**ATTRIBUTE-BASED ACCESS CONTROL MODELS AND IMPLEMENTATION IN CLOUD INFRASTRUCTURE AS A SERVICE**

APPROVED BY SUPERVISING COMMITTEE:

Ravi Sandhu, Ph.D., Co-Chair

Ram Krishnan, Ph.D., Co-Chair

Rajendra V. Boppana, Ph.D.

Hugh Maynard, Ph.D.

Jianwei Niu, Ph.D.

Accepted:

Dean, Graduate School

## DEDICATION

*This dissertation is dedicated to all my family, particularly to my wife, Xiaoyan, who patiently support me all the way. I must also thank all my friends.*

# ATTRIBUTE-BASED ACCESS CONTROL MODELS AND IMPLEMENTATION IN CLOUD INFRASTRUCTURE AS A SERVICE

by

XIN JIN, M.Sc

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
May 2014

## ACKNOWLEDGEMENTS

First, I would like to thank my advisors Prof. Ravi Sandhu and Prof. Ram Krishnan for their professional guidance. Without their help, I could not accomplish my research. They have provided guidance on how to do research and also provided great research topics which motivate me to grow and learn faster. They taught me to learn not only how to find and solve practical technical problems but also how to be a better person in life. They always emphasized that contributing to real world problems is more important than just submitting paper to top conferences, which is one of the most important takeaways from Ph.D life.

Second, I would like to express gratitude to my other committee members Prof. Rajendra V. Boppana, Prof. Hugh Maynard and Prof. Jianwei Niu for their valuable comments and suggestions.

Third, I would like to thank my colleagues and friends in the lab. We learned the tools for implementation together and they've provided great help when I got stuck in programming issues. Especially, I want to thank Farhan Patwa, director of our lab. Without him, I could not finish the implementation project, which is a crucial part of my dissertation. During this period, I gain industry level experience from which further benefit me in job hunting.

May 2014

**ATTRIBUTE-BASED ACCESS CONTROL MODELS AND IMPLEMENTATION IN CLOUD INFRASTRUCTURE AS A SERVICE**

Xin Jin, Ph.D.
The University of Texas at San Antonio, 2014

Supervising Professors: Ravi Sandhu, Ph.D. and Ram Krishnan, Ph.D.

Recently, there has been considerable interest in attribute-based access control (ABAC) to overcome the limitations of the classical access control models (i.e, discretionary-DAC, mandatory-MAC and role based-RBAC) while unifying their advantages. The general idea of ABAC is to determine access control based on the attributes of involved entities. Example user attributes are $department$, $clearance$ and $role$ and example object attributes are $size$, $createTime$ and $owner$. Authorization results are computed based on subject and object attributes and authorization policies. As attributes can be engineered to reflect appropriately detailed information about users, subjects and objects, ABAC ensures great flexibility in expressing fine-grained policies which are increasingly required by applications.

There has been considerable prior work for ABAC in various aspects such as formal models, enforcement models, implementation standards, policy composition languages and so on. However, there is no consensus on precisely what is meant by ABAC or the required features of ABAC. There is no widely accepted formal ABAC model as there are for DAC, MAC and RBAC. Questions such as what are the core components and configuration points of ABAC, and how are attributes assigned and modified remain to be fully investigated.

In this dissertation, we conduct a systematic study of ABAC models. Based on the sizable related work on ABAC and on existing classical access control models, we design models that cover operational and administrative ABAC. More specifically, the contributions are summarized into two parts. In the formal model part, we first define the $\text{ABAC}_\alpha$ model that has "just sufficient" features to be "easily and naturally" configured to do DAC, MAC and RBAC. We understand DAC to mean owner-controlled access control lists, MAC to mean lattice-based access control

with tranquility and RBAC to mean flat and hierarchical RBAC. We design basic components, configuration points and configuration languages for this model and give $\text{ABAC}_\alpha$ configurations for DAC, MAC and RBAC. To further extend the expressive power, we develop $\text{ABAC}_\beta$ model based on $\text{ABAC}_\alpha$. The basic motivation is to cover advanced features of the standard RBAC model as well as RBAC extensions. We show that without additional configuration points, $\text{ABAC}_\beta$ is able to unify numerous well-documented extended features for RBAC. We conjecture that $\text{ABAC}_\beta$ can serve as the most general ABAC model excluding attribute mutability as defined in usage control models. Secondly, based on the operational model, we design an administrative model called generalized user-role assignment model (GURA) to manage user attributes using administrative roles. We carry out comprehensive complexity analysis for the