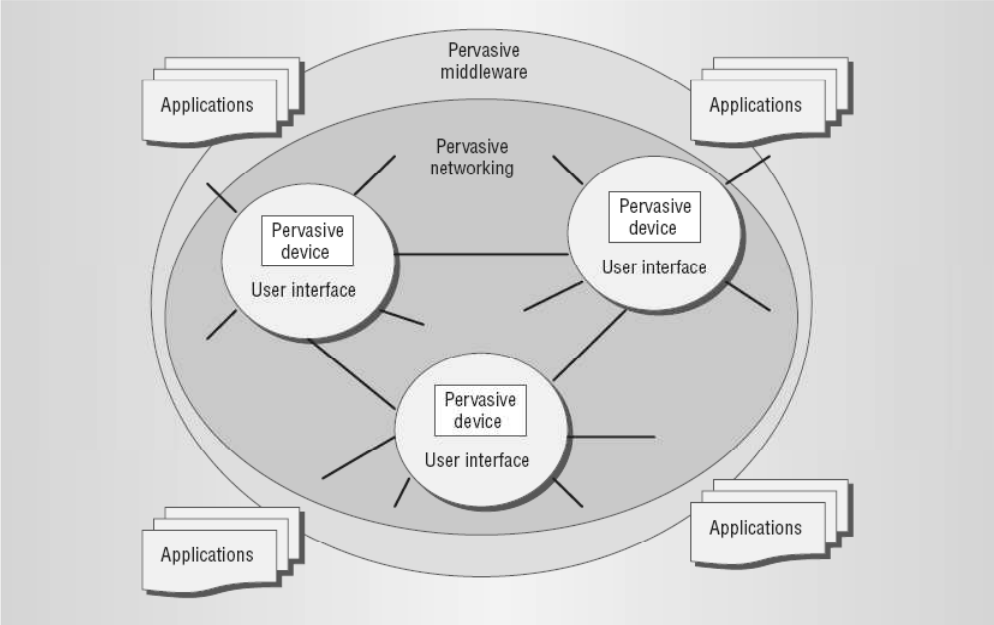


Figure 1.1: Pervasive computing framework. Middleware mediates interactions with the networking kernel on the user's behalf and keeps users immersed in the pervasive computing space.

be applied to all of these intelligent devices.

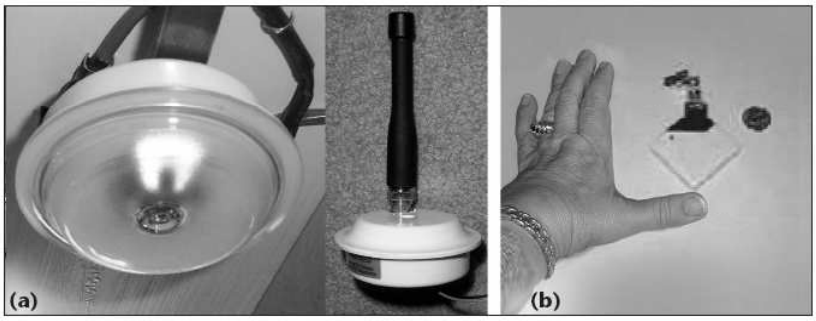


Figure 1.2: Example of pervasive devices. (a) infrared and radio frequency sensors for locator badges reside throughout the Elite Care environment; (b) residents use badges as apartment keys and to locate service or summon help.

Pervasive networking

Since the fundamental of the ubiquitous computing environment is based on the communication between various devices in the network, the rapid growth of the number of pervasive devices causes existing network technologies to be renovated. In addition to extending the

backbone infrastructure to meet the anticipated demand, global networks like the Internet also must modify existing applications to completely integrate these pervasive computing devices into existing social systems.

Pervasive middleware

Like distributed computing and mobile computing, pervasive computing requires a middleware "shell" to interface between the networking kernel and the end-user applications running on pervasive devices. As Figure 1.1 shows, this pervasive middleware will mediate interactions with the networking kernel on the user's behalf and will keep users connected to the pervasive computing space. The middleware will consist mostly of firmware and software bundles executing in either client-server or peer-to-peer mode. User interfaces are another aspect of middleware. Standard web browsers represent the high end of interface sophistication. Nonetheless, the usage of color, graphics, and controls are more than users typically expect on pervasive devices. As a result, mobile computing has already introduced microbrowsers. For example, phone.com's UP.Browser is implemented on several commercially available digital phones.

Pervasive applications

The unique property of the pervasive computing is that, it relies more on the surrounding context than both web-based and mobile computing. The application will interact based on the contextual information it perceives. Consider a heart patient wearing an implanted monitor that communicates wirelessly with computers trained to detect and report anomalies. The monitor should know when to raise the alarm, based on its knowledge about the environment and patient's health record. Such scenario requires much more than simple wireless communication.

1.1.2 Pervasive Computing Environment

Pervasive computing aims to simplify day-to-day life by providing mobile users with the means to carry out personal and business tasks via portable and embedded devices [44]. These

tasks range from the simple–switching on the lights in a conference room, checking e-mail, and organizing meetings–to the more complex–booking airline tickets, buying and selling stock, or managing bank accounts. Pervasive computing environments of the near future will involve the interaction, coordination, and cooperation of numerous, casually accessible, and often invisible computing devices and services. As Figure 1.3 [44] shows, these devices–whether carried on our person or located in our homes, businesses, and classrooms–will connect via wired and wireless links to one another as well as to the global networking infrastructure to provide more relevant information and integrated services.

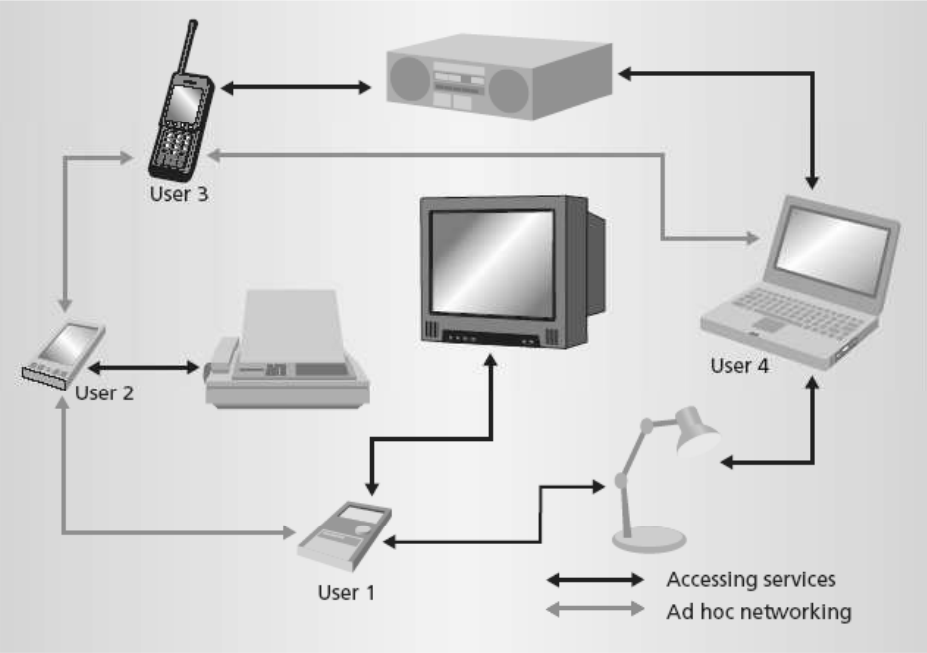


Figure 1.3: Pervasive computing environment

1.2 Problem Description and Motivation

The growth of pervasive computing technology will spawn applications such as, the Aware Home [22] and CMU's Aura [27], that will make life easier for people. Pervasive computing is revolutionary because it provides services and functionalities that use the knowledge of surrounding physical places. Pervasive computing applications typically involve many entities that may span different organizations interacting in complex and subtle ways. Unconstrained interactions result in security and privacy breaches. Application design requires understanding what resources an entity has access to, which entities it should interact with, what information can be released to an entity, how to protect the information used or produced by an entity, which entities can be trusted and to what extent, and how these trust relationships change over time.

Security and privacy are major concerns for such applications. Consider a cardiac patient living by himself in a smart home. Data collected by sensors is sent to a monitoring service which takes appropriate decisions when necessary. Preventing data transmission to the monitoring service or sending false data may be fatal. Sending too many false alarms can cripple emergency services. Disclosing the patients health data to prospective employers may cause financial hardship and disclosing the data to unapproved doctors causes breach of privacy. Comparing a patients report to unauthentic reports of other patients results in incorrect diagnosis. These severe consequences motivate the need to consider security and privacy issues when designing secure pervasive computing applications. Security policies and mechanisms developed for traditional applications are inadequate for pervasive computing applications for the following reasons:

1. Unlike traditional applications, pervasive computing applications have no definite security perimeters –the entities an application will interact with or the resources that will be accessed may not be known in advance.

2. These applications are also dynamic in nature–the accessing entities may change, resources requiring protection may be created or modified, and an entity's access to re-

sources may change during the course of the application. Protecting resources during application execution remains challenging.

3. Pervasive computing applications use the knowledge of surrounding physical spaces to provide services which requires security policies to use contextual information. For instance, access to a resource may be contingent upon the location of the user and time of day. This contextual information can be used to infer the activities of the user and cause a privacy breach. Contextual information must, therefore, be protected by security and privacy policies.

In the model which supports multiple features, such as, hierarchical structures, separation of duties constraints, or delegation of authority, it is possible that the different features of the model might result in inconsistencies and conflicts. Consequently, it is important to analyze and understand these models before it is widely deployed. With respect to this aspect, our second proposition is motivated by the following observations:

1. Nowadays, there are very few verification approaches proposed for the access control verification. Most of them are either non-automated, error-prone, or hard to use.

2. Interaction between various access control model features can lead to conflict which could result in the denial of service, or security breach. The existing researches focus more on modeling the functionality of access control model. To the best of our knowledge, none of the proposed works deal with the verification of the interaction among access control model functionalities.

1.3 Research Goals and Tasks

Motivated by the open issues listed in the previous section, in this Ph.D. dissertation, we propose access control models for pervasive computing applications, which are capable of:

1. Granting or denying an access decision in the pervasive computing systems where the entities an application will interact with or the resources that will be accessed may not be known in advance.

2. Granting or denying an access decision in such dynamic scenario where the accessing entities may change, resources requiring protection may be created or modified, and an entity's access to resources may change on the fly.

3. Using the knowledge of space and time to provide accessibility to resources for the user.

To ensure the correctness of the models, the proposed models must also be analyzed. In this Ph.D. dissertation, we propose a methodology to verify the correctness of access control models. The proposed methodology can:

1. Automatically detect the existence of conflicts between different features in the proposed access control model.

2. Detect conflicts taking into account the dynamic aspects of the model, when the entities and interactions between them are modified on the fly.

We decompose the research into three tasks. All these tasks are cohesive and related to each other, serving the major goals as: (i) to propose the context-aware access control models for pervasive computing environments; and (ii) to propose the verification methodology for the access control models. The set of tasks are described in details in the following Sections.

1.3.1 Task 1. Investigate and identify the types and characteristics of policies needed in pervasive computing environment and develop policy models

A pervasive computing application typically collects information from a wide variety of sources, aggregates it, processes it, and distributes it to different users. The nature of the interactions with different sources are not always well defined. Much of the information that is exchanged is sensitive and must be protected. Sensitive information is protected by different kinds of policies. In the first task, first we need to evaluate the kinds of policies needed and develop suitable policy models for use in pervasive computing applications.

A policy model formalizes the syntax and semantics of supported policies and provides guidelines for their development. Researchers have proposed different kinds of models to for-

malize policies, including the Bell-Lapadula (BLP) model [11] and the Biba model [14]. The Role-Based Access Control (RBAC) model [23, 24, 40, 75, 76] is used by commercial organizations and formalizes the access control policies in a commercial environment. However, traditional models cannot be used for pervasive computing applications because they do not capture the notion of physical context.

Since RBAC model is the de facto standard, flexible, and policy neutral, we decided to base our work on the RBAC model. There are several ways in which RBAC must be extended. We need to explicitly capture the notion of context. We must integrate the contextual information to the existing entities in RBAC and formalize the contextual constraints to support both authorization and delegation policies. Identifying the impact that the context has on entities and their relationships is a major concern in this task.

1.3.2 Task 2. Develop a model verification methodology

It is widely known that different features of RBAC such as, role hierarchy and SoD, interact in subtle ways resulting in inconsistencies and conflicts. Improper resolution of conflicts may cause security breaches. Consequently, it is important to analyze and detect the discrepancy before the model is deployed. In this task, we intend to develop a verification method to verify the correctness of the model.

Manual analysis is tedious and error-prone. Analyzers based on theorem proving are hard to use, require expertise, and need manual intervention. Model checkers are automated but are limited by the size of the system they can verify.

We will focus on the properties and interaction between different features of the access control model. We will analyze these properties and define a methodology to detect conflicts that may occur between the features of the access control model. We will classify all kinds of such conflicts with respect to different context.

Making a thorough analysis and giving a complete list of conflicts is the major challenge of this task. Developing a verification methodology for the model is another big challenge in this part.

1.4 Significance and Contributions

The research conducted in this dissertation is significant. In this research, we address the need of a novel access control model for pervasive computing environments. We then develop access control models to support the security requirement in the context-aware environment. This research is among the earliest works in extending RBAC to support contextual constraints.

Moreover, this research seems to be the first work in analyzing the possible conflicts among the constraints in RBAC model. We show how we can model the access control model, and automatically check for its consistency. Finally, we show how our approach can be adapted to the complicated real-world application which are typically modeled as workflows.

Contributions of this research are summarized below:

1. It proposes access control models that use contextual information to make access decisions.

2. It proposes access control models that are suitable for dynamic applications where access rules may change during the course of the application.

3. It illustrates how to describe the syntax and semantics of these models.

4. It provides techniques for analyzing the interaction of various features of the access control models.

5. It describes approaches for analyzing the interference of access control constraints with application requirements.

1.5 Dissertation Structure

The rest of the dissertation is organized as follows. Chapter 2 describes the related work. Chapter 3 discusses our Spatio-Temporal Role Based Access Control model. Chapter 4 discusses how we can analyze and verify correctness of our Spatio-Temporal Role Based Access Control model by using the automated tool called Alloy. Chapter 5 proposes the second model

called a Spatio-Temporal Aware Role-Based Access Control with Delegation (STARBACD) model. The development of the model is based on graph representation, which is well-formed semantics. Chapter 6 discusses the extension of the Spatio-Temporal Role Based Access Control model and its graph-theoretic representation. Chapter 7 describes how the model can be transformed into the form of Coloured Petri-Nets to enable the automatic verification. Chapter 8 discusses the other approach of developing the access control model for pervasive computing environment based on the trustworthiness between the entities. Chapter 9 demonstrates how such trustworthiness can be used in the delegation operation and how we can ensure the security of the system after the delegation was performed. Chapter 10 concludes the dissertation.

Chapter 2

Related Work

Our work consists of two research areas: access control model and access control model analysis. In this chapter, we provide an overview of the relevant work categorized by the areas of our research.

2.1 Access Control Model

In the past three decades, various types of access control models have been proposed. In this chapter, we review the background and describe different approaches of access control model and access control model analysis.

2.1.1 Access Control Matrix

The access control matrix was defined by Lampson in [47]. Access control matrix [25, 47, 52] is a two-dimensional matrix representing subjects on the rows and objects on the columns. Each entry in the matrix contains the *access attributes*, specifying the access privileges held by subject *S* to object *O*. Table 2.1 shows the example of access control matrix.

Table 2.1: Access Control Matrix

	File1	**File2**	**File3**	**Process1**
Alice	Read, Write	Read	Write	–
Bob	–	–	–	Suspend
Charlie	Read	Read	Read	–

From Table 2.1, subject *Alice* may read or write object *File1*, since 'Read' and 'Write' appear in the corresponding access control matrix entry. Similarly, subject *Bob* may suspend object *Process1*.

In a large system, the access matrix will be enormous in size, and most of its entries are likely to be empty. As a result, the access matrix is very rarely implemented as a matrix. We now discuss two common approaches to implementing the access matrix in practical systems [76].

2.1.1.1 Access Control List

In Access Control List (ACL) implementation, each object is associated with an ACL, indicating for each subject in the system the accesses the subject is authorized to execute on the object. This approach corresponds to storing the matrix by columns.

It is easy to determine which access privileges subjects are currently granted for that object by using the ACLs. In other words, ACLs provide for convenient access review with respect to an object. It is also easy to revoke all access to an object by replacing the existing ACL with an empty one. However, ACL implementation makes it difficult to determine all the accesses that a subject has. To do that, it is necessary to examine the ACL of every object in the system to do access review with respect to a subject. Similarly, if all accesses of a subject need to be revoked, all ACLs must be traversed. In practice, revocation of all accesses of a subject is often done by deleting the user account corresponding to that subject. This is acceptable if a user is leaving an organization. However, if a user is reassigned within the organization it would be more convenient to retain the account and change its privileges to match the changed assignment of the user. ACLs corresponding to the access control list in Table 2.1 is shown in Figure 2.1.

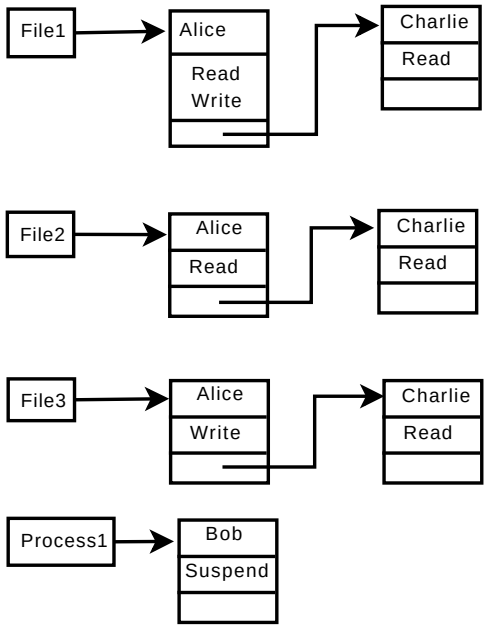


Figure 2.1: Access Control Lists

2.1.1.2 Capability List

The dual approach to the ACL is the *Capability List*. Each subject is associated with a list, called the capability list, indicating for each object in the system, the access privileges the subject is authorized to execute on the object. This approach corresponds to storing the access matrix by rows. The capability lists of Table 2.1 are shown in Figure 2.2. In this approach, it is easy to review all accesses that a subject is authorized to perform, by simply examining the subject's capability list. On the other hand, determination of all subjects who can access a particular object is cumbersome. It requires examination of each and every subject's capability list. Moreover, implementing this approach also causes the difficulties in adding or removing protected objects to the system. If such case should happen, the access privileges have to be updated to all capability lists in the system.

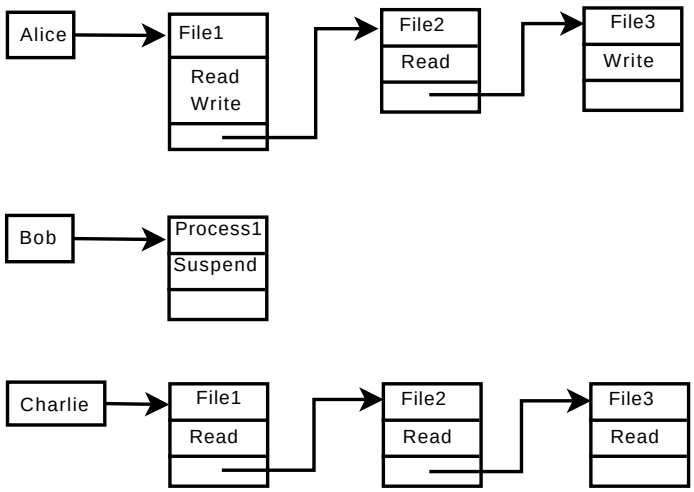


Figure 2.2: Capability Lists

2.1.1.3 HRU System Protection Model

Harrison et al [26] propose a HRU model–a formal protection system model based on the access matrix model. To manage the authorization policy, the protection system consists of (1) a finite set of generic rights R, and (2) a finite set C of commands of the form:

command $\alpha(X_1, X_2, \ldots, X_k)$
  if r_1 in (X_{s_1}, X_{o_1}) and
    r_2 in (X_{s_2}, X_{o_2}) and
    $\ldots$
    r_m in (X_{s_m}, X_{o_m})
  then
    op_1
    op_2
    $\ldots$
    op_n
end

Each command body consists of primitive operation op_i and the condition as shown above. The body of the command is allowed to execute only if the rights specified in the condition parts exist in the access control matrix. The authors discuss about the safety property of the HRU command which could affect the safety of the system. From [26], the formulation of safety system can be summarized as follow: The system is "unsafe" if there exists a command which causes the leakage of right from one place to another place in the access matrix. It is later shown in the literature that the safety problem of the system is, in general, *undecidable*. However, the work shows that the problem is *decidable* in the mono-operational case, where the body part of the command consists of only one primitive operation.

2.1.2 Discretionary Access Control Model

Discretionary Access Control (DAC) Model [26, 52, 76] restricts the accessibility to objects based on the identity of subjects and/or groups to which they belong. Each request of a user to access an object is checked against the specified authorizations in the access control matrix. If there exists an authorization stating that the user can access the object in the specific mode, the access is granted, otherwise it is denied. As the name implies, the controls are discretionary in the sense that a user or process given discretionary access to information is capable of passing that information along to another subject. To provide this discretionary control, DAC policies usually include a concept of object ownership, where the object owner has control permission to grant access permission to the object for other subjects.

DAC policies are very flexible and widely used in the industry. However, they do not provide a high security assurance for two reasons [23, 76]: First, the granting access is transitive. For example, a user who is able to read data can pass his read privilege to other users not authorized to read it unbeknownst to the object owner. Second, DAC policies are vulnerable to *Trojan Horse* attacks. A *Trojan Horse* program is the one that appears to be doing one thing on the surface but actually does something more underneath without the cognizance of the user. Because programs inherit the identity of the invoking user, the intruder can bypass the access control policies by giving the authorized user the *Trojan Horse* program, which on the surface

performs the desirable function for that user, while at the same time reads the contents of user's files and writes them to the reachable location for both the authorized user and the intruder. In this manner, the intruder can now access the information which was supposed to be protected from him.

2.1.3 Mandatory Access Control Model

The Mandatory Access Control (MAC) policies are known to be defined to prevent the *Trojan Horse* problem [23]. An important goal of MAC is to enforce information flow policies to ensure confidentiality [11] and integrity [14]. This can be done by augmenting the discretionary access control with the mandatory access control. To grant the accessibility, MAC takes a two-step approach. First, each subject's access privileges stored in the discretionary access control matrix are checked. These privileges can be modified by subjects as mentioned earlier in Section 2.1.2. However, having authorizations stored in the access control matrix is not sufficient to perform the operation. In addition, the operation must be authorized by the MAC policy, over which subjects have no control. MAC policies govern access on the basis of classification of subjects and objects in the system. With regard to this model, security levels are assigned to subjects and objects. The security level associated with an object, also called security classification, reflects the sensitivity of the information contained in the object, i.e, the potential damage which could result from unauthorized disclosure of the information. The security level associated with a subject, also called security clearance, reflects the subject's trustworthiness not to disclose sensitive information to subjects not cleared to see it [74, 76]. Security levels may related with each other through the dominance relationship. The dominance relationship is defined as follow [74]:

Definition 1 (Dominance)

$A \geq B$ (read as A dominates B) if and only if the information can flow from B to A. The strictly dominates relation $>$ is defined by $A > B$ if and only if $A \geq B$ and $A \neq B$. We say that A and B are *comparable* if $A \geq B$ or $B \geq A$, otherwise A and B are *incomparable*.

Together with the dominance relationship, these security levels generally form a lattice structure. Hence, MAC policy is sometimes referred to as a lattice-based policy [74]. We now discuss different types of the mandatory access control model.

2.1.3.1 The Bell-LaPadula Model

Bell and LaPadula [11, 74] formalized the model to protect the information confidentiality. With respect to the security level of objects and subjects, the Bell-LaPadula (BLP) model [11, 23, 74] grants accessibilities based on two properties [74]:

- *Simple-Security Property:* Subject *s* can read object *o* only if $\lambda(s) \geq \lambda(o)$ where $\lambda(s)$ and $\lambda(o)$ are security level of *s* and *o* (*no-readup* property)

- $\star$*-Property (Star-Property):* Subject *s* can write object *o* only if $\lambda(s) \leq \lambda(o)$ (*no-writedown* property)

With the simple-security property, we can prevent subjects from being able to read information that dominates their clearance level and the $\star$-property prevents subjects from writing the information to the lower security level. Satisfaction of both properties ensure the system confidentiality. However, the system still lacks the system integrity because the $\star$-property allows the subject at the dominated security level to write the information to an object belonging to the dominating security level. Hence, the subject can corrupt the information at dominating level.

2.1.3.2 The Biba's Integrity Model

As the name imply, Biba [14] designed the Biba model to achieve the information integrity. Unlike BLP model, the accessibility in Biba model is based on the integrity level. The access is granted with respect to two properties [74]:

- *Simple-Integrity Property:* Subject *s* can read object *o* only if $\omega(s) \leq \omega(o)$ where $\omega(s)$ and $\omega(o)$ are integrity level of *s* and *o* (*no-readdown* property)

- *Integrity $\star$-Property:* Subject *s* can write object *o* only if $\omega(s) \geq \omega(o)$ (*no-writeup* property)

Satifying both properties prevent the information from integrity violation. However, the model suffers from the confidentiality problem because the integrity $\star$-property allows the subject to write his data to the object of the lower integrity classification (lower secrecy). This can later lead to the Trojan Horse problem.

To overcome those problems discussed in Sections 2.1.3.1 and 2.1.3.2, a composite model which can achieve both confidentiality and integrity is needed. Sandhu describes in [74] how to combine the BLP model and the Biba model using lattices to achieve such model.

2.1.4 The Clark-Wilson Model

Clark and Wilson described the differences between commercial and military security requirements in [20]. The authors argue that MAC policies lack adequate flexibility, and the primary concern for most commercial applications is the information integrity, rather than secrecy. Integrity refers to the accuracy and authenticity of information, as well as the need to ensure that objects are modified only in authorized ways by authorized personnel [23].

To ensure the information integrity, the model relies on two principles [20, 23]:

- *Well-formed transactions:* This constraint ensures that all data that starts in the valid state will remain in valid state after the execution of the transaction.

- *Separation of duties:* This constraint prevents the authorized subjects from modifying the information in the improper way. This goal is achieved by separating all critical operations into multiple subparts and requiring different person perform each subpart. For example, to authorize the check, we divide the process into *check issuing* and *check authorizing*. *Check issuing* task has to be done by the clerk, while *check authorizing* task has to be done by the account manager.

Unlike the BLP and Biba models, where the accessibility relies on the information flow controlled at the operating system kernel level. In Clark-Wilson's approach, the model ensures

that information is modified only in authorized ways by authorized people. Such requirement relies on the application-level controls which yield more flexible control that cannot be achieve from the kernel level controls [23].

2.1.5 Role-Based Access Control Model

Role-based access control model [24] is used for addressing the access control needs of commercial organizations. In RBAC permissions are attached to roles and users must be assigned to roles to get the permissions. Permissions determine what operations can be carried out on resources under access control. A user must establish a session to activate a subset of roles to which the user is assigned. Each user can activate multiple sessions, however, each session is associated with only one user. The operations that a user can perform in a session depend on the roles activated in that session and the permissions associated with those roles. RBAC also supports role hierarchies. Role hierarchies define an inheritance relationship between roles. To prevent conflict of interests that arise in an organization, RBAC allows the specification of Static and Dynamic Separation of Duty constraints. The summarization of RBAC components can be shown in Figure 2.3.

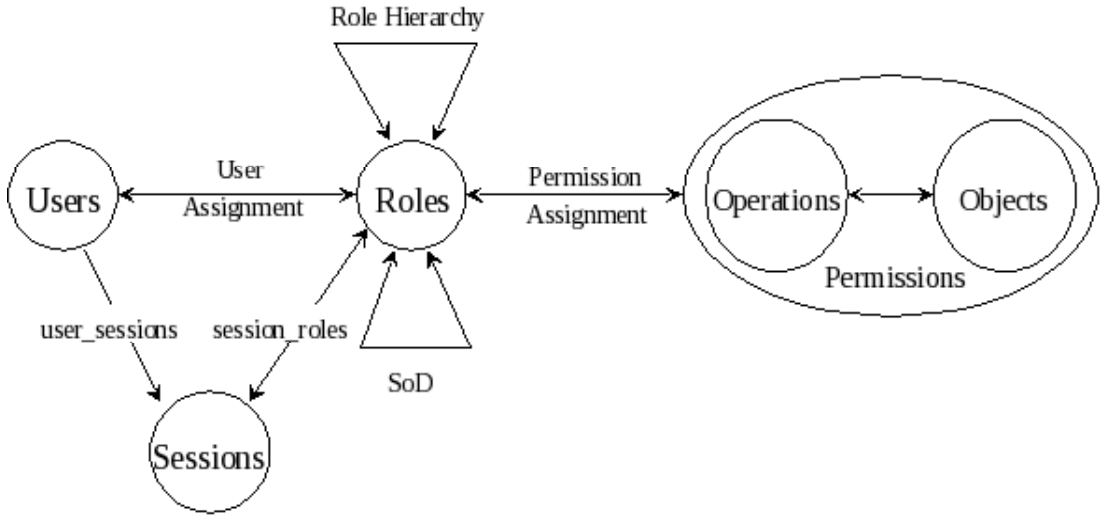


Figure 2.3: RBAC Components

RBAC approach provides several benefits [23, 24, 40, 75, 76] including:

- *Security Management:* RBAC model specifies user authorizations by breaking this task into two parts, one which assigns users to roles and one which assigns permissions for objects to roles. This greatly simplifies security management. For instance, suppose a user's responsibilities change, say, due to a promotion. The user's current roles can be taken away and new roles assigned as appropriate for the new responsibilities. Similarly, if there are any changes in the permission assignments, those changes can be done at the role level without having to apply the changes to all users. And since the role structure of the organization does not change frequently, assigning permissions to role make the permissions management task easier.

- *Data Abstraction:* Instead of the read, write, execute permissions typically provided by the operating systems, RBAC can establish abstract permissions, such as credit and debit on an account object.

- *Group Objects:* RBAC provides a classification of users according to the activities they execute. Similarly, such classification should be provided for objects. Objects could be classified according to their type (letters, manuals) or their application area (commercial letters, advertising letters). Access authorizations of roles should then be on the basis of object classes, not specific objects. For example, a secretary role can be given the authorization to read and write the entire class of letters, instead of giving it explicit authorization for each single letter. This approach has the advantage of making authorization administration much easier and better controlled. Moreover, the accesses authorized on each object are automatically determined according to the type of the object without the need of specifying authorizations upon each object creation [76].

- *Least Privilege Principle:* User is allowed to be assigned to multiple roles. This allows an user to sign on with the least privilege required for the particular task at hand. Users authorized to powerful roles do not need to exercise them until those privileges are actually needed. This minimizes the danger of damage due to inadvertent errors or by intruders masquerading as legitimate users [76].

- *De Facto Standard:* Nowadays, RBAC is widely used as an industrial standard [28, 58].

- *Support Role Hierarchy:* The structure of an organization in terms of lines of authority can be modeled as an hierarchy. This organization structure can be easily reflected in RBAC in the form of a role hierarchy [75]. Role hierarchy is a relation among roles. Roles higher up in the hierarchy are referred to as *senior roles* and those lower down are *junior roles*. The major motivation for adding role hierarchy to RBAC was to simplify role management. Senior roles can inherit the permissions of junior roles, or a senior role can activate a junior role, or do both depending on the nature of the hierarchy. This obviates the need for separately assigning the same permissions to all members belonging to a hierarchy.

- *Support Separation of Duties:* Separation of duties (SoD) enables the protection of the fraud that might be caused by the user [80]. SoD constraints ensure that the invocation of mutually exclusive roles be required to complete a sensitive task [75]. Hence, a deliberate fraud is more difficult to perpetrate because it requires collusion of two or more individuals or parties. RBAC supports invoking SoD constraints both statically and dynamically.

- *Policy Neutrality:* RBAC is policy neutral. It can be configured to model the specification of other access control e.g. MAC policies in which system administrator maintains the access matrix or DAC policies in which users create and update security policies for their devices [53, 59, 60, 73].

The Spatio-Temporal Role-Based Access Control (STRBAC) model proposed in this dissertation is an extension of NIST RBAC model [23, 24]. The extensions are with respect to augmenting the time dimension and location dimension to the core components of the existing RBAC model.

2.1.5.1 Context Aware Role-Based Access Control Model

With the increase in the growth of wireless networks and sensor and mobile devices, researchers have also worked on extending RBAC to recognize the context information to support the ubiquitous computing applications.

Sampemane et al. [71] present a new access control model for active spaces. Active space denotes the computing environment integrating physical spaces and embedded computing software and hardware entities. The active space allows interactive exchange of information between the user and the space. Environmental aspects are adopted into the access control model for active spaces, and the space roles are introduced into the implementation of the access control model based on RBAC. The model supports specification of MAC policies in which system administrator maintains the access matrix and DAC policies in which users create and update security policies for their devices.

Covington et al. [22] introduce environment roles in a generalized RBAC model (GRBAC) to help control access control to private information and resources in ubiquitous computing applications. The environments roles differ from the subject roles in RBAC but do have similar properties including role activation, role hierarchy and separation of duty. In the access control framework enabled by environment roles, each element of permission assignment is associated with a set of environment roles, and environment roles are activated according to the changing conditions specified in environmental conditions; in this way, environmental properties like time and location are introduced into the access control framework. In a subsequent work [21], Covington et al. describe the Context-Aware Security Architecture (CASA) which is an implementation of the GRBAC model. The access control is provided by the security services in the architecture. In CASA, polices are expressed as roles and managed by the security management service, authentication and authorization services are used to verify user credentials and determine access to the system resources. The environmental role activation services manage environmental role activation and deactivation according to the environment variables collected by the context management services.

Ya-Jun et al. [89] propose Trust Based Access Control (TBAC), the extension of the Role

Based Access Control model (RBAC), for ubiquitous computing application where users are not known in advance. The access privileges of a user depends on his trust level which in turn depend on contextual information. The model is based on the basic RBAC model and does not take into account the role hierarchy and separation of duty constraint. Our work also focuses on such feature of RBAC. Moreover, we also study on the effect of trust on the operation such as delegation of authorities.

Chakraborty et al. [17] propose another trust-based authorization model called TrustBAC. The model is the extension of the hierarchical RBAC model. In this model, user can activate the role and invoke the permissions assigned to that role based on his *trust level*. User's trust level can be obtained from the calculation based on three factors–user's past behavior, knowledge about user, and recommendation provided by others about the user. The trust level will be updated periodically. Chakraborty's model also introduce the concept of *trust dominance* which is equivalent to the inheritance hierarchy. The model however, does not take into account the activation hierarchy nor the separation of duty. Our trust-based access control model fills in this gap.

2.1.5.2 Temporal Role-Based Access Control Model

Other extensions to RBAC include the Temporal Role-Based Access Control Model (TRBAC) proposed by Bertino et al. [12]. This work adds the time dimension to the RBAC model. The authors in this paper introduce the concept of *role enabling* and *role disabling*. Temporal constraints determine when the roles can be enabled or disabled. A role can be activated only if it has been enabled. Joshi et al.[40, 41, 43] extend this work by proposing the Generalized Temporal Role Based Access Control Model (GTRBAC). The authors identify two basic types of temporal hierarchy. The first is the permission inheritance hierarchy where a senior role x inherits the permission of a junior role y. The second is the role activation hierarchy where a user assigned to a senior role can activate a junior role. The authors also propose Time-Based SoD. In [40, 42, 43], the authors discuss two forms of SSoD with the existing of temporal information–the *Weak Form* and *Strong Form*. The Weak Form states that no two conflicting

roles can be assigned to the same user at the same time. The Strong Form is equivalent to the non-temporal RBAC i.e. it states that no two conflicting roles can be assigned to the same user at any time. The same semantics can be applied to the DSoD. The model focus on the User-Role assignment only. The definition of SoD in our proposed model is based on the one in GTRBAC model. However, we enhance the constraints to support spatial information. Moreover, we fill the gap existing in GTRBAC model by introducing the definition of the other form of SSoD (Permission-Role assignment).

2.1.5.3 Spatial Role-Based Access Control Model

Researchers have also extended RBAC to incorporate spatial information [13, 64]. Bertino et al. propose the GEO-HRBAC–the GEO-RBAC model supporting the Spatial Role-Hierarchy in [13]. In GEO-HRBAC model, role activation is based on the location of the user. Moreover, the senior role can inherits permissions assigned to its junior role only when the user of the senior role is in junior role's enabled location. The model does not deal with separation of duties.

Another work incorporating spatial information is by Ray et al. [64]. Here again, the authors propose how each component of RBAC is influenced by location. The authors define their formal model using the Z specification language. Role hierarchy and separation of duties are not addressed in this paper. None of these works discusses the impact of time on location.

2.1.5.4 Spatio-Temporal Role-Based Access Control Model

Incorporating both time and location in RBAC is addressed by several works [18, 72]. Chandrans work combines the main features of GTRBAC and GEO-RBAC. Here again, role is enabled by time constraints. The user can activate the role if the role is enabled and the user satisfies the location constraints associated with role activation. Our Spatio-Temporal RBAC model is closely related to this work. The similarity is that in both the models role activation occurs when temporal and spatial constraints are satisfied. However, there are a number of points where we differ. First, in Chandran's work, role assignment is not dependent on location or time. A number of motivating examples indicate that role assignment should be

dependent on role and time. Consequently, we incorporate this feature in our model. Second, in Chandran's work, when a role can be activated all the permissions associated with the role can be invoked. This may not be true in real world. For instance, a system administrator's role can be activated from 9:00 a.m. to 9:00 p.m. everyday. However, he can perform backup only during 8:00 to 9:00 p.m. on Fridays. Chandran's model cannot express this situation. We associate a permission with additional location and temporal constraints that must be satisfied before a permission can be invoked. Third, Chandran's work does not discuss the impact of location and time on role hierarchy or separation of duty. We propose different types of time and location based hierarchy and separation of duty constraints in our model which will be useful for real-world applications.

Samuel et al. [72] propose GST-RBAC which incorporates topological spatial constraints to the existing GTRBAC model. The authors do this by augmenting GTRBAC operations, namely, role enabling, user-role assignment, role-permission assignment, and role-activation with spatial constraints. The operations are allowed only if the spatial and temporal constraints are satisfied. The model also introduces the notion of Spatial Role Hierarchy and Spatial Separation of Duty (spSoD) constraints. Although the goal of the model is similar to our work, Samuel's model is different from our work in various of points. First, again the spatial and temporal constraints are not applied to the permissions assigned to role. When a role can be activated all the permissions a ssociated with the role can be invoked. This may not be true in the real world as illustrated by the example in the summary of Chandran's work. Second, Samuel's work only discuss the permission inheritance type of role hierarchy. This may not be sufficient in the real world. For example, a project manager may be able to activate the code developer role but we should not allow him to inherit permissions from the developer role for the responsibility purpose. To resolve this scenario, we also include the role activation hierarchy in our work. Third, in Samuel's work, the hierarchical relationship mainly focus on the spatial constraints. The model assume that both senior role and junior role are temporally enabled i.e. both roles satisfy the temporal constraints. This also may not be true. For instance, the account auditor role may inherits all permissions from the accountant role. He can use the

inherited permissions at any time and at any place. We associate the time as well as location constraints in our model to handle such requirement. Fourth, Samuel's work discuss only the dynamic separation of duty. We argue that this might not be enough for the real world. For instance, we cannot allow the check writer and check authorizer role to be assigned to the same user. Consequently, we include the static separation of duty to our model. Fifth, Samuel's work does not incorporate time constraints into the separation of duty. Two conflict roles are allowed to activate if user is in the different location during the same time period when both roles are enabled. This may not sufficient for the real world situation. For example, if a user has activated the Graduate Teaching Assistant role in his office, he should not be able to activate the role of Lab Operator at anywhere during the same time period. To handle this situation, we also incorporate the time constraints into both types of separation of duties in our work.

Chen and Crampton develop the graph based representation for the spatio-temporal RBAC in [19]. All RBAC components are represented by vertices while the assignment and hierarchical relationships are represented by the edges of the directed graph. The model can be categorized into three types i.e. standard, strong, and weak model. For the standard model, component v_1 is said to be authorized to component v_n if all vertices along the authorization path satisfy the spatio-temporal constraints. For the strong model, component v_1 is said to be authorized to component v_n if all vertices together with the edges along the authorization path satisfy the spatio-temporal constraints. And in the weak model, component v_1 is said to be authorized to component v_n if both vertices satisfy the spatio-temporal constraints. The model has a well-defined semantics. However, it does not address separation of duty or delegation constraints. It also does not take into account the spatio-temporal attributes of the object before determining access.

2.1.6 Other Spatio-Temporal Access Control Models

Location-based access control has been addressed in other works not pertaining to RBAC [27, 48, 63]. Many researchers have developed the non-RBAC based access control which support the usage of spatio-temporal information.

Atluri and Chun [5, 6] propose the Geospatial Data Authorization Model (GSAM), which is the authorization model for the Geospatial information. The accessibility to the information is provided based on the relationship between the geospatial object and the credential of the requester, which is the requester geospatial information. To access the specific geospatial information of the object, the credential that the user own has to match with the corresponding credential expression defined as an authorisation for that object. The authorization is valid only during the specified period which defined by the temporal term of the authorization.

Ardagna et al. [4] present the Location-Based Access Control (LBAC) model. In this model, the requester can be granted or denied access by checking her location as well as her credentials. The examples of the location-based information of the requester used in the model are: the location of the requester, her velocity, and the number of people in that location. All these information form the location conditions which later can be used to determine the accessibility of the requester.

Yu et al. [90] propose LTAM, a location-temporal authorization model which focuses on controlling access to the different locations. For example, access rules may have temporal constraints that can specify when a user can enter or leave a location or how many times a user can enter a location. However, it does not address the issue of where and when a subject can access a given object. And since this model is based on DAC, authorization management is non-trivial.

Pu et al. [61] present the context access control model, called CACM, which integrates the context information to the $UCON_{ABC}$ usage control model. To access the resource, the user must satisfy the predefined combination of authorization, obligation, and condition constraints. The value of conditional status can be changed as the environmental situation is being changed (e.g. the change of time, location associated with user). Nonetheless, the impacts on the model components as a result from introducing the context information are not mentioned in the work.

Context Sensitive Access Control (CSAC) [29] proposed by Hulsebosch et al. focus on using context information such as time, location, velocity to control the accessibility of services while preserving the privacy of user information. Hengartner et al. [27] discuss how location

information pertaining to a user can be securely accessed.

2.2 Access Control Model Analysis

A lot of work also appears in the area of analysis of security policies. Researchers have used formal logic for specifying authorization policies so that they can be analyzed. Many work appears that attempt to analyze RBAC specifications. Some have used the Z modeling language for specifying RBAC [91] and LRBAC [64]. Although Z language can represent RBAC and its constraints in the formal manner, the language itself lacks the tool to support the automatic analysis of the formalized model. Others have used an extension of the Unified Modeling Language (UML) [65] called parameterized UML to visualize the properties of RBAC constraints. The model describes how one can visualize the conflicts that may occur with RBAC constraints. However, it still lacks the ability to perform automatic model analysis.

Researchers have also advocated the use of Alloy for modeling RBAC specifications. In [92], Zao et al. model basic features of RBAC, role hierarchy, and static separation of duties. The author briefly illustrates how to use Alloy to model the Bell-LaPadula access control model.

Schaad et al. model user-role assignment, role-permission assignment, role hierarchy, and static separation of duties features of RBAC extension using Alloy in [77]. The authors do not model role activation hierarchy, dynamic separation of duties or the delegation operation. The authors briefly describe how to analyze conflicts in the context of the model.

Samuel et al. [72] illustrate how GST-RBAC can be specified in Alloy. They describe how the various GST-RBAC functionalities, that is, user-role assignment, role-permission assignment, and user-role activation, can be specified by Alloy. However, this work does not focus on how to identify interactions between features that result in conflicts.

Although Alloy supports automated analysis, it has limitations with respect to the types of verifications it can perform. For example, analyzing and understanding the behavior of the application using Alloy is non-trivial. Such analysis is needed for dynamic systems where we need to ensure that the system does not enter an undesirable state. Towards this end, re-

searchers [7, 39, 46, 49, 62] have investigated alternate approaches, such as, Coloured Petri Nets (CPNs) [35, 38, 46] for automated analysis. CPN allows one to represent the model in a graphical language, has a well-defined semantics and has automated tools for doing simulation and verification.

Rasmussen and Singh [62] show how CPN is used in designing the PRISMA C96 intruder alarm system. The interactions of components were modeled and verified using CPN to detect if the configurations have any conflicts. CPN has also been used in access control model verification. Jiang et al. [39] develop a CPN model to verify the security properties of the Bell LaPadula (BLP) model. Laborde et al. propose the use of CPN for analyzing the traditional RBAC-based policies of network security mechanisms in [46]. This work focuses on verifying confidentiality, integrity, availability, and filtering rules.

Lu et al. [49] show how access control properties of workflows can be verified using CPNs. Specifically, they describe how to formalize the control flow, authorization rules, and separation of duty constraints in a workflow in the presence of role activation hierarchy. The authors first show how to model each part (namely, control flow, authorization rules, and separation of duty) in isolation. Subsequently, the authors propose an approach for producing the integrated model which allows one to study the interactions of the parts, such as RBAC authorization policy with separation of duty constraints. Reachability analysis is used to detect conflicts between the features. The size of the integrated model increases exponentially when new entities are added. To prevent state explosion during reachability analysis, the authors introduce two rules for reducing the size of the model. The model analyzed by the authors do not support many features which are needed in workflow applications: permission inheritance hierarchy, separation of duty for permission-role assignment, and delegation.

Atluri et al. [8] propose an authorization model to use for workflows. The model specifies constraints that allows authorized subjects to gain access on the required objects for the duration of the task. Subsequently, the authors extend this work to support task-based separation of duty constraints and show how this extended model can be specified using CPN [7]. The authors then show how to do a reachability analysis to check whether the given tasks can be

executed in the presence of authorization constraints.

Shafiq et al. [78] show how the various constraints of GTRBAC, such as, cardinality constraints, SoD constraints, and role hierarchy can be modeled using CPN. The reachability analysis reveals the presence of infeasible paths where an entity cannot invoke the privileges assigned to him. However, analyzing the interaction of constraints is not discussed in these works.

Samrat et al. [51] demonstrate how to analyze the properties of GTRBAC model using timed automata. The authors propose the methodology to transform GTRBAC into the state transition model. The model maps the behavior of the GTRBAC components such as users, roles, and permissions. The different behaviors of components are captured by creating the timed automaton corresponding to each types of interactions. A desirable set of security properties is constructed from the GTRBAC constraints. These properties are later used for the model verification process. The verification process is automatically done by using a time automata based verification tool called Uppaal [9, 10]. Although, this approach makes it possible to perform an automated analysis of the model, there are two major limitations. First, because the number of timed automaton is varied by the number of components, the approach is affected by the state space explosion problem. Therefore, verifying the large scale system by using this approach still remains a challenge. And second, because every time the changes occur to the components of the relationships between them, the set of the corresponding timed automatons have to be changed i.e. the existing timed automatons must be updated or the new timed automatons must be added. The process of adding or updating the timed automatons requires some time. This makes this approach not suitable for the pervasive computing application which the configuration is frequently changed. Our work present a methodology to reduce the scope of analysis to rectify the former issue. For the latter issue, our verification methodology can handle such changes in the access control model configuration by updating only the set of relationships or components affected by the modification of the access control model configuration while the core part of the verification model remains unchanged.

2.3 Chapter Summary

In this chapter, we briefly discuss the work related to two research areas in which this dissertation is focusing on, namely, the access control model and its analysis. Regarding the access control model, our research indicated that none of these work provide an access control model which fully support the requirements of the pervasive computing applications discussed in Chapter 1. In subsequent chapters, we will discuss our access control models which fulfill this gap.

Chapter 3

The Spatio-Temporal Role Based Access Control Model

Our preliminary investigation reveals the following: Pervasive computing applications need different kinds of policies for protecting sensitive resources. Authorization policies will be used to protect resources from unauthorized access. Simple authorization policies are specified by subject, object, and permissions, where the permissions specify what operations the subject can perform on the object. More complex authorization policies also have conditions specifying access constraints. Consider our running example the a cardiac patient living by himself in a smart home. Simple authorization policies may be defined that allow only the responsible doctor to be able to monitor the patient's data collected by sensors in the smart home. Policies are needed to prevent unauthorized disclosure or modification of the data that is transmitted to the different sensors.

For critical applications, delegation policies are needed. A delegation permits subjects to temporarily transfer some of their privileges to other subjects. Pervasive computing applications are dynamic in nature and the set of users and resources are not known in advance. It is possible that a user/role for doing a specific task is temporarily unavailable and another user/role must be granted access during this time to complete it. This necessitates that the model be able to support delegation. Moreover, different types of delegation needs to be supported because of the unpredictability of the application. A delegation policy may also have conditions suggesting the delegation constraints. For example, a delegation policy will enable

the emergency room doctor to temporarily put the cadio pacer monitoring the cardiac patient's heart rate to a sleep mode. If the local hospital does not have qualified cardiologists to evaluate the patient's condition, the job is delegated to external experts. The responsible cardiologist may specify the conditions the experts must satisfy before the job can be delegated (for instance, working experience in the area of cardiology).

In pervasive computing applications, the access decisions cannot be based solely on the attributes of users and resources. For instance, we may want access to the patient information be enabled when a doctor enters a patient room and it to be disabled when he leaves the room. Such types of access control can only be provided if we take environmental contexts, such as, time and location, into account before making access decisions. Thus, the access control model for pervasive computing applications must allow for the specification and checking of environmental conditions.

As a result of our study different access control models, we decided to base our work on the RBAC model for the following reasons: First, reinventing a new kind of access control is the time consuming process. Since RBAC is widely use as an industrial standard, developing the model based on RBAC reduces the complexity in the implementation phase. Second, RBAC model has powerful access control management. In RBAC, permissions are assign to roles not to users, hence, if there are any changes in the permission assignments, those changes can be done at the role level without having to apply the changes to all users. Since the role structure of the organization does not change frequently, assigning permissions to role make managing permissions easier. Furthermore, RBAC supports concept of role hierarchy. In role hierarchy, the senior role in the hierarchy inherits all permissions assigned to junior role. This idea helps us to avoid the redundancy in assigning the same set of permissions to the senior role. Third, RBAC supports separation of duty constraints and delegation of authority. Basically, separation of duty states that no single individual should have control over two or more phases of a transaction or operation. Thus, a deliberate fraud is more difficult to perpetrate because it requires collusion of two or more individuals or parties. Finally, RBAC is policy neutral. It can model the specification of other access control policies, such as MAC or DAC policies

[53, 59, 60, 73] as well.

In this chapter, we describe how we incorporate the spatio-temporal constraints to the components of RBAC model.

3.1 The Spatio-Temporal Role Based Access Control (STRBAC) Model

In order to extend the RBAC model to be able to use the spatio-temporal information to determine whether a user has access to a given object, we need to understand how RBAC components and their relationships are affected by spatio-temporal information.

From Section 3.1.1 to 3.1.2, we propose the STRBAC model entities. Next, we discuss the spatio-temporal role hierarchies in Section 3.2. Then, we describe the variations of spatio-temporal separation of duty constraints in 3.3. The preliminary version of this model has been proposed in [67, 83].

3.1.1 Representing Location and Time

3.1.1.1 Representing Location

In order to perform location-based access control, we need to perform operations on location information and protect the location information. In this section, we formalize the concept of location [13, 18] and propose the location comparison operators that are used in our model.

There are two types of locations: *physical* and *logical*. All users and objects are associated with locations that correspond to the physical world. These are referred to as the physical locations. A physical location is formally defined by a set of points in a three-dimensional geometric space.

Definition 2 (Physical Location)

A physical location $PLoc_i$ is a non-empty set of points $\{p_i, p_j, \ldots, p_n\}$ where a point p_k is represented by three co-ordinates.

Physical locations are grouped into symbolic representations that will be used by applications. We refer to these symbolic representations as logical locations. Examples of logical

locations are Fort Collins, Colorado etc.

Definition 3 (Logical Location)

A logical location is an abstract notion for one or more physical locations.

We assume the existence of two translation functions, m and m', that convert from logical locations to physical locations and vice-versa.

Definition 4 (Mapping Functions m and m')

m is a total function that converts a physical location into a logical one. m' is a total function that converts a logical location into a physical one. Let P be the set of all possible physical locations and L be the set of all logical locations. The following formalizes the functions.

- $m : P \rightarrow L$.
- $m' : L \rightarrow P$
- For any logical location Loc_i, $m(m'(Loc_i)) = Loc_i$.
- For any physical location $PLoc_j$, $m'(m(PLoc_j)) = PLoc_j$.

Different kinds of relationships may exist between a pair of locations. We discuss one such relationship, known as *containment*, that will be used in this paper. Containment formalizes the idea whether one location is contained within another. Intuitively, a physical location $ploc_j$ is contained in another physical location $ploc_k$, if all points in $ploc_j$ also belong to $ploc_k$. This is formalized as follows.

Definition 5 (Containment Relation)

A physical location $ploc_j$ is said to be contained in another physical location $ploc_k$, denoted as, $ploc_j \subseteq ploc_k$, if the following condition holds: $\forall p_i \in ploc_j, p_i \in ploc_k$. The location $ploc_j$ is called the contained location and $ploc_k$ is referred to as the containing or the enclosing location. A logical location $lloc_m$ is contained in $lloc_n$, denoted as, $lloc_m \subseteq lloc_n$, if and only if the physical location corresponding to $lloc_m$ is contained within that of $lloc_m$, that is $m'(lloc_m) \subseteq m'(lloc_n)$.

Note that, a physical location may be contained in a logical location or vice-versa. In such cases, we use the mapping functions to convert the logical locations into physical ones and then test whether one is contained within the other. We assume the existence of a logical location called *universe* that contains all other locations. In the rest of the section, we do not discuss physical locations any more. The locations referred to are logical locations.

3.1.1.2 Representing Time

Our model uses two kinds of temporal information. It is necessary to distinguish between these two kinds of information because they have very different semantics. The first is known as time instant and the other is time interval. Time can be represented as a set of discrete points on the time line.

Definition 6 (Time Instant)

A *time instant* is one discrete point on the time line.

The exact granularity of a time instant will be application dependent. For instance, in some application a time instant may be measured at the nanosecond level and in another one it may be specified at the millisecond level.

Definition 7 (Time Interval)

A *time interval* is a set of time instances. When the time instances making up an interval are consecutive, we refer to the interval as a *continuous* one. Otherwise, the interval is said to be *non-continuous*.

Example of a continuous interval is 9:00 a.m. to 3:00 p.m. on 25th December. Example of a non-continuous interval is 9:00 a.m. to 6:00 p.m. on Mondays to Fridays in the month of March. Some researchers refer to time intervals as time expressions. We use the notation $t_i \in d$ to mean that t_i is a time instance in the time interval d.

Two time intervals can be related by any of the following relations: *disjoint*, *equality*, and *overlapping*. Two time intervals tv_i and tv_j are disjoint if the intersection of the set of time instances in tv_i with those of tv_j results in the null set. Two time intervals tv_i and tv_j are equal if the set of time instances in tv_i is equal to those of tv_j. Two time intervals tv_i and tv_j are

overlapping if the intersection of the set of time instances in tv_i with those of tv_j results in a non-empty set. A special case of overlapping relation is referred to as *containment*. A time interval tv_i is contained in another interval tv_j if the set of time instances in tv_i is a subset of those in tv_j. We formally denote this as $tv_i \preceq tv_j$.

3.1.2 Relationship of Core-RBAC Entities with Time and Location

In this section, we discuss how the different entities of core RBAC, namely, *Users*, *Roles*, *Sessions*, *Permissions*, and *Objects*, are associated with location and time.

3.1.2.1 Users

We assume that each valid user, interested in doing some location-sensitive operation, carries a locating device which is able to track his location. The location of a user changes with time. The relation $UserLocation(u,t)$ gives the location of the user at any given time instant t. Since a user can be associated with only one location at any given point of time, we have the following constraint:

$$UserLocation(u,t) = l_i \wedge UserLocation(u,t) = l_j \Leftrightarrow (l_i \subseteq l_j) \vee (l_j \subseteq l_i)$$

We define a similar function $UserLocations(u,d)$ that gives the location of the user during the time interval d. Note that, a single location can be associated with multiple users at any given point of time.

3.1.2.2 Objects

Objects can be physical or logical. Example of a physical object is a computer. Files are examples of logical objects. Physical objects have devices that transmit their location information with the timestamp. Logical objects are stored in physical objects. The location and timestamp of a logical object corresponds to the location and time of the physical object containing the logical object. We assume that each object is associated with one location at any given instant of time. Each location can be associated with many objects. The function *ObjLocation(o,t)* takes as input an object o and a time instance t and returns the location associated with the

object at time t. Similarly, the function *ObjLocations(o,d)* takes as input an object *o* and time interval *d* and returns the location associated with the object.

3.1.2.3 Roles

We have three types of relations with roles. These are user-role assignment, user-role activation, and permission-role assignment. We begin by focusing on user-role assignment. Often times, the assignment of user to roles is location and time dependent. For instance, a person can be assigned the role of U.S. citizen only in certain designated locations and at certain times only. To get the role of conference attendee, a person must register at the conference location during specific time intervals. Thus, for a user to be assigned a role, he must be in designated locations during specific time intervals. In our model, a user must satisfy spatial and temporal constraints before roles can be assigned. We capture this with the concept of *role allocation*. A role is said to be *allocated* when it satisfies the temporal and spatial constraints needed for role assignment. A role can be assigned once it has been allocated. $RoleAllocLoc(r)$ gives the set of locations where the role can be allocated. $RoleAllocDur(r)$ gives the time interval where the role can be allocated. Some role *s* can be allocated anywhere, in such cases $RoleAllocLoc(s) = universe$. Similarly, if role p can be assigned at any time, we specify $RoleAllocDur(p) = always$.

Some roles can be activated only if the user is in some specific locations. For instance, the role of audience of a theater can be activated only if the user is in the theater when the show is on. The role of conference attendee can be activated only if the user is in the conference site while the conference is in session. In short, the user must satisfy temporal and location constraints before a role can be activated. We borrow the concept of *role-enabling* [12, 43] to describe this. A role is said to be *enabled* if it satisfies the temporal and location constraints needed to activate it. A role can be activated only if it has been enabled. $RoleEnableLoc(r)$ gives the location where role *r* can be activated and $RoleEnableDur(r)$ gives the time interval when the role can be activated.

The predicate $UserRoleAssign(u,r,d,l)$ states that the user u is assigned to role r during

the time interval d and location l. For this predicate to hold, the location of the user when the role was assigned must be in one of the locations where the role allocation can take place. Moreover, the time of role assignment must be in the interval when role allocation can take place.

$$UserRoleAssign(u,r,d,l)$$
$$\Rightarrow (UserLocation(u,d) = l) \wedge (l \subseteq RoleAllocLoc(r)) \wedge (d \subseteq RoleAllocDur(r))$$

The predicate $UserRoleActivate(u,r,d,l)$ is true if the user u activated role r for the interval d at location l. This predicate implies that the location of the user during the role activation must be a subset of the allowable locations for the activated role and all time instances when the role remains activated must belong to the duration when the role can be activated and the role can be activated only if it is assigned.

$$UserRoleActivate(u,r,d,l)$$
$$\Rightarrow (l \subseteq RoleEnableLoc(r)) \wedge (d \subseteq RoleEnableDur(r)) \wedge UserRoleAssign(u,r,d,l)$$

The additional constraints imposed upon the model necessitates changing the preconditions of the functions *AssignRole* and *ActivateRole*. The permission role assignment is discussed later.

3.1.2.4 Sessions

In mobile computing or pervasive computing environments, we have different types of sessions that can be initiated by the user. Some of these sessions can be location-dependent, others not. Thus, sessions are classified into different types. Each instance of a session is associated with some type of a session. The type of session instance s is given by the function $Type(s)$. The type of the session determines the allowable location. The allowable location for a session type *st* is given by the function $SessionLoc(st)$. When a user u wants to create a session *si*, the location of the user for the entire duration of the session must be contained within the location associated with the session. The predicate $SessionUser(u,s,d)$ indicates that a user u has initiated a session s for duration d.

$SessionUser(u,s,d) \Rightarrow (UserLocation(u,d) \subseteq SessionLoc(Type(s)))$

Since sessions are associated with locations, not all roles can be activated within some session. The predicate $SessionRole(u,r,s,d,l)$ states that user u initiates a session s and activates a role for duration d and at location l.

$SessionRole(u,r,s,d,l) \Rightarrow UserRoleActivate(u,r,d,l) \wedge l \subseteq SessionLoc(Type(s)))$

3.1.2.5 Permissions

The goal of our model is to provide more security than their traditional counterparts. This happens because the time and location of a user and an object are taken into account before making the access decisions. Our model also allows us to model real-world requirements where access decision is contingent upon the time and location associated with the user and the object. For example, a teller may access the bank confidential file if and only if he is in the bank and the file location is the bank secure room and the access is granted only during the working hours. Our model should be capable of expressing such requirements.

Permissions are associated with roles, objects, and operations. We associate three additional entities with permission to deal with spatial and temporal constraints: user location, object location, and time. We define three functions to retrieve the values of these entities. $PermRoleLoc(p,r)$ specifies the allowable locations that a user playing the role r must be in for him to get permission p. $PermObjLoc(p,o)$ specifies the allowable locations that the object o must be in so that the user has permission to operate on the object o. $PermDur(p)$ specifies the allowable time when the permission can be invoked.

We define another predicate which we term *PermRoleAcquire(p,r,d,l)*. This predicate is true if role r has permission p for duration d at location l. Note that, for this predicate to be true, the time interval d must be contained in the duration where the permission can be invoked and the role can be enabled. Similarly, the location l must be contained in the places where the permission can be invoked and role can be enabled.

$PermRoleAcquire(p,r,d,l)$

$$\Rightarrow (l \subseteq (PermRoleLoc(p,r) \cap RoleEnableLoc(r))) \wedge$$
$$(d \subseteq (PermDur(p) \cap RoleEnableDur(r)))$$

The predicate $PermUserAcquire(u,o,p,d,l)$ means that user u can acquire the permission p on object o for duration d at location l. This is possible only when the permission p is assigned some role r which can be activated during d and at location l, the user location and object location match those specified in the permission, the duration d matches that specified in the permission.

$$PermRoleAcquire(p,r,d,l) \wedge UserRoleActivate(u,r,d,l) \wedge$$
$$(ObjectLocation(o,d) \subseteq PermObjectLoc(p,o)) \Rightarrow PermUserAcquire(u,o,p,d,l)$$

3.2 Impact of Time and Location on Role-Hierarchy

The structure of an organization in terms of lines of authority can be modeled as an hierarchy. This organization structure is reflected in RBAC in the form of a role hierarchy [75]. Role hierarchy is a relation among roles. This relation is transitive, and anti-symmetric. Roles higher up in the hierarchy are referred to as senior roles and those lower down are junior roles. The major motivation for adding role hierarchy to RBAC was to simplify role management. A properly design role hierarchy allows efficient specification and management of access control structure of the system.

Senior roles can inherit the permissions of junior roles, or a senior role can activate a junior role, or do both depending on the nature of the hierarchy. This obviates the need for separately assigning the same permissions to all members belonging to a hierarchy.

In the environment where the accessibility is based on the time and location, it is necessary that the role hierarchy should evaluate the spatio-temporal information before allow the senior role to inherits the junior role's permissions or activate the junior role. In our model, each of these hierarchies may be constrained by location and temporal constraints.

3.2.0.6 The Spatio-Temporal Permission Inheritance Hierarchy

In the permission inheritance hierarchy, a senior role x inherits the permission of a junior role y. By incorporating various time and location constraints, we can have a number of different hierarchical relationships in our model that are described below.

Definition 8 (Unrestricted Permission Inheritance Hierarchy)

Let x and y be roles such that $x \geq y$, that is, senior role x has an unrestricted permission-inheritance relation over junior role y. In such a case, x inherits y's permissions but not the locations and time associated with it. This is formalized as follows:

$$(x \geq y) \wedge PermRoleAcquire(p, y, d, l) \Rightarrow PermRoleAcquire(p, x, always, universe)$$

Here, a senior role inherits the junior roles permissions but not the spatial and temporal constraints associated with it. For example, *account auditor* role inherits the permissions from the *accountant* role but he can use the permissions at any time and at any place.

Definition 9 (Time Restricted Permission Inheritance Hierarchy)

Let x and y be roles such that $x \geq_t y$, that is, senior role x has a time restricted permission-inheritance relation over junior role y. In such a case, x inherit's y's permissions together with the temporal constraints associated with the permission. This is formalized as follows:

$$(x \geq_t y) \wedge PermRoleAcquire(p, y, d, l) \Rightarrow PermRoleAcquire(p, x, d, universe)$$

Here, a senior role inherits the junior role's permissions but the duration when the permissions are valid are those that are associated with the junior role. For example, a *contact author* can inherit the permissions of the *author* until the paper is submitted.

Definition 10 (Location Restricted Permission Inheritance Hierarchy)

Let x and y be roles such that $x \geq_l y$, that is, senior role x has a location restricted permission-inheritance relation over junior role y. In such a case, x inherit's y's permissions together with the location constraints associated with the permission. This is formalized as follows:

$$(x \geq_l y) \wedge PermRoleAcquire(p, y, d, l) \Rightarrow PermRoleAcquire(p, x, always, l)$$

Here, a senior role inherits the junior roles permissions but these permissions are restricted to the locations imposed on the junior roles. For example, a *top secret scientist* inherits the permission of *top secret citizen* only when he is in top secret locations.

Definition 11 (Time Location Restricted Permission Inheritance Hierarchy)
Let x and y be roles such that $x \geq_{tl} y$, that is, senior role x has a time-location restricted permission-inheritance relation over junior role y. In such a case, x inherit's y's permissions together with the temporal and location constraints associated with the permission. This is formalized as follows:

$$(x \geq_{tl} y) \wedge PermRoleAcquire(p,y,d,l) \Rightarrow PermRoleAcquire(p,x,d,l)$$

Here, a senior role inherits the junior roles permissions together with the spatial and temporal constraints imposed upon those of the junior role. For example, *daytime doctor* role inherits permission of *daytime nurse* role only when he is in the hospital during the daytime.

3.2.0.7 The Spatio-Temporal Role Activation Hierarchy

In the role activation hierarchy, a user assigned to a senior role can activate a junior role. By incorporating various time and location constraints, we can have a number of different hierarchical relationships in our model that are described below.

Definition 12 (Unrestricted Activation Hierarchy)
Let x and y be roles such that $x \succcurlyeq y$, that is, senior role x has a role-activation relation over junior role y. Then, a user assigned to role x can activate role y at any time and at any place. This is formalized as follows:

$$(x \succcurlyeq y) \wedge UserRoleActivate(u,x,d,l) \Rightarrow UserRoleActivate(u,y,always,universe)$$

Here, a user who can activate a senior role can also activate a junior role at any time and at any place. For example, a *project manager* can activate the *code developer* role at any time and at any place.

Definition 13 (Time Restricted Activation Hierarchy)

Let x and y be roles such that $x \succcurlyeq_t y$, that is, senior role x has a role-activation relation over junior role y. Then, a user assigned to role x can activate role y only at the time when role y can be enabled. This is formalized as follows:

$$(x \succcurlyeq_t y) \wedge UserRoleActivate(u,x,d,l) \wedge (d \subseteq RoleEnableDur(y)) \\ \Rightarrow UserRoleActivate(u,y,d,universe)$$

Here again, a user who can activate a senior role can activate a junior role only during the time when the junior role can be activated. For example, a *program chair* can activate a *reviewer* role only during the review period.

Definition 14 (Location Restricted Activation Hierarchy)

Let x and y be roles such that $x \succcurlyeq_l y$, that is, senior role x has a role-activation relation over junior role y. Then, a user assigned to role x can activate role y only at the places when role y can be enabled. This is formalized as follows:

$$(x \succcurlyeq_l y) \wedge UserRoleActivate(u,x,d,l) \wedge l \subseteq RoleEnableLoc(y) \\ \Rightarrow UserRoleActivate(u,y,always,l)$$

Here again, a user who can activate a senior role can also activate a junior role but the activation is limited to the place where the junior role can be activated. For example, a *Department Chair* can activate a *Staff* role only when he is in the Department.

Definition 15 (Time Location Restricted Activation Hierarchy)

Let x and y be roles such that $x \succcurlyeq_{tl} y$, that is, senior role x has a role-activation relation over junior role y. Then, a user assigned to role x can activate role y only at the places and during the time when role y can be enabled. This is formalized as follows:

$$(x \succcurlyeq_{tl} y) \wedge UserRoleActivate(u,x,d,l) \wedge (d \subseteq RoleEnableDur(y)) \wedge (l \subseteq RoleEnableLoc(y)) \\ \Rightarrow UserRoleActivate(u,y,d,l)$$

Here again, a user who can activate a senior role can also activate a junior role but must obey the temporal and spatial constraints imposed on the activation of the junior role. For

example, user who has a role of *mobile user* can activate the *weekend mobile user* role only if he/she is in the US during the weekend.

When a senior role and a junior role are related with both permission inheritance and activation hierarchies, the application must choose the required type of inheritance and activation hierarchies.

3.3 Impact of Time and Location on Separation Of Duty

Separation of Duty (SoD) is a fundamental principle in security systems. SoD is basically states that *critical operations are divided among two or more people, so that no single individual can compromise security* [23]. SoD enables the protection of the fraud that might be caused by the user. There are two categories of SoD: Static (SSoD) and Dynamic (DSoD) [80]. SSoD constrains the User-Role and Permission-Role assignments, while DSoD constrains the User-Role activation. When SoD constraints are properly implemented, collusion of two or more parties is required to commit a damaging action. Therefore, the risk of critical damage is reduced. Moreover, SoD increases the opportunity for detecting errors, since two or more parties are involved in committing the fraud.

Unlike the traditional RBAC, where the SoD constraints are applied all the time at any location, with the existing of the context information, we can increase the flexibility of the model by apply the spatio-temporal information into SoD constraints.

Next, we propose variations of Spatio-Temporal SoD. The preliminary version of such constraints has been proposed in [67, 83].

3.3.0.8 The Spatio-Temporal Static Separation of Duty

Due to the presence of temporal and spatial constraints, we can have different flavors of separation of duty – some that are constrained by temporal and spatial constraints and others that are not. In the following, we describe the first form of static separation of duty constraints. This form is with respect to user role assignment.

Definition 16 (Weak Form of SSoD - User Role Assignment)

Let x and y be two roles such that $x \neq y$ and $x,y \in SSOD_w(ROLES)$, that is, x and y are two distinct roles that are related by the $SSOD_w$ relation. A user u assigned to role x during time d and location l cannot be assigned to role y at the same time and location if x and y are related by $SSOD_w$. This is formalized as follows:

$$(x,y) \in SSOD_w(ROLES) \Rightarrow (UserRoleAssign(u,x,d,l) \wedge UserRoleAssign(u,y,d,l) = False)$$

Here the same user cannot be assigned to two conflicting roles during the same time and at the same location. For example, a user should not be assigned the *audience* role and *mobile phone user* role at the same time and location.

Definition 17 (Strong Temporal Form of SSoD - User Role Assignment)

Let x and y be two roles such that $x \neq y$ and $(x,y) \in SSOD_t(ROLES)$, that is, x and y are two distinct roles that are related by the $SSOD_t$ relation. A user u assigned to role x during time d and location l cannot be assigned to role y at any time d' in the same location if x and y are related by $SSOD_t$. This is formalized as follows:

$$(x,y) \in SSOD_t(ROLES) \Rightarrow (UserRoleAssign(u,x,d,l) \wedge UserRoleAssign(u,y,d',l) = False)$$

Here the same user cannot be assigned to two conflicting roles at the same location at any time. The *consultant for oil company A* will never be assigned the role of *consultant for oil company B* in the same country.

Definition 18 (Strong Spatial Form of SSoD - User Role Assignment)

Let x and y be two roles such that $x \neq y$ and $(x,y) \in SSOD_l(ROLES)$, that is, x and y are two distinct roles that are related by the $SSOD_l$ relation. A user u assigned to role x during time d and location l cannot be assigned to role y at the same time at any location l' if x and y are related by $SSOD_l$. This is formalized as follows:

$$(x,y) \in SSOD_l(ROLES) \Rightarrow (UserRoleAssign(u,x,d,l) \wedge UserRoleAssign(u,y,d,l') = False)$$

Here the same user cannot be assigned to two conflicting roles at any location during the same time. For example, a person cannot be assigned the roles of *realtor* and *instructor* at the same time.

Definition 19 (Strong Form of SSoD - User Role Assignment)

Let x and y be two roles such that $x \neq y$ and $(x,y) \in SSOD_s(ROLES)$, that is, x and y are two distinct roles that are related by the $SSOD_s$ relation. A user u assigned to role x during time d and location l cannot be assigned to role y at any time d' or at any location l' if x and y are related by $SSOD_s$. This is formalized as follows:

$$(x,y) \in SSOD_s(ROLES) \Rightarrow (UserRoleAssign(u,x,d,l) \wedge UserRoleAssign(u,y,d',l') = False)$$

Here the same user cannot be assigned to two conflicting roles. For example, the same person cannot be assigned the role of *minority candidate* and *regular candidate* in a job application.

We next consider the second form of static separation of duty that deals with permission role assignment. The idea is that the same role should not acquire conflicting permissions. For instance, the same manager should not make a request for funding as well as approve it.

Definition 20 (Weak Form of SSoD - Permission Role Assignment)

Let p and q be two permissions such that $p \neq q$ and $(p,q) \in SSOD\_PRA_w$, that is, p and q are two distinct permissions that are related by the $SSOD\_PRA_w$ relation. A role x which has permission p at time d and location l cannot be assigned permission q at the same time and location if p and q are related by $SSOD\_PRA_w$. This is formalized as follows:

$$(p,q) \in SSOD\_PRA_w \Rightarrow (PermRoleAcquire(p,x,d,l) \wedge PermRoleAcquire(q,x,d,l) = False)$$

Here the same role cannot be assigned two conflicting permissions during the same time and at the same location. For example, a passenger role should not be assigned the permission to *go aboard the plane* and *use the cell phone* at the same place and time.

Definition 21 (Strong Temporal Form of SSoD - Permission Role Assignment)

Let p and q be two permissions such that $p \neq q$ and $(p,q) \in SSOD\_PRA_t$, that is, p and q are two distinct permissions that are related by the $SSOD\_PRA_t$ relation. A role x which has permission p at time d and location l cannot be assigned permission q at the any time d' in the same location if p and q are related by $SSOD\_PRA_t$. This is formalized as follows:

$$(p,q) \in SSOD\_PRA_t \Rightarrow (PermRoleAcquire(p,x,d,l) \wedge PermRoleAcquire(q,x,d',l) = False)$$

Here the same role cannot be assigned two conflicting permissions at the same location at any time. In the top secret base, if any role has a permission to *access the high confidential information,* the permission to *store or distribute information* should not be granted to that role.

Definition 22 (Strong Spatial Form of SSoD - Permission Role Assignment)

Let p and q be two permissions such that $p \neq q$ and $(p,q) \in SSOD\_PRA_l$, that is, p and q are two distinct permissions that are related by the $SSOD\_PRA_l$ relation. A role x which has permission p at time d and location l cannot be assigned permission q at the same time at any location l' if p and q are related by $SSOD\_PRA_l$. This is formalized as follows:

$$(p,q) \in SSOD\_PRA_l \Rightarrow (PermRoleAcquire(p,x,d,l) \wedge PermRoleAcquire(q,x,d,l') = False)$$

Here the same role cannot be assigned two conflicting permissions at any location during the time. For example, the permission to *access the exam paper* and *access the answer key* should not be assigned for the same time.

Definition 23 (Strong Form of SSoD - Permission Role Assignment)

Let p and q be two permissions such that $p \neq q$ and $(p,q) \in SSOD\_PRA_s$, that is, p and q are two distinct permissions that are related by the $SSOD\_PRA_s$ relation. A role x which has permission p at time d and location l cannot be assigned permission q at any time d' or at any location l' if p and q are related by $SSOD\_PRA_s$. This is formalized as follows:

$$(p,q) \in SSOD\_PRA_s \Rightarrow (PermRoleAcquire(p,x,d,l) \wedge PermRoleAcquire(q,x,d',l') = False)$$

Here the same role cannot be assigned two conflicting permissions. For example, The permission to *issue check* and permission to *authorize check* must not be assign to the same role.

3.3.0.9 The Spatio-Temporal Dynamic Separation of Duty

We next consider the dynamic separation of duty. Static separation of duty ensures that a user does not get assigned conflicting roles or a role is not assigned conflicting permissions. Dynamic separation of duty addresses the problem that a user is not able to activate conflicting roles during the same session.

Definition 24 (Weak Form of DSoD)

Let x and y be two roles such that $x \neq y$ and $(x,y) \in DSOD_s$, that is, two distinct roles x and y are related by $DSOD_w$. If roles x and y are related through weak DSoD and if user u has activated role x in some session s for duration d and location l, then u cannot activate role y during the same time d and in the same location l in session s. This is formalized as follows:

$$(x,y) \in DSOD_w \Rightarrow (SessionRole(u,x,s,d,l) \wedge SessionRole(u,y,s,d,l) = False)$$

This allows the same user to activate two conflicting roles in the same session but not during the same time and in the same location. A user can activate a sales assistant role and a customer role in the same session but not during the same time and in the same location.

Definition 25 (Strong Temporal Form of DSoD)

Let x and y be two roles such that $x \neq y$ and $(x,y) \in DSOD_t$, that is, two distinct roles x and y are related by $DSOD_t$. If roles x and y are related through strong temporal DSoD and if user u has activated role x in some session s, then u can never activate role y at any time d' at the same location in the same session. This is formalized as follows:

$$(x,y) \in DSOD_t \Rightarrow (SessionRole(u,x,s,d,l) \wedge SessionRole(u,y,s,d',l) = False)$$

This allows the same user to activate two conflicting roles in the same session but not in the same location at any time. For example, in a teaching session in a classroom, a user cannot activate the the grader role and the student role at any time.

Definition 26 (Strong Spatial Form of DSoD)

Let x and y be two roles such that $x \neq y$ and $(x,y) \in DSOD_l$, that is, two distinct roles x and y are related by $DSOD_l$. If roles x and y are related through strong DSoD and if user u has activated role x in some session s, then u can never activate role y in session s during the same time in any location. This is formalized as follows:

$$(x,y) \in DSOD_l \Rightarrow (SessionRole(u,x,s,d,l) \wedge SessionRole(u,y,s,d,l') = False)$$

This allows the same user to activate two conflicting roles in the same session but not at the same time in any location. If a user has activated the *Graduate Teaching Assistant* role in his office, he cannot activate the *Lab Operator* role at the same time in any location.

Definition 27 (Strong Form of DSoD)

Let x and y be two roles such that $x \neq y$ and $(x,y) \in DSOD_s$. If roles x and y are related through strong DSoD and if user u has activated role x in some session s, then u can never activate role y in the same session. This is formalized as follows:

$$(x,y) \in DSOD_s \Rightarrow (SessionRole(u,x,s,d,l) \wedge SessionRole(u,y,s,d',l') = False)$$

Here a user can never activate the roles related through strong DSoD. For example, a user cannot be both a *code developer* and a *code tester* in the same session.

3.4 Impact of Time and Location on Delegation

Many situations require the temporary transfer of access rights to accomplish a given task. For example, in a pervasive computing application, a doctor may give certain privilege to a trained nurse, when he is taking a short break. In such situations, the doctor can give a subset of his permission to the nurse for a given period of time. There are a number of different types of delegation. The entity that transfers his privileges temporarily to another entity is often referred to as the delegator. The entity who receives the privilege is known as the delegatee. The delegator (delegatee) can be either a user or a role. Thus, we may have four types of delegations: *user to user* (U2U), *user to role* (U2R), *role to role* (R2R), and *role to user* (R2U). System administrators are responsible for overseeing delegation when the delegator is a role. Individual users administer delegation when the delegator is an user. When a user is the delegator, he can delegate a subset of permissions that he possesses by virtue of being assigned to different roles. When a role is the delegator, he can delegate either a set of permissions or he can delegate the entire role. We can therefore classify delegation on the basis of role delegation or permission delegation. We identify the following types of delegation.

Definition 28 (U2U Unrestricted Permission Delegation)

In this type of delegation, the delegatee can invoke the delegator's permissions at any time and at any place where the delegator could invoke those permissions. Let the predicate *DelegateU2U_P$_u$(u, v, Perm)* be true if the user u is allowed to delegate the permissions in the set

Perm to user *v* without any temporal or spatial constraints. This will allow *v* to invoke the permissions at any time or at any place.

$$\forall p \in Perm, DelegateU2U\_P_u(u,v,Perm) \wedge PermUserAcquire(u,o,p,d,l)$$
$$\Rightarrow PermUserAcquire(v,o,p,d,l)$$

For example, the illness of the company president caused him to delegate his email reading privilege to his secretary.

Definition 29 (U2U Time Restricted Permission Delegation)

In this type of delegation, the delegator places time restrictions on when the delegatee can invoke the permissions. However, no special restrictions are placed with respect to location– the delegatee can invoke the permission at any place that the delegator could do so. Let $DelegateU2U\_P_t(u,v,Perm,d')$ be the predicate that allows user *u* to delegate the permissions in the set *Perm* to user *v* for the duration d'.

$$\forall p \in Perm, DelegateU2U\_P_t(u,v,Perm,d') \wedge PermUserAcquire(u,o,p,d,l) \wedge (d' \subseteq d)$$
$$\Rightarrow PermUserAcquire(v,o,p,d',l)$$

For example, the professor can delegate his permission to proctor an exam to the teaching assistant while he is on travel.

Definition 30 (U2U Location Restricted Permission Delegation)

A delegator can place spatial restrictions on when the delegatee can invoke the permissions. However, the only temporal restriction is that the delegatee can invoke the permissions during the period when the original permission is valid. Let $DelegateU2U\_P_l(u,v,Perm,l')$ be the predicate that allows user *u* to delegate the permissions in the set *Perm* to user *v* in the location l'.

$$\forall p \in Perm, DelegateU2U\_P_l(u,v,Perm,l') \wedge PermUserAcquire(u,o,p,d,l) \wedge (l' \subseteq l)$$
$$\Rightarrow PermUserAcquire(v,o,p,d,l')$$

For example, the teaching assistant can delegate the permission regarding lab supervision to the lab operator only in the Computer Lab.

Definition 31 (U2U Time Location Restricted Permission Delegation)

In this case, the delegator imposes a limit on the time and the location where the delegatee can invoke the permission. Let $DelegateU2U\_P_{tl}(u, v, Perm, d', l')$ be the predicate that allows user u to delegate the permissions in the set $Perm$ to user v in the location l' for the duration d'.

$$\forall p \in Perm, DelegateU2U\_P_{tl}(u, v, Perm, t', l') \wedge PermUserAcquire(u, o, p, d, l)$$
$$\wedge (d' \subseteq d) \wedge (l' \subseteq l) \Rightarrow PermUserAcquire(v, o, p, d', l')$$

For example, a nurse can delegate his permission to oversee a patient while he is resting in his room to a relative.

Definition 32 (U2U Unrestricted Role Delegation)

The delegator delegates a role to the delegatee. The delegatee can activate the roles at any time and place where the delegator can activate those roles. Let $DelegateU2U\_R_u(u, v, r)$ be the predicate that allows user u to delegate his role r to user v.

$$DelegateU2U\_R_u(u, v, r) \wedge UserRoleActivate(u, r, d, l) \Rightarrow UserRoleActivate(v, r, d, l)$$

For example, a manager before relocating can delegate his roles to his successor in order to train him.

Definition 33 (U2U Time Restricted Role Delegation)

In this case, the delegator delegates a role to the delegatee but the role can be activated only for a more limited duration than the original role. Let $DelegateU2U\_R_t(u, v, r, d')$ be the predicate that allows user u to delegate his role r to user v for the duration d'.

$$DelegateU2U\_R_t(u, v, r, d') \wedge UserRoleActivate(u, r, d, l) \wedge (d' \subseteq RoleEnableDur(r))$$
$$\wedge (d' \subseteq d) \Rightarrow UserRoleActivate(v, r, d', l)$$

For example, a user can delegate his role as a teacher to a responsible student while he is in a conference.

Definition 34 (U2U Location Restricted Role Delegation)

In this case, the delegator delegates a role to the delegatee but the role can be activated in more limited locations than the original role. Let $DelegateU2U\_R_l(u, v, r, l')$ be the predicate that allows user u to delegate his role r to user v in the location l'.

$$Delegate\_R_l(u,v,r,l') \wedge UserRoleActivate(u,r,d,l) \wedge (l' \subseteq RoleEnableLoc(r)) \wedge (l' \subseteq l)$$
$$\Rightarrow UserRoleActivate(v,r,d,l')$$

For example, a student can delegate his lab supervision role to another student in a designated portion of the lab only.

Definition 35 (U2U Time Location Restricted Role Delegation)

The delegator delegates the role, but the delegatee can activate the role for a limited duration in limited places. Let $DelegateU2U\_R_{tl}(u,v,r,d',l')$ be the predicate that allows user u to delegate his role r to user v in location l' and time d'.

$$DelegateU2U\_R_l(u,v,r,d',l') \wedge UserRoleActivate(u,r,d,l) \wedge (l' \subseteq RoleEnableLoc(r)) \wedge$$
$$(d' \subseteq RoleEnableDur(r)) \wedge (d' \subseteq d) \wedge (l' \subseteq l) \Rightarrow UserRoleActivate(v,r,d',l')$$

For example, a student can delegate his lab supervision role to another student only in the lab when he leaves the lab for emergency reasons.

Definition 36 (R2R Unrestricted Permission Delegation)

All users assigned to the delegatee role can invoke the delegator role's permissions at any time and at any place where the user of this delegator role could invoke those permissions. Let $DelegateR2R\_P_u(r_1,r_2,Perm)$ be the predicate that allows role r_1 to delegate the permissions in the set $Perm$ to role r_2 without any temporal or spatial constraints. This will allow users in the role r_2 to invoke the permissions at any time or at any place.

$$\forall p \in Perm, DelegateR2R\_P_u(r_1,r_2,Perm) \wedge PermRoleAcquire(p,r_1,d,l) \wedge$$
$$(d \subseteq RoleEnableDur(r_2)) \wedge (l \subseteq RoleEnableLoc(r_2)) \Rightarrow PermRoleAcquire(p,r_2,d,l)$$

For example, the Smart Home owner role may delegate the permission to check the status of security sensors of the home to the police officer role, so all police officers can detect the intruder at any time at any place.

Definition 37 (R2R Time Restricted Permission Delegation)

The delegator role can place temporal restrictions on when the users of the delegatee role can invoke the permissions. No special restrictions are placed with respect to location i.e.

the delegatee role's users can invoke the permissions at any place that the delegator role's users could do so. Let $DelegateR2R\_P_t(r_1, r_2, Perm, d')$ be the predicate that allows role r_1 to delegate the permissions in the set $Perm$ to role r_2 for the duration d'.

$$\forall p \in Perm, DelegateR2R\_P_t(r_1, r_2, Perm, d') \wedge (d' \subseteq d) \wedge PermRoleAcquire(p, r_1, d, l) \wedge$$
$$(l' \subseteq l) \wedge (d' \subseteq RoleEnableDur(r_2)) \wedge (l \subseteq RoleEnableLoc(r_2))$$
$$\Rightarrow PermRoleAcquire(p, r_2, d', l)$$

For example, CS599 teacher role can grant the permission to access course materials to CS599 student role for the specific semester.

Definition 38 (R2R Location Restricted Permission Delegation)

The delegator role places spatial constraints on where the users of the delegatee role can invoke the permissions. No special temporal constraints are placed, that is, the delegatee role's users can invoke the permissions at any time that the delegator role's users could do so. Let $DelegateR2R\_P_l(r_1, r_2, Perm, l')$ be the predicate that allows role r_1 to delegate the permissions in the set $Perm$ to role r_2 in the location l'.

$$\forall p \in Perm, DelegateR2R\_P_l(r_1, r_2, Perm, l') \wedge PermRoleAcquire(p, r_1, d, l) \wedge (l' \subseteq l) \wedge$$
$$(d \subseteq RoleEnableDur(r_2)) \wedge (l' \subseteq RoleEnableLoc(r_2)) \Rightarrow PermRoleAcquire(p, r_2, d, l')$$

For example, the librarian role may grant the permission to checkout the book to the student role only at the self-checkout station.

Definition 39 (R2R Time Location Restricted Permission Delegation)

The delegator role imposes a limit on the time and the location where the delegatee role's users could invoke the permissions. Let $DelegateR2R\_P_{tl}(r_1, r_2, Perm, d', l')$ be the predicate that allows role r_1 to delegate the permissions in the set $Perm$ to role r_2 in the location l' for the duration d'.

$$\forall p \in Perm, DelegateR2R\_P_{tl}(r_1, r_2, Perm, d', l') \wedge PermRoleAcquire(p, r_1, d, l) \wedge$$
$$(d' \subseteq RoleEnableDur(r_2)) \wedge (l' \subseteq RoleEnableLoc(r_2)) \wedge (d' \subseteq d) \wedge (l' \subseteq l)$$
$$\Rightarrow PermRoleAcquire(p, r_2, d', l')$$

For example, the daytime doctor role may delegate the permission to get his location information to the nurse role only when he is in the hospital during the daytime.

Definition 40 (R2R Unrestricted Role Delegation)

All users assigned to the delegatee role can activate the delegator role at any time and at any place where the user of this delegator role could activate the role. Let $DelegateR2R\_R_u(r_1, r_2)$ be the predicate that allows role r_1 to be delegated to role r_2.

$$DelegateR2R\_R_u(r_1, r_2) \wedge UserRoleActivate(u, r_2, d, l) \wedge (d \subseteq RoleEnableDur(r_1)) \wedge$$
$$(l \subseteq RoleEnableLoc(r_1)) \Rightarrow UserRoleActivate(u, r_1, d, l)$$

For example, in the case of company reorganization, the manager role can be delegated to the manager successor role in order to train him.

Definition 41 (R2R Time Restricted Role Delegation)

The delegator places temporal constraints on when the users of the delegatee role can activate the delegator role. No special spatial constraints are placed i.e. the delegatee role's users can activate the delegator role at any place that the delegator role's users could do so. Let $DelegateR2R\_R_t(r_1, r_2, d')$ be the predicate that allows role r_1 to be delegated to role r_2 for the duration d'.

$$DelegateR2R\_R_t(r_1, r_2, d') \wedge UserRoleActivate(u, r_2, d', l) \wedge (d \subseteq RoleEnableDur(r_1)) \wedge$$
$$(l \subseteq RoleEnableLoc(r_1)) \wedge (d' \subseteq d) \Rightarrow UserRoleActivate(u, r_1, d', l)$$

For example, the permanent staff role can be delegated to the contract staff role during the contract period.

Definition 42 (R2R Location Restricted Role Delegation)

The delegator role can place spatial restrictions on where the users of the delegatee role can activate the delegator role. No special restrictions are placed with respect to time i.e. the delegatee role's users can activate the delegator role at any time that the delegator role's users could do so. Let $DelegateR2R\_R_l(r_1, r_2, l')$ be the predicate that allows role r_1 to be delegated to role r_2 in the location l'.

$$DelegateR2R\_R_l(r_1, r_2, l') \wedge UserRoleActivate(u, r_2, d, l') \wedge (d \subseteq RoleEnableDur(r_1)) \wedge$$
$$(l \subseteq RoleEnableLoc(r_1)) \wedge (l' \subseteq l) \Rightarrow UserRoleActivate(u, r_1, d, l')$$

For example, the researcher role can be delegated to the lab assistant role at the specific area of the lab.

Definition 43 (R2R Time Location Restricted Role Delegation)

In this case, the delegator role imposes a limit on the time and the location where the delegatee role's users could activate the role. Let $DelegateR2R\_R_{tl}(r_1, r_2, d', l')$ be the predicate that allows role r_1 to be delegated to role r_2 in the location l' for the duration d'.

$$DelegateR2R\_R_{tl}(r_1, r_2, d', l') \wedge UserRoleActivate(u, r_2, d', l') \wedge (d' \subseteq d) \wedge (l' \subseteq l) \wedge$$
$$(d \subseteq RoleEnableDur(r_1)) \wedge (l \subseteq RoleEnableLoc(r_1)) \wedge (d' \subseteq d) \wedge (l' \subseteq l)$$
$$\Rightarrow UserRoleActivate(u, r_1, d', l')$$

For example, the full-time researcher role can be delegated to the part-time researcher role only during the hiring period in the specific lab.

3.5 Chapter Summary

Unlike the tradional access control model, in pervasive computing applications, the access decisions cannot be based solely on the attributes of users and resources. In such applications, the system might want to grant access base on the context information such as location and time. In this chapter, we propose our Spatio-Temporal Role Based Access Control (STRBAC) model. The model determine the accessibility of the user based on the time and location constraints. The model also supports various types of role hierarchy and separation of duties constraints. In a highly dynamic application such as the ubiquitous computing, it is necessary for the model to support the transfer of privileges via delegation operations. Our model also supports different types of delegation.

As mentioned above, our STRBAC model consists of various features to support the various application requirements. These features of the model may interact in subtle ways resulting in inconsistencies and conflicts. Such inconsistencies and conflicts must be detected before

the model can be deployed for real-world application. Manual analysis of the access control specifications of complex, real-world applications is tedious and error-prone. To resolve this issue, an automated analysis approach is needed. In Chapter 4, we propose the usage of Alloy analyzer tool for doing an automated analysis on our STRBAC model.

Chapter 4

The ALLOY Specification of STRBAC Model

Our STRBAC model consists of various features to support the various application requirements. Such features of the model may interact in subtle ways resulting in inconsistencies and conflicts. The access control constraints of an application using our model must be analyzed to ensure that such problems, which, in turn, may cause security breaches, do not occur. Manual analysis of the access control specifications of complex, real-world applications is tedious and error-prone. In this chapter, we propose the use of Alloy analyzer tool for doing automated analysis. The analysis can be done at two different levels–*the model level* and *the application level*. Analysing the model at the model level ensures that in general, our STRBAC model does not contain any inconsistencies or conflicts. The application level analysis guarantees that the implementation of our model is fully protect the real-world application and does not cause any security breaches. In this chapter, first we briefly discuss the Alloy specification language we will use to assist our model analysis in Section 4.1. Next, we show how to use Alloy to analyze the STRBAC model in Section 4.2. And finally, we show how to analyze the security properties of the STRBAC–embedded real-world application in Section 4.3.

4.1 Alloy Lightweight Modeling System

ALLOY ([30], [31], [32], [92]), is a textual language developed at MIT by Daniel Jackson and his team. Unlike a programming language, an Alloy model is declarative: it can describe

the effect of a behavior without giving its mechanism. This allows very succinct and partial models to be constructed and analyzed. It is similar in spirit to the formal specification languages Z, VDM, Larch, B, OBJ, etc, but, unlike all of these, is amenable to fully automatic analysis in the style of a model checker.

Z was a major influence on Alloy. Very roughly, Alloy can be viewed as a subset of Z. Unlike Z, Alloy is first order, which makes it analyzable (but also less expressive). Alloys composition mechanisms are designed to have the flexibility of Z's schema calculus, but are based on different idioms: extension by addition of fields, similar to inheritance in an object-oriented language, and reuse of formulas by explicit parameterization, similar to functions in a functional programming language. Alloy is a pure ASCII notation and does not require special typesetting tools.

An Alloy model consists of a number of signature and relation declarations. A signature specifies entities used to model the system, and relation declarations specify the dependencies between such entities, allowing the designer to capture complex structures. Alloy is supported by a fully automated constraint solver, called Alloy Analyzer, that analyzes system properties by searching for model instances that violate assertions about them. Alloy Analyzer translates the model into a Boolean expression, and analyzes it using embedded SAT-solvers. The user specifies a scope to the tool, which is an integer number used to bound the domain of model elements. Bounding enables the tool to create finite Boolean formulas for evaluation by the SAT-solver. If Alloy Analyzer produces an instance that violates the assertion (a counterexample), we can infer that the specified property is not satisfied. However, for a chosen scope, if no counterexample emerges, it is possible that the property is violated in a larger scope. The larger the scope, the more confidence is warranted, but the longer the analysis will take [33]. Experience has shown that design flaws are often discovered in small scopes. This is known as "small scope hypothesis" [33]. Choosing the right scope, and the degree of confidence a given scope provides, depends on the problem and the security property being analyzed. Currently, there are no generic guidelines on how to choose the scope for a given problem. However, when developing security-critical systems, where a higher degree of confidence is required, the Alloy

Analyzer can be used as a first line of defense to discover flaws in the design of a system. If the analyzer does not produce a counterexample, other techniques such as Model Checking and Theorem Proving can be used to ensure the security property is not violated. Such techniques are more time consuming and require human intervention and expertise. Our approach can therefore save time and resources by using the Alloy Analyzer to rapidly discover a number of flaws that would otherwise require much more time and resources to uncover. For more details on Alloy and its comparison with other formal methods please refer to [31, 32, 33].

4.2 STRBAC Model in ALLOY

An Alloy model consists of *signature* declarations, *fields*, *facts* and *predicates*. Each signature consists of a set of *atoms* which are the basic entities in Alloy. Atoms are *indivisible* (they cannot be divided into smaller parts), *immutable* (their properties do not change) and *uninterpreted* (they do not have any inherent properties). Each field belongs to a signature and represents a relation between two or more signatures. A relation denotes a set of tuples of atoms. Facts are statements that define constraints on the elements of the model. Predicates are parameterized constraints that can be invoked from within facts or other predicates.

To represent the STRBAC model, we The basic types in the access control model, such as, *User*, *Time*, *Location*, *Role*, *Permission* and *Object* are represented as signatures. For instance, the declarations shown below define a set named *User* and a set named *Role* that represents the set of all users and the set of all roles in the system. Inside the *Role* signature body, we have four relations, namely, *RoleAllocLoc*, *RoleAllocDur*, *RoleEnableLoc*, and *RoleEnableDur* which relates *Role* to other signatures.

```
sig Time{}
sig Location{}
sig User{}
sig Role{
     RoleAllocLoc: Location,
     RoleAllocDur: Time,
```

```
     RoleEnableLoc: Location,
     RoleEnableDur: Time}
sig Permission{
     PermRoleLoc: Role->Location,
     PermObjLoc: Object->Location,
     PermDur: Time}
sig Object{}

sig TimeLoc{
     dur : Time,
     loc : Location}
```

The different relationships between the STRBAC components are also expressed as signatures. *RoleEnable* has a field called *member* that maps to a cartesian product of *Role*, *Time* and *Location*. *UserRoleAssignment* has a field called *member* that maps to a cartesian product of *User*, *Role*, *Time* and *Location*. *RolePermissionAssignment* has a field called *member* that maps to a cartesian product of *Role*, *Permission*, *Time* and *Location*. *UserLocation* has a field called *member* that maps to a cartesian product of *User*, *Time* and *Location*. *ObjLocation* has a field called *member* that maps to a cartesian product of *Object*, *Time* and *Location*. *UserRoleActivate* has a field called *member* that maps to a cartesian product of *User*, *Role*, *Time* and *Location*. *PermRoleAcquire* has a field called *member* that maps to a cartesian product of *Role*, *Permission*, *Time* and *Location*. *PermUserAcquire* has a field called *member* that maps to a cartesian product of *User*, *Object*, *Permission* and *TimeLoc*. Note that for *PermUserAcquire*, instead of declare it as a cartesian product of product of *User*, *Object*, *Permission*, *Time* and *Location*, we have to define a special signature called *TimeLoc* which consists of two fields called *dur* and *loc* representing Time and Location, respectively. The rational behind this indirect declaration is to overcome the limitation of Alloy, which limits the dimension of cartesian product to 4. And finally, *RoleHierarchy* has a field *RHmember* that represents a relationship between *Role* and *Role*. Note that we use the *abstract* signature to represent role hierarchy, and

the different types of role hierarchy are modeled as the subsignatures of *RoleHierarchy*. By this way, the analyzer will recognize that role hierarchy consists of only these different types of role hierachy, and nothing else.

```
one sig RoleEnable {member : Role-> Time ->Location}
one sig UserRoleAssignment{member : User -> Role ->Time ->Location}
one sig RolePermissionAssignment{member : Role-> Permission ->Time->Location}
one sig UserLocation{member : User->Time->Location}
one sig ObjLocation{member : Object->Time->Location}
one sig UserRoleActivate{member : User-> Role->Time->Location}
one sig PermRoleAcquire{member : Role->Permission->Time->Location}
one sig PermUserAcquire{member : User->Object->Permission->TimeLoc}

abstract sig RoleHierarchy{member : Role -> Role}
sig UPIH, TPIH, LPIH, TLPIH, UAH, TAH, LAH, TLAH extends RoleHierarchy{}
```

The various invariants in the STRBAC model are represented as facts in Alloy. For instance, the fact *URActivate* states that for user *u* to activate role *r* during the time interval *d* and location *l*, this user has to be assigned to role *r* in location *l* during time *d*. Moreover, the location of the user must be a subset of the locations where the role is enabled, and the time must be in the time interval when role *r* can be enabled. This is specified in Alloy as shown below. Other invariants are modeled in a similar manner.

```
fact URActivate{
all u: User, r: Role, d: Time, l: Location, uras: UserRoleAssignment,
urac: UserRoleActivate |
((u->r->d->l) in urac.member) => (((u->r->d->l) in uras.member) &&
(l in r.RoleEnableLoc) && (d in r.RoleEnableDur))
}
```

To represents the effects of STRBAC hierarchical structure, we use Alloy's *fact* feature. The fact *UPIHFact* represents the Unrestricted Permission Inheritance Hierarchy's property. The fact states that senior role `sr` can acquire all permission assigned to itself together with all permissions assigned to junior role `jr`. Note that to be more specific, we also explicitly state that the permission assigned to junior role have never been assigned to the senior role.

```
//Unrestricted Permission Inheritance Hierarchy
fact UPIHFact{
     all sr, jr: Role, p: Permission, d: Time, l: Location, upih: UPIH,
         rpa: RolePermissionAssignment, pra: PermRoleAcquire |
         ((sr->jr in upih.member) && (jr->p->d->l in pra.member) &&
         (sr->p !in (rpa.member).Location.Time)) =>
              (sr->p->sr.RoleEnableDur->sr.RoleEnableLoc) in pra.member}
```

The separation of duty constraints are modeled as predicates. Consider the Weak form of Static Separation of Duties User Role Assignment. This constraint says that a user *u* assigned to role *r*1 du ring time *d* and location *l* cannot be assigned to its conflicts role *r*2 at the same time and location. The other forms are modeled in a separate manner.

```
pred W_SSoD_URA(u: User, disj r1, r2: Role,
ura: UserRoleAssignment.member, d: Time, l: Location){
     ((u->r1->d->l) in ura) => ((u->r2->d->l) not in ura)}
```

The different types of delegation are also modeled as predicates. Consider the U2U Unrestricted Permission Delegation. This type of delegation says that a user *dtr* delegates his permission *p* to user *dte*. User *dte* can invoke the delegator's permission at any time and at any place where the delegator could invoke the permission. The other forms are modeled in a separate manner.

```
//U2U Unrestricted Permission Delegation
pred u2uUPD(disj dtr, dte: User, p: Permission){
```

```
    all o: Object, tl: TimeLoc, puacq: PermUserAcquire |
        (dtr->o->p->tl in puacq.member) =>
              (dte->o->p->tl in puacq.member)}
```

Finally, we need to verify whether any conflicts occur between the features of the model. We rely on the powerful analysis capability of the ALLOY analyzer for this purpose. We create an *assertion* that specifies the properties we want to check. After we create the assertion, we will let ALLOY analyzer validate the assertion by using *check* command. If our assertion is wrong in the specified scope, ALLOY analyzer will show the counterexample.

For instance, to check the interaction of the Weak form of SSOD User Role Assignment and the Unrestricted Permission Inheritance Hierarchy, we make the assertion shown below. The assertion does not hold as illustrated by the counterexample shown in Figure 4.1.

```
// WSSoD_URA violation in the present of UPIH Hierarchy
assert TestConflict1_1{
no u: User, disj x, y: Role, upih: UPIH,
      d: Time, l: Location, ura: UserRoleAssignment |
((x->y in ^(upih.member))  &&
            (u->x->d->l in ura.member)) =>
W_SSoD_URA[u, x, y, u->(x+y)->d->l, d, l]
}
check TestConflict1_1
```

The counterexample shows one possible scenario. In this case, it uses the following instances to show the violation.

1. $Role = \{Role0, Role1, Role2\}$
2. $UPIH0 = \{Role0 \rightarrow Role1, Role2 \rightarrow Role0, Role2 \rightarrow Role1\}$
3. $Time = d, Location = l$

4. $UserRoleAssignment = \{User \rightarrow Role0 \rightarrow Time \rightarrow Location, User \rightarrow Role1 \rightarrow Time \rightarrow Location, User \rightarrow Role2 \rightarrow Time \rightarrow Location\}$

Substituting x and y in W_SSoD_URA predicate with $Role2$ and $Role1$ respectively, we get the violation.

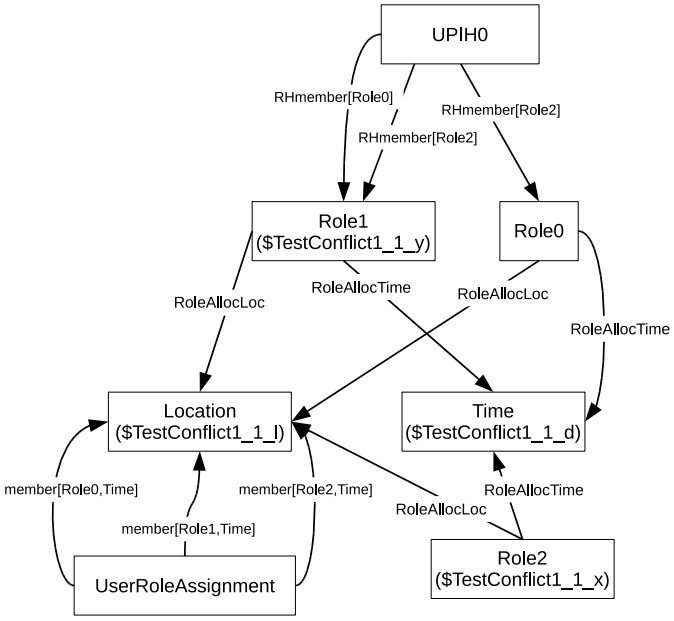


Figure 4.1: Counterexample for assertion TestConflict1_1

We checked the assertion on a HP-xw4400-Core2Duo-SATA with two Core2Duo 1.86Ghz CPU and 2 Gb memory running Linux 64. We used Version 4.1.2 Alloy Analyzer. The time taken to check this assertion was 25,916 ms.

Another example, to check the interaction of the Weak form of SSOD Permission Role Assignment and the R2R Unrestricted Permission Delegation, we make the assertion shown below. The assertion does not hold as illustrated by the counterexample shown in Figure 4.2.

```
// WSSoD_PRA violation in the present of R2R Unrestricted
// Permission Delegation
assert TestConflict14_1{
     all disj rdtr, rdte: Role, disj p, q: Permission, d: Time,
```

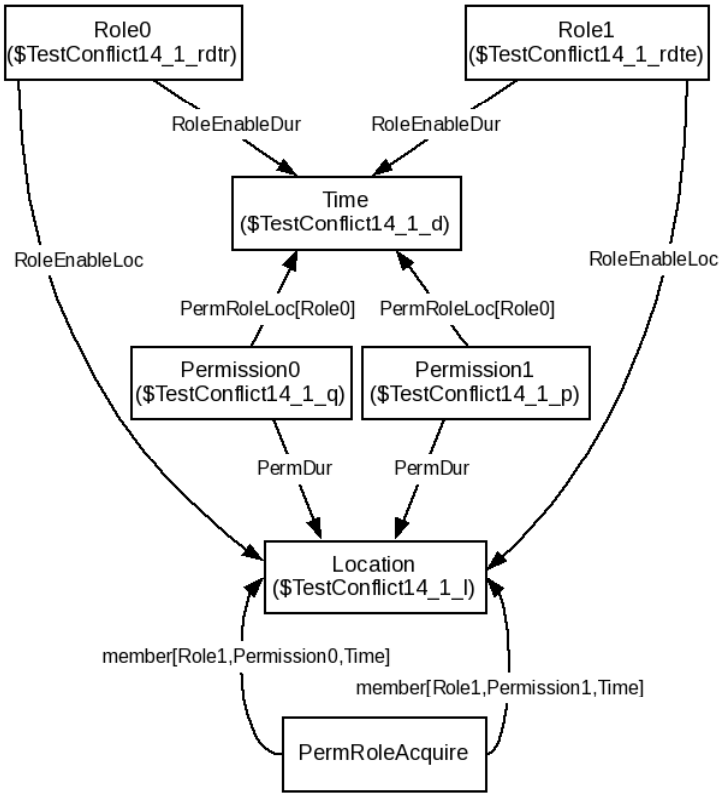


Figure 4.2: Counterexample for assertion TestConflict14_1

```
    l: Location |
         (r2rUPD[rdtr, rdte, p] && r2rUPD[rdtr, rdte, q]) =>
              W_SSoD_PRA[rdte, p, q, d, l]
}
check TestConflict14_1
```

The counterexample shows one possible scenario. In this case, it uses the following instances to show the violation.

1. $Role = \{Role0, Role1\}$
2. $Permission = \{Permission0, Permission1\}$
3. $Time = d$, $Location = l$

4. $PermRoleAcquire = \{Role1 \rightarrow Permission0 \rightarrow Time \rightarrow Location, Role1 \rightarrow Permission1 \rightarrow Time \rightarrow Location\}$

Substituting *rdtr*, *rdte*, *p*, *q*, *d*, and *l* in r2rUPD and W_SSoD_PRA predicates with *Role*0, *Role*1, *Permission*0, *Permission*1, *d* and *l* respectively, we get the violation. By using the same setup, the time taken to check this assertion was 20,572 ms.

With the advocate of the STRBAC model in Alloy shown in Appendix B, we could reveal the following types of conflict:

1. Conflict of Unrestricted, Time Restricted, Location Restricted, and Time Location Restricted Permission Inheritance Hierarchy with the Weak Form of Static Separation of Duties-User Role Assignment (detected by assertion *TestConflict1_1*, *TestConflict1_2*, *TestConflict1_3*, and *TestConflict1_4*, respectively)

2. Conflict of Unrestricted, Time Restricted, Location Restricted, and Time Location Restricted Permission Inheritance Hierarchy with the Strong Temporal Form of Static Separation of Duties-User Role Assignment (detected by assertion *TestConflict2_1*, *TestConflict2_2*, *TestConflict2_3*, and *TestConflict2_4*, respectively)

3. Conflict of Unrestricted, Time Restricted, Location Restricted, and Time Location Restricted Permission Inheritance Hierarchy with the Strong Spatial Form of Static Separation of Duties-User Role Assignment (detected by assertion *TestConflict3_1*, *TestConflict3_2*, *TestConflict3_3*, and *TestConflict3_4*, respectively)

4. Conflict of each type of Permission Inheritance Hierarchy with the Strong Form of Static Separation of Duties-User Role Assignment (detected by assertion *TestConflict4*)

5. Conflict of Unrestricted, Time Restricted, Location Restricted, and Time Location Restricted Permission Inheritance Hierarchy with the Weak Form of Static Separation of Duties-Permission Role Assignment (detected by assertion *TestConflict5_1*, *TestConflict5_2*, *TestConflict5_3*, and *TestConflict5_4*, respectively)

6. Conflict of Unrestricted, Time Restricted, Location Restricted, and Time Location Restricted Permission Inheritance Hierarchy with the Strong Temporal Form of Static Separation of Duties-Permission Role Assignment (detected by assertion *TestConflict6_1*, *TestConflict6_2*, *TestConflict6_3*, and *TestConflict6_4*, respectively)

7. Conflict of Unrestricted, Time Restricted, Location Restricted, and Time Location Restricted Permission Inheritance Hierarchy with the Strong Spatial Form of Static Separation of Duties-Permission Role Assignment (detected by assertion *TestConflict7_1*, *TestConflict7_2*, *TestConflict7_3*, and *TestConflict7_4*, respectively)

8. Conflict of each type of Permission Inheritance Hierarchy with the Strong Form of Static Separation of Duties-Permission Role Assignment (detected by assertion *TestConflict8*)

9. Conflict of Unrestricted, Time Restricted, Location Restricted, and Time Location Restricted Activation Hierarchy with the Weak Form of Dynamic Separation of Duties (detected by assertion *TestConflict9_1*, *TestConflict9_2*, *TestConflict9_3*, and *TestConflict9_4*, respectively)

10. Conflict of Unrestricted, Time Restricted, Location Restricted, and Time Location Restricted Activation Hierarchy with the Strong Temporal Form of Dynamic Separation of Duties (detected by assertion *TestConflict10_1*, *TestConflict10_2, TestConflict10_3,* and *TestConflict10_4*, respectively)

11. Conflict of Unrestricted, Time Restricted, Location Restricted, and Time Location Restricted Activation Hierarchy with the Strong Spatial Form of Static Separation of Duties (detected by assertion *TestConflict11_1*, *TestConflict11_2*, *TestConflict11_3*, and *TestConflict11_4*, respectively)

12. Conflict of each type of Activation Hierarchy with the Strong Form of Dynamic Separation of Duties (detected by assertion *TestConflict12*)

13. Conflict occurs during the permission role assignment operation i.e. the spatio-temporal

constraints of the permission assigned to the role are conflicted with the spatio-temporal constraints of the role (detected by assertion *TestConflict13*)

14. Conflict of R2R Unrestricted, R2R Time Restricted, R2R Location Restricted, and R2R Time Location Restricted Permission Delegation with the Weak Form of Static Separation of Duties-Permission Role Assignment (detected by assertion *TestConflict14_1*, *TestConflict14_2*, *TestConflict14_3*, and *TestConflict14_4*, respectively)

15. Conflict of R2R Unrestricted, R2R Time Restricted, R2R Location Restricted, and R2R Time Location Restricted Permission Delegation with the Strong Temporal Form of Static Separation of Duties-Permission Role Assignment (detected by assertion *TestConflict15_1*, *TestConflict15_2*, *TestConflict15_3*, and *TestConflict15_4*, respectively)

16. Conflict of R2R Unrestricted, R2R Time Restricted, R2R Location Restricted, and R2R Time Location Restricted Permission Delegation with the Strong Spatial Form of Static Separation of Duties-Permission Role Assignment (detected by assertion *TestConflict16_1*, *TestConflict16_2*, *TestConflict16_3*, and *TestConflict16_4*, respectively)

17. Conflict of R2R Unrestricted, R2R Time Restricted, R2R Location Restricted, and R2R Time Location Restricted Permission Delegation with the Strong Form of Static Separation of Duties-Permission Role Assignment (detected by assertion *TestConflict17_1*, *TestConflict17_2*, *TestConflict17_3*, and *TestConflict17_4*, respectively)

18. Conflict of R2R Unrestricted, R2R Time Restricted, R2R Location Restricted, and R2R Time Location Restricted Role Delegation with the Weak Form of Dynamic Separation of Duties (detected by assertion *TestConflict18_1*, *TestConflict18_2*, *TestConflict18_3*, and *TestConflict18_4*, respectively)

19. Conflict of R2R Unrestricted, R2R Time Restricted, R2R Location Restricted, and R2R Time Location Restricted Role Delegation with the Strong Temporal Form of Dynamic Separation of Duties (detected by assertion *TestConflict19_1*, *TestConflict19_2*, *TestConflict19_3*, and *TestConflict19_4*, respectively)

20. Conflict of R2R Unrestricted, R2R Time Restricted, R2R Location Restricted, and R2R Time Location Restricted Role Delegation with the Strong Spatial Form of Dynamic Separation of Duties (detected by assertion *TestConflict20_1*, *TestConflict20_2*, *TestConflict20_3*, and *TestConflict20_4*, respectively)

21. Conflict of R2R Unrestricted, R2R Time Restricted, R2R Location Restricted, and R2R Time Location Restricted Role Delegation with the Strong Form of Dynamic Separation of Duties (detected by assertion *TestConflict21_1*, *TestConflict21_2*, *TestConflict21_3*, and *TestConflict21_4*, respectively)

4.3 Using Alloy to Analyze the STRBAC-Embedded Application

The analysis approach we propose in Section 4.2 is performed at the model level, which means the model in general is free from the inconsistencies and conflicts listed in Section 4.2. In this section, we propose a methodology that describes how we can get assurance that an application is adequately protected. Since the applications are generally specified in UML, we use UML to specify our application and access control constraints as well. UML can be used in conjunction with OCL which is based on formal logic; this allows us to formally specify the constraints in our model. However, in order to get assurance that our application is adequately protected, we need to analyze our application together with the access control constraints. Manual analysis is tedious and error-prone, so we need to automate the verification process. Existing tools for automated analysis of UML models, such as USE and OCLE, cannot verify behavioral properties and so are inadequate. We propose an approach that will transform UML models with OCL constraints into an Alloy specification.

4.3.1 Model Transformation from UML to Alloy

There are clear similarities between Alloy and UML languages such as class diagrams and OCL. From a semantic point of view both Alloy and UML can be interpreted by sets of tuples [34, 69]. Alloy is based on first-order logic and is well suited for expressing constraints

on object-oriented models. Similarly, OCL has extensive constructs for expressing constraints as first order logic formulas. Considering such similarities, model transformation from UML class diagrams and OCL to Alloy seems straightforward. However, UML and Alloy have fundamental differences, which are deeply rooted in their underlying design decisions. For example, Alloy makes no distinction between sets, scalars and relations, while the UML makes a clear distinction between the three. Other examples include that UML supports a number of primitive types, whereas Alloy only supports integers. UML also supports aggregation and composition, but there is no counterpart in Alloy. All of this makes the transformation from UML to Alloy challenging.

Figure 4.3 depicts an outline of our approach. Using the Extended Backus-Naur Form (EBNF) representation of the Alloy grammar [34], we shall first generate a Meta Object Facility (MOF) compliant [55] metamodel for Alloy. We then select a subset of the class diagrams [57] and OCL [56] metamodels. To conduct the model transformation, a set of transformation rules has been defined. The rules map elements of the metamodels of class diagram and OCL into the elements of the metamodel of Alloy. The rules have been implemented into a prototype tool called UML2Alloy. If a UML class diagram, which conforms to the subset of UML we support, is provided as input to UML2Alloy, it automatically generates an Alloy model. For lack of space, we do not show how the EBNF representation of Alloy's grammar is transformed into a MOF compliant metamodel but refer the interested reader to [3].

4.3.2 Mapping Class diagram and OCL to Alloy

The transformation rules map elements of the UML class diagram and OCL metamodels to the Alloy metamodel. Due to space limitations the UML and OCL metamodels are not presented here, but can be found in the respective specification documents [57, p.29], [56].

Table 4.1 presents a table which provides an informal mapping between the most important elements of the UML and OCL metamodels and Alloy. More specifically a UML *Class* is translated to an Alloy signature declaration (*ExtendsSigDecl*), which defines a *SigId* with the same name. If the class is not a specialization the Alloy signature is not related to any *SigRef*.

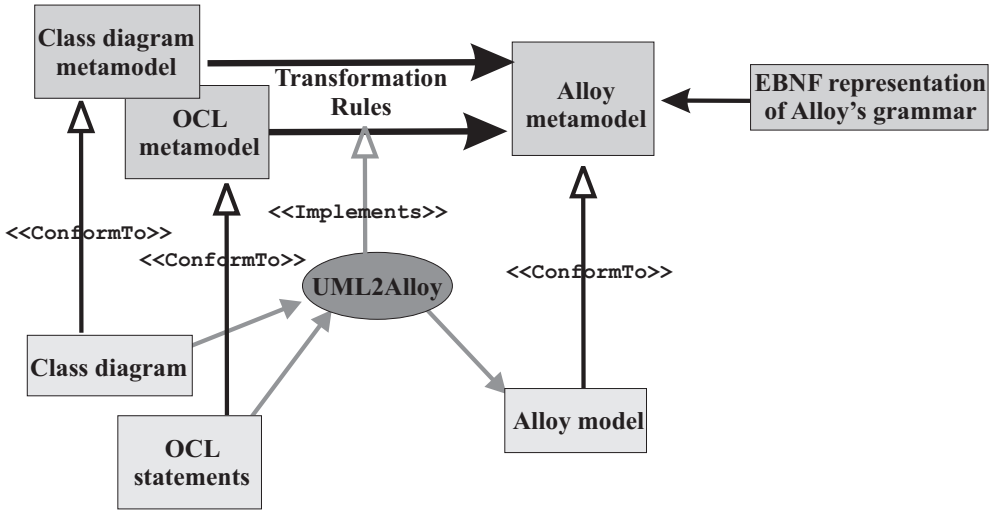


Figure 4.3: Outline of the transformation method.

UML+OCL metamodel element	Alloy metamodel element
Class	ExtendsSigDecl
Property	DeclExp
Operation	Predicate
Parameter	Decl
Enumeration	ExntedsSigDecl
EnumerationLiteral	ExtendsSigDecl
Constraint	Expression

Table 4.1: Informal mapping between UML and Alloy metamodel elements

Otherwise it might be related to a *SigRef*, which references the signature it might extend.

A *Property* is translated to a declaration expression (*declExp*), which is used to define a field in an Alloy model. An *Operation* is transformed to a *Predicate* and the *Parameters* of the operation are transformed to declarations (*Decl*). An Enumeration [57, p.63] is transformed to a signature declaration *SigDecl*, which declares an abstract signature. An *EnumerationLiteral* is transformed to a sub signature. A more complete transformation rules from UML to Alloy and their implementation are explained in our previous work [3].

4.3.3 UML2Alloy

UML2Alloy is the tool developed to transform the UML enriched with OCL constraints to Alloy. UML2Alloy makes use of Model Driven Architecture (MDA) [45] techniques to

perform this transformation. Both OCL and Alloy are based on first-order logic. They are therefore quite similar, and the translation from OCL to Alloy is straightforward when dealing with first-order logic statements. For example, the forAll OCL construct is translated to all in Alloy and the exists OCL construct to some in Alloy. Table 4.2 [16] shows subset of the transformation rules which UML2Alloy uses to transform UML and its associate OCL to Alloy.

Table 4.2: A Subset Of UML2Alloy Transformation Rules

UML	Alloy
Classes	Signature Declarations
Attributes	Relations of the Signature
Data Types	Signature Declarations
OCL Expressions	Formula Expressions
If Expressions	If Formulas
Operations that return a type	Functions
Operations that return void type	Predicates
Operation Parameters	Parameters of Predicates or Functions
Associations	Relations of a Signature

4.3.4 Example Scenario: Dengue Decision Support System

We illustrate our approach using a real-world Dengue Decision Support (DDS) system. The DDS helps state-level public health officials respond to local outbreaks of dengue. Response consists of vector control and vector surveillance, namely spraying for mosquitoes (control) and investigating locations where they might be breeding and living (surveillance) in areas where the level of confirmed dengue cases has increased above a prescribed threshold. Public health officials are organized in jurisdictions, based on population, and multiple jurisdictions are included in a single state. When the threshold is reached, officials at both levels respond. The jurisdiction officer activates vector control and surveillance teams that are local to the jurisdiction, with instructions regarding the specific control and surveillance protocols to follow and the locations where they are to be performed. The state officer releases materials for control to the team, and the local team then performs the controls and surveillance ordered. The jurisdiction and state vector control officials are often located in different buildings, although the vector control team is co-located with the jurisdiction officer. All control materials are

Table 4.3: DDS Tasks List

	Task		Task
1	Read Premise	10	Read VControl
2	Change Premise	11	Change VControl
3	Read Case	12	Read Work Record
4	Change Case	13	Change Work Record
5	Read Patient	14	Read VC Materials
6	Change Patient	15	Change VC Materials
7	Read Patient Names	16	Signal VC Need for DV
8	Read Schedule Work	17	Signal VC Need for DHF
9	Change Schedule Work		

located in warehouses elsewhere, and for coordination reasons are controlled by the state officer. Information about specific cases of dengue is retained in what is called an epidemiological study. This data includes information about the patient, the location where the patient lives (the premise), the case, and control and surveillance actions performed at the premise. The patient and case data are considered private information, and are only available to epidemiologists at the jurisdiction and state levels. The vector control team receives premise information along with orders for control and surveillance. However, the team also needs to have names associated with the premises in order to validate the location. The team therefore needs access to some of the patient data for a fixed period of time, in order to perform control and surveillance duties.

4.3.4.1 DDS Security Policies

Entities

DDS system consists of the following roles: *State Epidemiologist*, *Jurisdiction Epidemiologist*, *Clinic Epidemiologist*, *Clinician*, *State Vector Control*, *Jurisdiction Vector Control*, and *Local Jurisdiction VC Team*. Tasks user can perform are listed in Table 6.1. Each role can perform their own set of tasks in the designated location and time summarized in Table 6.2.

Role Hierarchy

Some roles in the DDS are related using unrestricted permission inheritance hierarchy. Using

Table 4.4: DDS Role Constraints

Role	Tasks	Location Constraint	Time Constraint
State Epi	16	A–State Office	a–Regular Hours
Juris Epi	1, 3	B–Juris Office	a–Regular Hours
	17	B–Juris Office	b–Any Time
Clinic Epi	17	C–Clinic	b–Any Time
Clinician	1, 2, 3, 4, 5, 6	C–Clinic	a–Regular Hours
State VC	11, 15	A–State Office	a–Regular Hours
Juris VC	1, 8, 9, 10, 12, 14	B–Juris Office	a–Regular Hours
Local VC Team	7	B–Juris Office, E–Emergency Location	c–24 Hours Window after signal to begin work received
	1, 9, 13	B–Juris Office, D–Field	a–Regular Hours

the STRBAC model, these relationships can be define as follow: *State Epi* $\geq$ *Juris Epi*, *Clinic Epi* $\geq$ *Clinician*, and *State VC* $\geq$ *Juris VC*.

Separation of Duty

There are two separation of duty constraints in DDS system. Both are the strong spatial form of static separation of duty.

1. User should not have permission to change VC protocols at the same time as he has permission to change VC materials.

2. User should not have permission to signal DV at the same time as signal DHF.

These can be represented in STRBAC as follow: $(11,15) \in SSOD\_PRA_l$ and $(16,17) \in SSOD\_PRA_l$.

4.3.4.2 DDS Model Analysis

The first step in formal security analysis is to abstract and transform the STRBAC model in the context of DDS into a UML class diagram and accompanying OCL. The class diagram depicts the entities that take part in the model, and defines their attributes related in the access control operations, such as the time and location attribute. OCL statements specify the invariants of the model such as the tasks assigned to role and security constraints that all entities in the model must satisfy. In the next step, we use UML2Alloy [15, 16] to automatically transform the class diagram and OCL statements into an Alloy model, which we subsequently analyze using Alloy Analyzer.

Stage 1: Model Abstraction

The first step of the abstraction is to simplify the original model by removing non-essential elements so that the translation to Alloy produces a model that only contains items necessary to reason about its security properties. For example, we remove the attributes which is not related with the security such as, *gender*, *birthdate*, *ssid* from the *Person* entity since these attributes are not related with the access control model. The resulting UML class diagram is shown in Figure 4.4.

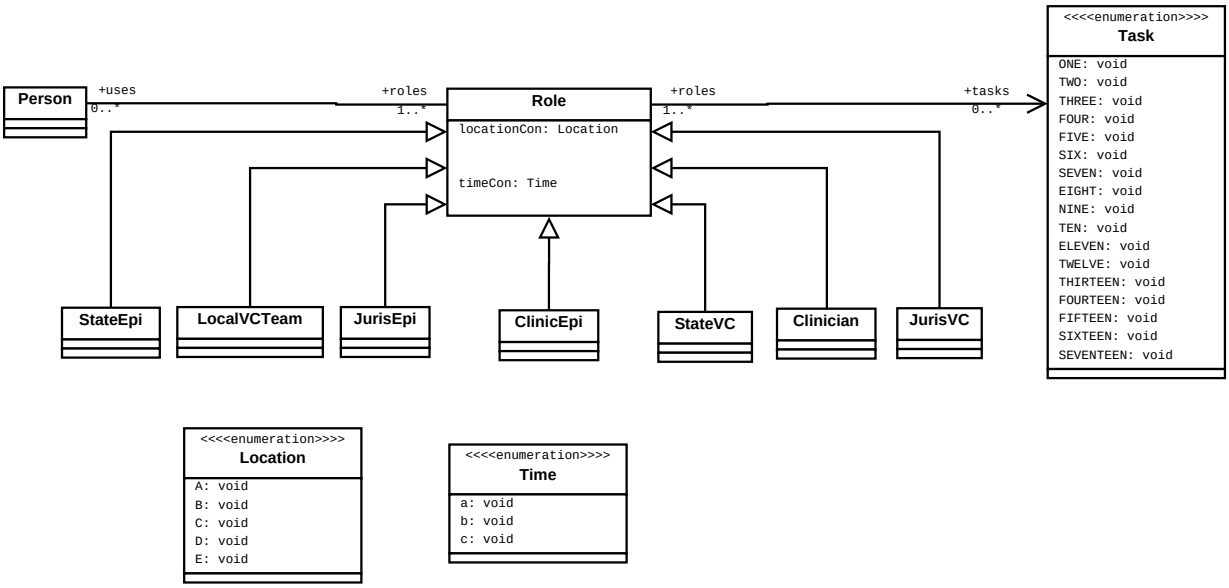


Figure 4.4: UML Model for the DDS's STRBAC

The permission role assignments are expressed as OCL constraints. The following OCL depicts the constraints for the permission role assignment for *Juris Epi* role.

```
context JurisEpi
inv jurisEpiCon : (self.tasks = (Task :: ONE ->
including (Task :: THREE)) and
self.location = Location :: B and
self.timeCon = Time :: a) or
(self.tasks = (Task :: SEVENTEEN -> including
(Task :: SEVENTEEN)) and
```

```
self.location = Location :: B and
self.timeCon = Time :: b )
```

The effect of permission inheritance hierarchy can also be expressed as OCL. The following OCL depicts the constraints for the permission role assignment for *State Epi* role.

```
context StateEpi
inv stateEpiCon : (self.tasks = (Task :: SIXTEEN ->
including (Task::SIXTEEN)) and
self.location = Location :: A and
self.timeCon = Time :: a) or
(self.tasks = (Task :: ONE -> including
(Task :: THREE)) and self.location = Location :: B and
self.timeCon = Time :: a) or
(self.tasks = (Task :: SEVENTEEN -> including
(Task :: SEVENTEEN)) and self.location = Location :: B
and self.timeCon = Time :: b)
```

Note that all permissions assigned to *Juris Epi*, which is the junior role of *State Epi* role are appended to the set of permissions assigned to *State Epi* role.

The separation of duty can also be modeled using OCL constraint. For instance, the constraint said that user should not have permission to change VC protocols at the same time as he has permission to change VC materials can be modeled as follow:

```
context Person
inv no_eleven_fifteen : self.roles ->
forAll (r1 , r2 : Role |
(r1.tasks -> includes (Task :: ELEVEN) implies
(r2.tasks -> excludes (Task :: FIFTEEN))) and
(r1.tasks -> includes (Task :: FIFTEEN) implies
r2.tasks -> excludes (Task :: ELEVEN)))
```

The complete list of the OCL constraints can be refered to Appendix B.1.

Stage 2: Model Transformation

The UML2Alloy tool is used to create an Alloy model from the class diagram and associated OCL specification.

When we apply UML2Alloy to the UML class diagram and its OCL specification, the class diagram will be transformed to the following *signatures* in Alloy corresponding to each class shown in Figure 4.4.

```
abstract  sig Role{
location:one Location,
timeCon:one Time,
tasks:some Task,
uses:set Person}

one sig StateEpi extends Role{}
one sig JurisEpi extends Role{}
one sig ClinicEpi extends Role{}
one sig Clinician extends Role{}
one sig StateVC extends Role{}
one sig JurisVC extends Role{}
one sig LocalVCTeam extends Role{}

some sig Person{roles:some Role}

abstract  sig Location{}
one sig A extends Location{}
one sig B extends Location{}
one sig C extends Location{}
```

```
one sig D extends Location{}
one sig E extends Location{}

sig Time{}
sig a in Time{}
sig b in Time{}
sig c in Time{}

abstract  sig Task{}
one sig ONE extends Task{}
one sig TWO extends Task{}
one sig THREE extends Task{}
one sig FOUR extends Task{}
one sig FIVE extends Task{}
one sig SIX extends Task{}
one sig SEVEN extends Task{}
one sig EIGHT extends Task{}
one sig NINE extends Task{}
one sig TEN extends Task{}
one sig ELEVEN extends Task{}
one sig TWELVE extends Task{}
one sig THIRTEEN extends Task{}
one sig FOURTEEN extends Task{}
one sig FIFTEEN extends Task{}
one sig SIXTEEN extends Task{}
one sig SEVENTEEN extends Task{}
```

The OCL constraint for the permission role assignment will be transformed to *fact* and *predicate* in Alloy. For example, the OCL constraint for the permission role assignment of the

Juris Epi role will be transformed to the following Alloy code.

```
fact JurisEpi_jurisEpiCon_fact{
all self: JurisEpi | JurisEpi_jurisEpiCon[self]}

pred JurisEpi_jurisEpiCon[self: JurisEpi]{
((self.tasks = ONE+THREE) && (self.location = B) &&
(self.timeCon = a)) || ((self.tasks = SEVENTEEN) &&
(self.location = B) && (self.timeCon in Time))}
```

The effect of role hierarchy represented in the OCL constraint will also be transformed to *fact* and *predicate* in Alloy. For example, the OCL constraint for the set of permissions that assigned to the *State Epi* role through the role hierarchy will be transformed to the following Alloy code.

```
fact StateEpi_stateEpiCon_fact{
all self: StateEpi | StateEpi_stateEpiCon[self]}

pred StateEpi_stateEpiCon[self: StateEpi]{
(self.tasks = SIXTEEN + ONE + THREE + SEVENTEEN) &&
(self.location = A) && (self.timeCon = a)}
```

The OCL constraint for the separation of duty constraint will be transformed to *predicate* in Alloy. For instance, the OCL constraint for the constraint said that user should not have permission to change VC protocols at the same time as he has permission to change VC materials will be transformed to the following Alloy code.

```
pred Person_no_eleven_fifteen[self: Person]{
all r1, r2: self.roles |
((ELEVEN in r1.tasks) => (FIFTEEN !in r2.tasks)) &&
((FIFTEEN in r1.tasks) => (ELEVEN !in r2.tasks))}
```

The complete Alloy code generated by UML2Alloy is shown in Appendix B.2.

Stage 3: Model Analysis

Alloy assertions must be formulated prior to analysis by Alloy Analyzer. Assertions are statements that capture properties we wish to verify. Alloy Analyzer automatically checks such assertions and if they fail it produces a counterexample. We have checked several assertions regarding the security properties of the example system. For example, it is crucial to ensure that no user can change VC protocols (task 11) at the same time as he has permission to change VC materials (task 15). To verify this, we create the following assertion:

```
assert NoConflictPermsSTVCAssigned{
all r: Person.roles, d: Time, l: Location|
((ELEVEN in r.tasks) && (d in r.timeCon) &&
(l in r.location)) =>
((FIFTEEN !in r.tasks) && (d in r.timeCon) &&
(l in r.location))}
```

We chose a value of 8 for the scope of analysis, and the assertion was checked for this scope. A scope of 8 means that the Alloy Analyzer will attempt to find an instance that violates the assertion, using up to 8 instances for each of the entities defined in the class diagram of Figure 4.4. The assertion produced no counterexample, meaning that it is valid for the given scope.

Next, we will check whether the SoD for role permission assignment is maintained. To do this, we create the following assertion:

```
assert NoConflictPermsSTVC{
all r: StateVC, d: Time, l: Location|
((ELEVEN in r.tasks) && (d in r.timeCon) &&
(l in r.location)) =>
((FIFTEEN !in r.tasks) && (d in r.timeCon) &&
(l in r.location))}
```

We chose a value of 8 for the scope of this analysis as well. However, this time the analyzer showed the counterexample, which means these conflicting permissions can be assigned to the same role. The counterexample is shown in Figure 4.5.

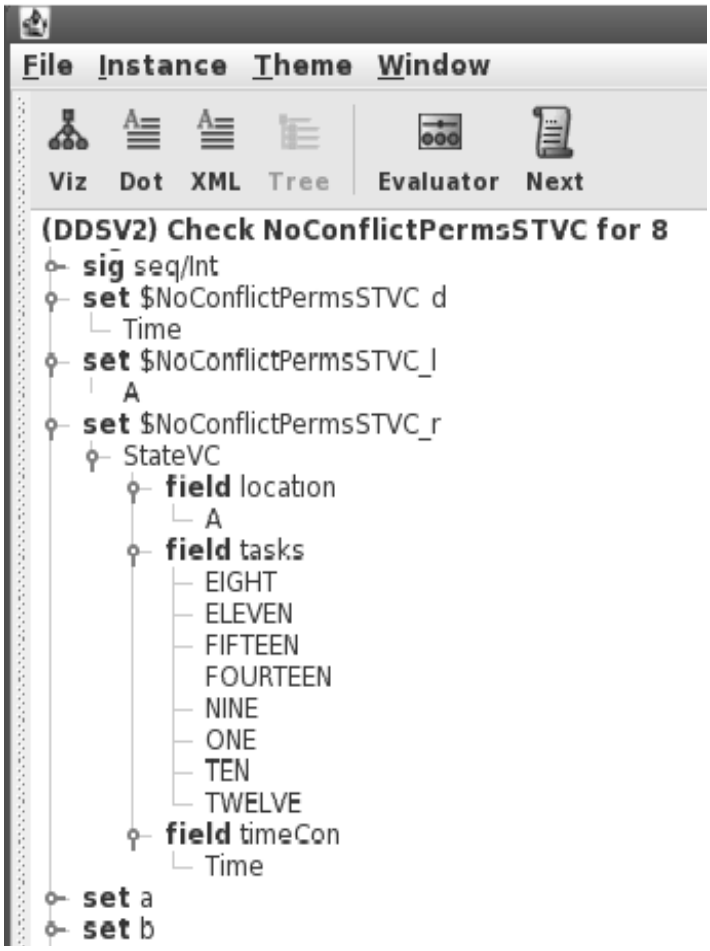


Figure 4.5: Counterexample for Assertion NoConflictPermsSTVC

4.4 Chapter Summary

In this chapter, we demonstrate how we can perform automated analysis on the STRBAC model. We propose the usage of Alloy analyzer tool for doing automated analysis. Our analysis can be done at two different levels–the model level and the application level. Analysing the model at the model level ensures that the different features of our STRBAC model does not contain any inconsistencies or conflicts. The application level analysis guarantees that the

implementation of our model is fully protect the real-world application and does not cause any security breaches.

Our STRBAC model is efficient in the aspect that it can represent the different kind of constraints and relationships as shown in Chapter 3. The transformation of the model to do the automated verification is feasible as demonstrated in Section 4.3. However, semantically, the model is extremely complicated. In Chapter 5, we define a model with well-defined semantics expressed in graph-theoretic notation. We name this new model as the Spatio-Temporal Aware Role-Based Access Control with Delegation (STARBACD) Model.

Chapter 5

A Spatio-Temporal Aware Role-Based Access Control with Delegation (STARBACD) Model

Chen and Crampton develop the graph based representation for the spatio-temporal RBAC in [19]. All RBAC components are represented by vertices while the assignment and hierarchical relationships are represented by the edges of the directed graph. The model can be categorized into three types i.e. standard, strong, and weak model. For the standard model, component v_1 is said to be authorized to component v_n if all vertices along the authorization path satisfy the spatio-temporal constraints. For the strong model, component v_1 is said to be authorized to component v_n if all vertices together with the edges along the authorization path satisfy the spatio-temporal constraints. And in the weak model, component v_1 is said to be authorized to component v_n if both vertices satisfy the spatio-temporal constraints. Although the authors developed a strong and clear semantics of the model, the model still lack of some useful functionalities. Firstly, the model does not consider the spatio-temporal constraint which may be applied to the object. As we mentioned earlier, this point is critical for the security in the pervasive computing environment. Secondly, the model does not implement the separation of duties constraints which is widely known as a useful function of the RBAC model. Thirdly, the delegation operation which is widely used nowadays is not supported in the model. We borrow the idea of graph representation from both [19] and [54], and then improve it to rectify their shortcomings.

In this chapter, we propose the second model based on graph representation, which is well-formed semantics. Based on this model, we will propose the model analysis algorithm in Section 5.4.

5.1 Spatio-Temporal Model

Our model extends the one proposed by Chen and Crampton [19] in the following ways. First, we believe that a spatio-temporal access control model must also support access control for moving objects, that is, objects whose physical location changes with time. Thus, access control should not only depend on the spatio-temporal coordinate-ordinate of the user but also of the object. Second, separation of duties must also be supported by access control models. Third, the model must also provide support for delegation which is an absolute necessity for access control in pervasive computing applications. In this and the next two sections, we show how the model by Chen and Crampton can be extended to support each of these features.

We first propose the model where access is dependent on the location of the user as well as that of the object. We extend the graph-theoretic notation of Chen and Crampton [19] in the following manner. In our work, the set of vertices $V = U \cup R \cup P \cup O$ correspond to the RBAC entities: Users (U), Roles (R), Permissions (P), and Objects (O). Like Chen and Crampton [19], our model assumes the existence of the following relationships of RBAC that constitute the set of edges $E = UA \cup PA \cup PO \cup RH_a \cup RH_u$ where

- User-Role Assignment (UA) = $U \times R$
- Permission-Role Assignment (PA) = $R \times P$
- Permission-Object Assignment (PO) = $P \times O$
- Role Hierarchy (RH) = $R \times R \times \{a, u\}$, which can be categorized to:
  - the activation hierarchy (RH_a) = $\{(r, r') : (r, r', a) \in RH\}$, and
  - the permission usage hierarchy (RH_u) = $\{(r, r') : (r, r', u) \in RH\}$

We define the notion of activation path, usage path and access path in a manner similar to that proposed by Chen and Crampton. An *activation path* (or *act-path*) between v_1 and v_n is

defined to be a sequence of vertices $v_1, \ldots, v_n$ such that $(v_1, v_2) \in UA$ and $(v_{i-1}, v_i) \in RH_a$ for $i = 3, \ldots, n$. A *usage path* (or *u-path*) between v_1 and v_n is defined to be a sequence of vertices $v_1, \ldots, v_n$ such that $(v_i, v_{i+1}) \in RH_u$ for $i = 1, \ldots, n-2$, and $(v_{n-1}, v_n) \in PA$. An *access path* (or *acs-path*) between v_1 and v_n is defined to be a sequence of vertices $v_1, \ldots, v_n$, such that (v_1, v_i) is an act-path, (v_i, v_{n-1}) is an u-path, and $(v_{n-1}, v_n) \in PO$. Note that, our access path definition is similar to the au-path definition of Chen and Crampton [19], except that we include the path to the object as part of our definition. This is important especially if we are dealing with objects whose location varies with time. Following Chen and Crampton's work [19], we assume the existence of a spatio-temporal domain $\mathcal{D}$. We also propose three models, namely, the standard model, the strong model, and the weak model. The models differ with respect to the spatio-temporal constraints that must be satisfied by the entities for the authorization to be successful.

5.1.1 Authorization in the Standard Model STARBACD$^{=}$

In the standard model, the individual entities, namely, users, roles, permissions, and objects, are associated with set of points in the spatio-temporal domain. These points indicate when and where the individual entities can be activated. The spatio-temporal points associated with the user describe when and where the user can create a session, those associated with a role specify when and where the role can be activated, those associated with a permission state when and where a permission can be invoked, and those associated with an object state when and where the object can be accessed. The standard model requires that if a user u can access an object o at some spatio-temporal point d, then d is contained in the set of spatio-temporal points associated with all the nodes in the path connecting u to o. These ideas are formalized below.

The spatio-temporal constraints in the *standard STARBACD model* (or STARBACD$^{=}$) are denoted with a function $\lambda : V \rightarrow 2^{\mathcal{D}}$. For $v \in V$, $\lambda(v) \subseteq \mathcal{D}$ denotes the set of points in space-time at which v can be invoked.

- if $u \in U$, then $\lambda(u)$ denotes the set of points in space-time at which u may create a session;
- if $r \in R$, then $\lambda(r)$ denotes the set of points in space-time at which r may be activated in

a session;

- if $p \in P$, then $\lambda(p)$ denotes the set of points in space-time at which p may be granted;
- if $o \in O$, then $\lambda(o)$ denotes the set of points in space-time at which o may be accessible.

Given a path $v_1, \ldots, v_n$ in the labeled graph $G = (V, E, \lambda)$, where $E = UA \cup PA \cup PO \cup RH_a \cup RH_u$, we write $\hat{\lambda}(v_1, \ldots, v_n) = \hat{\lambda}(v_1, v_n) \subseteq \mathcal{D}$ to denote $\bigcap_{i=1}^{n} \lambda(v_i)$. In other words, $\hat{\lambda}(v_1, v_n)$ is the set of points at which every vertex v_i is enabled. Note that semantics of λ and $\hat{\lambda}$ are consistent with those proposed by Chen and Crampton [19].

Authorization in STARBACD$^{=}$:

- A user $v \in U$ may activate role $v' \in R$ at point $d \in \mathcal{D}$ if and only if there exists an act-path $v = v_1, v_2, \ldots, v_n = v'$ and $d \in \hat{\lambda}(v, v')$;
- A role $v \in R$ is authorized for permission $v' \in P$ at point $d \in \mathcal{D}$ if and only if there exists an u-path $v = v_1, v_2, \ldots, v_n = v'$ and $d \in \hat{\lambda}(v, v')$;
- A user $v \in U$ is authorized for permission $v' \in P$ with respect to object $v'' \in O$ at point $d \in \mathcal{D}$ if and only if there exists an acs-path $v = v_1, v_2, \ldots, v_i, \ldots, v_{n-1} = v', v_n = v''$ such that $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_i, \ldots, v_{n-1}$ is a u-path, $(v_{n-1}, n) \in PO$, and $d \in \hat{\lambda}(v, v'')$;

5.1.2 Authorization in the Strong Model STARBACD^{+}

The strong model is used when not only the individual entities (users, roles, permissions, objects) involved must satisfy the spatio-temporal constraints, but the different relationships must also satisfy such constraints. For instance, consider the relation $(r, p) \in PA$. In this case, we not only have to take into account the spatio-temporal points at which the role r can be activated in a session and the points at which the permission p can be invoked, but we also must consider the spatio-temporal points when r can invoke p. This requires specifying another function in the strong model as described below.

The spatio-temporal constraints in the *strong STARBACD model* (or STARBACD$^+$) are denoted with a function $\mu : E \rightarrow 2^{\mathcal{D}}$. For $e = (v, v') \in E$, $\mu(v, v')$ denotes the set of points in space-time at which the association between v and v' is enabled.

- if $(u, r) \in UA$, then $\mu(u, r)$ denotes the set of points in space-time at which u is assigned to r;
- if $(r', r) \in RH_a$, then $\mu(r', r)$ denotes the set of points in space-time at which r' is senior to r in the activation hierarchy;
- if $(r', r) \in RH_u$, then $\mu(r', r)$ denotes the set of points in space-time at which r' is senior to r in the permission usage hierarchy;
- if $(r, p) \in PA$, then $\mu(r, p)$ denotes the set of points in space-time at which p is assigned to r.
- if $(p, o) \in PO$, then $\mu(p, o)$ denotes the set of points in space-time at which o is assigned to p.

Given a path $v_1, \ldots, v_n$ in the labeled graph $G = (V, E, \lambda, \mu)$, where $V = U \cup R \cup P \cup O$ and $E = UA \cup PA \cup PO \cup RH_a \cup RH_u$, we write $\hat{\mu}(v_1, \ldots, v_n) = \hat{\mu}(v_1, v_n) \subseteq \mathcal{D}$ to denote $\bigcap_{i=1}^{n-1} \mu(v_i, v_{i+1})$. The semantics imply that an edge can only be enabled if both endpoints are enabled. Hence, $\hat{\mu}(v_1, v_n)$ is the set of points at which every vertex and every edge in the path is enabled. Here again the semantics of μ and $\hat{\mu}$ are consistent with those proposed by Chen and Crampton [19].

Authorization in STARBACD$^+$:

- a user $v \in U$ may activate role $v' \in R$ at point $d \in \mathcal{D}$ if and only if there exists an act-path $v = v_1, v_2, \ldots, v_n = v'$ and $d \in \hat{\mu}(v, v')$;
- a role $v \in R$ is authorized for permission $v' \in P$ at point $d \in \mathcal{D}$ if and only if there exists an u-path $v = v_1, v_2, \ldots, v_n = v'$ and $d \in \hat{\mu}(v, v')$;
- a user $v \in U$ is authorized for permission $v' \in P$ with respect to object $v'' \in O$ at point $d \in \mathcal{D}$ if and only if there exists an acs-path $v = v_1, v_2, \ldots, v_i, \ldots, v_{n-1} = v', v_n = v''$ such

that $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_i, \ldots, v_{n-1}$ is an u-path, $(v_{n-1}, v_n) \in PO$ and $d \in \hat{\mu}(v, v'')$;

5.1.3 Authorization in the Weak Model STARBACD^{-}

The weak model is derived from the standard model. Recall that the standard model requires that each entity (users, roles, permissions, and objects) in the authorization path be associated with a set of spatio-temporal points and the intersections of all these sets be non-zero. In the weak model, the entity v is authorized for another entity v' provided there is overlap in their spatio-temporal points. There is no requirement that the intermediate nodes on the path satisfy the spatio-temporal constraints. Like STARBACD$^{=}$, the model is based on the labeled graph $G = (V, E, \lambda)$, where $V = U \cup R \cup P \cup O$ and $E = UA \cup PA \cup PO \cup RH_a \cup RH_u$.

Authorization in STARBACD^{-}:

- A user $v \in U$ may activate role $v' \in R$ at point $d \in \mathcal{D}$ if and only if there exists an act-path $v = v_1, v_2, \ldots, v_n = v'$ and $d \in \lambda(v) \cap \lambda(v')$;
- A role $v \in R$ is authorized for permission $v' \in P$ at point $d \in \mathcal{D}$ if and only if there exists a u-path $v = v_1, v_2, \ldots, v_n = v'$ and $d \in \lambda(v) \cap \lambda(v')$;
- A user $v \in U$ is authorized for permission $v' \in P$ with respect to object $v'' \in O$ at point $d \in \mathcal{D}$ if and only if there exists an acs-path $v = v_1, v_2, \ldots, v_i, \ldots, v_{n-1} = v', v_n = v''$ such that $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_i, \ldots, v_{n-1}$ is an u-path, $(v_{n-1}, v_n) \in PO$ and and $d \in \lambda(v) \cap \lambda(v_i) \cap \lambda(v') \cap \lambda(v'')$;

5.2 Separation of Duties Constraints

Separation of duties (SoD) prevents the occurrence of fraud arising out of conflicts of interests in organizations [80]. Separation of duties ensure that conflicting roles are not assigned to the same user or that conflicting permissions are not assigned to the same role.

Separation of Duty (SoD) comes in two varieties. First one is with respect to the mutual exclusion relations between two roles. This is to guarantee that no user can be assigned to two

conflicting roles. The second one is with respect to the mutual exclusion relations between two permissions. This is to guarantee that no role can be assigned two conflicting permissions. We denote these two types of SoD by using SD^R and SD^P edges, respectively. Since SoD is a symmetric relationship, the SD^R and SD^P edges are bi-directional.

We next define the separation of duties for the standard and weak models. The SoDs defined for the standard and weak models are expressed in terms of the graph $G = (V, E, \lambda)$, where $E = UA \cup PA \cup PO \cup RH_a \cup RH_u \cup SD^R \cup SD^P$ and $V = U \cup R \cup P \cup O$. For these cases, the SoD is similar to the SoD constraints in traditional RBAC. These are given below.

SoD Constraints for STARBACD$^-$ and STARBACD$^=$

- **User-Role Assignment** if $(r, r') \in SD^R$ then there are no two edges (u, r) and (u, r') such that $\{(u, r), (u, r')\} \subset UA$
- **Permission-Role Assignment** if $(p, p') \in SD^P$ then there are no two u-paths of the form $r = v_1, v_2, \ldots, v_n = p$ and $r = v'_1, v'_2, \ldots, v'_n = p'$

Similar to other associations, we defined the spatio-temporal constraint for the separation of duties with a function $\mu : E \rightarrow 2^{\mathcal{D}}$. For $e = (v, v') \in SD^R \cup SD^P$, $\mu(v, v')$ denotes the set of points in space-time at which the association between v and v' (in this case, the SoD) is enabled. In particular,

- if $(r, r') \in SD^R$, $\mu(r, r')$ denotes the set of points in space-time at which the role-role separation of duties constraint is valid;
- if $(p, p') \in SD^P$, $\mu(p, p')$ denotes the set of points in space-time at which the permission-permission separation of duties constraint is valid.

The strong model is defined over the labeled graph $G = (V, E, \lambda, \mu)$, where $E = UA \cup PA \cup PO \cup RH_a \cup RH_u \cup SD^R \cup SD^P$ and $V = U \cup R \cup P \cup O$. The strong model allows specification of weaker forms of SoD constraints than those supported by the traditional RBAC. Specifically, it allows one to specify the spatio-temporal points at which the SoD constraints are valid.

SoD Constraints for STARBACD$^+$

- **User-Role Assignment:** if $(r, r') \in SD^R$ then there are no two edges (u, r) and (u, r'), corresponding to some user u, where $\mu(u, r) \cap \mu(u, r') \cap \mu(r, r') \neq \emptyset$

- **Permission-Role Assignment:** if $(p, p') \in SD^P$ then there are no two u-paths $r = v_1, v_2, \ldots, v_n = p$ and $r = v'_1, v'_2, \ldots, v'_n = p'$ where $\hat{\mu}(v_1, v_n) \cap \hat{\mu}(v'_1, v'_n) \cap \mu(p, p') \neq \emptyset$

5.3 Delegation in STARBACD

Many situations require the temporary transfer or granting of access rights belonging to a user/role to another user/role in order to accomplish a given task. For example, a department chair may delegate his privilege to the assistant chair while he is traveling. The entity that transfers or grants his privileges temporarily to another entity is referred to as the delegator and the entity who receives the privilege is known as the delegatee. The delegator (delegatee) can be either an user or a role. Thus, we may have four types of delegations: *user to user* (U2U), *user to role* (U2R), *role to role* (R2R), and *role to user* (R2U). When a user is the delegator, he can delegate a subset of permissions that he possesses by virtue of being assigned to different roles. When a role is the delegator, he can delegate either a set of permissions or he can delegate the entire role. We can therefore classify delegation on the basis of role delegation or permission delegation. In the graphical representation of STARBACD, we define a function $\nu : (U \cup R) \times (R \cup P) \rightarrow (U \cup R)$ that maps the delegation to the delegator. We assume the existence of different types of relationship corresponding to the different types of delegation as follows:

- User to User Role Delegation ($Delegate^R_{U2U}$) = $U \times R$, $\nu(u, r') = u'$ denotes the delegator who is a user authorized for role r'.

- User to User Permission Delegation ($Delegate^P_{U2U}$) = $U \times P$, $\nu(u, p') = u'$ denotes the delegator who is a user authorized for permission p'.

- User to Role Role Delegation ($Delegate^R_{U2R}$) = $R \times R$, $\nu(r, r') = u'$ denotes the delegator who is a user authorized for role r'.

- User to Role Permission Delegation ($Delegate^{P}_{U2R}$) = $R \times P$, $\nu(r, p') = u'$ denotes the delegator who is a user authorized for permission p'.

- Role to Role Role Delegation ($Delegate^{R}_{R2R}$) = $R \times R$, $\nu(r, r'') = r'$ denotes the delegator which is a role authorized for role r''. Note that r' and r'' can be the same role.

- Role to Role Permission Delegation ($Delegate^{P}_{R2R}$) = $R \times P$, $\nu(r, p') = r'$ denotes the delegator which is a role authorized for permission p'

- Role to User Role Delegation ($Delegate^{R}_{R2U}$) = $U \times R$, $\nu(u, r'') = r'$ denotes the delegator which is a role authorized for role r''. Note that r' and r'' can be the same role.

- Role to User Permission Delegation ($Delegate^{P}_{R2U}$) = $U \times P$, $\nu(u, p') = r'$ denotes the delegator which is a role authorized for permission p'.

5.3.1 Delegation in the Standard Model STARBACD$^{=}$

To represent delegation in a graph-theoretic manner for the standard and weak models, we have the labeled graph $G = (V, E, \lambda)$, where $E = UA \cup PA \cup PO \cup RH_a \cup RH_u \cup Delegate^{R}_{U2U} \cup Delegate^{P}_{U2U} \cup Delegate^{R}_{U2R} \cup Delegate^{P}_{U2R} \cup Delegate^{R}_{R2R} \cup Delegate^{P}_{R2R} \cup Delegate^{R}_{R2U} \cup Delegate^{P}_{R2U}$, and $V = U \cup R \cup P \cup O$. We use the notations λ and $\hat{\lambda}$ as before. The constraints below describe the situations when delegation is possible in our spatio-temporal model. For instance, the first constraint gives the spatio-temporal constraints that must be satisfied when user u' wants to delegate role r' to another user u. It states that this delegation is possible only if there is some overlap among the set of spatio-temporal points associated with user u''s activation of role r' with those of user u's session creation. The other constraints are specified in a similar manner.

Delegation in STARBACD$^{=}$

- If $(u, r') \in Delegate^{R}_{U2U}$ and $\nu(u, r') = u'$, then there exists an act-path $u' = v_1, v_2, \ldots, v_n = r'$ such that $\hat{\lambda}(v_1, v_n) \cap \lambda(u) \neq \emptyset$

- If $(u, p') \in Delegate^{P}_{U2U}$ and $\nu(u, p') = u'$, then there exists a path $u' = v_1, v_2, \ldots, v_i, \ldots, v_n = p'$ such that $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_{i+1}, \ldots, v_n$ is a u-path such that $\hat{\lambda}(v_1, v_n) \cap \lambda(u) \neq \emptyset$

- If $(r, r') \in Delegate^{R}_{U2R}$ and $\nu(r, r') = u'$, then there exists an act-path $u' = v_1, v_2, \ldots, v_n = r'$ such that $\hat{\lambda}(v_1, v_n) \cap \lambda(r) \neq \emptyset$

- If $(r, p') \in Delegate^{P}_{U2R}$ and $\nu(r, p') = u'$, then there exists a path $u' = v_1, v_2, \ldots, v_i, \ldots, v_n = p'$ such that $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_{i+1}, \ldots, v_n$ is a u-path such that $\hat{\lambda}(v_1, v_n) \cap \lambda(r) \neq \emptyset$

- If $(r, r'') \in Delegate^{R}_{R2R}$ and $\nu(r, r'') = r'$, then there exists a path $r' = v_1, v_2, \ldots, v_n = r''$ where $(v_i, v_{i+1}) \in RH_a$ for $1 \leq i \leq (n-1)$ such that $\hat{\lambda}(v_1, v_n) \cap \lambda(r) \neq \emptyset$

- If $(r, p') \in Delegate^{P}_{R2R}$ and $\nu(r, p') = r'$, then there exists a u-path $r' = v_1, v_2, \ldots, v_n = p'$ such that $\hat{\lambda}(v_1, v_n) \cap \lambda(r) \neq \emptyset$

- If $(u, r'') \in Delegate^{R}_{R2U}$ and $\nu(u, r'') = r'$, then there exists a path $r' = v_1, v_2, \ldots, v_n = r''$ where $(v_i, v_{i+1}) \in RH_a$ for $1 \leq i \leq (n-1)$ such that $\hat{\lambda}(v_1, v_n) \cap \lambda(u) \neq \emptyset$

- If $(u, p') \in Delegate^{P}_{R2U}$ and $\nu(u, p') = r'$, then there exists a u-path $r' = v_1, v_2, \ldots, v_n = p'$ such that $\hat{\lambda}(v_1, v_n) \cap \lambda(u) \neq \emptyset$

5.3.2 Delegation in the Weak Model STARBACD^{-}

The weak model is defined on the same graph as the standard model given in Section 5.3.1. In the weak model, the entity v is authorized for the delegated entity v' if both entities v and v' are enabled. There is no requirement that the intermediate nodes on the path are enabled. The constraints shown below describe when delegation is possible in the weak model. The first constraint says that the user u' can delegate role r' to another user u if the user u' can activate the role r' in the weak model and provided the spatio-temporal points when user u can create a session and role r' can be activated have some overlap. The other constraints are defined in a similar manner.

Delegation in STARBACD$^-$

- If $(u, r') \in Delegate^R_{U2U}$ and $\nu(u, r') = u'$, then there exists an act-path $u' = v_1, v_2, \ldots, v_n = r'$ such that $\lambda(v_1) \cap \lambda(v_n) \neq \emptyset$ and $\lambda(u) \cap \lambda(r') \neq \emptyset$

- If $(u, p') \in Delegate^P_{U2U}$ and $\nu(u, p') = u'$, then there exists a path $u' = v_1, v_2, \ldots, v_i, \ldots, v_n = p'$ such that $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_{i+1}, \ldots, v_n$ is an u-path such that $\lambda(v_1) \cap \lambda(v_n) \neq \emptyset$ and $\lambda(u) \cap \lambda(p') \neq \emptyset$

- If $(r, r') \in Delegate^R_{U2R}$ and $\nu(r, r') = u'$, then there exists an act-path $u' = v_1, v_2, \ldots, v_n = r'$ such that $\lambda(v_1) \cap \lambda(v_n) \neq \emptyset$ and $\lambda(r) \cap \lambda(r') \neq \emptyset$

- If $(r, p') \in Delegate^P_{U2R}$ and $\nu(r, p') = u'$, then there exists a path $u' = v_1, v_2, \ldots, v_i, \ldots, v_n = p'$ where $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_{i+1}, \ldots, v_n$ is an u-path such that $\lambda(v_1) \cap \lambda(v_n) \neq \emptyset$ and $\lambda(r) \cap \lambda(p') \neq \emptyset$

- If $(r, r'') \in Delegate^R_{R2R}$ and $\nu(r, r'') = r'$, then there exists a path $r' = v_1, v_2, \ldots, v_n = r''$ where $(v_i, v_{i+1}) \in RH_a$ for $1 \leq i \leq (n-1)$ such that $\lambda(v_1) \cap \lambda(v_n) \neq \emptyset$ and $\lambda(r) \cap \lambda(r'') \neq \emptyset$

- If $(r, p') \in Delegate^P_{R2R}$ and $\nu(r, p') = r'$, then there exists a u-path $r' = v_1, v_2, \ldots, v_n = p'$ such that $\lambda(v_1) \cap \lambda(v_n) \neq \emptyset$ and $\lambda(r) \cap \lambda(p') \neq \emptyset$

- If $(u, r'') \in Delegate^R_{R2U}$ and $\nu(u, r'') = r'$, then there exists a path $r' = v_1, v_2, \ldots, v_n = r''$ where $(v_i, v_{i+1}) \in RH_a$ for $1 \leq i \leq (n-1)$ such that $\lambda(v_1) \cap \lambda(v_n) \neq \emptyset$ and $\lambda(u) \cap \lambda(r'') \neq \emptyset$

- If $(u, p') \in Delegate^P_{R2U}$ and $\nu(u, p') = r'$, then there exists a u-path $r' = v_1, v_2, \ldots, v_n = p'$ such that $\lambda(v_1) \cap \lambda(v_n) \neq \emptyset$ and $\lambda(u) \cap \lambda(p') \neq \emptyset$

5.3.3 Delegation in the Strong Model STARBACD$^+$

The spatio-temporal constraints enforced in the delegation of the STARBACD$^+$ model are denoted with a function $\mu : E \rightarrow 2^{\mathcal{D}}$. For $e = (v, v') \in E$, $\mu(v, v')$ denotes the set of points in space-time at which the association between v and v' is activated.

The strong model is defined over the labeled graph $G = (V, E, \lambda, \mu)$, where $E = UA \cup PA \cup PO \cup RH_a \cup RH_u \cup Delegate^R_{U2U} \cup Delegate^P_{U2U} \cup Delegate^R_{U2R} \cup Delegate^P_{U2R} \cup Delegate^R_{R2R} \cup Delegate^P_{R2R} \cup Delegate^R_{R2U} \cup Delegate^P_{R2U}$ and $V = U \cup R \cup P \cup O$. The constraints for the strong model are enumerated below. The first constraint says that when a user u' delegates role r' to user u, then the delegation is possible only if the spatio-temporal points for activating user u''s role r' overlap with those in which the delegation is valid.

Delegation in STARBACD$^+$

- If $(u, r') \in Delegate^R_{U2U}$ and $\nu(u, r') = u'$, then there exists an act-path $u' = v_1, v_2, \ldots, v_n = r'$ such that $\hat{\mu}(v_1, v_n) \cap \mu(u, r') \neq \emptyset$
- If $(u, p') \in Delegate^P_{U2U}$ and $\nu(u, p') = u'$, then there exists a path $u' = v_1, v_2, \ldots, v_i, \ldots, v_n = p'$ where $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_{i+1}, \ldots, v_n$ is an u-path such that $\hat{\mu}(v_1, v_n) \cap \mu(u, p') \neq \emptyset$
- If $(r, r') \in Delegate^R_{U2R}$ and $\nu(r, r') = u'$, then there exists an act-path $u' = v_1, v_2, \ldots, v_n = r'$ such that $\hat{\mu}(v_1, v_n) \cap \mu(r, r') \neq \emptyset$
- If $(r, p') \in Delegate^P_{U2R}$ and $\nu(r, p') = u'$, then there exists a path $u' = v_1, v_2, \ldots, v_i, \ldots, v_n = p'$ such that $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_{i+1}, \ldots, v_n$ is an u-path such that $\hat{\mu}(v_1, v_n) \cap \mu(r, p') \neq \emptyset$
- If $(r, r') \in Delegate^R_{R2R}$ and $\nu(r, r'') = r'$, then there exists a path $r' = v_1, v_2, \ldots, v_n = r''$ where $(v_i, v_{i+1}) \in RH_a$ for $1 \leq i \leq (n-1)$ such that $\hat{\mu}(v_1, v_n) \cap \mu(r, r') \neq \emptyset$
- If $(r, p') \in Delegate^P_{R2R}$ and $\nu(r, p') = r'$, then there exists an u-path $r' = v_1, v_2, \ldots, v_n = p'$ such that $\hat{\mu}(v_1, v_n) \cap \mu(r, p') \neq \emptyset$
- If $(u, r'') \in Delegate^R_{R2U}$ and $\nu(u, r'') = r'$, then there exists a path $r' = v_1, v_2, \ldots, v_n = r''$ where $(v_i, v_{i+1}) \in RH_a$ for $1 \leq i \leq (n-1)$ such that $\hat{\mu}(v_1, v_n) \cap \mu(u, r'') \neq \emptyset$
- If $(u, p') \in Delegate^P_{R2U}$ and $\nu(u, p') = r'$, then there exists an u-path $r' = v_1, v_2, \ldots, v_n = p'$ such that $\hat{\mu}(v_1, v_n) \cap \mu(u, p') \neq \emptyset$

5.4 Dynamism Analysis

The pervasive computing applications are dynamic in nature–the accessing entities may change, resources requiring protection may be created or modified, and an entity's access to resources may change during the course of the application. Such changes may result in the unreachable entity or the violation of separation of duty constraints. Regarding this, the analysis is needed to detect the conflicts that may arise in the model in the presence of such dynamism. Our study reveals the possible changes which may lead to the associated conflict as follow:

1. **Entity Removal** The entity could be either user, role, permission, or object. This type of change can cause the isolated entity.

2. **Relationship Removal** The relationship could be either User-Role Assignment, Permission Usage Hierarchy, Role Activation Hierarchy, Role-Permission Assignment, or Permission-Object Assignment. This type of change can cause the isolated entity.

3. **Entity and Relationship Creation** The new entity together with its corresponding new relationship can be created. The entity could be either user, role, permission, or object. The relationship could be either User-Role Assignment, Permission Usage Hierarchy, Role Activation Hierarchy, Role-Permission Assignment, Permission-Object Assignment, SoD, or Delegation. This type of change can cause the SoD constraints violation.

In this section, we present the algorithms used to detect these conflicts.

5.4.1 Algorithm for Detecting the Isolated Entity

5.4.1.1 Preliminaries

In STARBACD model, we define the isolated entity as the entity which cannot be used. The isolated entity can be determined by considering the *in-degree* and *out-degree* of each vertex. The *in-degree* of the vertex can be defined correspond with different type of STARBACD model as follow:

Definition 44 (In-degree)

STARBACD$^{=}$ and STARBACD^{-} In the labeled graph $G = (V, E, \lambda)$, where $V = U \cup R \cup P \cup O$ and $E = UA \cup PA \cup PO \cup RH_a \cup RH_u$, *in-degree* of a vertex v is the cardinality of the set $\{(v', v) | ((v', v) \in E) \wedge (\lambda(v') \cap \lambda(v) \neq \emptyset)\}$

STARBACD^{+} In the labeled graph $G = (V, E, \lambda, \mu)$, where $V = U \cup R \cup P \cup O$ and $E = UA \cup PA \cup PO \cup RH_a \cup RH_u$, *in-degree* of a vertex v is the cardinality of the set $\{(v', v) | ((v', v) \in E) \wedge (\lambda(v') \cap \lambda(v) \cap \mu(v', v) \neq \emptyset)\}$

Definition 45 (Source)

Source is a vertex v which *in-degree*$(v) = 0$

Similarly, we define the *out-degree* of the vertex as follow:

Definition 46 (Out-degree)

STARBACD$^{=}$ and STARBACD^{-} In the labeled graph $G = (V, E, \lambda)$, where $V = U \cup R \cup P \cup O$ and $E = UA \cup PA \cup PO \cup RH_a \cup RH_u$, *out-degree* of a vertex v is the cardinality of the set $\{(v, v') | ((v, v') \in E) \wedge (\lambda(v) \cap \lambda(v') \neq \emptyset)\}$

STARBACD^{+} In the labeled graph $G = (V, E, \lambda, \mu)$, where $V = U \cup R \cup P \cup O$ and $E = UA \cup PA \cup PO \cup RH_a \cup RH_u$, *out-degree* of a vertex v is the cardinality of the set $\{(v, v') | ((v, v') \in E) \wedge (\lambda(v) \cap \lambda(v') \cap \mu(v, v') \neq \emptyset)\}$

Definition 47 (Sink)

Sink is a vertex v which *out-degree*$(v) = 0$

Note that we do not consider neither separation of duty nor the delegation edges since the modifications of these edges do not result in changes of the isolated entity.

5.4.1.2 The Detection Algorithm

By considering the value of *in-degree* and *out-degree* of each vertex, we can determine the isolated entity based on different types of vertex as follow:

User For $v \in U$, v is the isolated entity iff *out-degree*$(v) = 0$

Role and Permission For $v \in R \cup P$, v is the isolated entity iff (*in-degree*$(v) = 0) \vee$ (*out-degree*$(v) = 0$)

Object For $v \in O$, v is the isolated entity iff *in-degree*$(v) = 0$

To get the *in-degree* and *out-degree*, we have to count the number of edges connected to each vertex. This could be done in $O(VE)$. However, we can improve this by recording the *in-degree* and *out-degree* of each vertex. Each time the vertex or the edge is added to or removed from the graph, we update the *in-degree* and *out-degree* of the related vertices. Since we do not allow the existence of multiple edges between each pair of vertices, this update process could be done in $O(V)$. After we have such values recorded for every vertex, the detection can be done in $O(V)$.

5.4.2 Algorithm for Detecting the Infeasible Path

5.4.2.1 Preliminaries

In STARBACD model, a user *u* is authorized for permission *p* through role *r* with respect to object *o* iff there exists a valid *acs-path* which contains *u*, *r*, *p*, and *o*. We define an infeasible path as an invalid *acs-path* i.e. an *acs-path* which cannot grant the authorization of any permission to user.

5.4.2.2 The Detection Algorithm

To detect the infeasible path, we assume that we store all source vertices in a list. Each member in the list maintain its own depth-first search (DFS) tree. To generate these trees, we perform DFS from each source. While performing the DFS, we check if there is any spatio-temporal conflicts between the nodes (for STARBACD$^{=}$ and STARBACD^{-}) or edges (for STARBACD^{+}). If there is any conflicts, then there exists an infeasible path. This step could be done in $O(VE)$. After the process we will have set of the initial DFS trees which are all consists of feasible paths. Next for each update operation of the graph, we ensure that the following conditions are satisfied:

- Only *user* vertices can be the root of each subtree.

- Only *object* vertices can be the leave node of each subtree.

For each update operation of the graph, we perform the following:
If any new entity v and its corresponding relationship have been added to the initial graph, we consider the following:

- If v is a new source, we check whether the spatio-temporal constraint between the source and its immediate successors is satisfied. If so, we add v to the source list and maintain its pointers to its immediate successors. If not, then this v will create an infeasible path. This step could be done in $O(E)$.

- If v is a new intermediate vertex, we check whether the spatio-temporal constraint between v, its immediate predecessors, and its immediate successors is satisfied. If so, we create pointer from v's immediate predecessors to v, and from v to its immediate successors. If not, then this v will create an infeasible path. This step could be done in $O(E)$.

- If v is a new sink, we check whether the spatio-temporal constraint between v and its immediate predecessors is satisfied. If so, we create pointer from its immediate predecessors to v. If not, then this v will create an infeasible path. This step could be done in $O(E)$.

If any entity and its corresponding relationship has been removed from the graph, we consider the following:

- If the $(u,v) \in E$ is removed from the graph, and cause either u or v to be an isolated entity, this will create an infeasible path. This detection could be done in $O(1)$.

- If $v \in V$ and its related edges are removed from the graph, we determine whether this cause any of its neighbor vertices to become an isolated entity. If so, then this will create an infeasible path. Otherwise, we determine whether v is in the source list. If it is in the list, then we remove it out. This detection could be done in $O(E)$.

5.4.3 Algorithm for Detecting the SoD Violation

5.4.3.1 Preliminaries

In STARBACD model, the SoD can be violated by two ways. First, if $(r_1, r_2) \in SD^R$, and there exists *acs-paths* from u_1 to r_1 and u_1 to r_2. Or, if $(p_1, p_2) \in SD^P$, and there exists *u-paths* from r_1 to p_1 and r_1 to p_2.

5.4.3.2 The Detection Algorithm

Consider the dynamic case where the edge can be add and delete from the graph. The naive algorithm can be done by performing the reverse DFS on each $(v, v') \in SD^R \cup SD^P$ of the modified graph to find the common predecessor. This could be done in $O(k|E|)$ time.

Our algorithm which will be proposed next is the special case of the algorithm to find the common predecessors in a Directed Acyclic Graph (DAG) described in detail in the Appendix C. In our algorithm, each entity except users will maintain a list of users authorized for it by performing the DFS from each user. Only users satisfied the spatio-temporal constraints will be added to the list. To determine whether the SoD $(v, v') \in SD^P \cup SD^R$ is violated, we compare whether $u \in U$ is in the authorized users list of both v and v', and $\lambda(u) \cap \mu(v, v') \neq \emptyset$. If this is evaluate to true, then there exists a SoD violation. Since the size of each list cannot exceeds the number of user vertices, the evaluation time is $O(|U|)$. Let k be number of SoD edges, the detection time for the static case where no adding or removing edges allow is equal to $O(k|U|)$. To label all vertices takes $O(|E||U|)$ time, yields the total running time in the static graph equal to $O((k+|E|)|U|)$. However, in case that all edges modifications are of same type i.e. only either adding edges or deleting edges are allowed, we can improve the running time by applying the following graph specification updating summarized below:

- When only adding edges is allow to be done with the graph, each time that new edge is added, we update only the label list of vertices belonged to the graph portion that have not been reached before by using the Incremental-DFS described in the Appendix C. All updates take $O(|E||U|)$ time, and detecting whether the SoD is violated take $O(|U|)$ per SoD edge. This yields the total processing time equal to $O((k+|E|)|U|)$.

- When only removing edges is allow to be done with the graph, we update only the label list of vertices that becomes unreachable by some user u after the edge removal. Following the algorithm described in the Appendix C, the removal of an edge takes $O(|E|log|V|)$ time for relabeling for each user vertex, and detecting whether the SoD is violated take $O(|U|)$ per SoD edge. This yields the total processing time equal to $O((k+|E|log|V|)|U|)$.

For the detail on graph specification updating algorithm and proof of correctness. We refer to the Appendix C.

5.5 Example Scenario

Military application is one scenario which STARBACD can be applied. The army application is dynamic in nature–during the period of war, soldiers might be under attack and cannot continue to pursue thier mission. In such scenario, it is important for the access control to support the delegation of authority from the soldier who is under attack to another soldier.

In this section, we will demonstrate a scenario where the STARBACD$^+$ model can be used to control the accessibility to the resource. Let assume that in the battlefield, each troop consists of military staff with the following responsibilities: The *Intelligent Officer* responsible for the process of acquiring enemy information then interpreting and exploiting it to the *Soldier* in his troop in order to perform a suitable attack. And finally, the *Clinical Officer* is in charge of monitoring the health information of his troop, evaluate the information to check whether the trooper's life is in danger, and send the SOS signal to the commander to get the proper help. The list of entities and the spatio-temporal relationships are shown in Tables 8.1 and 8.2 respectively.

The graph-theoretic representation is shown in Figure 5.1(a). We will only describe parts of this configuration. User *Alex* (u_1) can create session at any time and at any place as per Row 1 of Table 8.1. He is assigned the role of *Intelligence Officer* (r_1) which can be activated at any place at any time. During this time and at this location, he has permission (p_1) to access the *Surveillance Sensor Information* (o_1). Since *Intelligence Officer* is senior to *Soldier* role in the

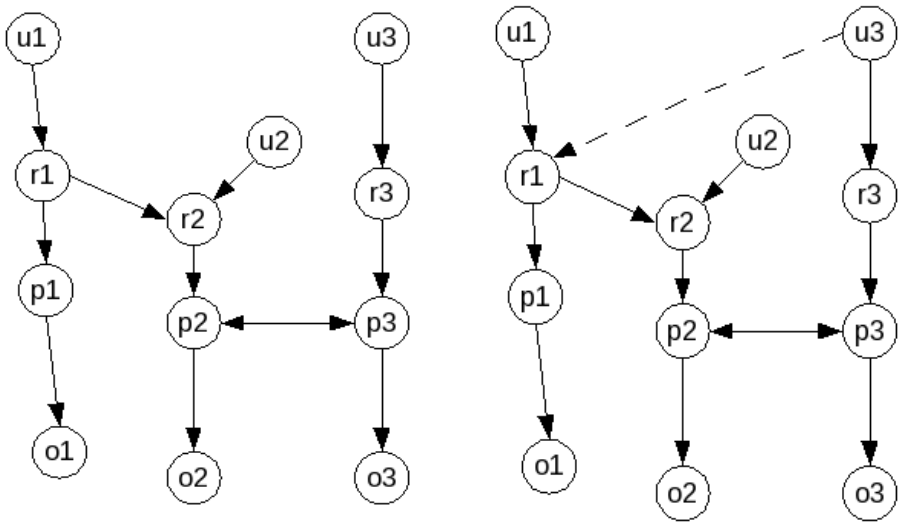
NAME	DESCRIPTION	SPATIO-TEMPORAL DOMAIN (λ)
u_1	Alex	$[Universe, Always]$
u_2	Ben	$[Universe, Always]$
u_3	Charlie	$[Universe, Always]$
r_1	Intelligence Officer	[*Universe, Always*]
r_2	Soldier	[*Field, Always*]
r_3	Clinical Officer	[*Universe, Always*]
p_1	Access Surveillance Sensor	[*Universe, Always*]
p_2	Manouver the Vehicle	[*Field, Always*]
p_3	Access Vital Sensor	[*Universe, Always*]
o_1	Surveillance Sensor Information	[*Universe, Always*]
o_2	Tank	[*Field, Always*]
o_3	Health Information	[*Universe, Always*]

Table 5.1: STARBACD Entities for the Example

NAME	DESCRIPTION	SPATIO-TEMPORAL DOMAIN (μ)
(u_1, r_1)	User-Role Assignment	[*Universe, Always*]
(u_2, r_2)	User-Role Assignment	[*Field, Always*]
(u_3, r_3)	User-Role Assignment	[*Universe, Always*]
(r_1, r_2)	Permission Usage Hierarchy	[*Field, Always*]
(r_1, p_1)	Permission-Role Assignment	[*Universe, Always*]
(r_2, p_2)	Permission-Role Assignment	[*Field, Always*]
(r_3, p_3)	Permission-Role Assignment	[*Universe, Always*]
(p_2, p_3)	Separation of Duties	[*Universe, Always*]
(p_3, p_2)	Separation of Duties	[*Universe, Always*]
(p_1, o_1)	Permission-Object Assignment	[*Universe, Always*]
(p_2, o_2)	Permission-Object Assignment	[*Field, Always*]
(p_3, o_3)	Permission-Object Assignment	[*Universe, Always*]

Table 5.2: STARBACD Relationships and Constraints

permission usage hierarchy, he can also get the permission to manouver the *Tank*. However, this permission is allowed only when the hierarchy is enabled on the battle field. During the war, Alex gets shot and cannot pursue his mission. So, he decides to delegate his role to Charlie for a month until he is fully recovered. This new graphical representation is shown in Figure 5.1(b) where the delegation edge is represented by the dash arrow. However, this delegation should not be allowed because our algorithm detects a violation of separation of duty constraint in the existence of this delegation.



(a) Configuration before delegation (b) Configuration after delegation

Figure 5.1: STARBACD Configuration for Example

5.6 Chapter Summary

In this chapter, we propose the second spatio-temporal access control model, namely the Spatio-Temporal Role Based Access Control Model with Delegation (STARBACD) model. The model is based on graph representation, which is well-formed semantics. Based on this model, we develop algorithms to detect the conflicts and inconsistencies based on graph theory as described in Section 5.4. The algorithm can reveal differenct types of conflicts, namely, isolated entity, infeasible path, and separation of duty violation. Nonetheless, this analysis approach is far from being automated.

In Chapter 6, we will extend our STRBAC model discussed in Chapter 3 to support the delegation chain. To get the well-formed semantics, the model will be transformed into the graph representation. The graph will then be transformed into the Coloured Petri-Nets format, which can be automatedly analyzed by using the analysis tool called CPN Tool.

Chapter 6

The Extended STRBAC Model

In this chapter, we present a comprehensive model with a well-defined semantics expressed in graph-theoretic notation and that can be automatically verified by existing tools.

6.1 Our Model

6.1.1 Representing Location and Time

Representing Location

In order to perform location-based access control, we need to formalize the concept of location [13, 18] and propose the location comparison operators that are used in our model. There are two types of locations: *physical* and *logical*. All users and objects are associated with locations that correspond to the physical world. These are referred to as the physical locations. A physical location is formally defined by a set of points in a three-dimensional geometric space. A *physical location* $PLoc_i$ is a non-empty set of points $\{p_i, p_j, \ldots, p_n\}$ where a point p_k is represented by three co-ordinates. The granularity of each co-ordinate is dependent upon the application. Physical locations are grouped into symbolic representations that will be used by applications. We refer to these symbolic representations as logical locations. Examples of logical locations are Fort Collins, Colorado etc. A *logical location* is an abstract notion for one or more physical locations. We assume the existence of a mapping function m that converts a logical location to a physical one.

Definition 48 (Mapping Functions m)

m is a total function that converts a logical location into a physical one. Formally, $m : L \rightarrow P$, where P is the set of all possible physical location and L is the set of all logical location.

Different kinds of operations can be performed on location data. We define two binary operators, namely, *containment* $\subseteq$, and *equality* $=$.A physical location $ploc_j$ is said to be *contained in* another physical location $ploc_k$, denoted as, $ploc_j \subseteq ploc_k$, if the following condition holds: $\forall p_i \in ploc_j, p_i \in ploc_k$. The location $ploc_j$ is called the contained location and $ploc_k$ is referred to as the containing or the enclosing location. Intuitively, a physical location $ploc_j$ is contained in another physical location $ploc_k$, if all points in $ploc_j$ also belong to $ploc_k$. Two physical locations $ploc_r$ and $ploc_s$ are *equal* if $ploc_r \subseteq ploc_s$ and $ploc_s \subseteq ploc_r$. Note that these operators are defined on physical locations. Thus, logical locations must be transformed into physical locations (using mapping function m defined above) before we can apply these operators. We define a logical location called *Universe* that contains all other locations.

Representing Time

Our model uses two kinds of temporal information. The first is known as time instant and the other is time interval. A *time instant* is one discrete point on the time line. The exact granularity of a time instant is application dependent. For instance, in some application a time instant may be measured at the nanosecond level and in another one it may be specified at the millisecond level. A *time interval* is a set of time instants. We use the notation $t_i \in d$ to mean that t_i is a time instant in the time interval d. Here again, we define operators containment $\subseteq$ and equality $=$ for operating on time intervals. A time interval d_j is said to be *contained in* another time interval d_k, denoted as, $d_j \subseteq d_k$, if the following condition holds: $\forall t_i \in d_j, t_i \in d_k$. The interval d_j is called the contained interval and d_k is referred to as the containing or the enclosing interval. Two time intervals d_s and d_r are said to be equal if $d_r \subseteq d_s$ and $d_s \subseteq d_r$. We define a time interval called *Always* that includes all other time intervals.

Representing Time and Location as Spatio-Temporal Points

In order to simplify our presentation, we use the concept of spatio-temporal points to represent time and location. A spatio-temporal point is represented as a pair of the form (d,l) where d represents the temporal component and l represents the spatial one. Note that, d and l represent time interval and location respectively. We say that a spatio-temporal point (d,l) is contained in another (d',l'), denoted by $(d,l) \subseteq (d',l')$ iff $(d \subseteq d') \wedge (l \subseteq l')$. The union of two spatio-temporal points, denoted as $(d,l) \cup (d',l')$, is given by $(d,l) \cup (d',l') = (d \cup d', l \cup l')$. The intersection of two spatio-temporal points, denoted as $(d,l) \cap (d',l')$, is given by $(d,l) \cap (d',l')$ $= (d \cap d', l \cap l')$.

6.1.2 Relationship of Core-RBAC Entities and Relationships with Time and Location

In this section, we describe how the entities in RBAC, namely, *Users*, *Roles*, *Sessions*, *Permissions*, and *Objects*, are associated with location and time.

Users

We assume that each valid user, interested in doing some location-sensitive operation, carries a locating device that is able to track his location. The location of a user changes with time. The relation $UserLocation(u,t)$ gives the location of the user at any given time instant t. Since a user can be associated with only one location at any given point of time, the following constraint must be true. Note that, in this and all the subsequent formulae, we omit the quantification symbols.

$$(UserLocation(u,t) = l_i) \wedge (UserLocation(u,t) = l_j) \Leftrightarrow (l_i \subseteq l_j) \vee (l_j \subseteq l_i)$$

We define a similar function $UserLocation(u,d)$ that gives the location of the user during the time interval d. Note that, a single location can be associated with multiple users at any given point of time.

Objects

Objects can be physical or logical. Example of a physical object is a computer. Files are examples of logical objects. Physical objects have devices that transmit their location information with the timestamp. Logical objects are stored in physical objects. The location and timestamp of a logical object corresponds to the location and time of the physical object containing the logical object. Each location can be associated with many objects. The function *ObjLocation(o,t)* takes as input an object o and a time instance t and returns the location associated with the object at time t. Similarly, the function *ObjLocation(o,d)* takes as input an object o and time interval d and returns the location associated with the object.

Roles

We have three types of relations with roles. These are user-role assignment, user-role activation, and permission-role assignment. We begin by focusing on user-role assignment. Often times, the assignment of user to roles is location and time dependent. For instance, a person can be assigned the on-campus student role only when he is in the campus during the semester. Thus, for a user to be assigned a role, he must be in designated locations during specific time intervals. In our model, a user must satisfy spatial and temporal constraints before roles can be assigned. We capture this with the concept of *role allocation*. A role is said to be *allocated* when it satisfies the temporal and spatial constraints needed for role assignment. A role can be assigned once it has been allocated. $RoleAllocTimeLoc(r)$ gives the set of spatio-temporal points where the role can be allocated.

The predicate $UserRoleAssign(u,r,d,l)$ states that the user u is assigned to role r during the time interval d and location l. For this predicate to hold, the location of the user when the role was assigned must be in one of the locations where the role allocation can take place. Moreover, the time of role assignment must be in the interval when role allocation can take place.

$$UserRoleAssign(u,r,d,l) \Rightarrow (UserLocation(u,d) = l) \wedge ((d,l) \subseteq RoleAllocTimeLoc(r))$$

Some roles can be activated only if the user is in some specific locations at given time.

For instance, the role of audience of a theater can be activated only if the user is in the theater when the show is on. The role of conference attendee can be activated only if the user is in the conference site while the conference is in session. In short, the user must satisfy temporal and location constraints before a role can be activated. We borrow the concept of *role-enabling* [12, 43] to describe this. A role is said to be *enabled* if it satisfies the temporal and location constraints needed to activate it. A role can be activated only if it has been enabled. $RoleEnableTimeLoc(r)$ gives the set of spatio-temporal points where role r can be activated.

The predicate $UserRoleActivate(u,r,d,l)$ is true if the user u activated role r for the interval d at location l. This predicate implies that the location of the user and the duration of role activation must be a subset of the allowable spatio-temporal points for the activated role and the role can be activated only if it is assigned.

$$UserRoleActivate(u,r,d,l) \Rightarrow ((d,l) \subseteq RoleEnableTimeLoc(r)) \wedge UserRoleAssign(u,r,d,l)$$

The permission-role assignment is discussed later.

Sessions

In mobile computing or pervasive computing environments, we have different types of sessions that can be initiated by the user. Some of these sessions can be time-dependent, location-dependent, or both. Thus, sessions are classified into different types. Each instance of a session is associated with some type of a session. The type of session instance s is given by the function $Type(s)$. The type of the session determines the allowable location and duration. The allowable spatio-temporal points where a session of type st can be created is denoted by $SessionTimeLoc(st)$.

When a user u wants to create a session s, the session duration d and the location of the user l must be contained within the spatio-temporal points associated with the session. The predicate $SessionUser(u,s,d,l)$ indicates that a user u has initiated a session s for duration d at location l.

$$SessionUser(u,s,d,l) \Rightarrow (d,l) \subseteq SessionTimeLoc(Type(s))$$

Since sessions are associated with time and locations, not all roles can be activated within some session. The predicate $SessionRoles(u,r,s,d,l)$ states that user u initiates a session s and activates a role r for duration d and at location l. This is possible only if user u can activate role r for duration d and at location l and the session can be created during the same time and at the same location.

$$SessionRole(u,r,s,d,l) \Rightarrow UserRoleActivate(u,r,d,l) \wedge (d,l) \subseteq SessionTimeLoc(Type(s))$$

Permissions

Our model allows us to specify real-world requirements where access decision is contingent upon the time and location associated with the user and the object. For example, a teller may access the bank confidential file only if he is in the bank, the file location is the bank secure room, and the time of access is during the working hours. Our model should be capable of expressing such requirements.

Permissions are associated with roles, objects, and operations. We associate additional entities with permission to deal with spatial and temporal constraints: user location, object location, and time. We define three functions to retrieve the values of these entities. $PermRoleLoc(p,r)$ specifies the allowable locations that a user playing the role r must be in for him to get permission p. $PermObjLoc(p,o)$ specifies the allowable locations that the object o must be in so that the user has permission to operate on the object o. $PermDur(p)$ specifies the allowable time when the permission can be invoked.

We define another predicate which we term $PermRoleAcquire(p,r,d,l)$. This predicate is true if role r has permission p for duration d at location l. Note that, for this predicate to be true, the spatio-temporal point (d,l) must be contained in the point where the role r can be enabled and where the permission p can be invoked by r.

$$PermRoleAcquire(p,r,d,l) \Rightarrow$$
$$(d,l) \subseteq RoleEnableTimeLoc(r) \cap (PermDur(p) \times PermRoleLoc(p,r))$$

The predicate $PermUserAcquire(u,o,p,d,l)$ means that user u can acquire the permission p on object o for duration d at location l. This is possible only when the permission p can be

acquired by role r during time d and at location l, user u can activate role r at the same time and location, and object location matches those specified in the permission.

$$PermRoleAcquire(p,r,d,l) \wedge UserRoleActivate(u,r,d,l)$$
$$\wedge(ObjectLocation(o,d) \subseteq PermObjectLoc(p,o)) \Rightarrow PermUserAcquire(u,o,p,d,l)$$

6.1.3 Impact of Time and Location on Role-Hierarchy

The structure of an organization in terms of lines of authority can be modeled as a hierarchy. This organization structure is reflected in RBAC in the form of a role hierarchy [75]. Role hierarchy is a transitive and anti-symmetric relation among roles. Roles higher up in the hierarchy are referred to as senior roles and those lower down are junior roles. The major motivation for adding role hierarchy to RBAC was to simplify role management. Senior roles can inherit the permissions of junior roles, or a senior role can activate a junior role, or do both depending on the nature of the hierarchy. This obviates the need for separately assigning the same permissions to all members belonging to a hierarchy.

Joshi et al. [43] identify two basic types of hierarchy. The first is the permission inheritance hierarchy where a senior role x inherits the permission of a junior role y. The second is the role activation hierarchy where a user assigned to a senior role can activate a junior role. Each of these hierarchies may be constrained by location and temporal constraints. Consequently, we have a number of different hierarchical relationships in our model.

[Unrestricted Permission Inheritance Hierarchy] Sometimes we want a senior role to inherit permissions of a junior role without any additional spatio-temporal constraints. For example, a contact author can inherit the permissions of the author without any extra spatio-temporal constraints. That is, the contact author can invoke the author's permission wherever and whenever the author can invoke them. Unrestricted permission inheritance hierarchy allows the senior role to acquire inherited permissions whenever and wherever the junior role can acquire them. Let x and y be roles such that $x \geq_{(Always,Universe)} y$, that is, senior role x has an unrestricted permission-inheritance relation over junior role y. In such a case, x inherits y's permissions without any additional spatio-temporal constraints. This is formalized as follows:

$$(x \geq_{(Always,Universe)} y) \wedge PermRoleAcquire(p,y,d,l) \Rightarrow PermRoleAcquire(p,x,d,l)$$

[Unrestricted Activation Hierarchy] Sometimes a senior role may want to activate a junior role without placing any additional constraints. For example, a user who has a role of mobile user can activate the weekend mobile user role only if he/she is in the US during the weekend. Unrestricted activation hierarchy allows the senior role to be activated whenever and wherever the junior role can be activated.

Let x and y be roles such that $x \succcurlyeq_{(Always,Universe)} y$, that is, senior role x has an unrestricted role-activation relation over junior role y. Then, a user assigned to role x can activate role y at any time and at any place that y can be activated. This is formalized as follows:

$$(x \succcurlyeq_{(Always,Universe)} y) \wedge UserRoleActivate(u,x,d,l) \wedge ((d,l) \subseteq RoleEnableTimeLoc(y)) \Rightarrow UserRoleActivate(u,y,d,l)$$

[Time Restricted Permission Inheritance Hierarchy] Sometimes a senior role can inherit a junior role only at certain times. For example, a company may have a policy that allows the project manager to inherit the permissions of the code developer role only when the product deadline date is less than a given threshold. Time restricted permission inheritance hierarchy allows the senior role to acquire the permissions of the junior role when the temporal constraints associated with the hierarchy hold and the senior role satisfies the spatio-temporal constraints that are needed by the junior role to invoke those permissions.

Let x and y be roles such that $x \geq_{(d',Universe)} y$, that is, senior role x has a time restricted permission-inheritance relation over junior role y. In such a case, x inherits y's permissions together with the temporal constraints associated with the permissions and the hierarchy. This is formalized as follows:

$$(x \geq_{(d',Universe)} y) \wedge PermRoleAcquire(p,y,d,l) \Rightarrow PermRoleAcquire(p,x,d \cap d',l)$$

[Time Restricted Activation Hierarchy] In some applications, the senior role may need to be activated only during specific periods. For example, the account auditor role can activate the accountant role only during the auditing period. Time restricted activation hierarchy allows

the senior role to activate the junior role when the temporal constraints associated with the hierarchy hold and the senior role satisfies the spatio-temporal constraints that are needed to activate the junior role.

Let x and y be roles such that $x \succcurlyeq_{(d',Universe)} y$, that is, senior role x has a role-activation relation over junior role y. Then, a user assigned to role x can activate role y only at the location and time when role y can be enabled and the additional temporal constraints are satisfied. This is formalized as follows:

$$(x \succcurlyeq_{(d',Universe)} y) \wedge UserRoleActivate(u,x,d,l) \wedge ((d,l) \subseteq RoleEnableTimeLoc(y)) \Rightarrow UserRoleActivate(u,y,d \cap d',l)$$

[Location Restricted Permission Inheritance Hierarchy] Sometimes a senior role can inherit a junior role only in certain locations. For example, a top secret nuclear scientist inherits the permissions of a nuclear scientist only in top secret locations. Location restricted permission inheritance allows the senior role to acquire the permissions of the junior role when the location constraints associated with the hierarchy hold and the senior role satisfies the spatio-temporal constraints that are needed by the junior role to invoke those permissions.

Let x and y be roles such that $x \geq_{(Always,l')} y$, that is, senior role x has a location restricted permission-inheritance relation over junior role y. In such a case, x inherits y's permissions together with the location constraints associated with the permission and the hierarchy. This is formalized as follows:

$$(x \geq_{(Always,l')} y) \wedge PermRoleAcquire(p,y,d,l) \Rightarrow PermRoleAcquire(p,x,d,l \cap l')$$

[Location Restricted Activation Hierarchy] Sometimes we want the senior role to be able to activate the junior role only at certain locations. For example, a department chair can activate a staff role only when he is in the department. Location restricted activation hierarchy allows the senior role to activate the junior role when the location constraints needed for the hierarchy activation hold and the senior role satisfies the spatio-temporal constraints needed to activate the junior role.

Let x and y be roles such that $x \succcurlyeq_{(Always,l')} y$, that is, senior role x has a role-activation relation over junior role y. Then, a user assigned to role x can activate role y only at the places when role y can be enabled and the location constraints of the hierarchy are satisfied. This is formalized as follows:

$$(x \succcurlyeq_{(Always,l')} y) \wedge UserRoleActivate(u,x,d,l) \wedge$$
$$((d,l) \subseteq RoleEnableTimeLoc(y)) \Rightarrow UserRoleActivate(u,y,d,l \cap l')$$

[Time Location Restricted Permission Inheritance Hierarchy] Sometimes we may want to place additional temporal as well as spatial constraints on the permission inheritance hierarchy. For instance, a doctor can inherit the daytime nurse role only when he is in the hospital at the daytime. Time-location restricted permission inheritance hierarchy allows the senior role to invoke the permissions of the junior role provided the senior role satisfies the spatio-temporal constraints of the inheritance hierarchy and also the spatio-temporal constraints needed to acquire the permissions of the junior role.

Let x and y be roles such that $x \geq_{(d',l')} y$, that is, senior role x has a time-location restricted permission-inheritance relation over junior role y. In such a case, x inherits y's permissions together with the temporal and location constraints associated with the permission together with the temporal and location constraints associated with the hierarchy. This is formalized as follows:

$$(x \geq_{(d',l')} y) \wedge PermRoleAcquire(p,y,d,l) \Rightarrow PermRoleAcquire(p,x,d \cap d', l \cap l')$$

[Time Location Restricted Activation Hierarchy] Sometimes additional spatial and temporal constraints must be satisfied for a senior role to activate a junior role. Emergency physicians can activate the role of primary care physicians only when the patient is in an emergency room. Time location restricted activation hierarchy allows the senior role to activate the junior role when the spatio-temporal constraints associated with the hierarchy are satisfied together with the spatio-temporal constraints associated with the invocation of the junior role.

Let x and y be roles such that $x \succcurlyeq_{(d',l')} y$, that is, senior role x has a role-activation relation over

junior role y. Then, a user assigned to role x can activate role y only at the places and during the time when role y can be enabled, and the additional spatio-temporal constraints assigned to the hierarchy are satisfied. This is formalized as follows:

$$(x \succcurlyeq_{(d',l')} y) \wedge UserRoleActivate(u,x,d,l) \wedge$$
$$((d,l) \subseteq RoleEnableTimeLoc(y)) \Rightarrow UserRoleActivate(u,y,d \cap d', l \cap l')$$

It is also possible for a senior role and a junior role to be related with both permission inheritance and activation hierarchies. In such a case, the application will choose the type of inheritance hierarchy and activation hierarchy needed.

6.1.4 Impact of Time and Location on Static Separation Of Duty Constraints

Separation of duty (SoD) protects against the fraud that may be caused from a user or role gaining too much power [80]. SoD can be either static or dynamic. Static Separation of Duty (SSoD) comes in two varieties. The first one, which is referred to as *SSoD - User Role Assignment (SSoD-URA)*, is with respect to user-role assignment. SSoD-URA is specified as a relation between roles – the same user cannot be assigned to the roles that are related by the SSoD-URA relation. The second one, which is referred to as *SSoD - Permission Role Assignment (SSoD-PRA)*, is with respect to permission-role assignment. SSoD-PRA is specified as a relation between permissions – the same role cannot be assigned to the permissions that are related by the SSoD-PRA relation. Due to the presence of temporal and spatial constraints, we can have different flavors of separation of duties – some that are constrained by temporal and spatial constraints and others that are not. In the following, we describe the different types of separation of duty constraints.

[Weak Form of SSoD - User-Role Assignment] Let x and y be two roles such that $x \neq y$. $(x,y) \in SSOD\_URA_w$ if the following condition holds:

$$UserRoleAssign(u,x,d,l) \Rightarrow \neg\, UserRoleAssign(u,y,d,l)$$

The above definition says that a user u assigned to role x during time d and location l cannot be assigned to role y at the same time and location if x and y are related by $SSOD\_URA_w$. An example where this form is useful is that a user should not be assigned the audience role and mobile user role at the same time and location.

[Strong Temporal Form of SSoD - User-Role Assignment]

Let x and y be two roles such that $x \neq y$. $(x,y) \in SSOD\_URA_t$ if the following condition holds:

$$UserRoleAssign(u,x,d,l) \Rightarrow \neg\ (\exists d' \subseteq always \bullet UserRoleAssign(u,y,d',l))$$

The above definition says that a user u assigned to role x during time d and location l cannot be assigned to role y at any time in the same location if x and y are related by $SSOD\_URA_t$. The consultant for oil company A will never be assigned the role of consultant for oil company B in the same country.

[Strong Spatial Form of SSoD - User-Role Assignment] Let x and y be two roles such that $x \neq y$. $(x,y) \in SSOD\_URA_l$ if the following condition holds:

$$UserRoleAssign(u,x,d,l) \Rightarrow \neg\ (\exists l' \subseteq Universe \bullet UserRoleAssign(u,y,d,l'))$$

The above definition says that a user u assigned to role x during time d and location l, he cannot be assigned to role y at the same time at any location if x and y are related by $SSOD\_URA_l$. A person cannot be assigned the roles of student and instructor of the same course at the same time.

[Strong Form of SSoD - User-Role Assignment] Let x and y be two roles such that $x \neq y$. $(x,y) \in SSOD\_URA_s$ if the following condition holds:

$$UserRoleAssign(u,x,d,l) \Rightarrow \neg\ (\exists l' \subset Universe, \exists d' \subseteq always \bullet UserRoleAssign(u,y,d',l'))$$

The above definition says that a user u assigned to role x during time d and location l, he cannot be assigned to role y at any time or at any location if x and y are related by $SSOD\_URA_s$. The same employee cannot be assigned the roles of minority and non-minority employee at any given corporation.

We next consider the second form of static separation of duty that deals with permission-role assignment. The idea is that the same role should not acquire conflicting permissions.

[Weak Form of SSoD - Permission-Role Assignment] Let p and q be two permissions such that $p \neq q$. $(p,q) \in SSOD\_PRA_w$ if the following condition holds:

$$PermRoleAcquire(p,x,d,l) \Rightarrow \neg\, PermRoleAcquire(q,x,d,l)$$

The above definition says that if permissions p and q are related through weak SSoD Permission-Role Assignment and x has permission p at time d and location l, then x should not be given permission q at the same time and location. The same role should not be assigned the permission of chairing the session and presenting the paper in the conference at the same location and at the same time.

[Strong Temporal Form of SSoD - Permission-Role Assignment] Let p and q be two permissions such that $p \neq q$. $(p,q) \in SSOD\_PRA_t$ if the following condition holds:

$$PermRoleAcquire(p,x,d,l) \Rightarrow \neg\, (\exists d' \subseteq always \bullet PermRoleAcquire(q,x,d',l))$$

The above definition says that if permissions p and q are related through strong temporal SSoD Permission-Role Assignment and x has permission p at time d and location l, then x should not get permission q at any time in location l. The accountant should not get both the permissions of modifying accounts and auditing accounts at the same branch location at any time.

[Strong Spatial Form of SSoD - Permission-Role Assignment] Let p and q be two permissions such that $p \neq q$. $(p,q) \in SSOD\_PRA_l$ if the following condition holds:

$$PermRoleAcquire(p,x,d,l) \Rightarrow \neg\ (\exists l' \subseteq Universe \bullet PermRoleAcquire(q,x,d,l'))$$

The above definition says that if permissions p and q are related through strong spatial SSoD Permission-Role Assignment and x has permission p at time d and location l, then x should not be given permission q at the same time. The same role should not be given the permission of grading the exam and taking the exam at the same time at any location.

[Strong Form of SSoD - Permission-Role Assignment] Let p and q be two permissions such that $p \neq q$. $(p,q) \in SSOD\_PRA_s$ if the following condition holds:

$$PermRoleAcquire(p,x,d,l) \Rightarrow \neg\ (\exists l' \subseteq Universe, \exists d' \subseteq always \bullet PermRoleAcquire(q,x,d',l'))$$

The above definition says that if permissions p and q are related through strong SSoD Permission-Role Assignment, then the same role should never be given the two conflicting permissions. The permission to authorize a check and issue it should not be given to the same role at any time and at any location.

6.1.5 Impact of Time and Location on Dynamic Separation of Duty Constraints

Dynamic separation of duty addresses the problem that a user is not able to activate conflicting roles during the same session.

[Weak Form of DSoD] Let x and y be two roles such that $x \neq y$. $(x,y) \in DSOD_w$ if the following condition holds:

$$SessionRole(u,x,s,d,l) \Rightarrow \neg\ SessionRole(u,y,s,d,l)$$

The above definition says that if roles x and y are related through weak DSoD and if user u has activated role x in some session s for duration d and location l, then u cannot activate

role y during the same time and in the same location in session s. In the same session, a user can activate a sales assistant role and a customer role. However, both these roles should not be activated at the same time in the same location.

[Strong Temporal Form of DSoD] Let x and y be two roles such that $x \neq y$. $(x,y) \in DSOD_t$ if the following condition holds:

$$SessionRole(u,x,s,d,l) \Rightarrow \neg\ (\exists d' \subseteq always, \bullet SessionRole(u,y,s,d',l))$$

The above definition says that if roles x and y are related through strong temporal DSoD and if user u has activated role x in some session s, then u can never activate role y any time at the same location in the same session. In a teaching session in a classroom, a user cannot activate the the grader role once he has activated the student role.

[Strong Spatial Form of DSoD] Let x and y be two roles such that $x \neq y$. $(x,y) \in DSOD_l$ if the following condition holds:

$$SessionRole(u,x,s,d,l) \Rightarrow \neg\ (\exists l' \subseteq Universe \bullet SessionRole(u,y,s,d,l'))$$

The above definition says that if roles x and y are related through strong DSoD and if user u has activated role x in some session s, then u can never activate role y in session s during the same time in any location. If a user has activated the Graduate Teaching Assistant role in his office, he cannot activate the Lab Operator role at the same time.

[Strong Form of DSoD] Let x and y be two roles such that $x \neq y$. $(x,y) \in DSOD_s$ if the following condition holds:

$$SessionRole(u,x,s,d,l) \Rightarrow \neg\ (\exists l' \subseteq Universe, \exists d' \subseteq always \bullet SessionRole(u,y,s,d',l'))$$

The above definition says that if roles x and y are related through strong DSoD and if user u has activated role x in some session s, then u can never activate role y in the same session. A user cannot be both an code developer and a code tester in the same session.

6.1.6 Impact of Time and Location on Delegation

Many situations require the temporary delegation of access rights to accomplish a given task. For example, a doctor may give certain privileges to a trained nurse when he is taking a break. In such situations, the doctor can give a subset of his permissions to the nurse for a given period of time. This requirement can be fulfilled by the delegation operation. The entity who delegates his privileges temporarily to another entity is referred to as the delegator. The entity who receives the privilege is known as the delegatee. Delegation can be either *grant* or *transfer*. Granting of privileges allows the delegator to temporarily assign his privileges to the delegatee without relinquishing his own privileges. Transferring of privileges allows the delegator to transfer his privileges temporarily to the delegatee. Note that, during the period of delegation the delegator does not have the privileges which he has transferred to the delegatee.

The delegator can be either a user or a role. System administrators are responsible for overseeing delegation when the delegator is a role. Individual users administer delegation when the delegator is an user. The delegator can delegate either a set of permissions that he possesses by virtue of being assigned to different roles or he can delegate a set of roles assigned to him directly by the user-role assignment or indirectly by the effect of the activation hierarchy. We can therefore classify delegation on the basis of role delegation or permission delegation. For role delegation, the delegatee can be either role or user. For permission delegation, the delegatee can be role only. This is to maintain the intent of RBAC – permissions should be assigned to user via role, not to user directly.

Role Delegation

A delegator (user or role) can delegate a role to a delegatee. Note that, for a delegator to delegate a role r for time d and at location l, the delegator must have been assigned to the role r during time d and location l either directly or indirectly. Depending on the type of delegation (grant or transfer), the delegator may or may not continue to enjoy the privileges he has delegated.

Let $Delegate_R(dtr, dte, r, \{g,t\}, d, l)$ be the predicate that allows the delegator $dtr \in U \cup R$

to grant (g) or transfer (t) a role r to the delegatee $dte \in U \cup R$ during time d and at location l. This will allow individual user (if $dte \in U$) or all users assigned to dte (if $dte \in R$) to be temporary assigned to role r at the specific location and time. The following specifies the various conditions under which user u' acquires role r for duration d' and location l' by virtue of delegation.

1. $Delegate_R(u,u',r,g,d',l') \Rightarrow UserRoleAssign(u',r,d',l')$
2. $Delegate_R(u,u',r,t,d',l') \Rightarrow UserRoleAssign(u',r,d',l') \wedge \neg UserRoleAssign(u,r,d',l')$
3. $Delegate_R(r',u',r,g,d',l') \Rightarrow UserRoleAssign(u',r,d',l')$
4. $Delegate_R(r',u',r,t,d',l') \wedge UserRoleAssign(u,r',d',l')$
   $\Rightarrow UserRoleAssign(u',r,d',l') \wedge \neg UserRoleAssign(u,r,d',l')$
5. $Delegate_R(u,r',r,g,d',l') \wedge UserRoleAssign((u',r',d',l') \Rightarrow UserRoleAssign(u',r,d',l')$
6. $Delegate_R(u,r',r,t,d',l') \wedge UserRoleAssign((u',r',d',l')$
   $\Rightarrow UserRoleAssign(u',r,d',l') \wedge \neg UserRoleAssign(u,r,d',l')$
7. $Delegate_R(r'',r',r,g,d',l') \wedge UserRoleAssign(u',r',d',l') \Rightarrow UserRoleAssign(u',r,d',l')$
8. $Delegate_R(r'',r',r,t,d',l') \wedge UserRoleAssign(u',r',d',l') \Rightarrow UserRoleAssign(u',r,d',l')$
   $\wedge UserRoleAssign(u,r'',d',l') \wedge \neg UserRoleAssign(u,r,d',l')$

The above eight conditions describe how user u' can be assigned to role r for duration d' and location l' under user to user, role to user, user to role and role to role delegation with the grant and transfer mode. Note that, the transfer mode causes the delegator to lose his privileges. With the effect of role activation hierarchy, the delegatee of a delegated role can also activate all junior roles in the activation hierarchy. Moreover, the delegatee inherits all permissions that the delegated role can acquire directly through the permission-role assignment and indirectly through the permission inheritance hierarchy.

Permission Delegation

A delegator (user or role) can delegate a permission to a delegatee. Note that, for a delegator to delegate a permission p for time d and at location l, the delegator must have acquired the privilege r during time d and location l either directly or indirectly. Depending on the type of

delegation grant or transfer, the delegator may or may not continue to enjoy the privileges he has delegated.

Let $Delegate_P(dtr, dte, p, \{g,t\}, d, l)$ be the predicate that allows the delegator $dtr \in U \cup R$ to grant or transfer a permission p to the delegatee $dte \in R$ during time d and at location l. The following specifies the various conditions that allow permission p to be delegated to role r' during time d' and location l'.

1. $Delegate_P(u, r', p, g, d', l') \Rightarrow PermRoleAcquire(p, r', d', l')$
2. $Delegate_P(r, r', p, g, d', l') \Rightarrow PermRoleAcquire(p, r', d', l')$
3. $Delegate_P(r, r', p, t, d', l') \Rightarrow PermRoleAcquire(p, r', d', l') \wedge \neg PermRoleAcquire(p, r, d', l')$

The first two conditions say that if a user u or role r has granted privilege p to role r' for duration d' and location l', then role r' acquires permission p for duration d' and location l'. The last condition says that if a role r' has transferred privilege p to role r' for duration d' and location l', then role r' acquires permission p for duration d' and location l', and role r loses permission p for duration d' and location l'. Note that, we have not specified transfer of privilege from user u to role r'. Since privileges are not directly assigned to any user, permissions cannot be removed directly from the user. The only way to remove permission from a user is to revoke the permission from the role assigned to the user and associated with the permission. However, this will impact all users assigned to this role. Consequently, we do not allow transfer of permission from user to role. Since privileges are not directly assigned to the user, we do not define the permission delegation in which the delegatee is the user.

Delegation Chains

In some cases, the delegator may allow the delegatee to further delegate the privileges that he has acquired by virtue of delegation. This could cause a sequence of delegations called the *delegation chain* or *delegation path* [41, 93]. Once a delegatee is granted a privilege, he can grant or transfer this privilege to another delegatee if the delegation chain is permitted by the delegator. However, if a delegatee is transferred a privilege, he can only transfer it to another delegatee in the presence of the delegation chain. Thus, the transfer operation is more

restrictive than grant operation ($grant \geqslant transfer$). We now formally define the two delegation chains that our model supports: *Monotonic Role Delegation Chain* and *Monotonic Permission Delegation Chain*.

[Monotonic Role Delegation Chain] Monotonic role delegation chain is the delegation chain of the form:

$$\bigwedge_{i=0}^{n-1} Delegate_R(dte_i, dte_{i+1}, r, gt_{i+1}, d_{i+1}, l_{i+1})$$

where dte_0 represents the original delegator, dte_i represents the delegatee in the i^{th} delegation, gt_i refers to grant or transfer, d_i, l_i refers to the time and location where the i^{th} delegation is valid, and $gt_i \geqslant gt_{i-1}$, $d_{i+1} \subseteq d_i$, and $l_{i+1} \subseteq l_i$. The above formalism implies that this delegation will gradually reduce the spatio-temporal points where the delegation can be granted or transferred. We define monotonic permission delegation chain in a similar manner.

[Monotonic Permission Delegation Chain] Monotonic permission delegation chain is the delegation chain of the form:

$$\bigwedge_{i=0}^{n-1} Delegate_P(dte_i, dte_{i+1}, p, gt_{i+1}, d_{i+1}, l_{i+1})$$

where dte_0 represents the original delegator, dte_i represents the delegatee in the i^{th} delegation, gt_i refers to grant or transfer, d_i, l_i refers to the time and location where the i^{th} delegation is valid, and $gt_i \geqslant gt_{i-1}$, $d_{i+1} \subseteq d_i$, and $l_{i+1} \subseteq l_i$.

The delegator may want to define on the length of the delegation chain. Let $\mathcal{DC}(dtr, e)$ denote the delegation chain starting from the original delegator *dtr* with respect to delegated entity *e*. The function *depth* when applied to this delegation chain, that is, $depth(\mathcal{DC}(dtr, e))$ gives the total number of delegation operations that occurs in $\mathcal{DC}(dtr, e)$.

6.2 Graph-Theoretic Representation of the Model

Although our proposed spatio-temporal model is syntaxically strong and can represent the spatio-temporal access control policies needed in the real-world application, we propose a

graph-theoretic representation that accurately reflects the semantics of the model.

Our graph-theoretic representation was inspired by the work of Chen and Crampton [19]. However, we adapt this model to better reflect our semantics. In our work, the set of vertices $V = U \cup R \cup P \cup O$ correspond to the RBAC entities: Users (U), Roles (R), Permissions (P), and Objects (O). The relationships of our spatio-temporal role-based access control model constitute the edges $E = UA \cup PA \cup PO \cup RH \cup SD \cup RD \cup PD$ where

- User-Role Assignment (UA) = $U \times R$
- Permission-Role Assignment (PA) = $R \times P$
- Permission-Object Assignment (PO) = $P \times O$
- Role Hierarchy (RH) = $R \times R$ which can be categorized into
  - activation hierarchy RH_a consisting of unrestricted activation RH_{au}, time restricted activation RH_{at}, location restricted activation RH_{al} and time location restricted activation RH_{atl} hierarchies.
  - permission inheritance hierarchy RH_i consisting of unrestricted permission inheritance RH_{iu}, time restricted permission inheritance RH_{it}, location restricted permission inheritance RH_{il}, and time location permission inheritance RH_{itl} hierarchies.
- Separation of Duty (SD) = $(R \times R) \cup (P \times P)$ which can be categorized into
  - static separation of duty for user role assignments $RSSD$ consisting of weak-form $RSSD_w$, strong temporal form $RSSD_t$, strong spatial form $RSSD_l$ and strong form $RSSD_s$.
  - static separation of duty for permission-role assignment $PSSD$ consisting of weak-form $PSSD_w$, strong temporal form $PSSD_t$, strong spatial form $PSSD_l$ and strong form $PSSD_s$.
  - dynamic separation of duty DSD consisting of weak-form DSD_w, strong temporal form DSD_t, strong spatial form DSD_l and strong form DSD_s.
- Role Delegation (RD), which can be categorized into
  - Role Delegation to User (RD_U) = $U \times R$

– Role Delegation to Role (RD_R) = $R \times R$

- Permission Delegation (PD) = $R \times P$

An *activation path* (or *act-path*) between v_1 and v_n is defined to be a sequence of vertices $v_1, \ldots, v_n$ such that $(v_1, v_2) \in (UA \cup RD_U)$ and $(v_{i-1}, v_i) \in (RH_a \cup RD_R)$ for $i = 3, \ldots, n$. A *usage path* (or *u-path*) between v_1 and v_n is defined to be a sequence of vertices $v_1, \ldots, v_n$ such that $(v_i, v_{i+1}) \in RH_i$ for $i = 1, \ldots, n-2$, and $(v_{n-1}, v_n) \in (PA \cup PD)$. An *access path* (or *acs-path*) between v_1 and v_n is defined to be a sequence of vertices $v_1, \ldots, v_n$, such that (v_1, v_i) is an act-path, (v_i, v_{n-1}) is an u-path, and $(v_{n-1}, v_n) \in PO$. An access path $(v_1, v_2, \ldots, v_{n-1}, v_n)$ allows user v_1 to access object v_n using permission v_{n-1} .

We define two functions, ρ and μ, on the edges E of the graph, where $E = UA \cup PA \cup PO \cup RH \cup SD \cup RD \cup PD$. Function ρ represents information associated with delegation edges and is specified as follows. $\rho : (RD_U \cup RD_R \cup PD) \rightarrow (U \cup R) \times \mathbb{N}$ that maps the delegation edge to the corresponding delegator and delegation depth. If a delegator further delegates his delegated entity, the delegation depth of the newly created delegation edge is calculated by subtracting one from the delegation depth of its immediate preceding delegation edge. μ represents the spatio-temporal constraints associated with all the edges in the graph and is defined as follows. $\mu : E \rightarrow 2^{\mathcal{D}}$ where $\mathcal{D}$ denotes the spatio-temporal domain. For $e = (v, v') \in E$, $\mu(v, v')$ denotes the set of spatio-temporal points at which the association between v and v' is enabled. In the following, we describe the value of μ for each type of edge in our graph.

- if $(u, r) \in UA$, then $\mu(u, r) = \{(d, l) | UserRoleActivate(u, r, d, l)\}$ denotes the set of spatio-temporal points in which user u can activate role r.
- if $(r, p) \in PA$, then $\mu(r, p) = \{(d, l) | PermRoleAcquire(p, r, d, l)\}$ denotes the set of spatio-temporal points in which permission p is assigned to role r.
- if $(p, o) \in PO$, then $\mu(p, o) = PermDur(p) \times PermObjLoc(p, o)$ denotes the set of spatio-temporal points at which object o can be accessed by virtue of permission p.
- if $(r', r) \in RH_{au} \cup RH_{iu}$, then $\mu(r', r) = RoleEnableTimeLoc(r)$ because senior role can activate the junior role, or inherit permissions of junior role at all the spatio-temporal points where the junior role can be enabled.

- if $(r',r) \in RH_{at} \cup RH_{it}$, then $\mu(r',r) = (d', Universe) \cap RoleEnableTimeLoc(r)$, where $r' \succcurlyeq_{(d',Universe)} r$ or $r' \geq_{(d',Universe)} r$, because senior role can activate the junior role, or inherit permissions of junior role when the junior role can be enabled and the hierarchy constraints are satisfied.
- if $(r',r) \in RH_{al} \cup RH_{il}$, then $\mu(r',r) = (Always, l') \cap RoleEnableTimeLoc(r)$, where $r' \succcurlyeq_{(Always,l')} r$ or $r' \geq_{(Always,l')} r$, because senior role can activate the junior role, or inherit permissions where the junior roles can be enabled and the hierarchy spatial constraints are satisfied.
- if $(r',r) \in RH_{atl} \cup RH_{itl}$, then $\mu(r',r) = (d', l') \cap RoleEnableTimeLoc(r)$, where $r' \succcurlyeq_{(d',l')} r$ or $r' \geq_{(d',l')} r$, because senior role can activate the junior role, or inherit permissions where and when both the roles can be enabled, and the spatio-temporal constraints of the hierarchy are satisfied.
- if $(r',r) \in RSSD_w \cup DSD_w$, then $\mu(r',r) = (d,l)$ denotes the set of points in space-time where no user should be assigned/allowed to activate roles r and r'.
- if $(r',r) \in RSSD_t \cup DSD_t$, then $\mu(r',r) = (Always, l)$ because the same user cannot be assigned/allowed to activate roles r and r' at specified location l at any time.
- if $(r',r) \in RSSD_l \cup DSD_l$, then $\mu(r',r) = (d, Universe)$ denotes the spatio-temporal points where the same user cannot be assigned or allowed to activate roles r and r' from any location.
- if $(r',r) \in RSSD_s \cup DSD_s$, then $\mu(r',r) = (Always, Universe)$ because no user can be assigned or allowed to activate roles r and r' from any place and at any time.
- if $(p',p) \in PSSD_w$, then $\mu(p',p) = (d,l)$ denotes the set of points in space-time where no role should be assigned to conflicting permissions p and p'.
- if $(p',p) \in PSSD_t$, then $\mu(p',p) = (Always, l)$ denotes the set of spatio-temporal points where the same role cannot be assigned to conflicting permissions p and p' at any time.
- if $(p',p) \in PSSD_l$, then $\mu(p',p) = (d, Universe)$ denotes the set of spatio-temporal points where the same role cannot be assigned to conflicting permissions p and p' at any location.
- if $(p',p) \in PSSD_s$, then $\mu(p',p) = (Always, Universe)$ because no role can be assigned

to conflicting permissions p and p' from any place and at any time.

- if $(u', r) \in RD_U$, then
  $\mu(u', r) = \{(d, l) | Delegate_R(u, u', r, \{g, t\}, d, l) \vee Delegate_R(r', u', r, \{g, t\}, d, l)\}$ denotes the set of points in space-time where user u' has been delegated role r.
- if $(r', r) \in RD_R$, then
  $\mu(r', r) = \{(d, l) | Delegate_R(u, r', r, \{g, t\}, d, l) \vee Delegate_R(r'', r', r, \{g, t\}, d, l)\}$ denotes the set of points in space-time where role r' has been delegated role r.
- if $(r, p) \in PD$, then
  $\mu(r, p) = \{(d, l) | Delegate_P(u, r, p, g, d, l) \vee Delegate_P(r'', r, p, \{g, t\}, d, l)\}$ denotes the set of points in space-time where role r has acquired permission p.

We write $\hat{\mu}(v_1, \ldots, v_n) = \hat{\mu}(v_1, v_n) \subseteq \mathcal{D}$ to denote $\bigcap_{i=1}^{n-1} \mu(v_i, v_{i+1})$. Hence, $\hat{\mu}(v_1, v_n)$ is the set of points at which every edge in the path is enabled. The authorization scheme in the access control graph can be summarized as follows:

- a user $v \in U$ may activate role $v' \in R$ at point $d \in \mathcal{D}$ if and only if there exists an act-path $v = v_1, v_2, \ldots, v_n = v'$ and $d \in \hat{\mu}(v, v')$;
- a role $v \in R$ is authorized for permission $v' \in P$ at point $d \in \mathcal{D}$ if there exists an u-path $v = v_1, v_2, \ldots, v_n = v'$ and $d \in \hat{\mu}(v, v')$;
- a user $v \in U$ is authorized for permission $v' \in P$ with respect to object $v'' \in O$ at point $d \in \mathcal{D}$ if and only if there exists an acs-path $v = v_1, v_2, \ldots, v_i, \ldots, v_{n-1} = v', v_n = v''$ such that $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_i, \ldots, v_{n-1}$ is an u-path, $(v_{n-1}, v_n) \in PO$ and $d \in \hat{\mu}(v, v'')$.

Note that, generating the access control graph consists of two steps. First, we have to create all vertices corresponding to the entities which takes $O(V)$ time. Next, we have to create all edges corresponding to the relationships between entities and constraints between entities. This step takes $O(E)$ time. Hence, total time to create the whole graph is $O(V + E)$.

6.3 Example Application

In this section, we present a revised version of the real-world application called the Dengue Decision Support (DDS) system previously discussed in Section 4.3.4 to illustrate our approach.

6.3.1 DDS Security Policies

Entities

DDS system consists of the following entities

- Users: *Alice*, *Bob*, *Ben*, *Charlie*, *Claire* and *David*
- Roles: State Epidemiologist (*State Epi*), Jurisdiction Epidemiologist (*Juris Epi*), Clinic Epidemiologist (*Clinic Epi*), Clinician (*Clinic*), State Vector Control (*State VC*), Jurisdiction Vector Control (*Juris VC*), and Local Jurisdiction VC Team (*Local VC Team*).
- Permissions: p_i where $1 \leq i \leq 17$ whose descriptions are given in Table 6.1.
- Objects are omitted from the example to keep it simple.

Table 6.1: DDS Permissions List

	Task		Task
p_1	Read Premise	p_{10}	Read VControl
p_2	Change Premise	p_{11}	Change VControl
p_3	Read Case	p_{12}	Read Work Record
p_4	Change Case	p_{13}	Change Work Record
p_5	Read Patient	p_{14}	Read VC Materials
p_6	Change Patient	p_{15}	Change VC Materials
p_7	Read Patient Names	p_{16}	Signal VC Need for Dengue Virus (DV)
p_8	Read Schedule Work	p_{17}	Signal VC Need for Dengue Hemorrhagic Fever (DHF)
p_9	Change Schedule Work		

Role Assignment The user-role assignments and permission-role assignments are specified as follows.

- User-role assignments: $UserRoleAssign(Alice, State\ Epi, b, A \cup B)$, $UserRoleAssign(Bob, Clinic\ Epi, b, C)$, $UserRoleAssign(Ben, Clinician, a, C)$, and $UserRoleAssign(Charlie, State\ VC, a, A \cup B)$.

Table 6.2: DDS Role-Permission Assignment Constraints

Role	Tasks	Location Constraint	Time Constraint
State Epi	p_{16}	A–State Office, B–Juris Office	a–Regular Hours
Juris Epi	p_1, p_3 p_{17}	B–Juris Office B–Juris Office	a–Regular Hours b–Always
Clinic Epi	p_{17}	D–Universe	b–Always
Clinician	p_1, p_2	C–Clinic	a–Regular Hours
State VC	p_{11}, p_{15}	A–State Office	a–Regular Hours
Juris VC	p_1, p_8	B–Juris Office	a–Regular Hours
Local VC Team	p_7	B–Juris Office, E–Emergency Location	a–Regular Hours, c–Emergency Hours

- Permission-role assignments are summarized in Table 6.2.

Role Hierarchy Two pairs of roles are related by the unrestricted permission inheritance hierarchy. These relationships are specified as follows:

- *State Epi* $\geq_{(Always,Universe)}$ *Juris Epi, State VC* $\geq_{(Always,Universe)}$ *Juris VC* and *Juris VC* $\geq_{(Always,Universe)}$ *Local VC Team.*

Separation of Duty There are three separation of duty constraints in DDS system:

- User should not have permission to change VC protocols at the same time as he has permission to change VC materials.
- User should not have permission to signal DV at the same time as signal DHF.
- User should not be assigned to both Epidemiologist and Vector Control roles at any place and time.

These can be represented as follows:

- $SSOD\_PRA_l = \{(p_{11}, p_{15}), (p_{16}, p_{17})\}$
- $SSOD\_URA_s =$ { (*State Epi,State VC*), (*Juris Epi, State VC*), (*Clinic Epi, State VC*), (*State Epi, Juris VC*), (*Juris Epi, Juris VC*), (*Clinic Epi, Juris VC*) }

Delegation Only one delegation constraint is specified for this application. The system administrator decided to transfer permission p_{17} from *Clinic Epi* role to *Clinician* role during

emergency hours at the clinic. The administrator does not allow the delegatee to delegate the permission further. This can be represented in our model as follows:

- $Delegate_P$(*Clinic Epi, Clinician,* p_{17}, t, c, C)
- $depth(\mathcal{DC}$(*Clinic Epi,* p_{17})) $= 1$

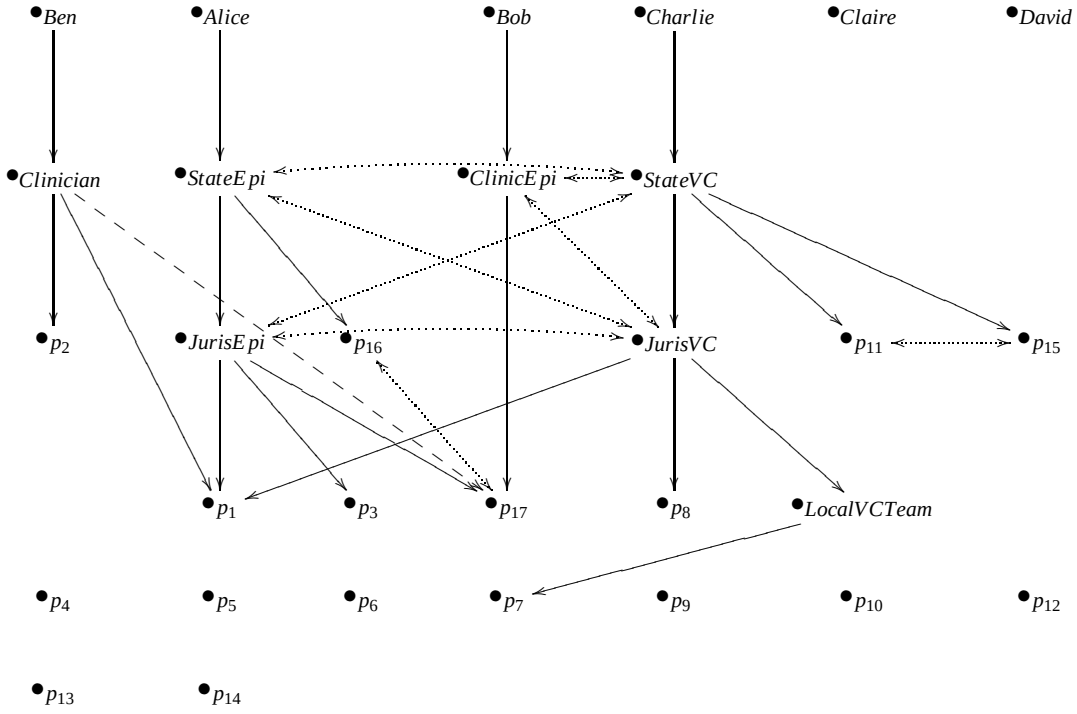


Figure 6.1: DDS System's Access Control Graph

The graph representation of the DDS security policies are shown in Figure 6.1. To avoid crowding the graph, we show the spatio-temporal and delegation constraints in Table 8.2. The *PD* edge is represented by dashed arrow. *SD* edges are represented by dotted bi-directional arrows. The activation paths and their associated spatio-temporal constraints are listed below:

- (*Alice, State Epi*) where $\hat{\mu}$(*Alice, State Epi*) = $[b, A \cup B]$
- (*Ben, Clinician*) where $\hat{\mu}$(*Ben, Clinician*) = $[a, C]$

- (*Bob, Clinic Epi*) where $\hat{\mu}$(*Bob, Clinic Epi*) = $[b, C]$

- (*Charlie, State VC*) where $\hat{\mu}$(*Charlie, State VC*) = $[a, A \cup B]$

Some examples of usage paths and their associated spatio-temporal constraints are given below:

- (*Clinician,* p_1) where $\hat{\mu}$(*Clinician,* p_1) = $[a, C]$

- (*Juris VC, Local VC Team*) where $\hat{\mu}$(*Juris VC, Local VC Team*) = $[b, A]$

- (*State VC, Juris VC,* p_1) where $\hat{\mu}$(*State VC, Juris VC,* p_1) = $[a, B]$

- (*State VC, Juris VC, Local VC Team,* p_7) where $\hat{\mu}$(*State VC, Juris VC, Local VC Team,* p_7) = $[a, \emptyset]$

Some examples of access paths are as follows:

- (*Alice, State Epi,* p_{16}) where $\hat{\mu}$(*Alice, State Epi,* p_{16}) = $[a, A \cup B]$

- (*Bob, Clinic Epi,* p_{17}) where $\hat{\mu}$(*Bob, Clinic Epi,* p_{17}) = $[b, C]$

- (*Charlie, State VC, JurisVC,* p_1) where $\hat{\mu}$(*Charlie, State VC, JurisVC,* p_1) = $[a, B]$

6.4 Chapter Summary

In this chapter, we propose the third model, namely, extended STRBAC model. The model remove the ambiguities from the first version and improve the functionality by including the delegation chains as a new feature. To strengthen the semantics of the model, we propose the methodology to transform the model into the graph-theoretic representation. From here, we can use the graph algorithm discussed in Chapter 5 to detect the conflicts and inconsistencies of the model. However, there is no tool which could advocate this task to be done automatically. To resolve this weakness, in Chapter 7 we introduce the approach to transform the access control graph into the Coloured-Petri Nets (CPNs). By using the CPN analysis application called *CPN Tools*, we can automatically analyze the STRBAC model at the application level.

NAME	DESCRIPTION	CONSTRAINTS	
		μ	ρ
$(Alice, State\ Epi)$	User-Role Assignment	$[b, A \cup B]$	
$(Bob, Clinic\ Epi)$	User-Role Assignment	$[b, C]$	
$(Ben, Clinician)$	User-Role Assignment	$[a, C]$	
$(Charlie, State\ VC)$	User-Role Assignment	$[a, A \cup B]$	
$(State\ Epi, Juris\ Epi)$	Permission Usage Hierarchy	$[b, B]$	
$(State\ VC, Juris\ VC)$	Permission Usage Hierarchy	$[a, B]$	
$(Juris\ VC, Local\ VC\ Team)$	Permission Usage Hierarchy	$[b, A]$	
$(State\ Epi, p_{16})$	Permission-Role Assignment	$[a, A \cup B]$	
$(Juris\ Epi, p_1)$	Permission-Role Assignment	$[a, B]$	
$(Juris\ Epi, p_3)$	Permission-Role Assignment	$[a, B]$	
$(Juris\ Epi, p_{17})$	Permission-Role Assignment	$[b, B]$	
$(Clinic\ Epi, p_{17})$	Permission-Role Assignment	$[b, D]$	
$(Clinician, p_1)$	Permission-Role Assignment	$[a, C]$	
$(Clinician, p_2)$	Permission-Role Assignment	$[a, C]$	
$(State\ VC, p_{11})$	Permission-Role Assignment	$[a, A]$	
$(State\ VC, p_{15})$	Permission-Role Assignment	$[a, A]$	
$(Juris\ VC, p_1)$	Permission-Role Assignment	$[a, B]$	
$(Juris\ VC, p_8)$	Permission-Role Assignment	$[a, B]$	
$(Local\ VC\ Team, p_7)$	Permission-Role Assignment	$[b, B \cup E]$	
$(Clinician, p_{17})$	R2R Permission Delegation	$[c, C]$	[*Clinic Epi*, 1]
$(State\ Epi, State\ VC)$	Role Static SoD	$[b, D]$	
$(State\ Epi, Juris\ VC)$	Role Static SoD	$[b, D]$	
$(Juris\ Epi, State\ VC)$	Role Static SoD	$[b, D]$	
$(Juris\ Epi, Juris\ VC)$	Role Static SoD	$[b, D]$	
$(Clinic\ Epi, State\ VC)$	Role Static SoD	$[b, D]$	
$(Clinic\ Epi, Juris\ VC)$	Role Static SoD	$[b, D]$	
(p_{11}, p_{15})	Permission Static SoD	$[a, D]$	
(p_{16}, p_{17})	Permission Static SoD	$[a, D]$	

Table 6.3: DDS Relationships and Constraints

Chapter 7

The Analysis of an Extended STRBAC Model

The model that we proposed in Chapter 6 has numerous features that can interact with each other to produce inconsistencies and conflicts. For example, incorrect spatio-temporal constraints may prevent a user from invoking his permission. Similarly, incorrect delegation may cause violation of separation of duty constraints. Thus, we must perform an analysis to ensure that inconsistencies or security violations do not occur when a given application is using our model. Manual analysis is error-prone and tedious. Towards this end, we show how Coloured Petri Nets (CPNs) can be used for detecting problems in the authorization specifications.

7.1 Coloured Petri Nets

Coloured Petri Nets (CPN) modelling language [35, 37, 38] has been widely used to model and analyze various domain of real-world applications. In CPN, the states of the system are represented as a set of circles or ellipses called *places*. The events which cause the change between states are represented as a rectangles called *transitions*. Between a state and its corresponding transition which leads to the next state will be linked with a directed arc. Figure 7.1 shows a simple example of coloured petri nets.

Modeling with CPN has a number of benefits. First, with a comprehensive graphical representation and well-defined semantics, CPN allows user to perform the formal analysis. Second, CPN has a well-defined semantics, which allows us to unambiguously define the behavior of

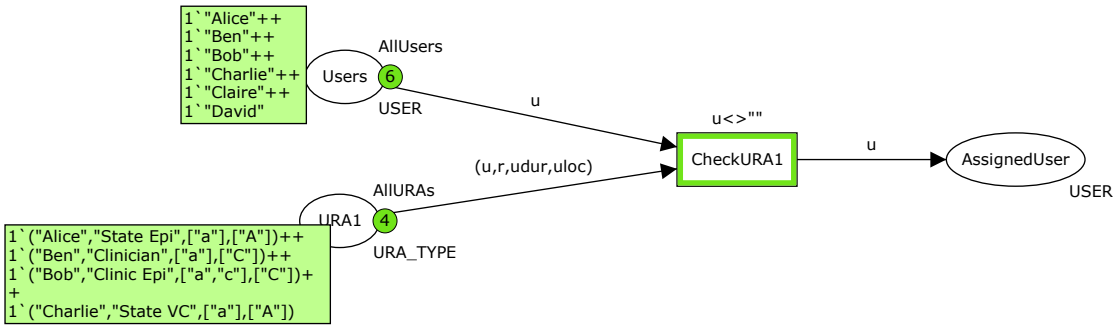


Figure 7.1: Simple example of CPN model

the model. Third, the Modeling Language (ML) supported by CPN is very general and can be used to describe a large various systems. Fourth, the language semantics is built upon true concurrency processing. Fifth, CPN supports a hierarchical description, which is helpful to model in a modular approach, where each system functions are modelled separately as sub CPNs. Finally, CPN also offers interactive simulation and has tools to support the drawing, simulation, and formal analysis.

Laborde et al. [46] have nicely described the characteristics of a CPN as follow. The states of a CPN are represented by the *places*, which are drawn as ellipses or circles. Each place has an associated type, which termed *color set* to determine the data type that the place may contain. A state of a CPN is called a *marking*. A marking consists of a number of tokens transfered to the individual places. Each token carries a value (color), which belongs to the type of the place on which the token resides. The tokens present on a particular place are called the marking of that place. The tokens of a CPN are distinguishable from each other. The marking of a place is, in general, a multi-set of token values. A multi-set is similar to a set, except that there may be several appearances of the same element. This means that a place may have several tokens with the same token value. We can formally define a CPN as follow [37]:

Definition 49 (A Coloured-Petri Net (CPN))

A CPN is a nine-tuple of the form $(P, T, A, \Sigma, V, C, G, E, I)$, where

1. P is a finite set of places.

2. T is a finite set of transitions.

3. $A \subseteq P \times T \cup T \times P$ is a set of directed arcs from places to transitions and from transitions to places.

4. Σ is a finite set of non-empty colour sets, a colour set is comparable to a data type in the programming paradigm.

5. V is a finite set of typed variables such that $Type[v] \in \Sigma$ for all variables $v \in V$.

6. $C : P \rightarrow \Sigma$ is a mapping function that assign colour set to each place.

7. $G : T \rightarrow EXPR_V$ is a guard function that assigns a guard to each transition t such that $Type[G(t)] = Boolean$. The guard is *enabled* and can be *fired* if and only if the guard function $G(t)$ is evaluate to *true*.

8. $E : A \rightarrow EXPR_V$ is an arc expression function that assigns an arc expression to each arc a such that $Type[E(a)] = C(p)_{MS}$, where $C(p)_{MS}$ represents the colour set of the place p connected to the arc a.

9. $I : P \rightarrow EXPR_\emptyset$ is an initialisation function that assigns an initialisation expression (, which evaluates to a set of *tokens* satified the initialisation function) to each place p such that $Type[I(p)] = C(p)_{MS}$.

For the complete version of the formal definition of Coloured Petri Nets, please refer to [37]. In this paper, we advocate the use of the coloured petri nets tool called CPNTools [36, 38] to develop and analyze the CPN model. CPNTools [38], is a computer tool for creating and analyzing the CPN models. Using CPNTools allow us to investigate the behavior of CPN model using the simulation.

7.2 The Extended STRBAC Model Analysis

We propose to use the CPN Tools [38] to detect problems in the access control specification. The CPN Tools allow one to perform *state space* analysis to identify potential problems. CPN

Tools will generate all possible states that are reachable together with the values of environmental variables that cause the change. Checking all the generated states is a time consuming and error-prone task. To solve this problem, we create queries using the Standard ML language [50] to select only those states which have the exact properties that we are interested in. To avoid state explosion, we develop a CPN model for each of the problems that we try to detect. The models are populated using values from the access control graph representing the access control policies of the organization.

We detect the following problems with the access control specification:

- Isolated entity occurs when an entity is not connected to any other entity.
- Infeasible path occurs when a user cannot access a permission or an object in an access path.
- Delegation constraint violation occurs when the spatio-temporal constraints associated with delegation or the delegation depth constraint is violated.
- Separation of duty violation occurs when a user is assigned conflicting roles, when a permission is assigned conflicting roles, or when a user is able to activate conflicting roles.

7.2.1 Isolated Entity Detection

Isolated entity occurs when an entity is disconnected from other entities in the access control graph, thus making it useless with respect to the access control specification. Consider the DDS example discussed in Section 9.4. If we look at the graph in Figure 6.1 representing the access control policies of the DDS, we find that users *Claire* and *David* are not connected to any roles or permissions–these are examples of isolated entities. A similar argument can be made for permissions p_4 and p_5. In our model, we can have three types of isolated entities, corresponding to users, roles, and permissions, as described below.

1. *Type 1:* User who is not assigned to any role,

2. *Type 2:* Role which is not assigned to any permission or junior role, and

3. *Type 3:* Permission which is not assigned to any role.

We develop CPN models to detect each of these types of isolated entities. In the following, we describe how to detect isolated users, that is, isolated entity of Type 1.

```
colset LOCATION = list STRING;
colset DURATION = list STRING;
colset VERTEX = STRING;
colset EDGETYPE = STRING;
colset DEPTH = INT;
colset EDGE = product VERTEX*VERTEX*EDGETYPE*DURATION*LOCATION*
              VERTEX*DEPTH;
```

All types of entities and relationships in our model are represented using color sets. From the declaration above, edge is represented by a tuple of vertices. The color set called *EDGETYPE* is used to distinguish between different types of edges. To represent μ function, we use the product of *DURATION* and *LOCATION*. Similarly to represent ρ function, we use the product of *STRING* and *DEPTH*.

We next model the states of the application that are of interest. The state of the application is represented using CPN's places which are drawn as ellipses or circles. Each place has an associated type, specified using color set, that determines the data type that the place may contain. In Figure 7.2, we have three places denoted by *User Assign Edges*, *Users*, and *Assigned Users* that have data types *USER*, *EDGE* and *USER* respectively. Each state of a CPN is called a marking. The marking of a place is represented by a multi-set of token values. The initial markings, representing the initial states, are initialized using values from the access control graph and are shown in the boxes adjoining the places. For example, the initial marking of the *Users* place, referred to as *AllUsers*, consists of six tokens corresponding to the users *Alice, Ben, Bob, Charlie, Claire* and *David* in the access control graph. *AllUsers* is described using a multi-set. Since all users are unique, the number of each multi-set member equals one. For example, the notation, `1‘("Alice")` indicates there is only one user Alice. The union operation

(++) is used to represent situations when there are more than one member, as in our example. The initial marking of place *User Assign Edges*, referred to as *AllUserAssign*, is specified in a similar manner and are populated using User-Role Assignment and Role to User Delegation edges from the access control graph. Here, we repeat the specifications of the initial markings.

```
val AllUsers=1‘("Alice")++1‘("Bob")++1‘("Ben")++1‘("Charlie")++
1‘("Claire")++1‘("David");

val AllUserAssign=1‘("Alice","State Epi","UA",["a","c"], ["A","B"],"",0)++
1‘("Bob","Clinic Epi","UA",["a","c"], ["C"],"",0)++
1‘("Ben","Clinician","UA",["a"], ["C"],"",0)++
1‘("Charlie","State VC","UA",["a"], ["A","B"],"",0);
```

The actions of the CPN are described by transitions, which are represented using rectangles. Arcs connect transitions and places. An activation (firing) of a transition removes tokens from places connected to the transition's incoming arcs (input places) and adds tokens to the places connected to the transition's outgoing arcs (output places). This results in the markings of the CPN, that symbolizes its state, to change. It is also possible to attach a boolean expression, referred to as a guard, to each transition. In such a case, the guard function must evaluate to true before it can be activated. The exact number of tokens added or removed by the firing of a transition and their respective data values are determined by the arc expressions. The transitions can be fired repeatedly. When the marking of a place can no longer be changed, it is referred to as dead marking.

Figure 7.2, shows one transition *Move Assigned User* that is activated when the arc expressions match on the *u* values and the guard function of *Move Assigned User* verifies that *u* is not null. The initial markings cause this transition to be fired. The corresponding *u*, (*u, v, etype, d*1, *l*1, *dtr, depth*) get removed from *Users* and *User Assign Edges* places respectively and *u* gets added to *Assigned Users*. The transitions are fired repeatedly until no more state change can take place. In the given example, the transitions are fired for users *Alice, Ben, Bob* and *Charlie*. The terminal state is reached when no more transitions can be fired.

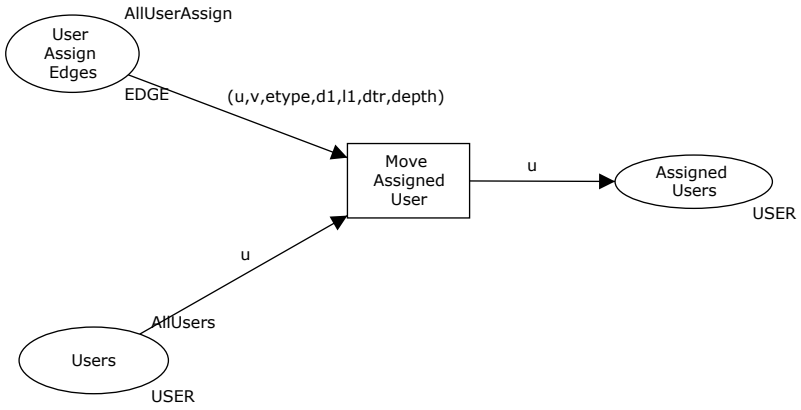


Figure 7.2: CPN Model for Isolated Entity Detection (Type 1)

Query 1

Show all terminal states

```
SearchNodes (EntireGraph,
fn n => (length(OutArcs(n)) = 0),
NoLimit,
fn n => n,
[],
op ::)
```

We use Query 1, which is the general query to show all terminal states, to detect isolated entity. This query is written using built-in query function of the State Space Tool called *SearchNodes* [36]. The first argument in *SearchNodes*, namely, `EntireGraph`, signify that we want to search the whole graph. The second argument, `fn n => (length(OutArcs(n)) = 0)`, states that we want to check all nodes that have no outgoing arcs, that is, the terminal nodes. The third argument, `NoLimit`, states that we want the query to return all possible results. The fourth argument, `fn n => n`, states that we do not want to change the value of the search result. The fifth argument states that the initial value of the result set is equal to empty list. The last argument, `op ::`, will combine all search results into one list. From the explanation above, Query 1 will return the state where the transition cannot proceed anymore. The result is the

state number 16 which can be viewed using the command: `print(NodeDescriptor 16)`. The content of each place in state number 16 can be shown below:

```
Users=1‘("Claire")++1‘("David");


User Assign Edges=empty;


Assigned User=1‘("Alice")++1‘("Ben")++1‘("Bob")++1‘("Charlie");
```

The result shows that tokens corresponding to users *Claire* and *David* are in places *Users* when the transitions cannot be fired anymore. These users cannot be transferred to the next state (*Assigned User*) and they are isolated entities. With trivial modification, we can develop the CPN models to detect the other types of isolated entity.

7.2.2 Infeasible Path Detection

Recall that in an access control graph, a user u is authorized for permission p through role r if there is an access path connecting u, r, and p. The spatio-temporal constraints may be specified in such a manner that it may not be possible for u to invoke r resulting in an infeasible path. Consider the following access path given in Figure 6.1: $(Ben, Clinician, p_{17})$. *Ben* is assigned to *Clinician* role during regular hours at the *Clinic*. However, the *Clinician* role is delegated permission p_{17} only during emergency hours at the *Clinic*. Thus, the temporal constraints prohibit *Ben* from ever invoking permission p_{17}. This is an example infeasible path.

Figure 7.3 shows the CPN model for detecting infeasible paths. This model is developed to perform a depth first search on the access control graph and calculate the $\hat{\mu}$ function of each acs-path. If there is an acs-path where the $\hat{\mu}$ function equal to empty set, then this acs-path is the infeasible path. In this CPN model we have a transition called *Get Initial Vertex*. This transition will get the first token needed to start the analysis. Moreover, it will prevent other tokens from being retrieved while the previous token is still in the analysis process. The transition *Retrieve Edge* will retrieve the authorization edge which starts at $v1$, then add it to

the *Authorization Path* place as a record. Then the transition *Calculate Mu Hat* will calculate the current $\hat{\mu}$ value. If either the spatial value or temporal value of $\hat{\mu}$ equals empty set, it will trigger the *Infeasible Path* transition to fire. This transition will send boolean value true to the *Infeasible Path* place, which will notify us that there exists an infeasible path in our policy. The initial marking of the *Users* place, denoted by *AllUsers*, consists of all users in the access control model. The initial marking of the *Authorization Edges* place, denoted by *AllAuthEdges*, consists of all edges except the SoD edges in the access control graph.

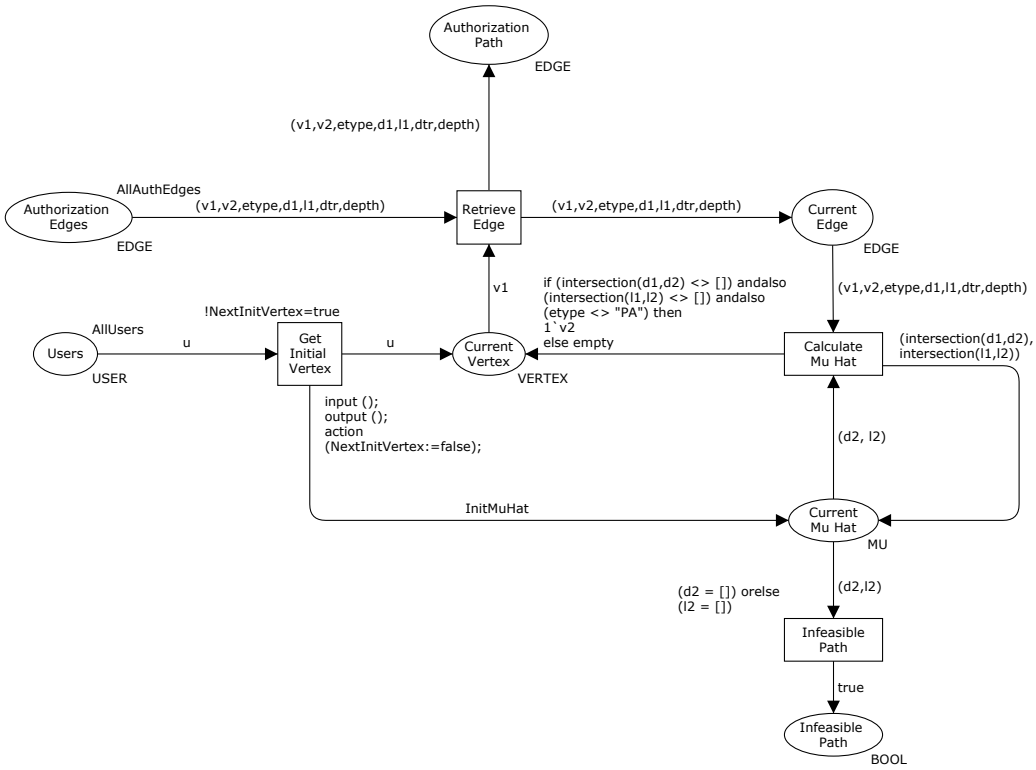


Figure 7.3: CPN Model for Infeasible Path Detection

Query 2

Infeasible Path

```
fun InfeasiblePath() : Node list
= SearchNodes(
EntireGraph,
fn n=>(
(size(Mark.UserInfeasiblePath’Infeasible_Path 1 n) <> 0)
```

```
),
NoLimit,
fn n=>n,
[],
op ::)
```

Query 2 checks the infeasible path that may occur due to incorrect specifications in the spatio-temporal constraints. The second argument in SearchNodes which represents a function states that we want to check the states where the number of tokens in `Infeasible Path` place is not equal to zero. The result shows that states 37 and 47 contain the infeasible path. To observe the result, we print the content of state number 47. Below is part of the content of state 47.

```
Authorization Path = 1‘("Charlie","State VC","UA",["a"], ["A","B"],"",0)++
1‘("State VC","Juris VC","RHI",["a"], ["B"],"",0)++
1‘("Juris VC","Local VC Team","RHI",["a","c"], ["A"],"",0);
```

The analysis reveals another infeasible path that exists in our DDS example: (*Charlie, State VC, Juris VC, Local VC Team*). This infeasible path is caused because no spatial constraints can be satisfied. *Charlie* is assigned the role *State VC* in the *State Office* and *Juris Office*. However, the *State VC* inherits *Juris VC*'s permissions only in the *Juris Office* and *Juris VC* inherits *Local VC Team*'s permission only in *State Office*. This prevents *Charlie* from invoking any of the *Local VC Team*'s permission. State 37 reveals another infeasible path (*Ben,Clinician, p17*) that exists in our application.

7.2.3 Delegation Constraint Violation Detection

A delegator can delegate only the roles or privileges assigned to him. Moreover, the delegation duration and location should satisfy the associated spatio-temporal constraints. In the context of our example, if *Clinic Epi* tried to delegate privilege p_3 (which he does not possess), then it would be an example of delegation constraint violation. Similarly, if the role *Juris*

Epi delegated permission p_3 to *Clinician* at location *A* (State Office) and time *c* (Emergency Hours), then it would violate the delegation constraint. This is because the role *Juris Epi* does not have permission p_3 in location *A* at time *c*.

The delegation should also not violate the delegation depth constraint. This type of violation occurs when there is a chain of delegation and the delegatee further delegates the privilege beyond the specified depth. For example, if the delegation depth is specified as one, then a delegation depth violation will occur if the delegatee is trying to further delegate the privileges he has acquired by virtue of delegation. In the context of our example, the role *Clinic Epi* transfers the permission p_{17} to *Clinician* at time *c* and location *C* and the delegation depth is specified as 1. Now, if the *Clinician* further delegates privilege p_{17} to some other role, then the delegation depth constraint will be violated.

Figure 7.4 shows the CPN model to detect the delegation constraint violation. This model is developed to ensure that both delegation depth constraint and delegation spatio-temporal constraint are satisfied by using the guard function of the transition *Check Delegation Depth* and *Check Delegation Constraint*, respectively. If the *Check Delegation Depth* transition is activated, then there exists a delegation depth violation. Similarly, if the *Check Delegation Constraint* is activated, then there exists a spatio-temporal delegation constraint violation, which will occurs if the delegator delegate the privileges that he has no accessibility. The model will send the problematic edge to the place corresponding to each type of error to notify the error.

The initial markings of the *Delegation Edges* place, denoted by *AllDelEdges*, consists of all delegation edges, that is, $RD \cup PD$. The initial markings of the *Delegator Authorization Edges* place, denoted by *AllDtrAuthEdges*, consists of all edges belonging to $UA \cup RH \cup PA$.

We then formulate Query 3 and 4 to check the delegation depth violation and the delegation constraint violation states, respectively. Both queries return empty list, which ensures that our model is free from both types of violation.

Query 3

Delegation Depth Violation

```
fun DepthViolation() : Node list
```

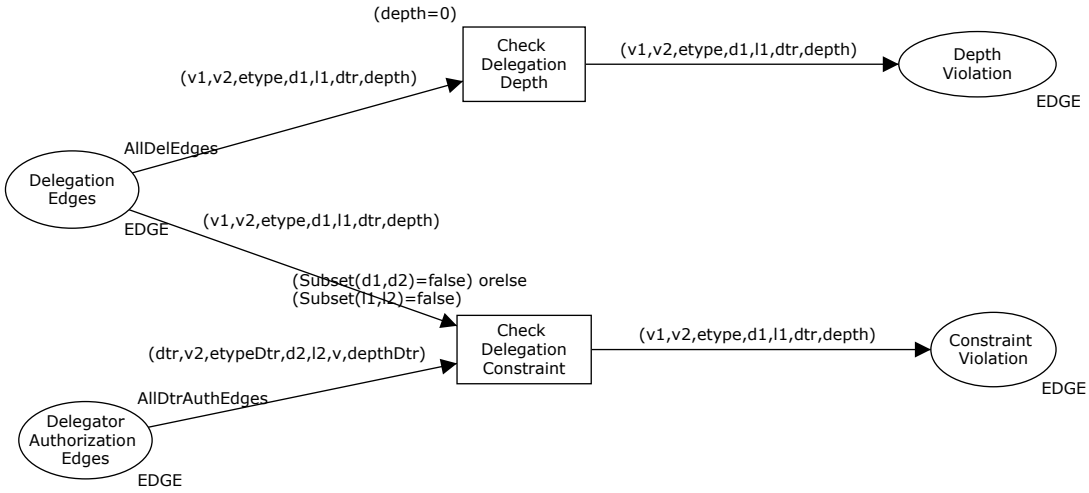


Figure 7.4: CPN Model for Delegation Constraint Violation Detection

```
= SearchNodes(
EntireGraph,
fn n=>(
(size(Mark.DelegationConstraint’Depth_Violation 1 n) <> 0)),
NoLimit,
fn n=>n,
[],
op ::)
```

Query 4

Delegation Constraint Violation

```
fun ConstraintViolation() : Node list
= SearchNodes(
EntireGraph,
fn n=>(
(size(Mark.DelegationConstraint’Constraint_Violation 1 n) <> 0)),
NoLimit,
fn n=>n,
[],
```

```
op ::)
```

7.2.4 SoD Violation Detection

Separation of duty violations can be static or dynamic. Static separation of duty can be with respect to the user-role assignment or permission-role assignment. In DDS system we have two different types of SSoDone with respect to user-role assignment and the other with respect to permission-role assignment. Let us take the example of SSoD for permission-role assignment. No role should have permissions p_{16} (Signal VC for Dengue Virus) and p_{17} (Signal VC for Dengue Hemorrhagic Fever) at the same time. Thus, if a role does have these conflicting permissions , SSoD will be violated.

Figure 7.5 shows the CPN model to detect the separation of duty violation. The model will perform a reverse depth first search starting from the vertices associated with the SoD edge. The ancestors of the two vertices will be stored in two separate places called *V1 Ancestors* and *V2 Ancestors* respectively, together with their corresponding $\hat{\mu}$ value. If there exist a common ancestor and there is an overlap of spatio-temporal points, then SoD is violated. The model will then send the problematic SoD and its ancestor to *SoD Violate* and *SoD Violate Ancestor* places respectively to notify the error.

The initial marking of the *SoD Edges*, denoted by *AllSoDEdges* is populated by all SoD edges in the access control model. The initial markings of *Authorization Edges V1* and *Authorization Edges V2*, denoted by *AllAuthEdges*, consists of all edges except the SoD edges in the access control graph. The content of *AllSoDEdges* is shown below.

```
val AllSoDEdges=
1‘("State Epi","State VC","RSSD",["a","c"],["A","B","C","E"],"",0)++
1‘("State Epi","Juris VC","RSSD",["a","c"],["A","B","C","E"],"",0)++
1‘("Juris Epi","State VC","RSSD",["a","c"],["A","B","C","E"],"",0)++
1‘("Juris Epi","Juris VC","RSSD",["a","c"],["A","B","C","E"],"",0)++
1‘("Clinic Epi","State VC","RSSD",["a","c"],["A","B","C","E"],"",0)++
1‘("Clinic Epi","Juris VC","RSSD",["a","c"],["A","B","C","E"],"",0)++
```

```
1‘("p11","p15","PSSD",["a"],["A","B","C","E"],"",0)++
1‘("p16","p17","PSSD",["a"],["A","B","C","E"],"",0);
```

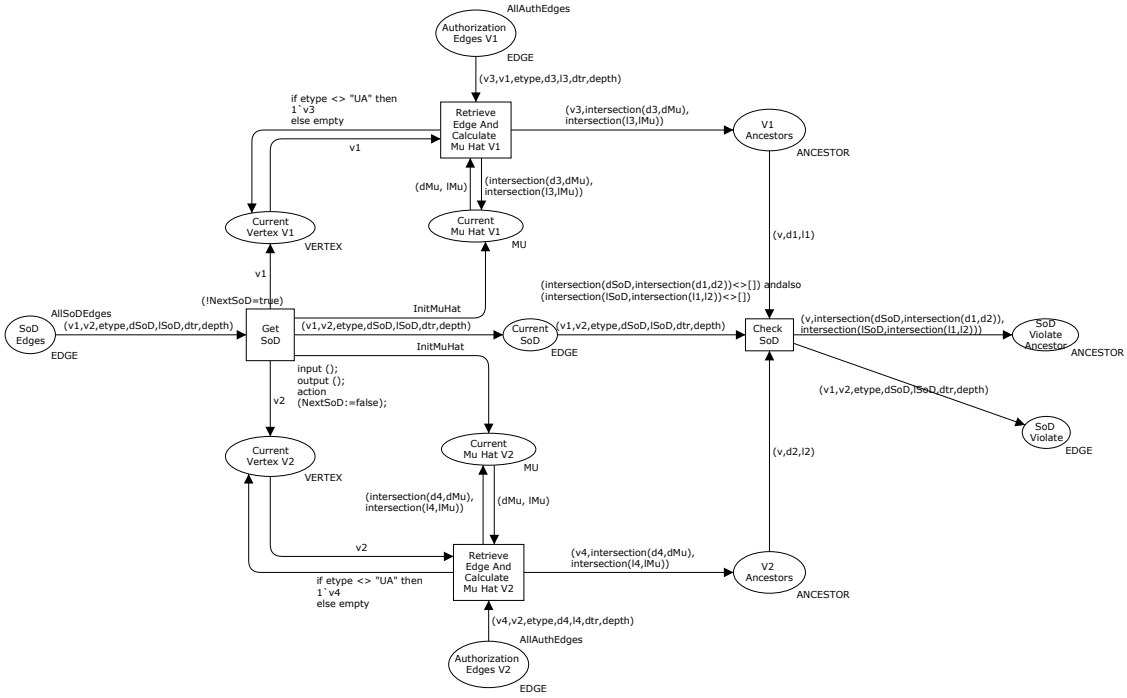


Figure 7.5: CPN Model for Separation of Duty Violation Detection

Query 5

SoD Violation

```
fun SOD() : Node list
= SearchNodes(
EntireGraph,
fn n=>(
(size(Mark.SoD’SoD_Violate 1 n) <> 0)),
NoLimit,
fn n=>n,
[],
op ::)
```

We then formulate Query 5 to check the SoD violation. Our analysis reveals various SoD violations. For example, there is a SoD violation caused by assigning the role *State VC* two conflicting permissions p_{11} and p_{15}. Similarly, there is another SoD violation caused because role *State Epi* gets conflicting permissions p_{16} and p_{17}. Our analysis reveals that there is no SoD violation caused by any user being assigned conflicting roles.

7.2.5 Soundness and Completeness

The set of problems that we considered in this work are by no means exhaustive. For example, it is quite possible that the spatio-temporal constraints have been incorrectly specified but this error does not lead to isolated entities, infeasible path, SoD violation or delegation constraint violation and will not be detected. However, with respect to the problems that we do detect, we can make a few comments about the soundness and completeness. If the CPN model has been correctly constructed and populated using the access control graph, then we can prove soundness and completeness properties with respect to the given problem.

Consider, for example, the problem of detecting isolated users as shown in Figure 7.2. Let us recall how this CPN will operate. The initial marking *AllUsers* are populated using the user entities in the access control graph. Similarly, the initial marking *AllUserAssign* are initialized using *UA* and RD_U edges in the access control graph. The transition *Move Assigned User* will fire as long as some user u matches the user u in the edge $(u, v, etype, d1, l1, dtr, depth)$. This firing results in removing u and $(u, v, etype, d1, l1, dtr, depth)$ from *Users* and *User Assign Edges* respectively and adding u to *Assigned Users*. When no more transitions can be fired, the terminal state has been reached and the place *Users* contain isolated users.

Suppose there is some isolated user u_i in the access control graph that is not detected by this CPN model. In other words, user u_i is not in the *Users* place when the terminal state is reached. This leads to two possibilities: either user u_i is in *Assigned Users* place in the terminal state or it is not. If u_i is in *Assigned Users*, then there exists an edge of type $(u_i, v, etype, d1, l1, dtr, depth)$ in the initial marking *AllUserAssign*. This is possible only if there is a corresponding *UA* or RD_U edge in the access control graph involving u_i; this, in turn, precludes u_i from being an

isolated user. If u_i is not in *Assigned Users* in the terminal state and it is also not in *Users*, then u_i was not in the initial marking *AllUsers*. This is possible only if the access control graph does not contain u_i. Since both the cases are not possible, it means that user u_i must be in the *Users* places when the terminal state is reached. Thus, all isolated users are detected by the CPN model.

Suppose u_j is detected as an isolated user by the CPN. This implies that u_j is in the place *Users* in the terminal state. In other words, there is no edge of the form $(u_j, v, etype, d1, l1, dtr, depth)$ in the initial marking *AllUserAssign*. In other words, there is no *UA* edge or RD_U edge associated with u_j in the access control graph. This implies that u_j is indeed an isolated user.

7.3 Improving the Analysis Performance

CPN explores the state space to check for violations of access control properties. Our investigations reveal that even a modest increase in the number of places and transitions cause a significant increment to the number of states of the state space; this substantially raises the verification time. We looked at the various CPN models that we generated for detecting problems with the access control specifications. We observed that the number of states generated in the CPN model were related to the number of edges traversed in the access control graph for detecting a specific problem. We looked at the number of states generated for each problem. For detecting delegation constraint violation, the number of states generated is of the order $O(|PD| + |RD|)$, where $|PD|$ and $|RD|$ represent the number of permission delegation edges and role delegation edges respectively. Since typically the number of delegation edges will be small, we did not think it necessary to produce further optimization. We next considered the problem of detecting infeasible paths. In this case, the number of states generated is of the order $O(|U||E| + |IP|)$ where $|U|$ is the number of users, $|E|$ is the number of edges in the access path, and $|IP|$ is the number of infeasible paths. Next, consider the problem of detecting SoD violations. Here, the number of states generated is of the order of $O(|SD||E| + |SoD|)$ where $|SD|$ is the number of SoD edges, $|E|$ is the number of edges in the access path, and $|SoD|$ is the number of SoD violations. Thus, one way to reduce the number of states is to decrease the

number of edges in the graph.

One way of reducing the number of edges is to flatten the hierarchy. We did some initial experiments in order to understand the effect of flattening the hierarchy on the state space. We created a very simple access control graph consisting of one user, one user-role assignment, one permission-role assignment, and multiple levels of hierarchy. With 10 levels of hierarchy the state space reduction was 40%, which is quite significant. T his motivated us to transform the access control graph to a smaller graph, which we term, the *privilege acquisition graph*.

7.3.1 Privilege Acquisition Graph

In order to generate a smaller number of states in the CPN model that does efficient verification, we propose to transform the access control graph into the privilege acquisition graph. The privilege acquisition graph essentially flattens out the hierarchical structure. It captures the following relationships: UA', PA', PO' and SD' where UA' represents the user-role assignment that occurs either directly or indirectly via hierarchy and delegation constraints, PA' represents permission-role assignment that occurs either directly or indirectly due to inheritance and delegation, PO' corresponds to the permission-object relationship (represented by PO in the access control graph), and SD' corresponds to separation of duty (represented by SD in the access control graph). Algorithm 1 shows the transformation process. Step 1 adds all the vertices of the access control graph to the privilege acquisition graph. Step 2 converts all the act-path in the access control graph to UA' edges in the privilege acquisition graph. Step 3 converts all the u-path in the access control graph to PA' edges in the privilege acquisition graph. Steps 4 and 5 adds all the PO, SD edges in the access control graph to PO', SD' edges in the privilege acquisition graph respectively. The time complexity to generate the privilege acquisition graph is $O(VE)$, where V is the number of vertices and E is the number of edges of the original access control graph.

Theorem 1

The role authorizations and user authorizations are equivalent in the access control graph $G(V, E, \mu, \rho)$ and its corresponding privilege acquisition graph $G'(V', E', \mu')$.

Proof First, let us consider the case of role authorizations. Suppose role $v \in R$ is authorized for permission $v' \in P$ in the access control graph G. This is possible if there exists an u-path in the access control graph $(v, v_1, v_2, \ldots, v_n, v')$ and $\hat{\mu}(v, v') \neq \emptyset$. By step 3 of Algorithm 1, if there exists an u-path $(v, v_1, v_2, \ldots, v_n, v')$ in the original graph G, it will be transformed to a PA' edge in its corresponding privilege acquisition graph G' with the same spatio-temporal constraint ($\mu'(v, v') = \hat{\mu}(v, v')$). Hence, for every u-path in G, there exists a PA' edge in G' that authorizes v to acquire permission v' at $\hat{\mu}(v, v')$. To show the converse, let us consider an edge $(v, v') \in PA'$ in G'. Since edge $(v, v') \in PA'$ in G' is created from some u-path in G, every role authorization in G' has a corresponding u-path in G that gives role v permission v' at the same spatio-temporal points. Thus, for every edge $(v, v') \in PA'$, there exists an u-path in G that gives role v permission v'.

Next, let us consider user authorizations. Let user $v \in U$ be authorized for permission $v' \in P$ with respect to object $v'' \in O$ in the access control graph G. This is possible if there exists an acs-path $(v, v_1, v_2, \ldots, v_i, \ldots, v', v'')$ such that $v_i \in R$ for some i, $(v_1, \ldots, v_i)$ is an act-path, $(v_i, \ldots, v')$ is an u-path, $(v', v'') \in PO$ and $\hat{\mu}(v, v'') \neq \emptyset$. Corresponding to this acs-path, the algorithm to generate the privilege acquisition graph creates three edges in Steps 2, 3, and 4. The edges created are $(v, v_i) \in UA'$ where $\mu'(v, v_i) = \hat{\mu}(v, v_i)$, $(v_i, v') \in PA'$ where $\mu'(v_i, v') = \hat{\mu}(v_i, v')$, and $(v', v'') \in PO'$ where $\mu'(v', v'') = \hat{\mu}(v', v'')$. Moreover, $\mu'(v, v_i) \cap \mu'(v_i, v') \cap \mu'(v', v'') = \hat{\mu}(v, v'')$. These three edges $(v, v_i), (v_i, v')$ and (v', v'') give the user v permission v' to access object v'' at points $\hat{\mu}(v, v'')$ in graph G'. To prove the converse, let us assume that the privilege acquisition graph G' provides some user u permission p for object o. This implies that there exists three edges of the $(u, r) \in UA'$, $(r, p) \in PA'$ and $(p, o) \in PO'$ and $\mu'(u, r) \cap \mu'(r, p) \cap \mu'(p, o) \neq \emptyset$. The existence of these three edges is possible only if there is an act-path $(u, v_1, v_2, \ldots, v_n, r)$, an u-path $(r, v'_1, v'_2, \ldots, v'_m, p)$, an edge $(p, o) \in PO$ in the corresponding access control graph. Moreover, $\mu'(u, r) = \hat{\mu}(u, r), \mu'(r, p) = \hat{\mu}(r, p)$, and $\mu'(p, o) = \mu(p, o)$. Thus, user u will get permission p to access object o at the same spatio-temporal points in graph G.

Lemma 1

Each isolated entity that exists in the access control graph G is also present in the corresponding

privilege acquisition graph G' and vice-versa.

Proof Let u_i be an isolated user in the access control graph. This means that there is no act-path starting at u_i. Consequently, there is no edge in UA' in the privilege acquisition graph of the form (u_i, v). Since the edges in UA' are the only edges joining users to roles in the privilege acquisition graph, u_i is also an isolated entity in the privilege graph. Conversely, let u_j be an isolated user in the privilege acquisition graph. Thus, there is no edge of the form (u_j, v) in UA'. This is possible only if there is not act-path starting at u_j in the access control graph, which implies that u_j is an isolated entity in the access control graph. We can make similar arguments for isolated roles and permissions.

Lemma 2

For each infeasible path that exists in the access control graph G, there exists a corresponding infeasible path in the privilege acquisition graph G' and vice-versa.

Proof Let $P = (v_1, v_2, \ldots, v_n)$ be an infeasible path in the access control graph where $(v_1, \ldots, v_i)$ is an act-path, $(v_i, \ldots, v_{n-1})$ is an u-path and (v_{n-1}, v_n) be in PO. Since P is an infeasible path, $\hat{\mu}(v_1, v_n) = \hat{\mu}(v_1, v_i) \cap \hat{\mu}(v_i, v_{n-1}) \cap \hat{\mu}(v_{n-1}, v_n) = (d, l)$ where either $d = \emptyset$ or $l = \emptyset$. The construction of the privilege graph from this acquisition graph generates the the following edges: $(v_1, v_i), (v_i, v_{n-1})$, and (v_{n-1}, v_n) where $(v_1, vi) \in UA', (v_i, v_{n-1}) \in PA'$ and $(v_{n-1}, v_n) \in PO$ and $\mu'(v_1, v_i) = \hat{\mu}(v_1, v_i), \mu'(v_i, v_{n-1}) = \hat{\mu}(v_i, v_{n-1})$, and $\mu'(v_{n-1}, v_n) = \hat{\mu}(vn-1, v_n)$. Thus, $\hat{\mu}'(v_1, v_i) = \hat{\mu}(v_1, v_n) = \hat{\mu}(v_1, v_i) \cap \hat{\mu}(v_i, v_{n-1}) \cap \hat{\mu}(v_{n-1}, v_n) = (d, l)$ where either $d = \emptyset$ or $l = \emptyset$. Thus, the path (v_1, v_i, v_{n-1}, v_n) is an infeasible path in the privilege acquisition graph. The converse can be similarly proved.

Lemma 3

For every SoD violation that exists in the access control graph G, there exists a corresponding SoD violation in the privilege acquisition graph G' and vice-versa.

Proof Suppose the access control graph has a SSoD role-permission violation of the form $(r_1, r_2, \ldots, r_n, p_i), (r_1, r'_2, \ldots, r'_n, p_j)$ and (p_i, p_j) where $(r_1, r_2, \ldots, r_n, p_i)$ and $(r_1, r'_2, \ldots, r'_n, p_j)$ are u-paths and (p_i, p_j) is a SoD edge and $\hat{\mu}(r_1, p_i) \cap \hat{\mu}(r_1, p_j) \cap \mu(p_i, p_j) = (d, l)$ where $d \neq \emptyset$

and $l \neq \emptyset$. By construction, the following edges are generated for the privilege acquisition graph: (r_1, p_i) and (r_1, p_j) are edges in PA' and (p_i, p_j) is an edge in SD'. Since $\mu'(r_1, p_i) = \hat{\mu}(r_1, p_i), \mu'(r_1, p_j) = \hat{\mu}(r_1, p_j)$, and $\mu'(p_i, p_j) = \mu(p_i, p_j)$, we have $\mu'(r_1, p_i) \cap \mu'(r_1, p_j) \cap \mu'(p_i, p_j) = (d, l)$. Thus, the edges $(r_1, p_i), (r_1, p_j)$ and (p_i, p_j) indicate there is a SoD violation. The converse can be proved in a similar manner. We can also prove the other types of SoD constraint violations similarly.

Theorem 2

The privilege acquisition graph accurately captures isolated entities, infeasible paths, and SoD violations.

Proof The proof follows from Lemmas 1, 2, and 3.

Note that, the privilege acquisition graph contains less information than the corresponding access control graph. For example, information about the role hierarchy is no longer present in the privilege acquisition graph. The CPN analysis of privilege acquisition graphs will detect the problems, but it may not have enough information to identify the source of the problem. Thus, if a problem exists, the access control graph or its subgraph related to the problem must be analyzed. For instance, if the analysis of the CPN corresponding to the privilege acquisition graph identifies that there is an infeasible path (v_1, v_i, v_{n-1}, v_n), then to detect where the spatio-temporal constraints have been violated we need to find the subgraph of the access control graph involving these vertices and analyze it. Similarly, if the CPN analysis of the privilege acquisition graph reveals a potential SoD violation involving edges $(r_1, p_i), (r_1, p_j)$, and (p_i, p_j), the corresponding subgraph of the access control graph must be analyzed to identify the source of the problem.

7.3.2 DDS Example Privilege Acquisition Graph

We use Algorithm 1 to transform the access control graph of the dengue decision support system into the privilege acquisition graph, shown in Figure 7.6. The new spatio-temporal constraints can be calculated from the $\hat{\mu}(v, v')$ function as described in Algorithm 1. For

Algorithm 1 Transform access control graph to privilege acquisition graph

{**Input:** Access control graph $G(V,E,\mu,\rho)$}
{**Output:** Privilege acquisition graph $G'(V',E',\mu')$}
BEGIN
$V' \leftarrow \emptyset$
$E' \leftarrow \emptyset$
$\mu' \leftarrow \emptyset$
{**Step 1:** Add all vertices of the access control graph to the privilege acquisition graph}
for all $v \in V$ **do**
  $V' \leftarrow V' \cup v$
end for
{**Step 2:** Transform all act-path started at each user vertex of the access control graph to the set of edges of the privilege acquisition graph (UA')}
for all $v \in U$ **do**
  for all act-path $act_i = (v,\ldots,v')$ **do**
    $EdgeType(v,v') \leftarrow UserRoleAuth$
    $E' \leftarrow E' \cup (v,v')$
    $\mu'(v,v') \leftarrow \hat{\mu}(v,v')$
    $\mu' \leftarrow \mu' \cup \mu'(v,v')$
  end for
end for
{**Step 3:** Transform all u-path started at each role vertex and ended at the permission vertex of the access control graph to the set of edges of the privilege acquisition graph (PA')}
for all $(v \in R) \wedge (v' \in P)$ **do**
  for all u-path $u_i = (v,\ldots,v')$ **do**
    $EdgeType(v,v') \leftarrow PermRoleAuth$
    $E' \leftarrow E' \cup (v,v')$
    $\mu'(v,v') \leftarrow \hat{\mu}(v,v')$
    $\mu' \leftarrow \mu' \cup \mu'(v,v')$
  end for
end for
{**Step 4:** Add all PO edges from the access control graph to the set of edges of the privilege acquisition graph (PO')}
for all $(v,v') \in PO$ **do**
  $EdgeType(v,v') \leftarrow PermObjAssign$
  $E' \leftarrow E' \cup (v,v')$
  $\mu'(v,v') \leftarrow \mu(v,v')$
  $\mu' \leftarrow \mu' \cup \mu'(v,v')$
end for
{**Step 5:** Add all SD edges from the access control graph to the set of edges of the privilege acquisition graph (SD')}
for all SD edge $sd_i = (v,v')$ **do**
  $EdgeType(v,v') \leftarrow EdgeType(sd_i)$
  $E' \leftarrow E' \cup (v,v')$
  $\mu'(v,v') \leftarrow \mu(v,v')$
  $\mu' \leftarrow \mu' \cup \mu'(v,v')$
end for
Return $G'(V',E',\mu')$
END

instance, $\mu(\textit{State Epi}, p_1)$ in the condensed graph can be calculated from $\hat{\mu}(\textit{State Epi}, p_1)$ of the original access control graph, which equals to $\mu(\textit{State Epi}, \textit{Juris Epi}) \cap \mu(\textit{Juris Epi}, p_1) = [b, B] \cap [a, B] = [b \cap a, B \cap B] = [a, B]$. Note that in this example duration b means *Always*, hence, $b \cap a = a$. We compute other spatio-temporal constraints in the same manner. All new spatio-temporal constraints are shown in Table 7.1.

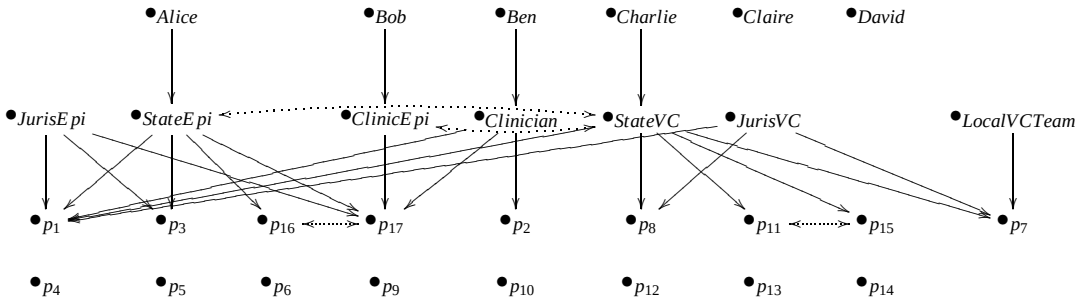


Figure 7.6: DDS System's Privilege Acquisition Graph

NAME	DESCRIPTION	SPATIO-TEMPORAL DOMAIN (μ)
$(Alice, State\ Epi)$	User-Role Authorization	$[b, A \cup B]$
$(Bob, Clinic\ Epi)$	User-Role Authorization	$[b, C]$
$(Ben, Clinician)$	User-Role Authorization	$[a, C]$
$(Charlie, State\ VC)$	User-Role Authorization	$[a, A \cup B]$
$(State\ Epi, p_1)$	Permission-Role Authorization	$[a, B]$
$(State\ Epi, p_3)$	Permission-Role Authorization	$[a, B]$
$(State\ Epi, p_{16})$	Permission-Role Authorization	$[a, A \cup B]$
$(State\ Epi, p_{17})$	Permission-Role Authorization	$[b, B]$
$(Juris\ Epi, p_1)$	Permission-Role Authorization	$[a, B]$
$(Juris\ Epi, p_3)$	Permission-Role Authorization	$[a, B]$
$(Juris\ Epi, p_{17})$	Permission-Role Authorization	$[b, B]$
$(Clinic\ Epi, p_{17})$	Permission-Role Authorization	$[b, D]$
$(Clinician, p_1)$	Permission-Role Authorization	$[a, C]$
$(Clinician, p_2)$	Permission-Role Authorization	$[a, C]$
$(Clinician, p_{17})$	Permission-Role Authorization	$[c, C]$
$(State\ VC, p_1)$	Permission-Role Authorization	$[a, B]$
$(State\ VC, p_7)$	Permission-Role Authorization	$[a, \emptyset]$
$(State\ VC, p_8)$	Permission-Role Authorization	$[a, B]$
$(State\ VC, p_{11})$	Permission-Role Authorization	$[a, A]$
$(State\ VC, p_{15})$	Permission-Role Authorization	$[a, A]$
$(Juris\ VC, p_1)$	Permission-Role Authorization	$[a, B]$
$(Juris\ VC, p_7)$	Permission-Role Authorization	$[b, \emptyset]$
$(Juris\ VC, p_8)$	Permission-Role Authorization	$[a, B]$
$(Local\ VC\ Team, p_7)$	Permission-Role Acquisition	$[c, B \cup E]$
$(State\ Epi, State\ VC)$	Role Static SoD	$[b, D]$
$(State\ Epi, Juris\ VC)$	Role Static SoD	$[b, D]$
$(Juris\ Epi, State\ VC)$	Role Static SoD	$[b, D]$
$(Juris\ Epi, Juris\ VC)$	Role Static SoD	$[b, D]$
$(Clinic\ Epi, State\ VC)$	Role Static SoD	$[b, D]$
$(Clinic\ Epi, Juris\ VC)$	Role Static SoD	$[b, D]$
(p_{11}, p_{15})	Permission Static SoD	$[a, D]$
(p_{16}, p_{17})	Permission Static SoD	$[a, D]$

Table 7.1: New Relationships and Constraints

7.3.3 Problem Detection using Privilege Acquisition Graph

In this section, we show how to detect infeasible paths and separation of duty violations using our modified approach.

7.3.3.1 Infeasible Path Detection

We define the types in the model as colorset as shown in Section 7.2. We use the privilege acquisition graph instead of the access control graph to populate the initial markings of our previous CPN model shown in Figure 7.3. The initial marking for *Authorization Edges*, denoted as *AllAuthEdges* is populated by the UA' and the PA' edges of the privilege acquisition graph. The rest of the initial markings for other places are the same as before.

We allow the execution of this model and run the queries to detect infeasible paths. The analysis result shows that the system contains infeasible path. The query shows that a set of states $\{42, 43\}$ suffers from the infeasible path. To check this, we use the *print* command to check the descriptor (environmental variables) of the state. For instance, let us check the state 43. Below is part of the content of state 43.

```
Authorization Path = 1‘("Ben","Clinician","UA",["a"], ["C"],"",0)++
1‘("Clinician","p17","PA",["c"], ["C"],"",0);
```

The result shows that the infeasible path occurs because user *Ben* cannot acquire permission 17 assigned to him via the *Clinician* role. The percentage reduction in the number of states when using the privilege acquisition graph instead of the access control graph is only 8 percent in this case.

In this analysis, we do not have enough information about how *Ben* was assigned the *Clinician* role, whether through direct assignment or indirect assignment by hierarchy or delegation. If we are interested in knowing the source of conflict, we have to verify the original graph. However, since we know that only p_{17} causes the problem, we can bypass the verification of other irrelevant entities. To do this, from the access control graph, we create a subgraph consists of all entities related with p_{17} by performing a reverse depth first search starting from p_{17}. The subgraph derived from the access control graph shown in Figure 7.7.

We then run the same CPN model for the derived subgraph, which is similar to Figure 7.3.

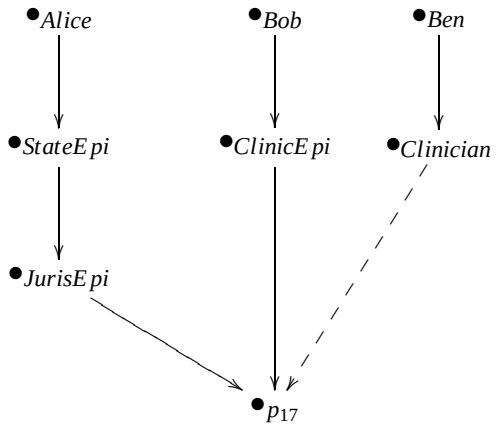


Figure 7.7: Subgraph of the related entities of p_{17}.

We observe the state variable, which shown that p_{17} is the delegated permission which has temporal conflict with role *Clinician*.

7.3.3.2 SoD Violation Detection

We run our previous CPN model for detecting the SoD violation shown in Figure 7.5 on the privilege acquisition graph. We then create the state space graph and execute the query to detect conflicts. The percentage reduction in the number of states obtained by using the privilege acquisition graph is 25 percent. The tools return a list of possible conflict states $\{38, 46, 48, 50, 53, 55, 56, 57, 58, 59\}$. We run the print command to show the value of environmental variables of state number 46. Below is part of the content of state 46 which shows that the conflict occurs between p_{16} and p_{17}.

```
SoD Violate Ancestor = 1‘("State Epi",["a"], ["B"]);
SoD Violate = 1‘("p16","p17","PSSD",["a"],
            ["A", "B", "C", "D", "E"], "", 0);
```

Since CPNs based on the privilege acquisition graph can detect conflicts but cannot identify the source, we create a subgraph from the access control graph by performing a reverse depth first search starting from node for p_{16} and then for p_{17}. The resulting subgraph is shown in Figure 7.8. This subgraph can be analyzed as described in Section 7.2 to reveal the source of

conflict. Since the subgraph is much smaller than the original access control graph, it will take significantly less time. We then run the model again on the derived subgraph. This time the model indicates that p_{17} is the inherited permission which together with the assigned p_{16} of role *State Epi* have violated the SSoD for permission-role assignment.

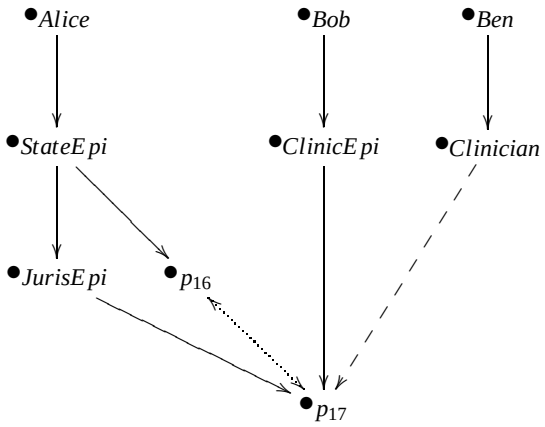


Figure 7.8: Subgraph of the related entities of permission 16 and 17.

7.4 Chapter Summary

In this chapter, we investigated how the various features of the extended STRBAC model may interact with each other in subtle ways resulting in conflicts and other inconsistencies. Consequently, we need to analyze the access control constraints of the application to ensure that such problems do not occur. Since manual analysis is tedious and error-prone, we show how the analysis can be automated using Coloured Petri Nets. For large complex applications, the analysis may take a significant amount of time. Towards this end, we show how to speed up the analysis by condensing the graph representing the application and verifying this condensed graph.

Chapter 8

A Trust-Based Access Control Model for Pervasive Computing Applications

From Chapter 3 to 7, we propose three types of the access control model for the pervasive computing. These models utilize the location and time constraint to determine the accessibility of the users. In the highly secure pervasive computing environment, where uncontrolled disclosure of information, unconstrained interaction among entities, or relying on untrustworthy entities may have very serious consequences, only the trustworthy user should be allowed to perform a critical operation independent of time and location constraints. In such scenario, the spatio-temporal access control model might not be the right choice. In this chapter, we propose another access control model for pervasive computing environment–the Trust-Based Access Control Model.

8.1 Trust Modeling and Computation

In our model, only the user (human users and devices) with adequate trustworthiness can be authorized to the roles and permissions. Trust values for each user are calculated based on the role the user performed previously. Trust value of a user can change based on her/his activities. Activities detrimental to the security of the system such as, committing fraud will decrease the user's trustworthiness. Other properties of users, such as, location, signal strength, or stability factor for sensors, and education level, age, etc for human users, can affect the trustworthiness. Finally, recommendations provided by others are evaluated to compute trustworthiness

of users.

In this section, we describe how trust is modeled between two entities with respect to a given context. We adapt the trust model proposed by Ray et al. in [66]. The authors formalize trust as a relationship between two entities, the truster, an entity that trusts the target entity, and the trustee, the target entity that is trusted. Initially, an entity *A* does not trust entity *B* completely. Entity *A* needs to evaluate a trust relationship with entity *B* in some context. The context in our model is the role to which a user will be assigned to. We will refer to the context as a role context *rc* throughout the trust model. The truster is the pervasive computing system while the trustee (the user) is either a human user (or its representative treated synonymously) or a device.

Users can be associated with multiple roles. In order to verify the authorization between a user and a role, a user's trust value is evaluated based on each role context separately. For instance, if a human user needs to be assigned as the *engineer in pH monitoring role* and *engineer in impeller speed monitoring role,* then two trust values need to be evaluated for each role in order to make a decision about role assignments. The trust relationship between human user or device user and a system in the role context *rc* depends on three factors: *properties*, *experience*, and *recommendations*. The semantics of these three factors are different for the human and the device user.

A trustee discloses a set of physical properties to be verified by the truster. A device may be associated with a set of properties such as CPU processing speed, memory capacity, transmission rate, signal strength, location of sensor, and physical security (such as is the sensor placed inside a tamper proof container). The properties associated with a human user could be age, gender, education level, specialization, credentials, and so on.

Experience is based on the set of events that had occurred in the past within a certain period of time in which the trustee was involved and that the truster has recollection about. For a device, this can be incidents like number of defects encountered, tamper occurrences, collected data quality, and alarms and control signals responsiveness. For the human user, this could be decisions made in the past, task execution time taken, finesse demonstrated, and so

on.

Recommendations are provided by trusted third-parties who have knowledge about the trustee with respect to the role context *rc*. Recommendations in case of a device can be provided by other organizations that have used the device under similar circumstances. For a human user, the recommendations can be provided by an organization that he was worked with in the same (or similar) role context *rc*.

The trustworthiness between a system and a human user is evaluated in the same way as between the system and a device user. In our trust model, we formally represent a trust relationship between truster A and trustee B on some role context rc as a triple $({}_Ab_B^{rc}, {}_Ad_B^{rc}, {}_Au_B^{rc})$, where ${}_Ab_B^{rc}$ is A's belief on B in role context rc, ${}_Ad_B^{rc}$ is A's disbelief on B in role context rc, and ${}_Au_B^{rc}$ is A's uncertainty on B in role context rc. Each of these components has a value between [0,1] and sum of these components is 1. In the following, we describe how trust relationship can be evaluated based on properties, experience, and recommendations factors. Later, we will use the resulting trust relationship to evaluate the trust value.

8.1.1 Quantifying Properties

Each role in an organization requires certain properties of a user. Some other properties are desirable. The properties are scored based on information provided by the user to the system at the initiation of the access request. Each role in our model is associated with a set of positive properties and negative properties, collectively called the role properties. Let a set of positive properties associated with role R be denoted by PS_R where $PS_R = \{ps_1, ps_2, \ldots, ps_n\}$. Each of these properties has different weight and depends on the organization policy. Let $w_{ps_1}, w_{ps_2}, \ldots, w_{ps_n}$ be the weights of the positive properties $ps_1, ps_2, \ldots, ps_n$ respectively, where $w_{ps_i} \in [0,1]$ and $\sum_{i=1}^{n} w_{ps_i} = 1$. The weights of positive properties indicate their effectiveness with respect to the role R. Similarly, let a set of negative properties associated with role R be denoted by NE_R where $NE_R = \{ne_1, ne_2, \ldots, ne_n\}$, and the weights of these negative properties are $w_{ne_1}, w_{ne_2}, \ldots, w_{ne_n}$, where $w_{ne_i} \in [0,1]$ and $\sum_{i=1}^{n} w_{ne_i} = 1$.

Let the set of properties possessed by a user B be $UP = up_1, up_2, \ldots, up_n$. Let $p_B =$

$\{UP \bigcap PS_R\}$ be the set of positive properties for the user that are relevant for the role, and $n_B = \{UP \bigcap NE_R\}$ be the set of negative properties. The contribution of the user's properties towards its trust is represented by (b_P, d_P, u_P) where b_i, d_i, u_i denotes belief, disbelief, and uncertainty respectively. Each positive property increases the belief factor b_P, whereas each negative property increases the disbelief factor d_P. Let w_{ps_i} be the weight of the positive property $p_{B_i} \in UP \bigcap PS_R$, and w_{ne_i} be the weight of the negative property $n_{B_i} \in UP \bigcap NE_R$, $m = |UP \bigcap PS_R|$, and $n = |UP \bigcap NE_R|$, the values of b_P,d_P and u_P are computed using the following formulas:

$$b_P = \frac{\sum_{i=1}^{m} w_{ps_i}}{\sum_{i=1}^{m} w_{ps_i} + \sum_{i=1}^{n} w_{ne_i}}; \; d_P = \frac{\sum_{i=1}^{n} w_{ne_i}}{\sum_{i=1}^{m} w_{ps_i} + \sum_{i=1}^{n} w_{ne_i}}; \text{ and } u_P = 1 - b_P - d_P.$$

8.1.2 Quantifying Experience

Experience constitutes an important factor in this model. We model experience in terms of a number of events encountered by a truster A regarding trustee B in particular context within a specific period of time $[t_0, t_n]$. The time period $[t_0, t_n]$ is equally divided into a set S_i of n intervals, $S_i = \{[t_0, t_1], [t_1, t_2], \ldots, [t_{n-1}, t_n]\}$. The intervals overlap at the boundary points only. The truster A keeps a history file of events performed by the trustee B within these intervals. Within each interval $[t_j, t_{j+1}] \in S_i$ where $j \in \mathbb{N}$, there exists a (possibly empty) set of events that transpired between the user and the system.

A user is granted a role based on the recorded events performed by the user with respect to the role context within a certain period. The length of the time period depends on the system implementation. We assume that the length of the time period is specified by the security administrator. Intuitively, events far back in time do not count as strongly as very recent events for evaluating trust relationship. We assume that truster A has a log file of the events performed by trustee B in the time period between t_0 and t_n. An event can be positive, negative, or neutral. Positive events increases the belief component of experience, negative event increase the disbelief component of experinece, and netural events equally increase both belief and disbelief

components. In addition, the neutral event contributes towards increase in the uncertainty component. In the following, we explain how to calculate the experience that a truster A has about trustee B with respect to the role context in period of time $[t_0, t_n]$. This formally denoted as ${}_AE_B^{rc} = (b_E, d_E, u_E)$ where b_E, d_E , u_E represent belief, disbelief, and uncertainty components with respect to the experience in the time period $[t_0, t_n]$ in the role context rc.

Let experience acquired from events occured in the at interval i where $1 \leq i \leq n$ be represented as (b_i, d_i, u_i) where b_i,d_i,u_i denote belief, disbelief, and uncertainly respectively. When no events occured in the interval i, then $u_i = 1$,$b_i = d_i = 0$. Let P_i be a set of all positive events,Q_i is a set of all negative events, and N_i is a set of all neutral events occured in the interval i. The values of b_i,d_i, and u_i are computed as follows:

$$b_i = \frac{|P_i| + \frac{|N_i|}{2}}{|P_i| + |Q_i| + |N_i|}; \; d_i = \frac{|Q_i| + \frac{|N_i|}{2}}{|P_i| + |Q_i| + |N_i|}; \text{ and } u_i = \frac{0.5 \times \frac{|N_i|}{2}}{|P_i| + |Q_i| + |N_i|}.$$

Note that events occured in the distant past has less effect than events occured recently. To accomodate this in the trust model, we assign non-negative weight to each interval. Let w_i denotes to the weight of i^{th} interval such that $w_i > w_j$ whenever $j < i$, $i, j \in \mathbb{N}$. Recent intervals in the experience policy are given more weights than those far back in time. We use the formula $w_i = \frac{k}{S}$ for all $k = 1, 2, \ldots, n$ where $S = \frac{n \times (n+1)}{2}$ to evaluate weights of the intervals. Formally, the experience of truster A about trustee B with respect to the role context rc in the time period $[t_0, t_n]$, represented by ${}_AE_B^{rc}$ is computed as ${}_AE_B^{rc} = (b_E, d_E, u_E)$, where the values of b_E, d_E, and u_e are given by: $b_E = \sum_{i=1}^{n} w_i \times b_i$, $d_E = \sum_{i=1}^{n} w_i \times d_i$, and $u_E = \sum_{i=1}^{n} w_i \times u_i$, where n is the number of intervals.

8.1.3 Quantifying Recommendations

Recommendations play major role on the trust evaluation when the truster does not know much about the trustee. Truster obtains recommendations from one or more recommender knowing about the trustee with respect to particular roles. The recommendation is evaluated based on the recommendations returned by recommender M about B as well as the trust relationship between truster A and the recommender M in providing a recommendation about

trustee B. The trust relationship between truster A and recommender M in the context of giving recommendation about user B with respect to particular role effects the weight of the recommendation given by the recommender. The trust relationship between truster A and recommender M to provide recommendation is formally represented as a 3×3 matrix. The rows of the matrix correspond to properties, experience, and recommendation. The columns correspond to belief, disbelief, and uncertainty on each of these factors. The matrix is normalized to (b,d,u) and it will be used to evaluate the recommendation sent by a recommender.

Let the triples $({}_{M}b_{B},{}_{M}d_{B},{}_{M}u_{B})$ be the recommendation sent by recommender M about user B to truster A, and the trust relationship between the truster A and recommender M is represented as a triples $({}_{A}b_{M},{}_{A}d_{M},{}_{A}u_{M})$, then the recommendation score ${}_{AM}R_{B}^{rc}$ generated by recommender M about a user B to the truster A in the role context rc is given by $({}_{AM}b_{B}^{rc},{}_{AM}d_{B}^{rc},{}_{AM}u_{B}^{rc})$. , where ${}_{AM}b_{B}^{rc} =_{A} b_{M} \times_{M} b_{B}$; ${}_{AM}d_{B}^{rc} =_{A} d_{M} \times_{M} d_{B}$; and ${}_{AM}u_{B}^{rc} =_{A} d_{M} +_{A} u_{M} +_{A} b_{M} \times_{M} u_{B}$.

Note that truster A could get recommendations about trustee B from several recommenders. Therefore, A's disbelief on the recommendation about user B is the average of the disbelief values of all recomendations. The same applies to both belief and uncertainty values. Let G be a set of recommendations about user B to truster A with respect to the role context rc, represented by triples $({}_{AG}b_{R},{}_{AG}d_{R},{}_{AG}u_{R})$, where the belief component ${}_{AG}b_{R}$, the disbelief component ${}_{AG}d_{R}$, and the uncertainty component ${}_{AG}u_{R}$ are computed as follow:

$${}_{AG}b_{R} = \frac{\sum_{i=1}^{n} {}_{Ai}b_{B}^{rc}}{n};\ {}_{AG}d_{R} = \frac{\sum_{i=1}^{n} {}_{Ai}d_{B}^{rc}}{n};\ \text{and}\ {}_{AG}u_{R} = \frac{\sum_{i=1}^{n} {}_{Ai}u_{B}^{rc}}{n}.$$

8.1.4 Computing Trustworthiness

We have determine trust vector for properties, experience, and recommendation components of trust relationship between truster A and trustee B with respect to a role context rc. The trust relationship between truster A and B is specified as:

$$(A \xrightarrow{rc} B) = \begin{pmatrix} b_P & d_P & u_P \\ b_E & d_E & u_E \\ {}_{AG}b_R & {}_{AG}d_R & {}_{AG}u_R \end{pmatrix}$$

Note that truster may come up with different values for the factors that influence trust for the same trustee with respect to different role contexts. This may happen because a truster may assign different weights to different factors that influence trust. A truster may give more weights to one of the component than others in computing trust relationship to assign particular role to the trustee. For example, truster may choose to focus more on experience than recommendation in computing trust with respect to the role context rc_i, whereas he may focus more on recommendation about the trustee in computing trust with respect to role context rc_j. A truster choose to emphasis on particular component of trust than others based on evaluation policy of the truster with respect to each role context. The evaluation policy of the truster is repersented by triple ${}_AW_B^{rc} = (W_P, W_E, W_R)$ where $W_P + W_E + W_R = 1$ and $W_P, W_E, W_R \in [0,1]$. The trust relationship between a truster A and trustee B for a particular role context rc is given by:

$$\begin{aligned}(A \xrightarrow{rc} B) &= {}_A W_B^{rc} \times (A \xrightarrow{rc} B) \\ &= (W_P, W_E, W_R) \times \begin{pmatrix} b_P & d_P & u_P \\ b_E & d_E & u_E \\ {}_{AG}b_R & {}_{AG}d_R & {}_{AG}u_R \end{pmatrix} \\ &= ({}_Ab_B^{rc}, {}_Ad_B^{rc}, {}_Au_B^{rc})\end{aligned}$$

where ${}_Ab_B^{rc} = (W_P \times b_P) + (W_E \times b_E) + (W_R \times_{AG} b_R)$, ${}_Ad_B^{rc} = (W_P \times d_P) + (W_E \times d_E) + (W_R \times_{AG} d_R)$, and ${}_Au_B^{rc} = (W_P \times u_P) + (W_E \times u_E) + (W_R \times_{AG} u_R)$. The elements of the trust relationship ${}_Ab_B^{rc}, {}_Ad_B^{rc}, {}_Au_B^{rc} \in [0,1]$, and ${}_Ab_B^{rc} +_A d_B^{rc} +_A u_B^{rc} = 1$. After evaluating the trust of the properties, experience, and recommendation factors in the previous subsections and the normalized trust relationship between a truster A and the trustee B with respect to the role context rc, the trust value is computed as follows:

$$T_{au} = \frac{{}_A b_B^{rc} +_A u_B^{rc}}{{}_A b_B^{rc} +_A d_B^{rc} +_A u_B^{rc}}$$

The value T will be in the range of [0,1]. The value closer to 0 indicates low trust value of user B with respect to role R, while the value closer to 1 indicates very high trust value of user with respect to role R.

8.2 Our Trust-Based RBAC Model

To represent our model, we adapt the graph-theoretic approach proposed by Chen and Crampton [19] in the following manner. In our work, the set of vertices $V = U \cup R \cup P$ correspond to the following RBAC entities:

- Users (U), which can be either human (U_h) or intelligent device (U_d);
- Roles (R), which can be categorized to human role (R_h) and device role (R_d), and
- Permisssions (P), which can be categorized to human permission (P_h) and device permission (P_d).

Our model assumes the existence of the following relationships of RBAC that constitute the set of edges $E = UA \cup PA \cup RH_a \cup RH_u$ where

- User-Role Assignment (UA) = $(U_h \times R_h) \cup (U_d \times R_d)$
- Permission-Role Assignment (PA) = $(R_h \times P_h) \cup (R_d \times P_d)$
- Role Hierarchy (RH) = $((R_h \times R_h) \cup (R_d \times R_d)) \times \{a, u\}$, which can be categorized to:
  - the activation hierarchy (RH_a) = $\{(r, r') : (r, r', a) \in RH\}$, and
  - the permission usage hierarchy (RH_u) = $\{(r, r') : (r, r', u) \in RH\}$

In our model, trust values for each user are calculated based on the role he performed previously. The information of the roles he used to perform will be stored in the User Role History, which is the history of roles that user used to performed. The values of trust can be changed from time to time based on user activities. Negative activities such as, committing the

fraud in the can decrease his trustworthiness. The calculation process is described in Section 8.1.

The system administrator will assign trust constraints in the form of a corresponding *trust interval* to roles, permissions, and other associations between entities based on different characteristics of each model. Trust interval is an interval $[l,1]$, where l is the lowest trust value that each role, permission or association is active.

Note that in the organization structure, users of the senior role can perform the same set of duties as its junior role, hence user who will be assigned to the senior role require more trustworthiness than the user who will be assigned to junior role only. Based on this observation, when we introduce the notion of trust value, we assume that the trust value of the senior role always dominate the trust value of its junior roles. Figure 8.1 shows components in our model.

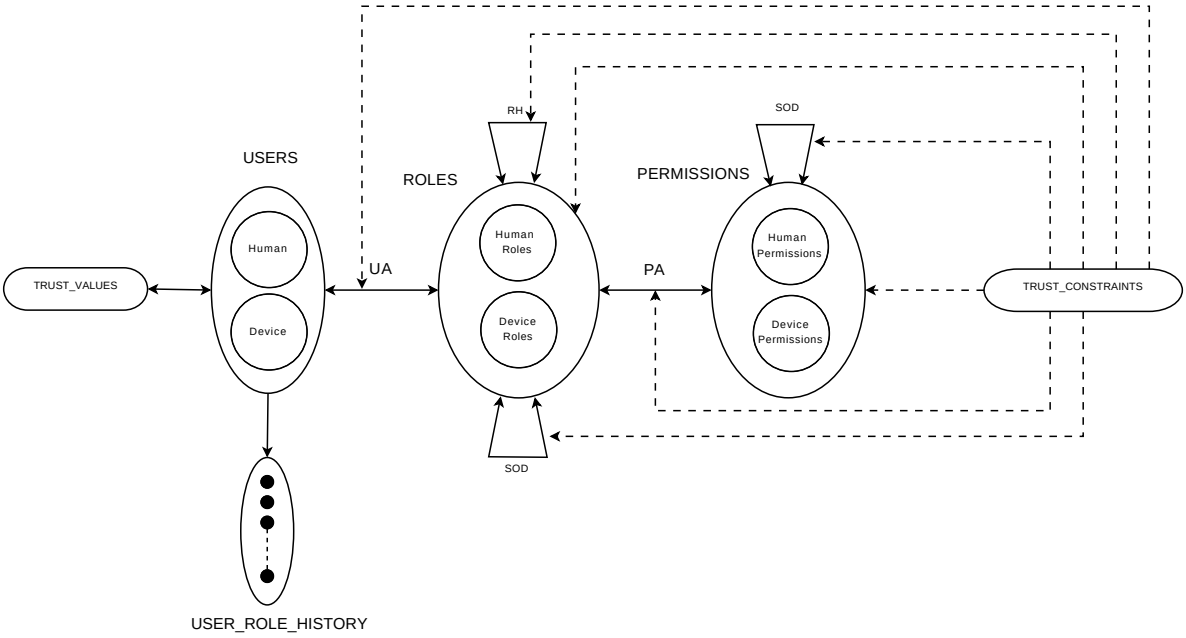


Figure 8.1: Trust RBAC Model

We define the notion of activation path, usage path and access path in a manner similar to that proposed by Chen and Crampton. An *activation path* (or *act-path*) between v_1 and v_n is defined to be a sequence of vertices $v_1,\ldots,v_n$ such that $(v_1,v_2) \in UA$ and $(v_{i-1},v_i) \in RH_a$ for $i = 3,\ldots,n$. A *usage path* (or *u-path*) between v_1 and v_n is defined to be a sequence of vertices $v_1,\ldots,v_n$ such that $(v_i,v_{i+1}) \in RH_u$ for $i = 1,\ldots,n-2$, and $(v_{n-1},v_n) \in PA$. An *access path* (or

acs-path) between v_1 and v_n is defined to be a sequence of vertices $v_1, \ldots, v_n$, such that (v_1, v_i) is an act-path, and (v_i, v_n) is an u-path. We assume the existence of a trust domain $\mathcal{D}$. The value of trust in the domain can be any real number vary from zero to one. Following Chen and Crampton's work [19], we also propose three models, namely, the standard model, the strong model, and the weak model. The models differ with respect to the trust constraints that must be satisfied by the entities for the authorization to be successful.

8.2.1 The Standard Model

In the standard model, individual entities, namely, users, roles, and permissions are associated with trust values. The trust values associated with the user describe how much the user is trusted to perform each specific role. The trust interval associated with a role specify the range of trust values with respect to that role which user has to acquire in order to activate the role. The trust interval associated with a permission specify the range of trust values with respect to the current role of the user which he has to acquire in order to invoke the permission. The standard model requires that if a user u can invoke a permission p, then the trust value of u is in the trust interval associated with all other nodes in the path connecting u to p. These ideas are formalized below.

The trust values for the user with respect to each role are denoted with a function $\mathcal{T} : ((U_h \times R_h) \cup (U_d \times R_d)) \rightarrow t \in \mathcal{D}$. The trust interval for role and permission are denoted with a function $\mathcal{L} : (R \cup P) \rightarrow [l, 1] \subseteq \mathcal{D}$.

- for $u \in U, r \in R$, $\mathcal{T}(u, r)$ denotes the trust value of u with respect to r;
- for $r \in R$, $\mathcal{L}(r)$ denotes the trust interval in which r is active;
- for $p \in P$, $\mathcal{L}(p)$ denotes the trust interval in which p is active.

Given a path $v_1, \ldots, v_n$ in the labeled graph $G = (V, E, \mathcal{T}, \mathcal{L})$, where $E = UA \cup PA \cup RH_a \cup RH_u$, we write $\hat{\mathcal{L}}(v_2, \ldots, v_n) = \hat{\mathcal{L}}(v_2, v_n) \subseteq \mathcal{D}$ to denote $\bigcap_{i=2}^{n} \mathcal{L}(v_i)$. In other words, $\hat{\mathcal{L}}(v_2, v_n)$ is the trust interval in which every vertex $v_i \in R \cup P$ is enabled.

Authorization in the Standard Model:

- A user $v_1 \in U$ may activate role $v_n \in R$ if and only if there exists an act-path $v_1, v_2, \ldots, v_n$ and $\mathcal{T}(v_1, v_2) \in \mathcal{L}(v_2)$;

- A role $v_1 \in R$ is authorized for permission $v_n \in P$ if and only if there exists an u-path $v_1, v_2, \ldots, v_n$ and $\mathcal{L}(v_1) \subseteq \hat{\mathcal{L}}(v_1, v_n)$;

- A user $v_1 \in U$ is authorized for permission $v_n \in P$ if and only if there exists an acs-path $v_1, v_2, \ldots, v_i, \ldots, v_n$ such that $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_i, \ldots, v_n$ is a u-path, v can activate v_i, and v_i is authorized for v'.

8.2.2 The Strong Model

The strong model is used when not only the individual entities (users, roles, permissions) involved must satisfy the trust constraints, but the different relationships must also satisfy such constraints. For instance, consider the relation $(r, p) \in PA$. In this case, we not only have to take into account the trust values at which the role r can be activated and the trust values at which the permission p can be invoked, but we also must consider the trust values when r can invoke p. This requires specifying another function in the strong model as described below.

The trust constraints in the *strong model* are denoted with a function $\mu : E \rightarrow [l, 1] \subseteq \mathcal{D}$. For $e = (v, v') \in E$, $\mu(v, v')$ denotes the trust interval in which the association between v and v' is active.

- if $(u, r) \in UA$, then $\mu(u, r)$ denotes the trust interval in which u is assigned to r;

- if $(r', r) \in RH_a$, then $\mu(r', r)$ denotes the trust interval in which r' is senior to r in the activation hierarchy;

- if $(r', r) \in RH_u$, then $\mu(r', r)$ denotes the trust interval in which r' is senior to r in the permission usage hierarchy;

- if $(r, p) \in PA$, then $\mu(r, p)$ denotes the trust interval in which p is assigned to r.

Given a path $v_1, \ldots, v_n$ in the labeled graph $G = (V, E, \mathcal{T}, \mathcal{L}, \mu)$, where $V = U \cup R \cup P$ and $E = UA \cup PA \cup RH_a \cup RH_u$, we write $\hat{\mu}(v_1, \ldots, v_n) = \hat{\mu}(v_1, v_n) \subseteq \mathcal{D}$ to denote $\bigcap_{i=1}^{n-1} \mu(v_i, v_{i+1})$. Hence, $\hat{\mu}(v_1, v_n)$ is the trust interval in which every edge in the path is enabled.

Authorization in the Strong Model:

- a user $v_1 \in U$ may activate role $v_n \in R$ if and only if there exists an act-path $v_1, v_2, \ldots, v_n$ and $\forall i = 2, \ldots, n \bullet \mathcal{T}(v_1, v_i) \in (\mathcal{L}(v_1) \cap \mathcal{L}(v_i) \cap \hat{\mu}(v_1, v_i))$;

- a role $v_1 \in R$ is authorized for permission $v_n \in P$ if and only if there exists an u-path $v_1, v_2, \ldots, v_n$ and $\mathcal{L}(v_1) \subseteq (\hat{\mathcal{L}}(v_1, v_n) \cap \hat{\mu}(v_1, v_n))$;

- A user $v_1 \in U$ is authorized for permission $v_n \in P$ if and only if there exists an acs-path $v_1, v_2, \ldots, v_i, \ldots, v_n$ such that $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_i, \ldots, v_n$ is a u-path, v_1 can activate v_i, and v_i is authorized for v_n.

8.2.3 The Weak Model

The weak model is derived from the standard model. Recall that the standard model requires that each entity (users, roles, and permissions) in the authorization path be associated with a trust value and in order to be authorized to access other entities, the requester's trust value must be included in the trust interval of the entity he wants to access, together with other entities along the path. In the weak model, the entity v is authorized for another entity v' if the trust value of v is included in the trust interval of v'. There is no requirement that the intermediate nodes on the path satisfy the trust constraints. Like the standard model, the model is based on the labeled graph $G = (V, E, \mathcal{T}, \mathcal{L})$, where $V = U \cup R \cup P$ and $E = UA \cup PA \cup RH_a \cup RH_u$.

Authorization in the Weak Model:

- A user $v_1 \in U$ may activate role $v_n \in R$ if and only if there exists an act-path $v_1, v_2, \ldots, v_n$ and $\mathcal{T}(v_1, v_n) \in \mathcal{L}(v_n)$;

- A role $v_1 \in R$ is authorized for permission $v_n \in P$ if and only if there exists a u-path $v_1, v_2, \ldots, v_n$ and $\mathcal{L}(v_1) \subseteq \mathcal{L}(v_n)$;

- A user $v_1 \in U$ is authorized for permission $v_n \in P$ if and only if there exists an acs-path $v_1, v_2, \ldots, v_i, \ldots, v_n$ such that $v_i \in R$ for some i, $v_1, \ldots, v_i$ is an act-path, $v_i, \ldots, v_n$ is a u-path, v_1 can activate v_i, and v_i is authorized for v_n.

8.3 Separation of Duties Constraints

Separation of duties (SoD) prevents the occurrence of fraud arising out of conflicts of interests in organizations [80]. Separation of duties ensure that conflicting roles are not assigned to the same user or that conflicting permissions are not assigned to the same role.

Separation of Duty (SoD) comes in two varieties. First one is with respect to the mutual exclusion relations between two roles. This is to guarantee that no user can be assigned to two conflicting roles. The second one is with respect to the mutual exclusion relations between two permissions. This is to guarantee that no role can be assigned two conflicting permissions. We denote these two types of SoD by using SD^R and SD^P edges, respectively. Since SoD is a symmetric relationship, the SD^R and SD^P edges are bi-directional.

We next define the separation of duties for the standard and weak models. The SoDs defined for the standard and weak models are expressed in terms of the graph $G = (V, E, \mathcal{T}, \mathcal{L})$, where $E = UA \cup PA \cup RH_a \cup RH_u \cup SD^R \cup SD^P$ and $V = U \cup R \cup P$. For these cases, the SoD is similar to the SoD constraints in traditional RBAC. These are given below. **SoD Constraints for the Weak and Standard Model**

- **User-Role Assignment** if $(r, r') \in SD^R$ then there are no two edges (u, r) and (u, r') such that $\{(u, r), (u, r')\} \subset UA$

- **Permission-Role Assignment** if $(p, p') \in SD^P$ then there are no two u-paths of the form $r, v_1, v_2, \ldots, p$ and $r, v'_1, v'_2, \ldots, p'$

In the organization, sometimes we want the user who gain the very high trust to be able to bypass the SoDs. To do this, we defined the trust constraint for the separation of duties with a function $\delta : E \rightarrow [l, 1] \subseteq \mathcal{D}$. For $e = (v, v') \in SD^R \cup SD^P$, $\delta(v, v')$ denotes the trust interval in which the SoD constraint can be ignored. In particular,

- if $(r, r') \in SD^R$, $\delta(r, r')$ denotes the trust interval in which the role-role separation of duties constraint can be ignored;

- if $(p, p') \in SD^P$, $\delta(p, p')$ denotes the trust interval in which the permission-permission separation of duties constraint can be ignored.

The strong model is defined over the labeled graph $G = (V, E, \mathcal{T}, \mathcal{L}, \mu, \delta)$, where $E = UA \cup PA \cup RH_a \cup RH_u \cup SD^R \cup SD^P$ and $V = U \cup R \cup P$. The strong model allows specification of weaker forms of SoD constraints than those supported by the traditional RBAC. Specifically, it allows one to specify the trust interval in which the SoD constraints can be ignored. **SoD Constraints for the Strong Model**

- **User-Role Assignment:** if $(r, r') \in SD^R$ then there are no two edges (u, r) and (u, r'), corresponding to some user u, where $\mathcal{T}(u, r) \notin (\mathcal{L}(u) \cap \mathcal{L}(r) \cap \mu(u, r) \cap \delta(r, r'))$ and $\mathcal{T}(u, r') \notin (\mathcal{L}(u) \cap \mathcal{L}(r') \cap \mu(u, r') \cap \delta(r, r'))$;

- **Permission-Role Assignment:** if $(p, p') \in SD^P$ then there are no two u-paths $r, v_1, v_2, \ldots, p$ and $r, v'_1, v'_2, \ldots, p'$ where $\mathcal{L}(r) \nsubseteq (\hat{\mathcal{L}}(r, p) \cap \hat{\mu}(r, p) \cap \delta(p, p'))$ and $\mathcal{L}(r) \nsubseteq (\hat{\mathcal{L}}(r, p') \cap \hat{\mu}(r, p') \cap \delta(p, p'))$.

8.4 Example Scenario

Consider a typical pervasive computing application–hazardous chemical monitoring and control in a chemical plant. The general environment within the plant is continuously monitored by various chemical sensors for chemical leaks. These sensors are linked with various controllers that can regulate the flow of chemicals to various machineries. Engineers monitor variables such as temperature, pH values, liquid levels, feed flow rates, and speed of impellers through other sensors. Back end systems analyze the sensed information to actuate different equipment. Operators in control room oversee various aspects of the plant operating computers. They can override automatic control or tune different operational parameters.

In this section we will demonstrate how our model can support the requirements described above. We describe a set of entities in the process control application for the chemical plant

with their corresponding trust values in Table 8.1, the trust constraints for each relationship between entities are shown in Table 8.2, and the configuration of the access control is shown in Figure 8.2.

NAME	DESCRIPTION	TRUST VALUES ($\mathcal{T}$)
u_1	Alice	$\mathcal{T}(u_1, r_1) = 0.95$
u_2	Bob	$\mathcal{T}(u_2, r_2) = 0.7$
u_3	Charlie	$\mathcal{T}(u_3, r_3) = 0.8$
u_4	Central Control Server	$\mathcal{T}(u_4, r_4) = 0.9$
u_5	Sensor Server	$\mathcal{T}(u_5, r_5) = 0.8$
u_6	Actuator Server	$\mathcal{T}(u_6, r_6) = 0.95$
NAME	DESCRIPTION	TRUST CONSTRAINTS ($\mathcal{L}$)
r_1	Senior Engineer	$\mathcal{L}(r_1) = [0.9, 1]$
r_2	Operator	$\mathcal{L}(r_2) = [0.7, 1]$
r_3	Junior Engineer	$\mathcal{L}(r_3) = [0.8, 1]$
r_4	Sensors and Actuators Manager	$\mathcal{L}(r_4) = [0.9, 1]$
r_5	Sensor Interface	$\mathcal{L}(r_1) = [0.8, 1]$
r_6	Actuator Interface	$\mathcal{L}(r_1) = [0.85, 1]$
p_1	Analyze Sensors Data	$\mathcal{L}(p_1) = [0.9, 1]$
p_2	Control Actuator	$\mathcal{L}(p_2) = [0.8, 1]$
p_3	Observe Sensors Data	$\mathcal{L}(p_3) = [0.7, 1]$
p_4	Update Event Log	$\mathcal{L}(p_4) = [0.9, 1]$
p_5	Analyze Sensors Alarm	$\mathcal{L}(p_5) = [0.6, 1]$
p_6	Retrieve Sensors Data	$\mathcal{L}(p_6) = [0.7, 1]$
p_7	Send Control Signal to Actuator	$\mathcal{L}(p_7) = [0.9, 1]$

Table 8.1: Entities and Trust Values

First, let us assume that we use the standard model to secure the process control. Here, Alice can activate the senior engineer role since her trust value with respect to the senior engineer role (0.95) satisfies the trust constraints of the senior role ($[0.9, 1]$). She also inherits the permission assigned to junior engineer role through the permission usage hierarchy, and since her trust value satisfies the trust constraints of permission assigned to junior engineer role, she is allowed to invoke such permission. Moreover, she is also trusted to activate the operator role and use the permission assigned to that role. Similarly, the central control server is trusted to perform the sensors and actuators manager role. However, the server cannot invoke both permission to *Retrieve Sensors Data* and *Send Control Signal to Actuators* because it is prohibited by the SoD. Charlie is assigned to the role of junior role and his trust value allow him to per-

NAME	DESCRIPTION	TRUST CONSTRAINTS
(u_1, r_1)	User-Role Assignment	$\mu(u_1, r_1) = [0.95, 1]$
(u_2, r_2)	User-Role Assignment	$\mu(u_2, r_2) = [0.8, 1]$
(u_3, r_3)	User-Role Assignment	$\mu(u_3, r_3) = [0.8, 1]$
(u_4, r_4)	User-Role Assignment	$\mu(u_4, r_4) = [0.9, 1]$
(u_5, r_5)	User-Role Assignment	$\mu(u_5, r_5) = [0.8, 1]$
(u_6, r_6)	User-Role Assignment	$\mu(u_6, r_6) = [0.9, 1]$
(r_1, r_2)	Activation Hierarchy	$\mu(r_1, r_2) = [0.98, 1]$
(r_1, r_3)	Permission Usage Hierarchy	$\mu(r_1, r_3) = [0.9, 1]$
(r_4, r_5)	Permission Usage Hierarchy	$\mu(r_4, r_5) = [0.9, 1]$
(r_4, r_6)	Permission Usage Hierarchy	$\mu(r_4, r_6) = [0.9, 1]$
(r_1, p_1)	Permission-Role Assignment	$\mu(r_1, p_1) = [0.9, 1]$
(r_2, p_2)	Permission-Role Assignment	$\mu(r_2, p_2) = [0.8, 1]$
(r_3, p_1)	Permission-Role Assignment	$\mu(r_3, p_1) = [0.95, 1]$
(r_3, p_3)	Permission-Role Assignment	$\mu(r_3, p_3) = [0.8, 1]$
(r_4, p_4)	Permission-Role Assignment	$\mu(r_4, p_4) = [0.9, 1]$
(r_4, p_5)	Permission-Role Assignment	$\mu(r_4, p_5) = [0.9, 1]$
(r_5, p_6)	Permission-Role Assignment	$\mu(r_5, p_6) = [0.8, 1]$
(r_6, p_7)	Permission-Role Assignment	$\mu(r_6, p_7) = [0.9, 1]$
(p_6, p_7)	Separation of Duties	$\delta(p_6, p_7) = [0.9, 1]$
(p_7, p_6)	Separation of Duties	$\delta(p_7, p_6) = [0.9, 1]$

Table 8.2: Relationships and Trust Constraints

form this role. However, he cannot analyze the data because this permission requires a higher trust level ([0.9,1]) than his own (0.8). Charlie can later be authorized for this permission in the future after his trust level is at least 0.9.

Later, there is an emergency situation at the chemical plant where the higher trustworthiness is required for accessibility. Under this circumstance, the system administrator decide to switch to the strong model. Now, not only we have to satisfy the trust constraints assigned to entities, but the related trust constraints assigned to the relationships in Table 8.2 must be satisfied as well. As a result, now Alice can still perform the role of senior engineer and use the junior engineer's permission. However, she cannot activate the operator role anymore because her trust level is not satisfied the trust constraints of the role hierarchy. For Bob, now he cannot perform the operator role which he is allowed to do in the standard model. Next, let us assume that before this crisis, Charlie has performed a very good job in his junior engineer role, and his trust value with respect to the role is increased to 0.92. Using the standard model, now he

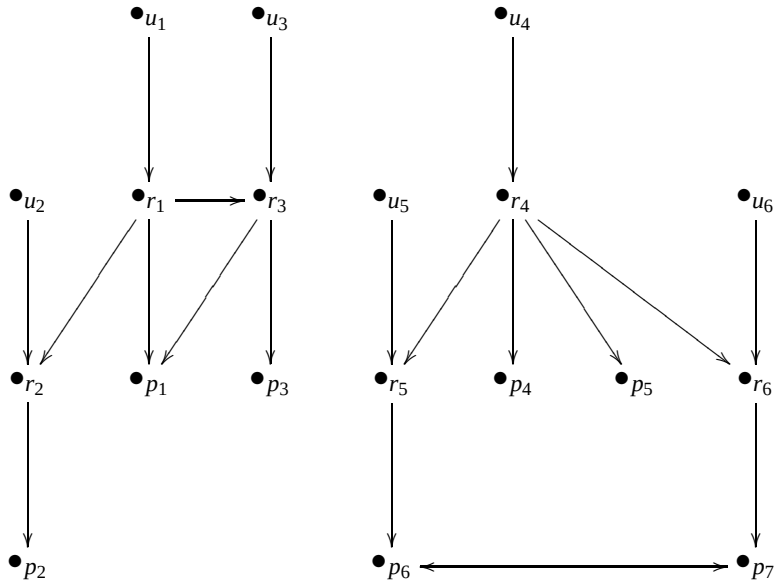


Figure 8.2: Access Control Model Configuration for Example

should be able to analyze the data. However, according to the new policy, he is not satisfies for the permission role assignment's trust constraint ([0.95,1]) and hence still cannot perform the data analysis. Next, let us investigate the impact of the model to the SoD constraints. Using the strong model, now the central control server has enough trustworthiness to bypass the SoD constraint and invoke the permission to *Retrieve Sensors Data* and *Send Control Signal to Actuators*. The weak model can be implemented using the similar idea hence will be left out of discussion.

8.5 Chapter Summary

In this chapter, we proposed a trust based access control model based on RBAC. First, we propose the methodology to evaluate the trust worthiness of the user entity in RBAC. We then identified the entities and relations in RBAC and investigated their dependence on trust. This dependency necessitates changes in the invariants and the operations of RBAC. The configuration of the model is formalized using graph-theoretic notation.

The usage of trust presented in this chapter is just one way of using the trust. We can

apply the trustworthiness to the operation in the access control such as delegation. Delegation operation is crucial in the pervasive computing environment, it allows the user to transfer his privileges to another user in the emergency situation. Although a lot of research appears in extending RBAC to support delegation, not much appears on providing a formal basis for choosing delegatees. However, automatically choosing delegatee is important for pervasive computing application. In the organization, choosing delegatee who is not suitable or not trustworthy for the task could put that organization business into jeopardy. In Chapter 9, we provide an approach that allows one to assess the trustworthiness of potential delegatees in the context of the task that is to be delegated. Our approach also ensure that the choice of the delegatee does not cause any security breaches.

Chapter 9

Trustworthy Delegation in Role-Based Access Control Model

The usage of trust to grant or deny the accessibility presented in Chapter 8 is just one way of using the trust. We can apply the trustworthiness to the operation in the access control such as delegation. The need to delegate, which allows the temporary grant or transfer of access rights, arise in many applications. Although a lot of research appears in extending Role-Based Access Control (RBAC) to support delegation, not much appears on providing a formal basis for choosing delegatees. In this chapter, we provide an approach that allows one to assess the trustworthiness of potential delegatees in the context of the task that is to be delegated. It is also important to ensure that the choice of the delegatee does not cause any security policy violation. Towards this end, we show how to formally analyze the application using Alloy analyzer to get assurance that our choice of delegatee does not cause a security breach.

9.1 Trust Modeling and Computation

Delegator refers to the role or user whose privileges are being transferred or granted to another role or user and the recipient of the privileges is termed *delegatee*. We show how the delegator can compute the trustworthiness of various entities in the context of the task that he is about to delegate.

Trust is a relationship between a truster and trustee with respect to a given context. The context in the case of delegation is the task for which delegation is needed. Trust relationship

for a given context depends on three factors: *properties*, *experiences* and *recommendations*. Properties are verifiable characteristics of the trustee. For instance, it may be the role and credentials possessed by the trustee. Experiences are the past interactions that the truster had with the trustee. Recommendations are provided by third-parties whom the truster trusts about the capabilities of the trustee. In the following, we describe how the trust relationship is quantified.

9.1.1 Quantifying Properties

Properties depend on the attributes of the entity and also the role associated with it.

9.1.1.1 Measuring Necessary Attributes $\mathcal{A}$

Every task in an organization requires some attributes of the user. For example, the task of performing surgery requires the user to be a certified surgeon. A task may require one or more attributes. The information about user attributes is contained in the credentials belonging to the user. Credentials are unforgeable and verifiable. Measuring necessary attributes requires evaluating what percentage of the necessary attributes are possessed by the user.

Let the set of attributes needed for task T_i be denoted by TA_i where $TA_i = \{a_{i1}, a_{i2}, \ldots, a_{in}\}$. Let wa_{i1}, wa_{i2}, ..., wa_{in} be the weights of attributes a_{i1}, a_{i2}, ..., a_{in} respectively. The weights of the attributes indicate their relative importance with respect to task T_i and $\sum_{r=1}^{n} wa_{ir} = 1$. Each user profile contains the credentials possessed by the user. Let the set of all attributes possessed by the user U_j be given by UA_j, where $UA_j = \{a_{j1}, a_{j2}, \ldots, a_{jm}\}$. Let $p = |TA_i \cap UA_j|$. The attribute value for user j with respect to task T_i, denoted by $\mathcal{A}_{ij}$, is calculated as follows: $\mathcal{A}_{ij} = \sum_{k=1}^{p} wa_k$ where wa_k $(1 \leq k \leq p)$ is the weight associated with attribute a_k and $a_k \in TA_i \cap UA_j$.

9.1.1.2 Measuring Role Attribute $\mathcal{R}$

The roles in the organization are arranged in the form of a hierarchy. The hierarchy can be represented as a labeled directed acyclic graph where the nodes represent the roles and the edges denote the hierarchical relationship. Note that, edges are drawn only for direct senior and junior relationship; transitive edges are not explicitly added. The edges in the hierarchy

are labeled with a number in the range (0,1] which indicates the closeness relationship between the roles. A number close to 0 indicates that the two roles are very distant, whereas a number close to 1 denotes that the roles are very close. We assume that the assignment of the numbers is done by the system administrator who has knowledge about the relationships between roles. If there is a path between role i and role j, the closeness relationship, denoted by $dist(r_i, r_j)$, is calculated by taking the product of all the edges constituting this path. Note that, if there are multiple paths connecting role i and role j, both the paths should give the same value. Otherwise, the role graph is said to be inconsistent. The formal definition of the role graph appears below.

Definition 50 (Weighted Role Hierarchy Graph)
Weighted role hierarchy graph, denoted by $WRH = (V, A)$, is a weighted directed acyclic graph where V is a set of nodes corresponding to the roles, and A is a set of arcs corresponding to the hierarchical relationship; $(v_i, v_j) \in A$ indicates that role v_j is directly senior to the role v_i. The weight of the edge (v_i, v_j), denoted by $w(v_i, v_j)$, is a number in the range (0,1] that gives a measure of the closeness of the two roles.

Each task T_i is associated with a set of roles TR_i who are authorized to execute this task. The roles associated with a task include roles who have the direct permission to execute those tasks, as well as those authorized by virtue of role hierarchy. Each user U_j also has a set of roles UR_j assigned to him. We choose the role belonging to the user that is closest to some role associated with the task. The distance between these two roles gives the role attribute $\mathcal{R}_{ij}$ of user U_j with respect to task T_i.

9.1.1.3 Computing the Properties Value

Some organizations may give greater importance to the role factor, whereas others may consider attribute factor to be more useful. Let w_a and w_r be the weights assigned to attributes and roles respectively, where $w_a, w_r \in [0,1]$ and $w_a + w_r = 1$. The exact values of w_a and w_r will be decided by the organization's policies. We use these weights to compute the property

value $\mathcal{P}_{ij}$ of user U_j with respect to task T_i: $\mathcal{P}_{ij} = w_a * \mathcal{A}_{ij} + w_r * \mathcal{R}_{ij}$

9.1.2 Quantifying Experience

Experience constitutes an important factor in delegation. A delegator is more likely going to choose a candidate as a delegatee if the delegatee has prior experience of doing the task. Two factors contribute towards experience. One factor is when the task was performed, and the second factor is how well the task was performed. Note that, information about these factors is stored in the users' profile, UP. Events that have occurred in the recent past have more influence than that occurred in the distant past. To accommodate this, we give the most recent slot has the highest weight and the most distant slot has the lowest one. For each time slot t_k, we get

Algorithm 2 Measuring Experience

Input: No. of slots n, User Profile UP_j
Output: $\mathcal{P}_{ij}$
Procedure:

$performance = 0$
for all $k : 1 \leq k \leq n$ **do**
  $weight\_slot_k = k$
end for
$total\_weight = n(n+1)/2$
for all $k : 1 \leq k \leq n$ **do**
  $w_k = (2 * k)/(n(n+1))$
end for
for all $k : 1 \leq k \leq n$ **do**
  $experience = experience + w_k * p_k$
end for
RETURN *experience*

the value for performance p_i. Recall that, performance on the task measures how well the task has been performed. The performance on the task can be graded on a scale of [0,1]. A value closer to 0 indicates poor performance, while that closer to 1 indicates excellent performance. Not performing the task in a slot, gives a performance value equal to 0. Algorithm 2 shows how to assign weights to the various time slots and evaluate the experience. Sometimes the past experience may not exactly match the the task, but is related to it. We show how to extrapolate the trust value in such cases in Section 9.3.

9.1.3 Quantifying Recommendation

A truster may obtain recommendation from one or more recommenders about the trustee with respect to its ability to perform the given task. In order to quantify the recommendation obtained from each recommender, we need to evaluate two factors. First, we need to obtain the trust value that the truster has with respect to the recommender providing recommendation about the trustee with respect to the given task. If the recommender is sufficiently trusted, then we need to get from him the recommendation value for the trustee. Algorithm 3 shows how to compute the recommendation component.

Algorithm 3 Measuring Recommendation

Input: Sequence of recommendations for user $U_j =< r_{1j}, r_{2j}, \ldots, r_{mj} >$, sequence of trust values for recommenders = $< t_1, t_2, \ldots, t_m >$
Output: $\mathcal{R}_{ij}$
Procedure:

$reco = 0; total = 0$
for all $k : 1 \leq k \leq m$ **do**
  $reco = reco + t_k * r_{kj}$
end for
for all $k : 1 \leq k \leq m$ **do**
  $total = total + t_k$
end for
$reco = reco/total$
RETURN *reco*

9.1.4 Computing Trustworthiness

Trust, with respect to a given task T_i for user U_j, denoted by $\mathcal{T}_{ij}$, depends on three factors, namely, properties $\mathcal{P}_{ij}$, experiences $\mathcal{E}_{ij}$, and recommendations, $\mathcal{R}_{ij}$. The exact weight assigned to each factor will be decided by the organization. Let w_p, w_e, and w_r be the weights assigned to the three factors respectively where $w_p, w_e, w_r \in [0,1]$ and $w_p + w_e + w_r = 1$. $\mathcal{T}_{ij}$ is given by, $\mathcal{T}_{ij} = w_p * \mathcal{P}_{ij} + w_e * \mathcal{E}_{ij} + w_r * \mathcal{R}_{ij}$. Note that $\mathcal{T}_{ij}$ will evaluate to some value in the range [0,1]. The delegator can choose a threshold value for trust $\mathcal{H}$. If $\mathcal{H} \leq \mathcal{T}_{ij}$, then user U_j can be a potential delegatee.

9.2 Using Trust Values in Delegation Chains

The privilege that a user receives can be further delegated resulting in what is known as a delegation chain. In some cases, we may want to limit the level of delegation. This level of delegation can be decided by the trustworthiness of the users involved in the delegation chain. Thus, delegation chain is dependent on the concept of trust chains. Trust chains are formalized using the concept of trust graphs defined below.

Definition 51 (Trust Graph)

Let $TG =< N, E >$ be the directed acyclic graph that represents trust relationship for a given context. The set of nodes N correspond to the entities in the system, and the set of edges E represent the trust relationship between the nodes. The edge (n_i, n_j) represents the trust relationship that node n_i has for node n_j with respect to the given task. The weight of the edge, denoted by $w(n_i, n_j)$, where $0 < w(n_i, n_j) \leq 1$, represents the trust value that node n_i has with respect to node n_j. Note that, the absence of a trust relationship between nodes n_r and n_s is indicated by the missing edge (n_r, n_s).

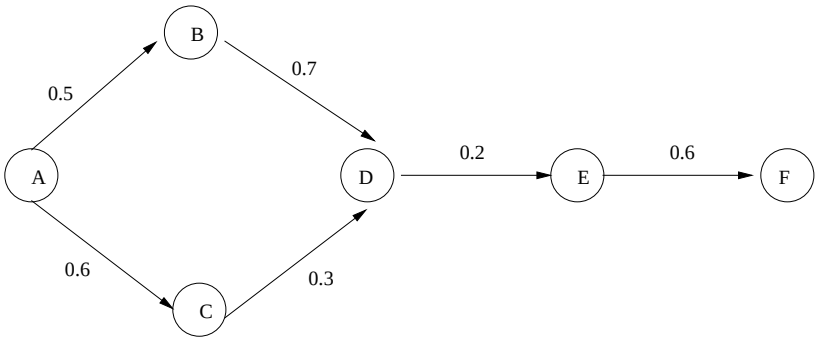


Figure 9.1: Example of a Trust Graph

Given a trust graph, we define two types of operators to compute transitive trust. One is the sequential operator, and the other is the parallel operator. Sequential and parallel operators and their desirable properties have been proposed by Agudo et al. [2].

Definition 52 (Sequential Operator)

Sequential operator, denoted by $\bigotimes$, is a binary operator that takes as input two trust values and returns a trust value that is the product of the two input values. Formally, $\bigotimes : [0,1] \times [0,1] \rightarrow$

$[0,1]$.

The sequential operator is used for computing the transitive trust value in a single path in the trust graph. Algorithm 4 gives the description of how transitive trust is computed. For instance, to compute the transitive trust that D has about F with respect to the given context is the product of 0.2 and 0.6 which equals 0.12.

Algorithm 4 Computing Transitive Trust in a Single Path

Input: Trust Path $(n_1, n_2, \ldots, n_k)$
Output: Transitive trust between nodes n_1 and n_k
Procedure:

$trans\_trust = 0$
for all $i : 1 \leq i \leq (k-2)$ **do**
  $trans\_trust = trans\_trust * w(n_i, n_{i+1}) \bigotimes w(n_{i+1}, n_{i+2})$
end for
RETURN $trans\_trust$

The sequential operator is not adequate for calculating transitive trust when multiple paths are involved. For example, in Figure 9.1, computing transitive trust that A has about E using the path (A,B,D,E) gives a different value than that obtained using the path (A,C,D,E). The value is 0.07 for the path (A,B,D,E) and it is 0.036 for the path (A,C,D,E). Such differences are reconciled using the parallel operator. The parallel operator becomes useful when there are multiple paths from one node to another.

Definition 53 (Parallel Operator)

Parallel operator, denoted by $\bigoplus$, is a binary operator that takes as input two trust values and returns a trust value that is the minimum of the two input values. Formally, $\bigoplus : [0,1] \times [0,1] \rightarrow [0,1]$.

Algorithm 5 shows how to compute transitive trust when the source and destination are connected by parallel paths. The transitive trust that A has for D, computed using this algorithm, equals 0.18.

The delegator can specify an acceptable level of trust to support delegation chains. Delegation is disallowed if the transitive trust value computed from the chain of delegation is below this minimum threshold.

Algorithm 5 Computing Transitive Trust in the Presence of Multiple Paths

Input: Trust Paths $(n_1, n_{2_1}, \ldots, n_{(k-1)_1}, n_k)$, $(n_1, n_{2_2}, \ldots, n_{(k-1)_2}, n_k)$, $\ldots$, $(n_1, n_{2_j}, \ldots, n_{(k-1)_j}, n_k)$

Output: Transitive trust between nodes n_1 and n_k

Procedure:

```
min = 1;
for all l : 1 ≤ l ≤ j do
   trans_trust_l = 0
end for
for all l : 1 ≤ l ≤ j do
   for all i : 1 ≤ i ≤ (k−2) do
      trans_trust_l = trans_trust_l + w(n_i, n_{i+1}) ⊗ w(n_{i+1}, n_{i+2})
   end for
end for
for all l : 1 ≤ l ≤ j do
   if trans_trust_l < min then
      min = trans_trust_l
   end if
end for
RETURN min
```

9.3 Extrapolating Trust Values

Sometimes the delegator may not have enough information to assess the trustworthiness of a user with respect to some given task. Although the user is not associated with a given task, it is possible that he has done some related tasks. To handle such scenarios, we define the different relationships that can exist among the tasks in an organization.

9.3.0.1 Specialization Relation

Different tasks may be related by the generalization/specialization relationship which is anti-symmetric and transitive. We use the notation $\mathcal{T}_i \subset \mathcal{T}_j$ to indicate that task $\mathcal{T}_i$ ($\mathcal{T}_j$) is a generalization (specialization) of task $\mathcal{T}_j$ ($\mathcal{T}_i$). For instance, *Surgery Treatment* $\subset$ *Heart Bypass Surgery Treatment* and *Heart Treatment* $\subset$ *Heart Bypass Surgery Treatment*. However, the degree of specialization is different in the two cases. The *degree of specialization* captures this difference. The degree of specialization is denoted as a fraction whose value is determined using domain knowledge.

9.3.0.2 Composition Relation

Sometimes tasks can be linked together using the composition relation. A task can either be *elementary* or *composite*. An elementary task is one which cannot be subdivided into other tasks, whereas a composite task is one that is composed from other tasks. The individual tasks that form a composite one are referred to as the *component* tasks. A component task can either be composite or elementary. We use the notation $\mathcal{T}_i \ll \mathcal{T}_j$ to indicate that the task $\mathcal{T}_i$ is a component of task $\mathcal{T}_j$. For instance, we may have the component tasks *operation* and *medication* that are part of the composite task *Catheter-assisted Procedures*. This is denoted as *operation* $\ll$ *Catheter-assisted Procedures*.

Sometimes a composite task $\mathcal{T}_i$ may be composed from the individual tasks $\mathcal{T}_j$, $\mathcal{T}_k$ and $\mathcal{T}_m$. All these tasks may not contribute equally to form $\mathcal{T}_i$. The *degree of composition* captures this idea. A degree of composition is associated with each composition relation. Since two tasks related by composition will not be exactly identical, the degree of composition is denoted as a fraction. The sum of all these fractions equals one if $\mathcal{T}_i$ is composed of $\mathcal{T}_j$, $\mathcal{T}_k$, and $\mathcal{T}_m$ only. If $\mathcal{T}_i$ is composed of $\mathcal{T}_j$, $\mathcal{T}_k$, and $\mathcal{T}_m$ and also other component contexts, then the sum of fractions associated with $\mathcal{T}_j$, $\mathcal{T}_k$, and $\mathcal{T}_m$ must be equal to or less than one. The exact value of the fraction representing the degree of composition will be determined by domain knowledge.

The generalization/specialization and composition relations are formally specified using the notion of *task graphs* defined below.

Definition 54 (Task Graph)
A task graph $\mathcal{TG} = \langle \mathcal{N}, \mathcal{E}_c \cup \mathcal{E}_s \rangle$ is a weighted directed acyclic graph satisfying the following conditions.

- $\mathcal{N}$ is a set of nodes where each node n_i is associated with a task $\mathcal{T}_i$.
- The set of edges in the graph can be partitioned into two sets $\mathcal{E}_c$ and $\mathcal{E}_s$. For each edge $(n_i,n_j) \in \mathcal{E}_c$, the task $\mathcal{T}_i$ corresponding to node n_i is a component of the task $\mathcal{T}_j$ corresponding to node n_j. The weight of the edge (n_i, n_j), denoted by $w(n_i, n_j)$, indicates the percentage of component task that makes up the composite one. For each edge (n_i,n_j)

$\in \mathcal{E}_s$, the task $\mathcal{T}_i$ corresponding to node n_i is a specialization of task $\mathcal{T}_j$ corresponding to node n_j. The weight of the edge (n_i, n_j), denoted by $w(n_i, n_j)$, indicates the degree of specialization.

9.3.1 Computing the Degree of Specialization and Composition

Consider two tasks $\mathcal{T}_i$ and $\mathcal{T}_j$ where $\mathcal{T}_i \subset \mathcal{T}_j$, that is, $\mathcal{T}_j$ is a specialization of $\mathcal{T}_i$. The degree of specialization is computed as follows. Let n_i, n_j be the nodes corresponding to tasks $\mathcal{T}_i$ and $\mathcal{T}_j$ in the weighted graph. Let the path from n_i to n_j consisting of specialization edges be denoted as $(n_i, n_{i+1}, n_{i+2}, \ldots, n_{j-1}, n_j)$. The degree of specialization = $\Pi_{p=i}^{j-1} w(n_p, n_{p+1})$. This corresponds to our notion that the similarity decreases as the length of the path from the generalized node to the specialized node increases. Note that, in real world there may be multiple paths from $\mathcal{T}_i$ to $\mathcal{T}_j$. In such cases, it is important that the degree of specialization yield the same values when any of these paths are used for computation.

Consider two tasks $\mathcal{T}_i$ and $\mathcal{T}_j$ such that $\mathcal{T}_j$ is a component of $\mathcal{T}_i$. Degree of composition captures what portion of $\mathcal{T}_i$ is made up of $\mathcal{T}_j$. The degree of composition is computed as follows. Let n_i, n_j be the nodes corresponding to contexts $\mathcal{T}_i$ and $\mathcal{T}_j$ in the task graph. Let there be m paths consisting of composition edges from n_i to n_j. Let the qth path ($1 \leq q \leq m$) from n_i to n_j be denoted as $(n_i, n_{i_q+1}, n_{i_q+2}, \ldots, n_{j_q-1}, n_j)$. The degree of composition $= \sum_{q=1}^{m} (w(n_i, n_{i_q+1}) \times w(n_{j_q-1}, n_j) \times \Pi_{p=i_q+1}^{j_q-2} w(n_p, n_{p+1}))$.

9.4 Trust Computation for Example Application

Consider a small healthcare organization that has six roles, namely, *senior doctor*, *junior doctor*, *cardiologist*, *surgeon*, *physician's assistant* and *patient*. *senior doctor* is senior to *junior doctor*, and *junior doctor* is senior to *cardiologist* and *physician's assistant*. Allen and Miller are assigned to *senior doctor*, Bell and Nelson are assigned to *junior doctor*, Cox is assigned to *cardiologist*, and Davis is assigned to *physician's assistant*. Allen is also assigned to *surgeon* and Evans is assigned to *patient*. Allen is the assigned surgeon for performing Coronary Artery Disease Angioplasty (CAD type A) surgery on patient Evans. Since Allen

has to leave town for family emergency, he must delegate the surgeon role to another doctor. He cannot delegate the surgeon role to his two trusted colleagues, Miller and Nelson, because they will be on vacation. The hospital policy requires that a person assigned to a doctor role or senior can be delegated the role of surgeon. This rules out Davis. Thus, he computes trust values for the only two viable candidates, Bell and Cox.

Quantifying Properties: To perform the CAD type A surgery, the hospital requires the following attributes from the candidates. First, the candidate should be a doctor ($a_1 = doctor$) and he should be able to perform a CAD type A surgery ($a_2 = Surgery_A$). So, $TA = \{doctor, Surgery_A\}$. The hospital policy ranks the ability to perform a CAD type A surgery higher than the doctor position, so the policy administrator assigned $w_{Surgery_A} = 0.7$ and $w_{doctor} = 0.3$. The hospital administrator assigned the value of closeness equal to 0.6 between roles *Senior Doctor* and *Junior Doctor* ($dist$(*Senior Doctor, Junior Doctor*)=0.6), and that between roles *Junior Doctor* and *Cardiologist* equals 0.3 ($dist$(*Junior Doctor, Cardiologist*)=0.3). Hence, by using the computation method explained in Section 9.3, we get the value of closeness between role *Senior Doctor* and *Cardiologist* equals to $0.6 * 0.3 = 0.18$ ($dist$(*Senior Doctor, Cardiologist*)=0.18). The hospital policy ranks the importance of necessary attributes and role attributes equally, hence $w_a = w_r = 0.5$.

Now, we quantify the properties of both candidates. Bell is a doctor who can perform the CAD type A surgery ($UA_{Bell} = \{doctor, Surgery_A\}$), and Cox is a cardiologist who can perform a bypass surgery ($UA_{Cox} = \{cardiologist, Surgery_B\}$). So, $\mathcal{A}_{Bell} = w_{Surgery_A} + w_{doctor} = 0.7 + 0.3 = 1$ and $\mathcal{A}_{Cox} = w_{doctor} = 0.3$. Since Bell is a junior doctor, $\mathcal{R}_{Bell} = dist$ (*Senior Doctor, Junior Doctor*)=0.6. Since Cox is a cardiologist, $\mathcal{R}_{Cox} = dist$ (*Senior Doctor, Cardiologist*)=0.18.

Using this information, we calculate the properties value of the candidates:

$\mathcal{P}_{Bell} = w_a * \mathcal{A}_{Bell} + w_r * \mathcal{R}_{Bell} = 0.5 * 1 + 0.5 * 0.6 = 0.8$, and

$\mathcal{P}_{Cox} = w_a * \mathcal{A}_{Cox} + w_r * \mathcal{R}_{Cox} = 0.5 * 0.3 + 0.5 * 0.18 = 0.24$.

Quantifying Experience: Here the experience is quantified based on the number of heart operations the candidates have done in the past five years and the unit of the slot of the time period is equal to one year. The weight for each time slot where $slot_1$ represents the time

period closest to the present time is defined by policy as follow: $w_{slot_1} = 1, w_{slot_2} = 0.8, w_{slot_3} = 0.6, w_{slot_4} = 0.4$, and $w_{slot_5} = 0.2$. Bell has performed surgery once 300 days ago ($slot_1$) with performance 0.7 ($p_{Bell_{slot_1}} = 0.7$) and Cox has performed surgery once 700 days ago ($slot_2$) with performance 0.8 ($p_{Cox_{slot_2}} = 0.8$). Thus, the experience value of both candidates can be calculated as follow:

$\mathcal{E}_{Bell} = \sum_{i=1}^{5} w_{slot_i} * p_{Bell_{slot_i}} = 1 * 0.7 + 0 + 0 + 0 + 0 = 0.7$, and

$\mathcal{E}_{Cox} = \sum_{i=1}^{5} w_{slot_i} * p_{Cox_{slot_i}} = 0 + 0.8 * 0.8 + 0 + 0 + 0 = 0.64.$

Quantifying Recommendation: Here, we have two recommenders–Miller and Nelson. According to hospital policy, the recommendation coming from senior doctor is more trustworthy than the one coming from junior doctor. So, the administrator set the trust value that hospital has about Miller (t_{Miller}) to 0.8 and the trust value that hospital has about Nelson (t_{Nelson}) to 0.2. Miller recommendation for Bell ($r_{MillerBell}$) and Cox ($r_{MillerCox}$) are 0.4 and 0.6, respectively. Nelson recommendation for Bell ($r_{NelsonBell}$) and Cox ($r_{NelsonCox}$) are 0.9 and 0.2, respectively. The computation results yield the recommendation for Bell and Cox as follow:

$\mathcal{R}_{Bell} = \frac{t_{Miller} * r_{MillerBell} + t_{Nelson} * r_{NelsonBell}}{t_{Miller} + t_{Nelson}} = \frac{0.8 * 0.4 + 0.2 * 0.9}{0.8 + 0.2} = 0.5$, and

$\mathcal{R}_{Cox} = \frac{t_{Miller} * r_{MillerCox} + t_{Nelson} * r_{NelsonCox}}{t_{Miller} + t_{Nelson}} = \frac{0.8 * 0.6 + 0.2 * 0.2}{0.8 + 0.2} = 0.52.$

Computing Trustworthiness: Allen prefers the delegatee with more experience. So, he set the weights for properties (w_p), experience (w_e), and recommendation (w_r) to 0.2, 0.6, and 0.2, respectively. The trustworthiness of Bell and Cox can be computed as follow:

$\mathcal{T}_{Bell} = w_p * \mathcal{P}_{Bell} + w_e * \mathcal{E}_{Bell} + w_r * \mathcal{R}_{Bell} = 0.2 * 0.8 + 0.6 * 0.7 + 0.2 * 0.5 = 0.68$, and

$\mathcal{T}_{Cox} = w_p * \mathcal{P}_{Cox} + w_e * \mathcal{E}_{Cox} + w_r * \mathcal{R}_{Cox} = 0.2 * 0.24 + 0.6 * 0.64 + 0.2 * 0.52 = 0.54$

Bell is selected to be the delegatee after comparing the trustworthiness values between both candidates.

9.5 Model Analysis

Once we have determined the most trustworthy candidate, we need to formally ensure that the choice of this delegatee does not cause a security breach. We do the formal analysis

using the Alloy Analyzer. An Alloy model consists of *signature* declarations, *fields*, *facts* and *predicates*. Each signature consists of a set of *atoms* which are the basic entities in Alloy. Atoms are *indivisible* (they cannot be divided into smaller parts), *immutable* (their properties do not change) and *uninterpreted* (they do not have any inherent properties). Each field belongs to a signature and represents a relation between two or more signatures. A relation denotes a set of tuples of atoms. Facts are statements that define constraints on the elements of the model. Predicates are parameterized constraints that can be invoked from within facts or other predicates.

The basic types in the access control model, such as, *User*, and *Role* are represented as signatures. For instance, the declarations shown below define a set named *User*, and a set named *Role* that represents the set of all users, and roles in the system. Note that we use the *abstract* signature to represent these sets, and the different of users, and roles are modeled as the subsignatures of each signature. The analyzer will then recognize that users, and roles consist of only these different types, and nothing else.

```
abstract sig User{}
one sig Allen, Bell, Cox, Davis, Evans,
            Miller, Nelson extends User{}


abstract sig Role{}
one sig SeniorDoctor, JuniorDoctor, Assistant,
            Cardiologist, Surgeon, Patient extends Role{}
```

The different relationships between the RBAC components are also expressed as signatures. Signature *UserRoleAssign* which represents the roles assigned to user has a field called *URAsmember* that maps to a cartesian product of *User* an d *Role*. Signature *UserRoleAcquire* which represents the roles user can acquire through the assignment and role hierarchy has a field called *URAcqmember* that maps to a cartesian product of *User* and *Role*. We use the signature *RoleHierarchy* to represent role hierarchy relationship.

```
one sig UserRoleAssign{URAsmember: User -> Role}
```

```
one sig UserRoleAcquire{URAcqmember: User -> Role}
one sig RoleHierarchy{RHmember : Role -> Role}
```

The various invariants in the RBAC model are represented as facts in Alloy. For instance, the fact *URAcq* states that user can acquire all roles assign ed to him together with all of his junior roles. This is specified in Alloy as shown below. Other invariants are modeled in a similar manner.

```
fact URAcq{
UserRoleAcquire.URAcqmember =
UserRoleAssign.URAsmember +
(UserRoleAssign.URAsmember).^(RoleHierarchy.RHmember)}
```

The policy constraints are modeled as predicates. First, consider the cardinality constraint. The following constraint says that role *r* can be assigned to only one user.

```
pred Cardinality(r: Role, uracq: User->Role){
        (#((uracq).r) >= 1) &&
        (#((uracq).r) <= 1)}
```

Next, consider the prerequisite constraint that says that if a user *u* can acquire role *r*1, then he can also acquire role *r*2. The other forms are modeled in a separate manner.

```
pred Prerequisite(u:User, r1, r2: Role,
uracq: User->Role){
    (u->r2 in uracq) => (u->r1 in uracq)}
```

The separation of duty constraint says that if a user *u* can acquire role *r*1, then he cannot acquire the conflicting role *r*2.

```
pred SoD(u:User, r1, r2: Role, uracq: User->Role){
    (u->r1 in uracq) => not (u->r2 in uracq)}
```

The different types of delegation are also modeled as predicates. The grant and transfer operation can be modeled as follows:

```
pred Grant[u: User, r: Role,
   uracq, uracq': User->Role]{
     uracq' = uracq + (u->r)}

pred Transfer[u1, u2: User, r: Role,
   uracq, uracq': User->Role]{
     uracq' = uracq + (u2->r) - (u1->r)}
```

Finally, we need to verify whether the selected delegatee could cause any security policy violation. We create an *assertion* that specifies the properties we want to check. After we create the assertion, we will let ALLOY analyzer validate the assertion by using *check* command. If our assertion is wrong in the specified scope, ALLOY analyzer will show the counterexample. For example, suppose we want to check whether separation of duty constraint is violated when Allen delegates his role to Bell. The assertion below will check whether the separation of duty constraint is violated after the transfer operation. The separation of duty constraint says that user cannot be assigned both *Assistant* and *Surgeon* roles. The counterexample illustrates that even though user *Bell* is not assigned to *Assistant* role, he can still acquire it from the effect of role hierarchy.

```
assert TestConflict3{
     all u1, u2: User, r: Role, uracq, uracq': User->Role|
         ((u1 = Allen) && (u2 = Bell) && (r=Surgeon) &&
         (uracq = UserRoleAcquire.URAcqmember) &&
         (u1->r in UserRoleAcquire.URAcqmember) &&
         (u2->Assistant not in UserRoleAssign.URAsmember) &&
         Transfer[u1, u2, r, uracq, uracq']) =>
            SoD[u2, r, Assistant, uracq']}
```

```
check TestConflict3
```

The result shown that, although Bell is the most trustworthy candidate, we cannot choose him as Allen's delegatee. Next, we verify the situation where Cox, another candidate with the lower trustworthiness, is chosen as the delegatee. The assertion below will check whether the separation of duty constraint is violated after the transfer operation.

```
assert TestConflict4{
     all u1, u2: User, r: Role, uracq, uracq': User->Role|
         ((u1 = Allen) && (u2 = Cox) && (r=Surgeon) &&
         (uracq = UserRoleAcquire.URAcqmember) &&
         (u1->r in UserRoleAcquire.URAcqmember) &&
         (u2->Assistant not in UserRoleAssign.URAsmember) &&
         Transfer[u1, u2, r, uracq, uracq']) =>
            SoD[u2, r, Assistant, uracq']}
check TestConflict4
```

Here, the analyzer cannot find the counterexample, which means the separation of duty constraint defined in the model is not violated. This indicates that Cox is a more suitable delegatee for Allen. The complete version of the Alloy model of the small healthcare organization is shown in Appendix D.

9.6 Chapter Summary

Delegation gives temporary privilege to one or more users, that allows critical tasks to be completed. In this chapter, we provide a formal approach for choosing delegatees. The approach evaluates the trustworthiness of candidates, and then ensures that the chosen candidate does not cause a security breach. We also illustrate how trustworthiness can be used to decide on the length of the delegation chain.

Chapter 10

Conclusions and Future Work

10.1 Contributions

Traditional access control models are not sufficient for pervasive computing applications for several reasons. First, unlike traditional applications, pervasive computing applications usually do not have well-defined security perimeter–the entities an application will interact with or the resources that will be accessed may not be known in advance. Second, these applications are also dynamic in nature–the accessing entities may change, resources requiring protection may be created or modified, and an entity's access to resources may change during the course of the application, which make the resources protection during application execution extremely challenging. Third, pervasive computing applications use the knowledge of surrounding physical spaces to provide services; security policies designed for such applications must therefore use contextual information. Thus, new access control models and technologies are needed for pervasive computing applications.

With the growing use of wireless networks and mobile devices, we are moving towards an era where spatial and temporal information will be necessary for access control. The use of such information can be used for enhancing the security of an application, and it can also be exploited to launch attacks. For critical applications, a model for spatio-temporal-based access control is needed that increases the security of the application and ensures that the location information cannot be exploited to cause harm.

Motivated by this issue, we propose STRBAC model [67, 83], which incorporates time

and location constraints to the traditional RBAC entities and relationships. In this model, the access control is contingent upon the role of the user, his location, location of the object and the time of access. In a highly dynamic system such as the pervasive computing environment, it is necessary for the user to be able to transfer his privileges to another user during his absence to perform a critical task. To rectify this issue, we extend our model to incorporate the notion of delegation [68]. Our STRBAC model is efficient in the aspect that it can represent the different kind of constraints and relationship in the access control model. The transformation of the model to do the automate verification is feasible. However, semantically, the model is extremely complicated. To rectify this, we develop a well-formed semantic model called the Spatio-Temporal Role Based Access Control with Delegation (STARBACD) model based on graph-theoretic notation.

Delegation is needed in pervasive computing environments where some user or role may be temporarily unavailable to do some tasks. We proposed a formal approach for choosing delegatees. Our approach assessed the trustworthiness of potential delegates in the context of the task that is to be delegated. It also ensures using existing SAT-solvers that the choice of delegatee does not cause any security policy violation. Results of this work appeared in [87].

Our proposed spatio-temporal model has numerous constraints that are not orthogonal, but interact with each other in subtle ways. Thus, it is important to understand the interaction of the constraints before the model can be deployed. Manual analysis is tedious and error-prone. Towards this end, we investigated how Alloy can be used for automated analysis. Alloy is based on first-order logic and is supported by tools. The specification is converted into a boolean expression that is automatically verified by an embedded SAT-solver. Our analysis reveals that the various forms of spatio-temporal hierarchy conflicts with the numerous forms of spatio-temporal separation of duty constraints [83]. Subsequently, we performed the analysis on the spatio-temporal role-based access control model that supports delegation [84]. The analysis revealed that the delegation constraints may also conflict with separation of duty constraints.

Knowing the relationship among constraints is necessary, but not sufficient if it is to be used for applications. We need to do application-level analysis to ensure that the policies for

the given application have been correctly specified. Typically, applications are specified using the Unified Modeling Language (UML), and so are the policies. However, UML does not have automated tool support. Towards this end, we investigated how the policies specified in UML can be automatically converted to Alloy using UML2Alloy and the resulting specification verified using Alloy [86].

Although Alloy supports automated analysis, it has limitations with respect to the types of verification it can perform. For example, analyzing and understanding the behavior of the application using Alloy is non-trivial. Such analysis is needed for dynamic systems where we need to ensure that the system does not enter an undesirable state. Towards this end, we have shown how Coloured Petri Nets can be used for analyzing an application using our spatio-temporal role-based access control model [85]. We show how the STRBAC model can be transformed into a graph-theoretic notation called access control graph, which will be used as an input to create the CPNs. Our investigations reveal that even a modest increase in the number of places and transitions cause a significant increment to the number of states of the state space; this substantially raises the verification time. In order to generate a smaller number of states in the CPN model that does efficient verification, we propose to transform the access control graph into the privilege acquisition graph, which is essentially flattens out the hierarchical structure.

The entities a pervasive computing application will interact with, or the resources that it will access may not always be known in advance. Moreover, the accessing entities may change, resources requiring protection may be created or modified, and an entity's access to resources may change during the course of the application. Under this scenario, we need to ensure that each entity participating in the interaction is trustworthy. Towards this end, we proposed a class of trust-based access control models and expressed their semantics using graph theory. The models differ with respect to the features they provide, and the types of the trust constraints that they can support. This work appeared in [82].

10.2 Future Research

10.2.1 The Representation of the Location Constraints

In this dissertation, we have made some simplifying assumptions. We have assumed that the precise locations of subjects and objects are known at any given point of time. This might not be true in the pervasive computing environments, where the subjects are continually moving and objects are frequently relocated [79]. For example, a user might want to access the system resources while he is traveling on the plane. Here, the user's exact location cannot be determined, and hence our current model cannot grant the accessibility to the user based on his location. This motivates the need of the novel representation of the location constraints which could handle the uncertainty of the location.

Ardagna et al [4] propose three types of condition to capture the location uncertainty:

- *position-based conditions* this type of condition evaluates whether one entity is in a certain building or city or in the proximity of other entities
- *movement-based conditions* this type of condition evaluates the mobility of the entities such as their velocity, acceleration, or direction where they are headed
- *interaction-based conditions* this type of condition relating multiple entities; for instance, the number of entities within a given area

With the advocate of these three types of condition, we can specify access control policy which supports the uncertainty location constraints. For instance, user may access the information about the Golden Gate bridge from the plane only if his plane is headed toward San Francisco.

Towards this end, we plan to develop a new representation of the location constraints, which support these three types of condition.

10.2.2 The Representation of the Time Constraints

Our time constraints proposed in this dissertation are based on the simplifying assumptions that the time intervals are distinctively defined and the only time interval relationship supported in the model is the containment relationship. These assumptions may not be valid in the time-sensitive application such as workflow. Workflow consists of a set of the related tasks which must be done sequentially within the specific time. For instance, the time period of the *check authorizing* task must start after the time period of the *check issuing task* started and it must end within 2 days after the time period of the *check issuing* task ended. Such temporal constraint cannot be represented by using our current model. Towards this end, we plan to improve our representation of the time constraints to handle such complicated temporal relationships.

10.2.3 Extension to Dynamic Workflow

Pervasive computing applications are typically modeled using workflow or other advanced transactions. Typically, a workflow is a representation of a given process that is made up of well defined collection of activities, referred to as tasks. Each of these tasks, which is serving a given function in the overall process represented by the workflow, has certain information input requirement and may produce information as part of its output. Tasks that make up a workflow are usually related and dependent on one another. These task dependencies are known as *intra-workflow* dependencies. Task dependencies may also exist across workflows, such dependencies are referred to as *inter-workflow* dependencies. Control flow dependencies, value dependencies and external dependencies are types of task dependencies [1]. Figure 10.1 demonstrates such dependencies. Here, the control flow dependency states that user cannot reserve the hotel unless he bought the plane ticket first. Next, the value dependency specifies that the budget he could spend on the hotel depends on how much he paid for the plane ticket. Finally, the external dependency states that all these tasks must be done during the office hours.

In such applications, security is one of the main concerns. We need to ensure that the user who executes task is the one who is authorized for that task. Also, we should not allow any user to be assigned to two conflicting tasks which permit the user to commit a fraud to the

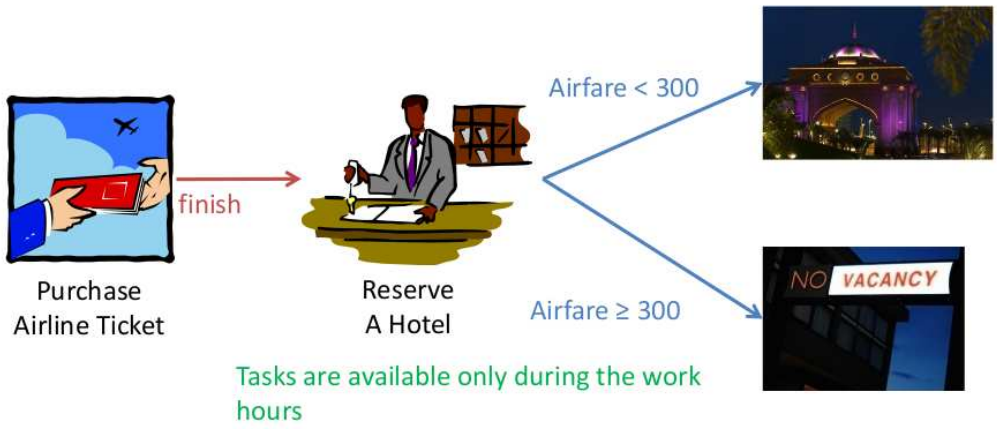


Figure 10.1: Example of task dependencies in workflow

organization or cause the security breaches. Next, the access control must support the mandatory constraints existing in the workflow environment such as task dependencies, and binding constraints. With regarding to this, an access control is needed to perform these functions. In most of the situation, these constraints are unavoidably related with the time and location constraints. For example, the task *check approving* must be done at the account manager office within 10 days after task *check issuing* completed. To be useful, we not only must fulfill the requirements mentioned earlier, this access control model should be effective in the sense of policy administration. We believe that our proposed STRBAC can be implemented to serve these objectives. Towards this end, we plan to apply the STRBAC model to the workflow system.

10.2.4 Model Analysis

In addition to the analysis of the interactions between different features in the access control model, we have to focus on analyzing the properties of workflow dependencies; classify the dependencies according to different criteria; discuss the interaction and impacts of dependencies on execution of tasks. More importantly, we will formally analyze all kinds of dependency conflict. Dependency conflict could be either between the dependencies themselves or between the dependency and access control constraints. The conflicts could take various forms and vary

from case to case.

Model analysis is needed to ensure the executability and security compliance of the dynamic workflow. Additional work is needed for the model analysis of the STRBAC-embedded workflow. We plan to develop the verification methodology for the integrated model for workflow.

In this dissertation, we have proven that our analysis methodology can be used to analyze our model. We plan to implement our analysis approach to other types of access control model.

10.2.5 Dynamism Analysis

Pervasive computing applications are dynamic in nature. While the application is executing, the entities requiring access or the resources needing protection may change. In the face of such dynamism, it is essential to ensure that access control breaches do not occur. Since the analysis must be done in real-time, it is important to minimize the verification time. Towards this end, we provide techniques for incremental analysis with good time complexity results. For example, to detect SoD violations in a dynamic graph, we need to find whether the nodes connected by SoD constraints have a common predecessor. Applying a naive algorithm based on Depth First Search, requires $O(kE)$ time for each change applied to the graph, where k is the number of SoD constraints and E is the number of edges. We improve upon this result significantly by proposing a new common predecessor detecting algorithm in a dynamic graph. However, there still be a limitation to this approach, which is, the changes of access control configuration must be done in a monotonic fashion. For instance, if the access control configuration is changed by adding the new relationships to the model, then the subsequent changes must be the addition of relationships only, the deletion of relationship is not allowed. To complete the analysis approach, we plan to improve our approach so that it could analyze the access control model with a sequence of different types of configuration modification.

10.2.6 Implementation

Implementation will require us to investigate additional issues, such as, how to store location and time information and perform operations involving spatio-temporal constraints in an efficient manner. The same goes for the trust based access control model. Defining entities' negative and positive actions which could affect the accessibility between entities is a challenging task. Implementing the model for real-world applications will further help refine our model and make it more useful.

Appendix A

ALLOY Specification of the Spatio-Temporal Role-Based Access Control Model

```
module STRBAC

sig Time{}
sig Location{}

// For solving the dimension limitation problem
sig TimeLoc{
     dur : Time,
     loc : Location}

sig User{}
sig Role{
     RoleAllocLoc: Location,
     RoleAllocDur: Time,
     RoleEnableLoc: Location,
     RoleEnableDur: Time}
sig Permission{
     PermRoleLoc: Role->Location,
     PermObjLoc: Object->Location,
     PermDur: Time
}
sig Object{}

one sig RoleEnable {member : Role-> Time ->Location}
one sig UserRoleAssignment{member : User -> Role ->Time ->Location}
one sig RolePermissionAssignment{member : Role-> Permission ->Time->Location}
one sig UserLocation{member : User->Time->Location}
one sig ObjLocation{member : Object->Time->Location}
one sig UserRoleActivate{member : User-> Role->Time->Location}
one sig PermRoleAcquire{member : Role->Permission->Time->Location}
one sig PermUserAcquire{member : User->Object->Permission->TimeLoc}
```

```
abstract sig RoleHierarchy{member : Role -> Role}
sig UPIH, TPIH, LPIH, TLPIH, UAH, TAH, LAH, TLAH extends RoleHierarchy{}

fact ULoc{
     all u: User, uloc: UserLocation, d: Time, l1, l2: Location |
          (((u->d->l1) in uloc.member) && ((u->d->l2) in uloc.member)) <=>
          ((l1 in l2) || (l2 in l1))}

fact ObjLoc{
     all o: Object, oloc: ObjLocation, d: Time, l1, l2: Location |
          (((o->d->l1) in oloc.member) && ((o->d->l2) in oloc.member)) <=>
          ((l1 in l2) || (l2 in l1))}

// Each user must has role assigned to him
fact UserRole{
     all u: User, uras: UserRoleAssignment | some r: Role |
          u->r in (uras.member).Location.Time}

fact URAssign{
     all u: User, r: Role, d: Time, l: Location, ura: UserRoleAssignment,
     uloc: UserLocation |
          ((u->r->d->l) in ura.member) => (((u->d->l) in uloc.member) &&
          (l in r.RoleAllocLoc) && (d in r.RoleAllocDur))}

fact URActivate1{
     all u: User, sr, jr: Role, d: Time, l: Location, uras: UserRoleAssignment,
     urac: UserRoleActivate,
     uah: UAH, tah: TAH, lah: LAH, tlah: TLAH |
          ((u->jr->d->l) in urac.member) && (u->sr in (uras.member).Location.Time) &&
          (jr !in sr.^((uah + tah + lah + tlah).member)) =>
          (((u->jr->d->l) in uras.member) && (l in jr.RoleEnableLoc) &&
          (d in jr.RoleEnableDur))}

fact URActivate2{
     all u: User, sr, jr: Role, d: Time, l: Location, uras: UserRoleAssignment,
     uract: UserRoleActivate, uah: UAH, tah: TAH, lah: LAH, tlah: TLAH |
          ((u->jr->d->l in uract.member) && (u->sr in (uras.member).Location.Time) &&
          (u->jr !in (uras.member).Location.Time)) =>
          (jr in sr.^((uah + tah + lah + tlah).member))}

fact NocycleRH{
     all r: Role, RH: RoleHierarchy| r !in r.^(RH.member)}


// All types of hierarchy are disjointed
fact ScopeRH{
     all rh: RoleHierarchy, upih: UPIH, tpih: TPIH, lpih: LPIH, tlpih: TLPIH, uah: UAH,
     tah: TAH, lah: LAH, tlah: TLAH |
          (upih.member = rh.member - (tpih.member + lpih.member + tlpih.member +
                          uah.member + tah.member + lah.member + tlah.member)) &&
          (tpih.member = rh.member - (upih.member + lpih.member + tlpih.member +
                          uah.member + tah.member + lah.member + tlah.member)) &&
          (lpih.member = rh.member - (upih.member + tpih.member + tlpih.member +
```

```
                        uah.member + tah.member + lah.member + tlah.member)) &&
          (tlpih.member = rh.member - (upih.member + tpih.member + lpih.member +
                        uah.member + tah.member + lah.member + tlah.member)) &&
          (uah.member = rh.member - (upih.member + tpih.member + lpih.member +
                        tlpih.member + tah.member + lah.member + tlah.member)) &&
          (tah.member = rh.member - (upih.member + tpih.member + lpih.member +
                        tlpih.member + uah.member + lah.member + tlah.member)) &&
          (lah.member = rh.member - (upih.member + tpih.member + lpih.member +
                        tlpih.member + uah.member + tah.member + tlah.member)) &&
          (tlah.member = rh.member - (upih.member + tpih.member + lpih.member +
                        tlpih.member + uah.member + tah.member + lah.member))}


// Each role must has at least one permission assigned to it
fact RoleFact{
     all r: Role, rpa: RolePermissionAssignment |
          r in (rpa.member).Location.Time.Permission}

// All permissions assigned to roles can be acquired
fact RPAFact{
     all disj r: Role, p: Permission, d: Time, l : Location,
     rpa: RolePermissionAssignment, pra : PermRoleAcquire |
          (r->p->d->l in rpa.member) =>
             (r->p->d->l in pra.member)}

// All roles can acquire only their own assigned or inherited permissions
fact PRAFact{
     all disj r1, r2: Role, p: Permission, d1, d2: Time, l1, l2 : Location,
     rpa: RolePermissionAssignment, pra : PermRoleAcquire,
     upih: UPIH, tpih: TPIH, lpih: LPIH, tlpih: TLPIH |
          (r1->p->d1->l1 in pra.member) =>
             ((r1->p->d1->l1 in rpa.member) ||
             ((r2->p->d2->l2 in rpa.member) &&
             (r1->r2 in ((upih + tpih + lpih + tlpih).member))))}

// Permission User Acquire
fact PUAFact{
     all r: Role, p: Permission, u: User, d: Time, l : Location, o: Object,
     tl: TimeLoc, pra : PermRoleAcquire, puacq: PermUserAcquire,
     ol: ObjLocation, urac: UserRoleActivate |
          ((r->p->d->l in pra.member) &&
          (u->r->d->l in urac.member) &&
          (o->d->l in ol.member) &&
          (o->l in p.PermObjLoc) &&
          (tl.dur = d) && (tl.loc = l)) =>
              (u->o->p->tl in puacq.member)
}

//Unrestricted Permission Inheritance Hierarchy
fact UPIHFact{
     all sr, jr: Role, p: Permission, d: Time, l: Location, upih: UPIH,
         rpa: RolePermissionAssignment, pra: PermRoleAcquire |
         ((sr->jr in upih.member) && (jr->p->d->l in pra.member) &&
         (sr->p !in (rpa.member).Location.Time)) =>
```

```
            (sr->p->sr.RoleEnableDur->sr.RoleEnableLoc) in pra.member}

//Time Restricted Permission Inheritance Hierarchy
fact TPIHFact{
     all sr, jr: Role, p: Permission, d: Time, l: Location, tpih: TPIH,
         rpa: RolePermissionAssignment, pra: PermRoleAcquire |
         ((sr->jr in tpih.member) && (jr->p->d->l in pra.member) &&
         (sr->p !in (rpa.member).Location.Time)) =>
              (sr->p->d->sr.RoleEnableLoc) in pra.member}

//Location Restricted Permission Inheritance Hierarchy
fact LPIHFact{
     all sr, jr: Role, p: Permission, d: Time, l: Location, lpih: LPIH,
         rpa: RolePermissionAssignment, pra: PermRoleAcquire |
         ((sr->jr in lpih.member) && (jr->p->d->l in pra.member) &&
         (sr->p !in (rpa.member).Location.Time)) =>
              (sr->p->sr.RoleEnableDur->l) in pra.member}

//Time Location Restricted Permission Inheritance Hierarchy
fact TLPIHFact{
     all sr, jr: Role, p: Permission, d: Time, l: Location, tlpih: TLPIH,
         rpa: RolePermissionAssignment, pra: PermRoleAcquire |
         ((sr->jr in tlpih.member) && (jr->p->d->l in pra.member) &&
         (sr->p !in (rpa.member).Location.Time)) =>
              (sr->p->d->l) in pra.member}

//Unrestricted Activation Hierarchy
fact UAHFact{
     all disj sr, jr: Role, u: User, d: Time, l: Location, uah: UAH,
         uras: UserRoleAssignment, uract: UserRoleActivate |
         ((sr->jr in uah.member) && (u->sr->d->l in uract.member) &&
         (u->jr !in (uras.member).Location.Time) && (d in sr.RoleEnableDur) &&
         (l in sr.RoleEnableLoc)) =>
              (u->jr->d->l) in uract.member}

//Time Restricted Activation Hierarchy
fact TAHFact{
     all disj sr, jr: Role, u: User, d, d': Time, l: Location, tah: TAH,
         uras: UserRoleAssignment, uract: UserRoleActivate |
         ((sr->jr in tah.member) && (u->sr->d->l in uract.member) &&
         (u->jr !in (uras.member).Location.Time) && (d in sr.RoleEnableDur) &&
         (d' in jr.RoleEnableDur) && (l in sr.RoleEnableLoc)) =>
              (u->jr->d'->l) in uract.member}

//Location Restricted Activation Hierarchy
fact LAHFact{
     all disj sr, jr: Role, u: User, d: Time, l, l': Location, lah: LAH,
         uras: UserRoleAssignment, uract: UserRoleActivate |
         ((sr->jr in lah.member) && (u->sr->d->l in uract.member) &&
         (u->jr !in (uras.member).Location.Time) && (d in sr.RoleEnableDur) &&
         (l in sr.RoleEnableLoc) && (l' in jr.RoleEnableLoc)) =>
              (u->jr->d->l') in uract.member}

//Time Location Restricted Activation Hierarchy
```

```
fact TLAHFact{
     all disj sr, jr: Role, u: User, d, d': Time, l, l': Location, tlah: TLAH,
         uras: UserRoleAssignment, uract: UserRoleActivate |
         ((sr->jr in tlah.member) && (u->sr->d->l in uract.member) &&
         (u->jr !in (uras.member).Location.Time) && (d in sr.RoleEnableDur) &&
         (d' in jr.RoleEnableDur) && (l in sr.RoleEnableLoc) &&
         (l' in jr.RoleEnableLoc)) =>
              (u->jr->d'->l') in uract.member}

//Weak Form of SSoD-User Role Assignment
pred W_SSoD_URA(u: User, disj r1, r2: Role, ura: UserRoleAssignment.member,
d: Time, l: Location){
     ((u->r1->d->l) in ura) => ((u->r2->d->l) not in ura)}

//Strong Temporal Form of SSoD-User Role Assignment
pred ST_SSoD_URA(u: User, disj r1, r2: Role, ura: UserRoleAssignment.member,
d, d': Time, l: Location){
     ((u->r1->d->l) in ura) => ((u->r2->d'->l) not in ura)}

//Strong Spatial Form of SSoD-User Role Assignment
pred SS_SSoD_URA(u: User, disj r1, r2: Role, ura: UserRoleAssignment.member,
d: Time, l, l': Location){
     ((u->r1->d->l) in ura) => ((u->r2->d->l') not in ura)}

//Strong Form of SSoD-User Role Assignment
pred S_SSoD_URA(u: User, disj r1, r2: Role, ura: UserRoleAssignment.member,
d, d': Time, l, l': Location){
     ((u->r1->d->l) in ura) => ((u->r2->d'->l') not in ura)}

//Weak Form of SSoD-Permission Role Assignment
pred W_SSoD_PRA(r: Role, disj p, q : Permission,
d: Time, l: Location){
     all pra: PermRoleAcquire.member |
          ((r->p->d->l) in pra) => ((r->q->d->l) not in pra)}

//Strong Temporal Form of SSoD-Permission Role Assignment
pred ST_SSoD_PRA(r: Role, disj p, q : Permission,
d, d': Time, l: Location){
     all pra: PermRoleAcquire.member |
          ((r->p->d->l) in pra) => ((r->q->d'->l) not in pra)}

//Strong Spatial Form of SSoD-Permission Role Assignment
pred SS_SSoD_PRA(r: Role, disj p, q : Permission,
d: Time, l, l': Location){
     all pra: PermRoleAcquire.member |
          ((r->p->d->l) in pra) => ((r->q->d->l') not in pra)}

//Strong Form of SSoD-Permission Role Assignment
pred S_SSoD_PRA(r: Role, disj p, q : Permission,
d, d': Time, l, l': Location){
     all pra: PermRoleAcquire.member |
          ((r->p->d->l) in pra) => ((r->q->d'->l') not in pra)}

//Weak Form of DSoD
```

```
pred W_DSoD(u: User, disj r1, r2: Role, d: Time, l: Location){
     all urac: UserRoleActivate.member |
          ((u->r1->d->l) in urac) => ((u->r2->d->l) not in urac)}

//Strong Temporal Form of DSoD
pred ST_DSoD(u: User, disj r1, r2: Role, d, d': Time, l: Location){
     all urac: UserRoleActivate.member |
          ((u->r1->d->l) in urac) => ((u->r2->d'->l) not in urac)}

//Strong Spatial Form of DSoD
pred SS_DSoD(u: User, disj r1, r2: Role, d: Time, l, l': Location){
     all urac: UserRoleActivate.member |
          ((u->r1->d->l) in urac) => ((u->r2->d->l') not in urac)}

//Strong Form of DSoD
pred S_DSoD(u: User, disj r1, r2: Role, d, d': Time, l, l': Location){
     all urac: UserRoleActivate.member |
          ((u->r1->d->l) in urac) => ((u->r2->d'->l') not in urac)}

//U2U Unrestricted Permission Delegation
pred u2uUPD(disj dtr, dte: User, p: Permission){
     all o: Object, tl: TimeLoc, puacq: PermUserAcquire |
         (dtr->o->p->tl in puacq.member) =>
               (dte->o->p->tl in puacq.member)}

//U2U Time Restricted Permission Delegation
pred u2uTPD(disj dtr, dte: User, p: Permission, d': Time){
     all o: Object, tl, tl': TimeLoc, puacq: PermUserAcquire |
         ((d' in tl.dur) && (d' != tl.dur) && (tl'.dur = d') &&
         (tl'.loc = tl.loc) && (dtr->o->p->tl in puacq.member)) =>
               (dte->o->p->tl' in puacq.member)}

//U2U Location Restricted Permission Delegation
pred u2uLPD(disj dtr, dte: User, p: Permission, l': Location){
     all o: Object, tl, tl': TimeLoc, puacq: PermUserAcquire |
         ((tl'.dur = tl.dur) && (l' in tl.loc) && (l' != tl.loc) && (tl'.loc = l') &&
         (dtr->o->p->tl in puacq.member)) =>
               (dte->o->p->tl' in puacq.member)}

//U2U Time Location Restricted Permission Delegation
pred u2uTLPD(disj dtr, dte: User, p: Permission, d': Time, l': Location){
     all o: Object, tl, tl': TimeLoc, puacq: PermUserAcquire |
         ((d' in tl.dur) && (d' != tl.dur) && (l' in tl.loc) && (l' != tl.loc) &&
         (tl'.dur = d') && (tl'.loc = l') &&
         (dtr->o->p->tl in puacq.member)) =>
               (dte->o->p->tl' in puacq.member)}

//U2U Unrestricted Role Delegation
pred u2uURD(disj dtr, dte: User, r: Role){
     all d: Time, l: Location, urac: UserRoleActivate |
         (dtr->r->d->l in urac.member) =>
               (dte->r->d->l in urac.member)}

//U2U Time Restricted Role Delegation
```

```
pred u2uTRD(disj dtr, dte: User, r: Role, d': Time){
     all d: Time, l: Location, urac: UserRoleActivate |
         ((dtr->r->d->l in urac.member) && (d' in r.RoleEnableDur) &&
         (d' in d)) =>
               (dte->r->d'->l in urac.member)}

//U2U Location Restricted Role Delegation
pred u2uLRD(disj dtr, dte: User, r: Role, l': Location){
     all d: Time, l: Location, urac: UserRoleActivate |
         ((dtr->r->d->l in urac.member) && (l' in r.RoleEnableLoc) &&
         (l' in l)) =>
               (dte->r->d->l' in urac.member)}

//U2U Location Restricted Role Delegation
pred u2uTLRD(disj dtr, dte: User, r: Role, d': Time, l': Location){
     all d: Time, l: Location, urac: UserRoleActivate |
         ((dtr->r->d->l in urac.member) && (d' in r.RoleEnableDur) &&
         (d' in d) && (l' in r.RoleEnableLoc) && (l' in l)) =>
               (dte->r->d->l' in urac.member)}

//R2R Unrestricted Permission Delegation
pred r2rUPD(disj rdtr, rdte: Role, p: Permission){
     all d: Time, l: Location, pracq: PermRoleAcquire |
         ((rdtr->p->d->l in pracq.member) && (d in rdte.RoleEnableDur) &&
         (l in rdte.RoleEnableLoc)) =>
               (rdte->p->d->l in pracq.member)}

//R2R Time Restricted Permission Delegation
pred r2rTPD(disj rdtr, rdte: Role, p: Permission, d': Time){
     all d: Time, l: Location, pracq: PermRoleAcquire |
         ((rdtr->p->d->l in pracq.member) && (d' in d) &&
         (d' in rdte.RoleEnableDur) &&
         (l in rdte.RoleEnableLoc)) =>
               (rdte->p->d'->l in pracq.member)}

//R2R Location Restricted Permission Delegation
pred r2rLPD(disj rdtr, rdte: Role, p: Permission, l': Location){
     all d: Time, l: Location, pracq: PermRoleAcquire |
         ((rdtr->p->d->l in pracq.member) && (l' in l) &&
         (d in rdte.RoleEnableDur) &&
         (l' in rdte.RoleEnableLoc)) =>
               (rdte->p->d->l' in pracq.member)}

//R2R Time Location Restricted Permission Delegation
pred r2rTLPD(disj rdtr, rdte: Role, p: Permission, d': Time, l': Location){
     all d: Time, l: Location, pracq: PermRoleAcquire |
         ((rdtr->p->d->l in pracq.member) && (d' in d) && (l' in l) &&
         (d' in rdte.RoleEnableDur) &&
         (l' in rdte.RoleEnableLoc)) =>
               (rdte->p->d'->l' in pracq.member)}

//R2R Unrestricted Role Delegation
pred r2rURD(disj rdtr, rdte: Role){
     all u: User, d: Time, l: Location, urac: UserRoleActivate |
```

```
        ((u->rdte->d->l in urac.member) && (d in rdtr.RoleEnableDur) &&
        (l in rdtr.RoleEnableLoc))=>
              (u->rdtr->d->l in urac.member)}

//R2R Time Restricted Role Delegation
pred r2rTRD(disj rdtr, rdte: Role, d': Time){
     all u: User, l: Location, urac: UserRoleActivate |
        ((u->rdte->d'->l in urac.member) && (d' in rdtr.RoleEnableDur) &&
        (l in rdtr.RoleEnableLoc))=>
              (u->rdtr->d'->l in urac.member)}

//R2R Location Restricted Role Delegation
pred r2rLRD(disj rdtr, rdte: Role, l': Location){
     all u: User, d: Time, urac: UserRoleActivate |
        ((u->rdte->d->l' in urac.member) && (d in rdtr.RoleEnableDur) &&
        (l' in rdtr.RoleEnableLoc))=>
              (u->rdtr->d->l' in urac.member)}

//R2R Time Location Restricted Role Delegation
pred r2rTLRD(disj rdtr, rdte: Role, d': Time, l': Location){
     all u: User, urac: UserRoleActivate |
        ((u->rdte->d'->l' in urac.member) && (d' in rdtr.RoleEnableDur) &&
        (l' in rdtr.RoleEnableLoc))=>
              (u->rdtr->d'->l' in urac.member)}

// Conflicts with the Weak Form of SSOD-User Role Assignment: Condition 1
assert TestConflict1_1{
     no u: User, disj x, y: Role, upih: UPIH,
     d: Time, l: Location, ura: UserRoleAssignment |
          ((x->y in ^(upih.member))  &&
            (u->x->d->l in ura.member)) =>
               W_SSoD_URA[u, x, y, u->(x+y)->d->l, d, l]
}
check TestConflict1_1

// Conflicts with the Weak Form of SSOD-User Role Assignment: Condition 2
assert TestConflict1_2{
     all u: User, disj x, y: Role, tpih: TPIH, d: Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(tpih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur)) =>
               W_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->y.RoleAllocDur->l), d, l]
}
check TestConflict1_2

// Conflicts with the Weak Form of SSOD-User Role Assignment: Condition 3
assert TestConflict1_3{
     all u: User, disj x, y: Role, lpih: LPIH, d: Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(lpih.member)) && (u->x->d->l in ura.member) &&
          (l in y.RoleAllocLoc)) =>
               W_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d->y.RoleAllocLoc), d, l]
}
check TestConflict1_3
```

```
// Conflicts with the Weak Form of SSOD-User Role Assignment: Condition 4
assert TestConflict1_4{
     all u: User, disj x, y: Role, tlpih: TLPIH, d: Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(tlpih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur) && (l in y.RoleAllocLoc)) =>
          W_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->y.RoleAllocDur->y.RoleAllocLoc), d, l]
}
check TestConflict1_4

// Conflicts with the Strong Temporal Form of SSOD-User Role Assignment: Condition 1
assert TestConflict2_1{
     all u: User, disj x, y: Role, upih: UPIH, d, d': Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(upih.member)) && (u->x->d->l in ura.member) &&
          (l in y.RoleAllocLoc)) =>
               ST_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d->l), d, d', l]
}
check TestConflict2_1

// Conflicts with the Strong Temporal Form of SSOD-User Role Assignment: Condition 2
assert TestConflict2_2{
     all u: User, disj x, y: Role, tpih: TPIH, d, d': Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(tpih.member)) && (u->x->d->l in ura.member) &&
          (l in y.RoleAllocLoc)) =>
               ST_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->y.RoleAllocDur->l), d, d', l]
}
check TestConflict2_2

// Conflicts with the Strong Temporal Form of SSOD-User Role Assignment: Condition 3
assert TestConflict2_3{
     all u: User, disj x, y: Role, lpih: LPIH, d, d': Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(lpih.member)) && (u->x->d->l in ura.member) &&
          (l in y.RoleAllocLoc)) =>
               ST_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d->y.RoleAllocLoc), d, d', l]
}
check TestConflict2_3

// Conflicts with the Strong Temporal Form of SSOD-User Role Assignment: Condition 4
assert TestConflict2_4{
     all u: User, disj x, y: Role, tlpih: TLPIH, d, d': Time, l: Location,
     ura: UserRoleAssignment |
           ((y in x.^(tlpih.member)) && (u->x->d->l in ura.member) &&
           (l in y.RoleAllocLoc)) =>
                ST_SSoD_URA[u, x, y,
                    (u->x->d->l) + (u->y->y.RoleAllocDur->y.RoleAllocLoc), d, d', l]
}
check TestConflict2_4

// Conflicts with the Strong Spatial Form of SSOD-User Role Assignment: Condition 1
assert TestConflict3_1{
```

```
     all u: User, disj x, y: Role, upih: UPIH, d: Time, l, l': Location,
     ura: UserRoleAssignment |
          ((y in x.^(upih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur)) =>
               SS_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d->l), d, l, l']
}
check TestConflict3_1

// Conflicts with the Strong Spatial Form of SSOD-User Role Assignment: Condition 2
assert TestConflict3_2{
     all u: User, disj x, y: Role, tpih: TPIH, d: Time, l, l': Location,
     ura: UserRoleAssignment |
          ((y in x.^(tpih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur)) =>
               SS_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->y.RoleAllocDur->l), d, l, l']
}
check TestConflict3_2

// Conflicts with the Strong Spatial Form of SSOD-User Role Assignment: Condition 3
assert TestConflict3_3{
     all u: User, disj x, y: Role, lpih: LPIH, d: Time, l, l': Location,
     ura: UserRoleAssignment |
          ((y in x.^(lpih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur)) =>
               SS_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d->y.RoleAllocLoc), d, l, l']
}
check TestConflict3_3

// Conflicts with the Strong Spatial Form of SSOD-User Role Assignment: Condition 4
assert TestConflict3_4{
     all u: User, disj x, y: Role, tlpih: TLPIH, d: Time, l, l': Location,
     ura: UserRoleAssignment |
          ((y in x.^(tlpih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur)) =>
               SS_SSoD_URA[u, x, y,
                    (u->x->d->l) + (u->y->y.RoleAllocDur->y.RoleAllocLoc), d, l, l']
}
check TestConflict3_4

// Conflicts with the Strong Form of SSOD-User Role Assignment
assert TestConflict4{
     all u: User, disj x, y: Role, d, d': Time, l, l': Location,
     ura: UserRoleAssignment,
     upih: UPIH, tpih: TPIH, lpih: LPIH, tlpih: TLPIH |
          ((y in x.^(upih.member + tpih.member + lpih.member + tlpih.member)) &&
          (u->x->d->l in ura.member)) =>
               S_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d'->l'), d, d', l, l']
}
check TestConflict4

// Conflicts with the Weak Form of SSOD-Permission Role Assignment: Condition 1
assert TestConflict5_1{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, upih: UPIH|
```

```
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(upih.member))) =>
               W_SSoD_PRA[x, p, q, d, l]
}
check TestConflict5_1

// Conflicts with the Weak Form of SSOD-Permission Role Assignment: Condition 2
assert TestConflict5_2{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, tpih: TPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(tpih.member)) &&
          (l & l' != none)) =>
               W_SSoD_PRA[x, p, q, d, l]
}
check TestConflict5_2

// Conflicts with the Weak Form of SSOD-Permission Role Assignment: Condition 3
assert TestConflict5_3{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, lpih: LPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(lpih.member)) &&
          (l & l' != none)) =>
               W_SSoD_PRA[x, p, q, d, l]
}
check TestConflict5_3

// Conflicts with the Weak Form of SSOD-Permission Role Assignment: Condition 4
assert TestConflict5_4{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, tlpih: TLPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(tlpih.member)) &&
          (l & l' != none) && (d & d' != none)) =>
               W_SSoD_PRA[x, p, q, d, l]
}
check TestConflict5_4

// Conflicts with the Strong Temporal Form of SSOD-Permission Role Assignment: Condition 1
assert TestConflict6_1{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, upih: UPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(upih.member))) =>
               ST_SSoD_PRA[x, p, q, d, d', l]
}
check TestConflict6_1

// Conflicts with the Strong Temporal Form of SSOD-Permission Role Assignment: Condition 2
assert TestConflict6_2{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, tpih: TPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
```

```
          (y->q->d'->l' in rpa) && (y in x.^(tpih.member))) =>
               ST_SSoD_PRA[x, p, q, d, d', l]
}
check TestConflict6_2

// Conflicts with the Strong Temporal Form of SSOD-Permission Role Assignment: Condition 3
assert TestConflict6_3{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, lpih: LPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(lpih.member)) &&
          (l & l' != none)) =>
               ST_SSoD_PRA[x, p, q, d, d', l]
}
check TestConflict6_3

// Conflicts with the Strong Temporal Form of SSOD-Permission Role Assignment: Condition 4
assert TestConflict6_4{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, tlpih: TLPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(tlpih.member)) &&
          (l & l' != none)) =>
               ST_SSoD_PRA[x, p, q, d, d', l]
}
check TestConflict6_4

// Conflicts with the Strong Spatial Form of SSOD-Permission Role Assignment: Condition 1
assert TestConflict7_1{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, upih: UPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(upih.member))) =>
               SS_SSoD_PRA[x, p, q, d, l, l']
}
check TestConflict7_1

// Conflicts with the Strong Spatial Form of SSOD-Permission Role Assignment: Condition 2
assert TestConflict7_2{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, tpih: TPIH |
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(tpih.member)) &&
          (d & d' != none)) =>
               SS_SSoD_PRA[x, p, q, d, l, l']
}
check TestConflict7_2

// Conflicts with the Strong Spatial Form of SSOD-Permission Role Assignment: Condition 3
assert TestConflict7_3{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, lpih: LPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(lpih.member))) =>
```

```
               SS_SSoD_PRA[x, p, q, d, l, l’]
}
check TestConflict7_3

// Conflicts with the Strong Spatial Form of SSOD-Permission Role Assignment: Condition 4
assert TestConflict7_4{
     all x, y: Role, disj p, q: Permission, d, d’: Time, l, l’: Location,
     rpa: RolePermissionAssignment.member, tlpih: TLPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d’->l’ in rpa) && (y in x.^(tlpih.member)) &&
          (d & d’ != none)) =>
               SS_SSoD_PRA[x, p, q, d, l, l’]
}
check TestConflict7_4

// Conflicts with the Strong Form of SSOD-Permission Role Assignment
assert TestConflict8{
     all x, y: Role, disj p, q: Permission, d, d’: Time, l, l’: Location,
     rpa: RolePermissionAssignment.member,
     upih: UPIH, tpih: TPIH, lpih: LPIH, tlpih: TLPIH |
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d’->l’ in rpa) &&
          (y in x.^(upih.member + tpih.member + lpih.member + tlpih.member))) =>
               S_SSoD_PRA[x, p, q, d, d’, l, l’]
}
check TestConflict8

// Conflicts with the Weak Form of DSOD: Condition 1
assert TestConflict9_1{
     all u: User, disj x, y: Role, d: Time, l: Location,
     ura: UserRoleActivate.member, uah: UAH|
          (((u->x->d->l) in ura) && (y in x.^(uah.member))) =>
               W_DSoD[u, x, y, d, l]
}
check TestConflict9_1

// Conflicts with the Weak Form of DSOD: Condition 2
assert TestConflict9_2{
     all u: User, disj x, y: Role, d: Time, l: Location,
     ura: UserRoleActivate.member, tah: TAH|
          (((u->x->d->l) in ura) && (y in x.^(tah.member)) &&
          (d in y.RoleEnableDur)) =>
               W_DSoD[u, x, y, d, l]
}
check TestConflict9_2

// Conflicts with the Weak Form of DSOD: Condition 3
assert TestConflict9_3{
     all u: User, disj x, y: Role, d: Time, l: Location,
     ura: UserRoleActivate.member, lah: LAH|
          (((u->x->d->l) in ura) && (y in x.^(lah.member)) &&
          (l in y.RoleEnableLoc)) =>
               W_DSoD[u, x, y, d, l]
}
```

```
check TestConflict9_3

// Conflicts with the Weak Form of DSOD: Condition 4
assert TestConflict9_4{
     all u: User, disj x, y: Role, d: Time, l: Location,
     ura: UserRoleActivate.member, tlah: TLAH|
          (((u->x->d->l) in ura) && (y in x.^(tlah.member)) &&
          (d in y.RoleEnableDur) && (l in y.RoleEnableLoc)) =>
               W_DSoD[u, x, y, d, l]
}
check TestConflict9_4

// Conflicts with the Strong Temporal Form of DSOD: Condition 1
assert TestConflict10_1{
     all u: User, disj x, y: Role, d, d': Time, l: Location,
     ura: UserRoleActivate.member, uah: UAH|
          (((u->x->d->l) in ura) && (y in x.^(uah.member))) =>
               ST_DSoD[u, x, y, d, d', l]
}
check TestConflict10_1

// Conflicts with the Strong Temporal Form of DSOD: Condition 2
assert TestConflict10_2{
     all u: User, disj x, y: Role, d, d': Time, l: Location,
     ura: UserRoleActivate.member, tah: TAH|
          (((u->x->d->l) in ura) && (y in x.^(tah.member))) =>
               ST_DSoD[u, x, y, d, d', l]
}
check TestConflict10_2

// Conflicts with the Strong Temporal Form of DSOD: Condition 3
assert TestConflict10_3{
     all u: User, disj x, y: Role, d, d': Time, l: Location,
     ura: UserRoleActivate.member, lah: LAH|
          (((u->x->d->l) in ura) && (y in x.^(lah.member)) &&
          (l in y.RoleEnableLoc)) =>
               ST_DSoD[u, x, y, d, d', l]
}
check TestConflict10_3

// Conflicts with the Strong Temporal Form of DSOD: Condition 4
assert TestConflict10_4{
     all u: User, disj x, y: Role, d, d': Time, l: Location,
     ura: UserRoleActivate.member, tlah: TLAH|
          (((u->x->d->l) in ura) && (y in x.^(tlah.member)) &&
          (l in y.RoleEnableLoc)) =>
ST_DSoD[u, x, y, d, d', l]
}
check TestConflict10_4

// Conflicts with the Strong Spatial Form of DSOD: Condition 1
assert TestConflict11_1{
     all u: User, disj x, y: Role, d: Time, l, l': Location,
     ura: UserRoleActivate.member, uah: UAH|
```

```
          (((u->x->d->l) in ura) && (y in x.^(uah.member))) =>
          SS_DSoD[u, x, y, d, l, l']
}
check TestConflict11_1

// Conflicts with the Strong Spatial Form of DSOD: Condition 2
assert TestConflict11_2{
     all u: User, disj x, y: Role, d: Time, l, l': Location,
     ura: UserRoleActivate.member, tah: TAH|
          (((u->x->d->l) in ura) && (y in x.^(tah.member)) &&
          (d in y.RoleEnableDur)) =>
               SS_DSoD[u, x, y, d, l, l']
}
check TestConflict11_2

// Conflicts with the Strong Spatial Form of DSOD: Condition 3
assert TestConflict11_3{
     all u: User, disj x, y: Role, d: Time, l, l': Location,
     ura: UserRoleActivate.member, lah: LAH|
          (((u->x->d->l) in ura) && (y in x.^(lah.member))) =>
               SS_DSoD[u, x, y, d, l, l']
}
check TestConflict11_3

// Conflicts with the Strong Spatial Form of DSOD: Condition 4
assert TestConflict11_4{
     all u: User, disj x, y: Role, d: Time, l, l': Location,
     ura: UserRoleActivate.member, tlah: TLAH|
          (((u->x->d->l) in ura) && (y in x.^(tlah.member)) &&
          (d in y.RoleEnableDur)) =>
               SS_DSoD[u, x, y, d, l, l']
}
check TestConflict11_4

// Conflicts with the Strong Form of DSOD
assert TestConflict12{
     all u: User, disj x, y: Role, d, d': Time, l, l': Location,
     ura: UserRoleActivate.member, uah: UAH, tah: TAH, lah: LAH, tlah: TLAH|
          (((u->x->d->l) in ura) && (y in x.^((uah + tah + lah + tlah).member)) &&
          (d in y.RoleEnableDur)) =>
               S_DSoD[u, x, y, d, d', l, l']
}
check TestConflict12


// Conflicts in Permission Role Assignment
assert TestConflict13{
     all p: Permission, r: Role, d: Time, l: Location, rpa: RolePermissionAssignment,
     re: RoleEnable |
          (r->p->d->l in rpa.member) => (r->d->l in re.member)
}
check TestConflict13

// Conflicts between r2rUPD and the Weak Form of SSOD-PRA
```

```
assert TestConflict14_1{
     all rdtr, rdte: Role, disj p, q: Permission, d: Time, l: Location |
          (r2rUPD[rdtr, rdte, p] && r2rUPD[rdtr, rdte, q]) =>
               W_SSoD_PRA[rdte, p, q, d, l]
}
check TestConflict14_1

// Conflicts between r2rTPD and the Weak Form of SSOD-PRA
assert TestConflict14_2{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l: Location |
          (r2rTPD[rdtr, rdte, p, d'] && r2rTPD[rdtr, rdte, q, d']) =>
               W_SSoD_PRA[rdte, p, q, d, l]
}
check TestConflict14_2

// Conflicts between r2rLPD and the Weak Form of SSOD-PRA
assert TestConflict14_3{
     all rdtr, rdte: Role, disj p, q: Permission, d: Time, l, l': Location |
          (r2rLPD[rdtr, rdte, p, l'] && r2rLPD[rdtr, rdte, q, l']) =>
               W_SSoD_PRA[rdte, p, q, d, l]
}
check TestConflict14_3

// Conflicts between r2rTLPD and the Weak Form of SSOD-PRA
assert TestConflict14_4{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rTLPD[rdtr, rdte, p, d', l'] && r2rTLPD[rdtr, rdte, q, d', l']) =>
               W_SSoD_PRA[rdte, p, q, d, l]
}
check TestConflict14_4

// Conflicts between r2rUPD and the Strong Temporal Form of SSOD-PRA
assert TestConflict15_1{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l: Location |
          (r2rUPD[rdtr, rdte, p] && r2rUPD[rdtr, rdte, q]) =>
               ST_SSoD_PRA[rdte, p, q, d, d', l]
}
check TestConflict15_1

// Conflicts between r2rTPD and the Strong Temporal Form of SSOD-PRA
assert TestConflict15_2{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l: Location |
          (r2rTPD[rdtr, rdte, p, d'] && r2rTPD[rdtr, rdte, q, d']) =>
               ST_SSoD_PRA[rdte, p, q, d, d', l]
}
check TestConflict15_2

// Conflicts between r2rLPD and the Strong Temporal Form of SSOD-PRA
assert TestConflict15_3{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rLPD[rdtr, rdte, p, l'] && r2rLPD[rdtr, rdte, q, l']) =>
               ST_SSoD_PRA[rdte, p, q, d, d', l]
}
check TestConflict15_3
```

```
// Conflicts between r2rTLPD and the Strong Temporal Form of SSOD-PRA
assert TestConflict15_4{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rTLPD[rdtr, rdte, p, d', l'] && r2rTLPD[rdtr, rdte, q, d', l']) =>
               ST_SSoD_PRA[rdte, p, q, d, d', l]
}
check TestConflict15_4

// Conflicts between r2rUPD and the Strong Spatial Form of SSOD-PRA
assert TestConflict16_1{
     all rdtr, rdte: Role, disj p, q: Permission, d: Time, l, l': Location |
          (r2rUPD[rdtr, rdte, p] && r2rUPD[rdtr, rdte, q]) =>
               SS_SSoD_PRA[rdte, p, q, d, l, l']
}
check TestConflict16_1

// Conflicts between r2rTPD and the Strong Spatial Form of SSOD-PRA
assert TestConflict16_2{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rTPD[rdtr, rdte, p, d'] && r2rTPD[rdtr, rdte, q, d']) =>
               SS_SSoD_PRA[rdte, p, q, d, l, l']
}
check TestConflict16_2

// Conflicts between r2rLPD and the Strong Spatial Form of SSOD-PRA
assert TestConflict16_3{
     all rdtr, rdte: Role, disj p, q: Permission, d: Time, l, l': Location |
          (r2rLPD[rdtr, rdte, p, l'] && r2rLPD[rdtr, rdte, q, l']) =>
               SS_SSoD_PRA[rdte, p, q, d, l, l']
}
check TestConflict16_3

// Conflicts between r2rTLPD and the Strong Spatial Form of SSOD-PRA
assert TestConflict16_4{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rTLPD[rdtr, rdte, p, d', l'] && r2rTLPD[rdtr, rdte, q, d', l']) =>
               SS_SSoD_PRA[rdte, p, q, d, l, l']
}
check TestConflict16_4

// Conflicts between r2rUPD and the Strong Form of SSOD-PRA
assert TestConflict17_1{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rUPD[rdtr, rdte, p] && r2rUPD[rdtr, rdte, q]) =>
               S_SSoD_PRA[rdte, p, q, d, d', l, l']
}
check TestConflict17_1

// Conflicts between r2rTPD and the Strong Form of SSOD-PRA
assert TestConflict17_2{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
           (r2rTPD[rdtr, rdte, p, d'] && r2rTPD[rdtr, rdte, q, d']) =>
               S_SSoD_PRA[rdte, p, q, d, d', l, l']
```

```
}
check TestConflict17_2

// Conflicts between r2rLPD and the Strong Form of SSOD-PRA
assert TestConflict17_3{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rLPD[rdtr, rdte, p, l'] && r2rLPD[rdtr, rdte, q, l']) =>
               S_SSoD_PRA[rdte, p, q, d, d', l, l']
}
check TestConflict17_3

// Conflicts between r2rTLPD and the Strong Form of SSOD-PRA
assert TestConflict17_4{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rTLPD[rdtr, rdte, p, d', l'] && r2rTLPD[rdtr, rdte, q, d', l']) =>
               S_SSoD_PRA[rdte, p, q, d, d', l, l']
}
check TestConflict17_4

// Conflicts between r2rURD and the Weak Form of DSOD
assert TestConflict18_1{
     all u: User, disj rdtr, rdte: Role, d: Time, l: Location |
        r2rURD[rdtr, rdte] => W_DSoD[u, rdtr, rdte, d, l]
}
check TestConflict18_1

// Conflicts between r2rTRD and the Weak Form of DSOD
assert TestConflict18_2{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l: Location |
        r2rTRD[rdtr, rdte, d'] => W_DSoD[u, rdtr, rdte, d, l]
}
check TestConflict18_2

// Conflicts between r2rLRD and the Weak Form of DSOD
assert TestConflict18_3{
     all u: User, disj rdtr, rdte: Role, d: Time, l, l': Location |
          r2rLRD[rdtr, rdte, l'] => W_DSoD[u, rdtr, rdte, d, l]
}
check TestConflict18_3

// Conflicts between r2rTLRD and the Weak Form of DSOD
assert TestConflict18_4{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTLRD[rdtr, rdte, d', l'] => W_DSoD[u, rdtr, rdte, d, l]
}
check TestConflict18_4

// Conflicts between r2rURD and the Strong Temporal Form of DSOD
assert TestConflict19_1{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l: Location |
     r2rURD[rdtr, rdte] => ST_DSoD[u, rdtr, rdte, d, d', l]
}
check TestConflict19_1
```

```
// Conflicts between r2rTRD and the Strong Temporal Form of DSOD
assert TestConflict19_2{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l: Location |
     r2rTRD[rdtr, rdte, d'] => ST_DSoD[u, rdtr, rdte, d, d', l]
}
check TestConflict19_2

// Conflicts between r2rLRD and the Strong Temporal Form of DSOD
assert TestConflict19_3{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rLRD[rdtr, rdte, l'] => ST_DSoD[u, rdtr, rdte, d, d', l]
}
check TestConflict19_3

// Conflicts between r2rTLRD and the Strong Temporal Form of DSOD
assert TestConflict19_4{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTLRD[rdtr, rdte, d', l'] => ST_DSoD[u, rdtr, rdte, d, d', l]
}
check TestConflict19_4

// Conflicts between r2rURD and the Strong Spatial Form of DSOD
assert TestConflict20_1{
     all u: User, disj rdtr, rdte: Role, d: Time, l, l': Location |
          r2rURD[rdtr, rdte] => SS_DSoD[u, rdtr, rdte, d, l, l']
}
check TestConflict20_1

// Conflicts between r2rTRD and the Strong Spatial Form of DSOD
assert TestConflict20_2{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTRD[rdtr, rdte, d'] => SS_DSoD[u, rdtr, rdte, d, l, l']
}
check TestConflict20_2

// Conflicts between r2rLRD and the Strong Spatial Form of DSOD
assert TestConflict20_3{
     all u: User, disj rdtr, rdte: Role, d: Time, l, l': Location |
          r2rLRD[rdtr, rdte, l'] => SS_DSoD[u, rdtr, rdte, d, l, l']
}
check TestConflict20_3

// Conflicts between r2rTLRD and the Strong Spatial Form of DSOD
assert TestConflict20_4{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTLRD[rdtr, rdte, d', l'] => SS_DSoD[u, rdtr, rdte, d, l, l']
}
check TestConflict20_4

// Conflicts between r2rURD and the Strong Form of DSOD
assert TestConflict21_1{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rURD[rdtr, rdte] => S_DSoD[u, rdtr, rdte, d, d', l, l']
}
```

```
check TestConflict21_1

// Conflicts between r2rTRD and the Strong Form of DSOD
assert TestConflict21_2{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTRD[rdtr, rdte, d'] => S_DSoD[u, rdtr, rdte, d, d', l, l']
}
check TestConflict21_2

// Conflicts between r2rLRD and the Strong Form of DSOD
assert TestConflict21_3{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rLRD[rdtr, rdte, l'] => S_DSoD[u, rdtr, rdte, d, d', l, l']
}
check TestConflict21_3

// Conflicts between r2rTLRD and the Strong Form of DSOD
assert TestConflict21_4{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTLRD[rdtr, rdte, d', l'] => S_DSoD[u, rdtr, rdte, d, d', l, l']
}
check TestConflict21_4
```

Appendix B

Specification of the STRBAC Model for the Dengue Decision Support (DDS) System

B.1 OCL Constraints for DDS's STRBAC Model

```
context JurisEpi
inv jurisEpiCon : (self.tasks = (Task :: ONE ->
including (Task :: THREE)) and
self.location = Location :: B and
self.timeCon = Time :: a) or
(self.tasks = (Task :: SEVENTEEN -> including
(Task :: SEVENTEEN)) and
self.location = Location :: B and
self.timeCon = Time :: b)

context StateEpi
inv stateEpiCon : (self.tasks = (Task :: SIXTEEN ->
including (Task::SIXTEEN)) and
self.location = Location :: A and
self.timeCon = Time :: a)
-- The following OCL excerpt represents that the
-- StateEpi inherits the permissions of the JurisEpi.
or (self.tasks = (Task :: ONE -> including
(Task :: THREE)) and self.location = Location :: B and
self.timeCon = Time :: a) or
(self.tasks = (Task :: SEVENTEEN -> including
(Task :: SEVENTEEN)) and self.location = Location :: B
and self.timeCon = Time :: b)

context LocalVCTeam
inv localVCTeam : (self.tasks = ( Task :: SEVEN ->
including (Task :: SEVEN)) and
```

```
(self.location = Location :: B or
self.location = Location :: E ) and
self.timeCon = Time :: c) or
(self.tasks = (Task :: THIRTEEN ->
including (Task :: THIRTEEN)) and
(self.location = Location :: B or
self.location = Location :: D) and
self.timeCon = Time :: a) or
(self.tasks = (Task :: ONE ->
including (Task :: NINE)) and
(self.location = Location :: B or
self.location = Location :: D ) and
self.timeCon = Time :: a )

context Clinician
inv clinicialCon : (self.tasks = (Task :: ONE ->
including (Task :: TWO) -> including
(Task :: THREE) -> including (Task :: FOUR) ->
including (Task :: FIVE) -> including
(Task :: SIX)) and self.location = Location :: C
and self.timeCon = Time :: a)

context ClinicEpi
inv clinicEpiCon : (self.tasks = (Task :: SEVENTEEN ->
including (Task :: SEVENTEEN)) and
self.location = Location :: C and
self.timeCon = Time :: b ) or
((self.tasks = ( Task :: ONE -> including
(Task :: TWO) -> including (Task :: THREE) ->
including (Task :: FOUR) -> including (Task :: FIVE) ->
including (Task :: SIX)) and
self.location = Location :: C and
self.timeCon = Time :: a))

context JurisVC
inv jurisVCCon : (self.tasks = (Task :: ONE ->
including (Task :: EIGHT) -> including
(Task :: NINE ) -> including (Task :: TEN) ->
including (Task :: TWELVE) -> including
(Task :: FOURTEEN)) and
self.location = Location :: B and
self.timeCon = Time :: a)

context StateVC
inv stateVCCon : (self.tasks = (Task :: ELEVEN ->
including (Task :: ELEVEN)) and
self.location = Location :: A and
```

```
self.timeCon = Time :: a) or
(self.tasks = (Task :: FIFTEEN -> including
(Task :: FIFTEEN)) and self.location = Location :: A and
self.timeCon = Time :: a ) or
(self. tasks = ( Task :: ONE -> including (Task :: EIGHT) ->
including (Task :: NINE) -> including(Task :: TEN) ->
including (Task :: TWELVE) -> including
(Task :: FOURTEEN)) and self.location = Location :: B and
self.timeCon = Time :: a)

context Person
inv no_eleven_fifteen : self.roles ->
forAll(r1 , r2 : Role |
(r1.tasks -> includes (Task :: ELEVEN) implies
(r2.tasks -> excludes (Task :: FIFTEEN))) and
(r1.tasks -> includes (Task :: FIFTEEN) implies
r2.tasks -> excludes (Task :: ELEVEN)))

inv no_sixteen_seventeen : self.roles ->
forAll(r1 , r2 : Role |
(r1.tasks -> includes (Task :: SIXTEEN) implies
(r2.tasks -> excludes (Task :: SEVENTEEN))) and
(r1.tasks -> includes (Task :: SEVENTEEN) implies
r2.tasks -> excludes (Task :: SIXTEEN)))
```

B.2 Generated Alloy Model for DDS's STRBAC Model

```
module DDSV2
abstract  sig Role{
location:one Location,
timeCon:one Time,
tasks:some Task,
uses:set Person}

one sig StateEpi extends Role{}
one sig JurisEpi extends Role{}
one sig ClinicEpi extends Role{}
one sig Clinician extends Role{}
one sig StateVC extends Role{}
one sig JurisVC extends Role{}
one sig LocalVCTeam extends Role{}

some sig Person{roles:some Role}

abstract  sig Location{}
one sig A extends Location{}
one sig B extends Location{}
```

```
one sig C extends Location{}
one sig D extends Location{}
one sig E extends Location{}

sig Time{}
sig a in Time{}
sig b in Time{}
sig c in Time{}


abstract  sig Task{}
one sig ONE extends Task{}
one sig TWO extends Task{}
one sig THREE extends Task{}
one sig FOUR extends Task{}
one sig FIVE extends Task{}
one sig SIX extends Task{}
one sig SEVEN extends Task{}
one sig EIGHT extends Task{}
one sig NINE extends Task{}
one sig TEN extends Task{}
one sig ELEVEN extends Task{}
one sig TWELVE extends Task{}
one sig THIRTEEN extends Task{}
one sig FOURTEEN extends Task{}
one sig FIFTEEN extends Task{}
one sig SIXTEEN extends Task{}
one sig SEVENTEEN extends Task{}

fact StateEpi_stateEpiCon_fact{
all self: StateEpi | StateEpi_stateEpiCon[self]}

fact JurisEpi_jurisEpiCon_fact{
all self: JurisEpi | JurisEpi_jurisEpiCon[self]}

fact ClinicEpi_clinicEpiCon_fact{
all self: ClinicEpi | ClinicEpi_clinicEpiCon[self]}

fact Clinician_clinicialCon_fact{
all self: Clinician | Clinician_clinicialCon[self]}

fact StateVC_stateVCCon_fact{
all self: StateVC| StateVC_stateVCCon[self]}

fact JurisVC_jurisVCCon_fact{
all self: JurisVC | JurisVC_jurisVCCon[self]}
```

```
fact LocalVCTeam_localVCTeam_fact{
all self: LocalVCTeam | LocalVCTeam_localVCTeam[self]}

fact Person_no_eleven_fifteen_fact{
all self: Person | Person_no_eleven_fifteen[self]}

fact Person_no_sixteen_seventeen_fact{
all self: Person | Person_no_sixteen_seventeen[self]}

fact Asso_Role_role_tasks_Task{
Role <: tasks in  ( Role) set->some ( Task)}

fact Asso_Person_uses_roles_Role{
Person <: roles in  ( Person) set->some ( Role) &&
Role <: uses in  ( Role) some->set ( Person)}

fact Person_uses_roles_Role_symmetry{
Role <: uses = ~(Person <: roles)}

pred StateEpi_stateEpiCon[self: StateEpi]{
(self.tasks = SIXTEEN + ONE + THREE + SEVENTEEN) &&
(self.location = A) && (self.timeCon = a)}


pred JurisEpi_jurisEpiCon[self: JurisEpi]{
((self.tasks = ONE+THREE) && (self.location = B) &&
(self.timeCon = a)) || ((self.tasks = SEVENTEEN) &&
(self.location = B) && (self.timeCon in Time))}

pred ClinicEpi_clinicEpiCon[self: ClinicEpi]{
((self.tasks = SEVENTEEN) && (self.location = C) &&
(self.timeCon in Time)) ||
((self.tasks = ONE+TWO+THREE+FOUR+FIVE+SIX) &&
(self.location = C) && (self.timeCon = a))}

pred Clinician_clinicialCon[self: Clinician]{
(self.tasks = ONE+TWO+THREE+FOUR+FIVE+SIX) &&
(self.location = C) && (self.timeCon = a)}

pred StateVC_stateVCCon[self: StateVC]{
((self.tasks = ELEVEN+FIFTEEN+ONE+EIGHT+NINE+
TEN+TWELVE+FOURTEEN) &&
(self.location = A) && (self.timeCon = a))}

pred JurisVC_jurisVCCon[self: JurisVC]{
((self.tasks = ONE+EIGHT+NINE+TEN+TWELVE+
FOURTEEN) && (self.location = B) &&
```

```
(self.timeCon = a))}

pred LocalVCTeam_localVCTeam[self: LocalVCTeam]{
((self.tasks = SEVEN) && ((self.location = B) ||
(self.location = E)) && (self.timeCon = c)) ||
((self.tasks = THIRTEEN) && ((self.location = B) ||
(self.location = D)) && (self.timeCon = a)) ||
((self.tasks = ONE+NINE) && ((self.location = B) ||
(self.location = D)) && (self.timeCon = a))}

pred Person_no_eleven_fifteen[self: Person]{
all r1, r2: self.roles |
((ELEVEN in r1.tasks) => (FIFTEEN !in r2.tasks)) &&
((FIFTEEN in r1.tasks) => (ELEVEN !in r2.tasks))}

pred Person_no_sixteen_seventeen[self: Person]{
all r1, r2: self . roles |
((SIXTEEN in r1.tasks) => (SEVENTEEN !in r2.tasks)) &&
((SEVENTEEN in r1.tasks) => (SIXTEEN !in r2.tasks))}

assert NoConflictPermsSTVCAssigned{
all r: Person.roles, d: Time, l: Location|
((ELEVEN in r.tasks) && (d in r.timeCon) &&
(l in r.location)) =>
((FIFTEEN !in r.tasks) && (d in r.timeCon) &&
(l in r.location))}
check NoConflictPermsSTVCAssigned for 8

assert NoConflictPermsSTVC{
all r: StateVC, d: Time, l: Location|
((ELEVEN in r.tasks) && (d in r.timeCon) &&
(l in r.location)) =>
((FIFTEEN !in r.tasks) && (d in r.timeCon) &&
(l in r.location))}
check NoConflictPermsSTVC for 8

assert NoConflictPermsSTEpiAssigned{
all r: Person.roles, d: Time, l: Location|
((SIXTEEN in r.tasks) && (d in r.timeCon) &&
(l in r.location)) =>
((SEVENTEEN !in r.tasks) && (d in r.timeCon) &&
(l in r.location))}
check NoConflictPermsSTEpiAssigned for 8

assert NoConflictPermsSTEpi{
all r: StateEpi, d: Time, l: Location|
((SIXTEEN in r.tasks) && (d in r.timeCon) &&
```

```
(l in r.location)) =>
((SEVENTEEN !in r.tasks) && (d in r.timeCon) &&
(l in r.location))}
check NoConflictPermsSTEpi for 8
```

Appendix C

STARBACD SoD Violation Detection Algorithm

C.1 Finding common predecessors in a DAG

Let $V = \{1, 2, \ldots, n\}$. Given a subset S of V, the *characteristic vector* of S is a bit vector that has a 1 in position i if and only if $i \in S$. Representing a subset's characteristic vector with a bit array allows one to determine whether $i \in S$ in $O(1)$ time.

In a directed graph $G = (V, E)$, let the *in-degree* of a vertex v be the number of edges directed into v. That is, the in-degree of v is the cardinality of the set $\{(u, v) | (u, v) \in E\}$. Similarly, the out-degree of v is the number of edges directed out of v, that is, the cardinality of the set $\{(v, u) | (v, u) \in E\}$.

An undirected graph is a special case of a directed graph where, for every directed edge (u, v), (v, u) is also a directed edge. In this case, we denote the pair $\{(u, v), (v, u)\}$ by uv. The *underlying undirected graph* of a directed graph is the graph obtained by adding (v, u) as an edge whenever (u, v) is an edge.

In a directed graph, and *in-neighbor* is a vertex u that has a directed edge (u, v) to v, and an *out-neighbor* is a vertex w such that v has a directed edge (v, w) to w. If G is a DAG, let a *predecessor* of vertex v be any vertex w such that there is a directed path from w to v. Similarly, a *successor* of v is any vertex u such that there is a directed path from v to u. A vertex is a *source* in a DAG if its in-degree is 0 and a *sink* if its out-degree is 0. A *path tree rooted at vertex w* is a subset of the edges of G that form a tree rooted at w, oriented away from w, and

reaching every successor of w. (A DFS or BFS tree is a special case of a path tree.)

We consider variants of the following problem: Given a DAG $G = (V,E)$ and a pair $\{u,v\}$ of vertices, determine whether u and v have a common predecessor.

Let us call $\{u,v\}$ a *query*. Let $n = |V|$ be the number of vertices, $m = |E|$ be the number of edges, k be the number of queries. Let p denote the number of sources of G. We may assume that every vertex has either in-degree or out-degree greater than zero, since vertices failing this property are irrelevant to the problem and can be removed from the graph in linear time. Therefore $n = O(m)$, and a time bound of $O(n+m)$ can be simplified to $O(m)$.

Let G^T denote the *transpose* of G, which is obtained by reversing the directions of all edges of G. That is, $G^T = (V,E')$, where $E' = \{(v,u)|(u,v) \in E\}$. Given an adjacency-list representation of G, it is well known that it takes $O(n+m)$ time to find the adjacency lists of the transpose G^T by radix sorting the edges using source vertex as the secondary sort key and destination vertex as the primary sort key. An adjacency-list representation of a graph gives, for each vertex, a list of out-neighbors; this gives, for each vertex, a list of in-neighbors.

C.1.1 A naive algorithm for the static and dynamic cases

If the number k of queries is 1, the query can be answered in $O(m)$ time by performing depth-first search from u in G^T, marking all visited vertices, and then performing depth-first search from v in G^T, determining whether any marked vertices are encountered.

A sequence of k queries on k graphs, each with $O(m)$ edges, takes $O(km)$ time. This gives a time bound of $O(km)$ for the *dynamic* case where edges can be added to or deleted from G between queries, and m is the maximum number of edges the graph has at any point.

Some improvements when k is large

Note that k can be quadratic in the number of vertices. If G is dynamic, then queries may be repeated as G changes, and there is no upper bound on k. We consider the possibility of better bounds than $O(km)$ in these cases.

In the static case, we observe that two vertices have a common predecessor if and only

if they have a common predecessor that is a source. For each source w, we may label all successors of w by depth-first search. This gives each vertex at most p labels. Moreover, if a source is added to a vertex's list of labels, we add it to the back of the list. That way, all lists of labels are sorted in the order in which the sources were processed. This labeling takes $O(pm)$ time, and a query now takes $O(p)$ time to determine whether the two query vertices share a common label. Summarizing, this gives an $O((k+m)p)$ algorithm to add the labels and then process the k queries.

If $k = o(p)$, this is $O(mp)$, which is worse than the $O(km)$ bound we got above. If $p = o(k)$ and $k = O(m)$, the bound is still $O(mp)$, but this is asymptotically better than the $O(km)$ bound we got above, and if $m = o(k)$, then the bound is $O(kp)$ which is also better than $O(km)$, since $p = O(n) = O(m)$. Summarizing, this approach gives a better asymptotic bound when $p = o(k)$.

Adding edges to G between queries

Let us now consider how we might do better than $O(km)$ for the dynamic case if k is large and edges may be added, but not deleted. In this case, we may maintain a path tree rooted at each source w. Below, we see that we maintain the invariant that the path tree is a DFS tree. Vertices once again carry w in a sorted list of source labels if they are a successor of w, that is, if they are in w's path tree.

Initially, we compute a path tree from each source using DFS. Let (u,v) be an added edge. For each vertex w in u's list of labels, we extend the DFS tree rooted at w by performing a depth-first search from v, retreating whenever a vertex labeled w is encountered. Let us call this an *incremental DFS*. Using $O(p)$ space to store the characteristic bit vector of the set of labels at each vertex allows us to look up whether w is a label of a vertex in $O(1)$ time, using a total of $O(m+np)$ space. If v was previously a source, we may remove the tree rooted at v, since it is no longer a source. It is trivial to do this in $O(n) = O(m)$ time.

Each edge is traversed once over all incremental DFS's on w, giving a bound of $O(m)$ to update the path tree rooted at w over all edge insertions. The addition of an edge never creates

a new source. This therefore gives an $O(mp)$ bound for all updates to trees, where p is the initial number of sources.

Determining whether two vertices lie in a common DFS tree once again takes $O(p)$ time, for a total of $O((k+m)p)$, where m is the final number of edges in G. The analysis of when this is better than the $O(km)$ bound we obtained above is the same as it is for the $O((k+m)p)$ bound we got for the static case.

In fact, it is possible to implement this algorithm without recording the trees, and only making use of the labels to guide the incremental DFS operations. However, if the trees are maintained, an interesting observation is that this maintains the invariant that each tree is a DFS tree. Suppose this is true for w's tree T_w before it was extended to T'_w due to the addition of (u,v), giving a new graph G'. It is easy to see that in a DFS of G' where (u,v) is the last edge considered at u, T_w is the state of the depth-first tree on G' during a run of DFS just before (u,v) is considered, where (u,v) is the last edge in u's adjacency list.

C.1.2 Deleting edges from G between queries

Let us now suppose that edges are only deleted from G. For this, we maintain, for each source, a path tree rooted at the source. Once again, each vertex is labeled with a sorted list of sources that it is a successor of.

When a new source is created by the removal of an edge, we use DFS to get an initial path tree for the new source.

It remains to describe how to update an existing path tree T_w for a single source w after deletion of an edge; path trees for all sources are updated with the same procedure.

When an edge (u,v) is removed from G, we find whether (u,v) is an edge of T_w. If it is not, T_w remains a path tree, and we are done.

Definition 55

Let w be a source, let T_w be the current path tree rooted at w, and let (u,v) be an edge of G that is also an edge in T_w and that is deleted from G, yielding G'. Removal of (u,v) splits T_w into two subtrees, the subtree T_v of T_w rooted at v, and the remainder T'_w of T_w. The *status* of a

vertex of G' is whether it is reachable from w in G'.

Before the deletion of (u, v) the status of all vertices of G is known: the ones that are reachable from w are just the vertices in T_w, and the remaining vertices are not reachable.

Lemma 4

Deletion of (u, v) can only change the status of some vertices in T_v from *reachable* to *unreachable*. If a vertex x of T_v continues to be reachable, then so does every vertex in the subtree T_x of T_v rooted at x.

Proof Removal of an edge cannot make a vertex reachable from w if it was not reachable before, so the status of vertices not in T_w does not change. The status of vertices in T'_w does not change, since the edges of this tree give paths in G' from w to every vertex in T'_w. Let y be a vertex in x's subtree of T_v. If x continues to be reachable, then there is a path P of G' from w to x. Appending the unique tree path from x to y in T_x to P yields a directed path from w to y, which implies that y continues to be reachable.

The goal is to determine the new status of each node x in T_v. Our strategy is to process the vertices of unknown status in an order such that when it is time to make the status of a vertex x known, the status of all in-neighbors of x is known. This reduces the problem of determining x's status to that of determining whether it has an in-neighbor that is known to be a successor of w.

Lemma 5

Let $(v_1, v_2, \ldots v_n)$ be a topological sort of G. If the status of vertices of unknown status is made known in the order in which they appear in this sort, then when it is time to make the status of a vertex v_i known, the status of all in-neighbors is known.

Proof By induction on the number of vertices whose status is made known, when it is time to make v_i's status known, the status of all earlier vertices in topological order is known. All in-neighbors of a vertex are earlier in topological order.

Let us give an overview of our strategy. Before beginning any operations on our initial DAG, we assign topological sort numbers to the vertices. Deletion of an edge does not invalidate a topological sort, so this numbering remains a valid topological sort after any number of edge deletions. Our strategy for obtaining our time bound is to take advantage of Lemma 5 by using a priority queue, keyed on topological-sort numbers, to dispense vertices of T_v in topological order. By Lemma 5, when a vertex x is dispensed from the priority queue, the reachability status of all in-neighbors is known. We determine whether x is reachable from w by determining whether it has an in-neighbor that is reachable from w. Moreover, when an in-neighbor is found to be unreachable, no subsequent edge deletion will make it reachable, so after each edge deletion, if a vertex x is inserted and dispensed from the priority queue, we may resume the search of its in-neighbor list where we left off the last time x was inserted and dispensed from the priority queue. This ensures that over all edge deletions, each element of x's in-neighbor list is examined only once to determine whether it is reachable.

A critical element for our time bound is to observe the following constraint on which vertices we can touch:

- **Constraint:** We touch a vertex x of T_v only if it becomes unreachable or has no reachable parent in T_v.

We accomplish this by inserting x to the priority queue only if $x = v$ or the status of its parent in T_v is found to be unreachable from w. If a vertex x is determined to have a reachable in-neighbor z, then, by Lemma 4, all vertices in T_x are reachable from w, so we can include all of them in the new path tree rooted at T_w in $O(1)$ time by adding (z,x) to the tree. This observes the constraint by avoiding touching lower vertices in T_x. If x is found not to be reachable, then we can touch the children of x in T_v. We insert these children in the priority queue.

Lemma 6

Vertices are dispensed from the priority queue in topological order.

Proof When x is extracted, it has an earlier topological number of any vertex in the priority queue. If it is determined to be a successor of w, no new vertices are inserted before another

extraction. If it is determined not to be a successor of w, its children in T_v are inserted, and since there is an edge of G from x to each of these children, they have larger topological numbers than x does. In either case, the minimum topological number in the priority queue increases every time a vertex is extracted.

We can now give the detailed implementation that gives the time bound. A given vertex x might be inserted to the priority queue any time an edge is deleted. After the first time x is inserted to the priority queue, we maintain a pointer x_w into x's in-neighbor list. The pointer initially points to the beginning of x's in-neighbor list, and satisfies the following invariants:

- All elements of x's in-neighbor list that precede x_w are known to be non-successors of w or have ceased to be in-neighbors of x.
- After T_w is updated, if x is a node of T_w, then x_w points to its parent in T_w.

Whenever x is inserted in the priority queue, it has lost its parent in T_w or the parent has ceased to be reachable from w. In either case, we can advance x_w without violating the first constraint. We iteratively advance x_w until we find an in-neighbor z that is known to be a successor of of w, or reach the end of the in-neighbor list. If z is found, we leave x_w pointing to z, and make z x's parent, satisfying the second invariant. If it is not found, we label x as unreachable and insert its children in the priority queue.

Lemma 7

The foregoing algorithm correctly updates the status of all nodes as reachable or not reachable from w after an edge deletion, and modifies T_w to be a correct path tree.

Proof That the invariants are maintained on x_w follows from the fact that once an in-neighbor is labeled as unreachable, it is never relabeled as reachable, since edge insertions are not allowed. That the status of the in-neighbors of x are all correctly labeled whenever x is extracted from the priority queue follows from Lemmas 5 and 6. It follows that x is correctly labeled. If x remains reachable, that the descendants of x remain correctly labeled and included in a correct path tree follows from Lemma 4. Since x is an arbitrary node of T_v that is inserted to and

extracted from the priority queue, and all nodes of T_v are either inserted and extracted from the priority queue, it follows that all vertices of T_v are either correctly labeled as non-successors of w, or are correctly linked into a new path tree rooted at w.

Lemma 8

The above algorithm takes $O(m \log n)$ time over all edge deletions.

Proof Every time a node x is inserted to the priority queue, x_w is advanced in its adjacency list. The time spent over all insertions of x in the priority queue is $O(\log n)$ times the in-degree of x. The sum of in-degrees of all vertices is $O(m)$, and the bound follows.

Lemma 9

Let p be the number of sources that appear during edge deletions on G. It takes $O(pm \log n)$ time to maintain the data structures for common-predecessor queries over all edge deletions, and they support queries in $O(p)$ time.

Proof The above algorithm for a given source w is carried out for each of the $O(p)$ sources whenever an edge is deleted, in sorted order of sources. This allows us to label each vertex with a sorted list of sources that it has ceased to be a successor of as a result of the edge deletion. These can then be removed from its list of sources that it is reachable from in $O(p)$ time. A common-predecessor query takes $O(p)$ time to determine, for the two given vertices, whether the two sorted lists of sources that they are successors of contain a common element.

Appendix D

ALLOY Specification of the Small Healthcare Organization

```
module TrustRBACV5
open util/integer

abstract sig User{}
one sig Allen, Bell, Cox, Davis, Evans, Miller, Nelson extends User{}

abstract sig Role{}
one sig SeniorDoctor, JuniorDoctor, Assistant, Cardiologist, Surgeon,
        Patient extends Role{}

// User Role assignment
one sig UserRoleAssign{URAsmember: User -> Role}
// User Role acquire
one sig UserRoleAcquire{URAcqmember: User -> Role}
// Role Hierarchy
one sig RoleHierarchy{RHmember : Role -> Role}

// User Role Assignments
```

```
fact URAs{
    UserRoleAssign.URAsmember =
        Allen->(SeniorDoctor + Surgeon) + (Bell+Nelson)->(JuniorDoctor) +
        Cox->(Cardiologist) +
        Davis->(Assistant) + Evans->(Patient) + Miller->(SeniorDoctor)}

// Role Hierarchy
fact RH{
    RoleHierarchy.RHmember = SeniorDoctor->JuniorDoctor +
        JuniorDoctor->Assistant}

// User Role Acquire
fact URAcq{
       UserRoleAcquire.URAcqmember = UserRoleAssign.URAsmember +
          (UserRoleAssign.URAsmember).^(RoleHierarchy.RHmember)}

// Role Hierarchy are acyclic
fact NocycleRH{
all r: Role, RH: RoleHierarchy| r !in r.^(RH.RHmember)}

// Cardinality Constraint
pred Cardinality(r: Role, uracq: User->Role){
         (#((uracq).r) >= 1) &&
         (#((uracq).r) <= 1)}

// Relation Constraint: Bi Complementary
pred Complement(u:User, r1, r2: Role, uracq: User->Role){
    (u->r2 in uracq) <=> (u->r1 in uracq)}
```

```
// Relation Constraint: Bi Conflict
pred SoD(u:User, r1, r2: Role, uracq: User->Role){
    (u->r1 in uracq) => not (u->r2 in uracq)}

//Delegation (Grant)
pred Grant[u: User, r: Role, uracq, uracq': User->Role]{
     uracq' = uracq + (u->r)}

//Delegation (Transfer)
pred Transfer[u1, u2: User, r: Role, uracq, uracq': User->Role]{
     uracq' = uracq + (u2->r) - (u1->r)}
run Transfer

//Relation violation in the presence of delegation (Transfer)
assert TestConflict3{
     all u1, u2: User, r: Role, uracq, uracq': User->Role|
         ((u1 = Allen) && (u2 = Bell) && (r=Surgeon) &&
         (uracq = UserRoleAcquire.URAcqmember) &&
         (u1->r in UserRoleAcquire.URAcqmember) &&
         (u2->Assistant not in UserRoleAssign.URAsmember) &&
         Transfer[u1, u2, r, uracq, uracq']) =>
            SoD[u2, r, Assistant, uracq']}
check TestConflict3

//Relation violation in the presence of delegation (Transfer)
assert TestConflict4{
     all u1, u2: User, r: Role, uracq, uracq': User->Role|
```

```
          ((u1 = Allen) && (u2 = Cox) && (r=Surgeon) &&
          (uracq = UserRoleAcquire.URAcqmember) &&
          (u1->r in UserRoleAcquire.URAcqmember) &&
          (u2->Assistant not in UserRoleAssign.URAsmember) &&
          Transfer[u1, u2, r, uracq, uracq’]) =>
             SoD[u2, r, Assistant, uracq’]}
check TestConflict4
```

REFERENCES

[1] Nabil R. Adam, Vijayalakshmi Atluri, and Wei-Kuang Huang. Modeling and Analysis of Workflows Using Petri Nets. *Journal of Intelligent Information Systems*, 10(2):131–158, March-April 1998.

[2] Isaac Agudo, M. Carmen Fernández Gago, and Javier Lopez. A model for trust metrics analysis. In *Proceedings of the 5th International Conference on Trust, Privacy and Security in Digital Business*, pages 28–37, 2008.

[3] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, 2007. To Appear.

[4] Claudio A. Ardagna, Marco Cremonini, Ernesto Damiani, Sabrina De Capitani di Vimercati, and Pierangela Samarati. Supporting location-based conditions in access control policies. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pages 212–222, Taipei, Taiwan, March 2006.

[5] Vijayalakshmi Atluri and Soon Ae Chun. An authorization model for geospatial data. *IEEE Transactions on Dependable and Secure Computing*, 1(4):238–254, October-December 2004.

[6] Vijayalakshmi Atluri and Soon Ae Chun. A geotemporal role-based authorisation system. *International Journal of Information and Computer Security*, 1(1/2):143–168, January 2007.

[7] Vijayalakshmi Atluri and Wei-Kuang Huang. A petri net based safety analysis of workflow authorization models. *Journal of Computer Security*, 8(2,3):209–240, August 2000.

[8] Vijayalakshmi Atluri and Wei kuang Huang. An Authorization Model for Workflows. In *Proceedings of the 4th European Symposium on Research in Computer Security*, pages 44–64, Rome, Italy, September 1996.

[9] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A Tutorial on Uppaal. In *4th International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 200–236, Bertinoro, Italy, September 2004.

[10] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, pages 125–126, Riverside, California, USA, September 2006.

[11] David E. Bell and Leonard J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical report, The MITRE Corporation, March 1976.

[12] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: a temporal role-based access control model. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*, pages 21–30, Berlin, Germany, July 2000.

[13] Elisa Bertino, Barbara Catania, Maria Luisa Damiani, and Paolo Perlasca. GEO-RBAC: a spatially aware RBAC. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*, pages 29–37, Stockholm, Sweden, June 2005.

[14] Kenneth J. Biba. Integrity Considerations for Secure Computer Systems. Technical report, MITRE Corporation, April 1977.

[15] Behzad Bordbar and Kyriakos Anastasakis. MDA and Analysis of Web Applications. In *Trends in Enterprise Application Architecture (TEAA) 2005*, volume 3888 of *Lecture notes in Computer Science*, pages 44–55, Trondheim, Norway, August 2005.

[16] Behzad Bordbar and Kyriakos Anastasakis. UML2ALLOY: A tool for lightweight modelling of discrete event systems. In *Proceedings of the IADIS International Conference on Applied Computing*, pages 209–216, Algarve, Portugal, February 2005.

[17] Sudip Chakraborty and Indrajit Ray. TrustBAC: integrating trust relationships into the RBAC model for access control in open systems. In *Proceedings of the 11th ACM symposium on Access control models and technologies*, pages 49–58, Lake Tahoe, California, USA, 2006. ACM.

[18] Suroop Mohan Chandran and James B. D. Joshi. *LoT-RBAC*: A Location and Time-Based RBAC Model. In *Proceedings of the 6th International Conference on Web Information Systems Engineering*, pages 361–375, New York, NY, USA, November 2005.

[19] Liang Chen and Jason Crampton. On Spatio-Temporal Constraints and Inheritance in Role-Based Access Control. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, pages 205–216, Tokyo, Japan, March 2008.

[20] David D. Clark and David R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, USA, April 1987.

[21] Michael J. Covington, Prahlad Fogla, Zhiyuan Zhan, and Mustaque Ahamad. A Context-Aware Security Architecture for Emerging Applications. In *Proceedings of the Annual Computer Security Applications Conference* , pages 249–260, Las Vegas, NV, USA, December 2002.

[22] Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind Dey, Mustaque Ahamad, and Gregory Abowd. Securing Context-Aware Applications Using Environment Roles. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*, pages 10–20, Chantilly, VA, USA, May 2001.

[23] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, Inc., Norwood, MA, USA, 2003.

[24] David F. Ferraiolo, Ravi S. Sandhu, Serban I. Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and Systems Security*, 4(3):224 – 274, August 2001.

[25] G. Scott Graham and Peter J. Denning. Protection–Principles and Practice. In *Proceedings of the Spring Joint Computer Conference*, pages 417–429, May 1972.

[26] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[27] Urs Hengartner and Peter Steenkiste. Implementing Access Control to People Location Information. In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies*, pages 11–20, Yorktown Heights, NY, USA, June 2004.

[28] Burkhard Hilchenbach. Observations on the Real-World Implementation of Role-Based Access Control, September 1997. At http://csrc.nist.gov/nissc/1997/proceedings/341.pdf.

[29] R. J. Hulsebosch, A. H. Salden, M. S. Bargh, P. W. G. Ebben, and J. Reitsma. Context sensitive access control. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*, pages 111–119, New York, NY, USA, 2005.

[30] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 130–139, San Diego, CA, USA, November 2000.

[31] Daniel Jackson. *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*. At http://alloy.mit.edu/alloy2website/reference-manual.pdf , 2002.

[32] Daniel Jackson. *Alloy 3.0 reference manual*. At http://alloy.mit.edu/reference-manual.pdf, 2004.

[33] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[34] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, London, England, 2006.

[35] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[36] Kurt Jensen, Søren Christensen, and Lars M. Kristensen. CPN Tools State Space Manual Last updated: January 2006, 2006. At http://wiki.daimi.au.dk/cpntools-help/_files/manual.pdf.

[37] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.

[38] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3):213–254, May 2007.

[39] Yixin Jiang, Chuang Lin, Hao Yin, and Zhangxi Tan. Security analysis of mandatory access control model. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, volume 6, pages 5013–5018 vol.6, October 2004.

[40] James B. D. Joshi. *A Generalized Temporal Role Based Access Control Model for Developing Secure Systems*. PhD thesis, Purdue University, August 2003.

[41] James B. D. Joshi and Elisa Bertino. Fine-grained role-based delegation in presence of the hybrid role hierarchy. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, pages 81–90, Lake Tahoe, California, USA, June 2006.

[42] James B. D. Joshi, Basit Shafiq, Arif Ghafoor, and Elisa Bertino. Dependencies and separation of duty constraints in GTRBAC. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies*, pages 51–64, New York, NY, USA, 2003.

[43] James B.D. Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. A Generalized Temporal Role-Based Access Control Model. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):4–23, January 2005.

[44] Lalana Kagal, Tim Finin, and Anupam Joshi. Trust-Based Security in Pervasive Computing Environments. *Computer*, 34(12):154–157, December 2001.

[45] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture–Practice and Promise*. The Addison-Wesley Object Technology Series. Addison-Wesley, 2003.

[46] Romain Laborde, Bassem Nasser, Frédéric Grasset, François Barrère, and Abdelmalek Benzekri. A Formal Approach for the Evaluation of Network Security Mechanisms Based on RBAC Policies. *Electronic Notes in Theoretical Computer Science*, 121:117–142, February 2005.

[47] Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. reprinted in ACM SIGOPS Operating Systems Review, 8,1, January 1974, pp. 18 - 24.

[48] Ulf Leonhardt and Jeff Magee. Security Consideration for a Distributed Location Service. *Imperial College of Science, Technology and Medicine, London, UK*, 1997.

[49] Yahui Lu, Li Zhang, and Jiaguang Sun. Using colored Petri nets to model and analyze workflow with separation of duty constraints. *The International Journal of Advanced Manufacturing Technology*, 40(1,2):179–192, January 2009.

[50] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

[51] Samrat Mondal, Shamik Sural, and Vijayalakshmi Atluri. Towards Formal Security Analysis of GTRBAC using Timed Automata. In *Proceedings of the 14th ACM Symposium on Access control Models and Technologies*, pages 33–42, Stresa, Italy, June 2009.

[52] National Computer Security Center. A Guide to Understanding Discretionary Access Control in Trusted Systems, September 1987.

[53] Matunda Nyanchama and Sylvia Osborn. Modeling Mandatory Access Control in Role-Based Security Systems. In *Proceedings of the 9th annual IFIP TC11 WG11.3 working conference on Database security IX : status and prospects*, pages 129–144, Rennselaerville, NY, USA, August 1995.

[54] Matunda Nyanchama and Sylvia Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security*, 2(1):3–33, 1999.

[55] OMG. MOF Core v. 2.0. Document Id: formal/06-01-01. `http://www.omg.org`.

[56] OMG. OCL Version 2.0. Document id: formal/06-05-01. `http://www.omg.org`.

[57] OMG. UML: Superstructure. Version 2.0. Document id: formal/05-07-04. `http://www.omg.org`.

[58] Oracle Corporation. Data Sheet Oracle User Management 11i, October 2004. At http://www.oracle.com/technology/products/applications/security/OracleUserManagementDataSheet.pdf.

[59] Sylvia Osborn. Mandatory Access Control and Role-Based Access Control Revisited. In *Proceedings of the 2nd ACM workshop on Role-Based Access Control*, pages 31–40, Fairfax, VA, USA, November 1997.

[60] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, May 2000.

[61] Fang Pu, Daoqin Sun, Qiying Cao, Haibin Cai, and Fan Yang. Pervasive Computing Context Access Control Based on $UCON_{ABC}$ Model. In *Proceedings of the 2nd International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 689–692, Pasadena, CA, December 2006.

[62] Jens Linneberg Rasmussen and Mejar Singh. Designing a Security System by Means of Coloured Petri Nets. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, pages 400–419, London, UK, June 1996. Springer-Verlag.

[63] Indrakshi Ray and Mahendra Kumar. Towards a Location-Based Mandatory Access Control Model. *Computers & Security*, 25(1), February 2006.

[64] Indrakshi Ray, Mahendra Kumar, and Lijun Yu. LRBAC: A Location-Aware Role-Based Access Control Model. In *Proceedings of the 2nd International Conference on Information Systems Security*, pages 147–161, Kolkata, India, December 2006.

[65] Indrakshi Ray, Na Li, Robert France, and Dae-Kyoo Kim. Using UML to Visualize Role-Based Access Control Constraints. In *Proceedings of the 9th ACM symposium on Access Control Models and Technologies*, pages 115–124, Yorktown Heights, NY, USA, June 2004.

[66] Indrakshi Ray, Indrajit Ray, and Sudip Chakraborty. An interoperable context sensitive model of trust. *Journal of Intelligent Information Systems*, 32(1):75–104, February 2009.

[67] Indrakshi Ray and Manachai Toahchoodee. A Spatio-temporal Role-Based Access Control Model. In *Proceedings of the 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 211–226, Redondo Beach, CA, July 2007.

[68] Indrakshi Ray and Manachai Toahchoodee. A Spatio-Temporal Access Control Model Supporting Delegation for Pervasive Computing Applications. In *Proceedings of the 5th International Conference on Trust, Privacy & Security in Digital Business*, pages 48–58, Turin, Italy, September 2008.

[69] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universitaet Bremen, 2002. Logos Verlag, Berlin, BISS Monographs, No. 14.

[70] Debashis Saha and Amitava Mukherjee. Pervasive Computing: A Paradigm for the 21st Century. *Computer*, 36(3):25–31, March 2003.

[71] Geetanjali Sampemane, Prasad Naldurg, and Roy H. Campbell. Access Control for Active Spaces. In *Proceedings of the Annual Computer Security Applications Conference* , pages 343–352, Las Vegas, NV, USA, December 2002.

[72] Arjmand Samuel, Arif Ghafoor, and Elisa Bertino. A Framework for Specification and Verification of Generalized Spatio-Temporal Role Based Access Control Model. Technical report, Purdue University, February 2007. CERIAS TR 2007-08.

[73] Ravi Sandhu and Qamar Munawer. How to do Discretionary Access Control using Roles. In *Proceedings of the 3rd ACM Workshop on Role-Based Access Control*, pages 47–54, Fairfax, VA, USA, October 1998.

[74] Ravi S. Sandhu. Lattice-Based Access Control Models. *Computer*, 26(11):9–19, November 1993.

[75] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[76] Ravi S. Sandhu and Pierangela Samarati. Access Control: Principle and Practice. *IEEE Communications Magazine*, 32(9):40–48, September 1994.

[77] Andreas Schaad and Jonathan D. Moffett. A Lightweight Approach to Specification and Analysis of Role-Based Access Control Extensions. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, pages 13–22, Monterey, CA, USA, June 2002.

[78] Basit Shafiq, James B. D. Joshi, and Arif Ghafoor. Petri-net model for verification of RBAC Policies. Technical report, Purdue University, 2002.

[79] Heechang Shin and Vijayalakshmi Atluri. Spatiotemporal Access Control Enforcement under Uncertain Location Estimates. In *Proceedings of the 23rd Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 159–174, Montreal, P.Q., Canada, 2009.

[80] Richard Simon and Mary Ellen Zurko. Separation of Duty in Role-based Environments. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 183–194, Rockport, MA, USA, June 1997.

[81] Vince Stanford. Using pervasive computing to deliver elder care. *IEEE Pervasive Computing*, 1(1):10–13, January 2002.

[82] Manachai Toahchoodee, Ramadan Abdunabi, Indrakshi Ray, and Indrajit Ray. A Trust-Based Access Control Model for Pervasive Computing Systems. In *Proceedings of the 23rd Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 307–314, Montreal, Canada, July 2009.

[83] Manachai Toahchoodee and Indrakshi Ray. On the Formal Analysis of a Spatio-Temporal Role-Based Access Control Model. In *Proceedings of the 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 17–32, London, U.K., July 2008.

[84] Manachai Toahchoodee and Indrakshi Ray. Using Alloy to Analyze a Spatio-Temporal Access Control Model Supporting Delegation. *IET Information Security*, 3(3):75–113, September 2009.

[85] Manachai Toahchoodee and Indrakshi Ray. On the Formalization and Analysis of a Spatio-Temporal Role-Based Access Control Model. *Journal of Computer Security (to appear)*, 2010.

[86] Manachai Toahchoodee, Indrakshi Ray, Kyriakos Anastasakis, Geri Georg, and Behzad Bordbar. Ensuring Spatio-Temporal Access Control for Real-World Applications. In *Proceedings of the 14th ACM Symposium on Access control Models and Technologies*, pages 13–22, Stresa, Italy, June 2009.

[87] Manachai Toahchoodee, Xing Xie, and Indrakshi Ray. Towards Trustworthy Delegation in Role-Based Access Control Models. In *Proceedings of the 12th Information Security Conference*, pages 379–394, Pisa, Italy, September 2009.

[88] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mobile Computing and Communications Review*, 3(3):3–11, July 1999.

[89] Guo Ya-Jun, Hong Fan, Zhang Qing-Guo, and Li Rong. An Access Control Model for Ubiquitous Computing Application. In *Proceedings of the 2nd International Conference on Mobile Technology, Applications and Systems*, pages 1–6, Guangzhou, China, November 2005.

[90] Hai Yu and Ee-Peng Lim. LTAM: A Location-Temporal Authorization Model. In *Secure Data Management*, volume 3178 of *Lecture Notes in Computer Science*, pages 172–186, Toronto, Canada, August 2004.

[91] Chunyang Yuan, Yeping He, Jianbo He, and Zhouyi Zhou. A Verifiable Formal Specification for RBAC Model with Constraints of Separation of Duty. In *Proceedings of the 2nd SKLOIS Conference on Information Security and Cryptology*, pages 196–210, Beijing, China, November 2006.

[92] John Zao, Hoetech Wee, Jonathan Chu, and Daniel Jackson. *RBAC Schema Verification Using Lightweight Formal Model and Constraint Analysis*. At http://alloy.mit.edu/publications.php, 2002.

[93] Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chu. A rule-based framework for role-based delegation and revocation. *ACM Transactions on Information and System Security*, 6(3):404–441, August 2003.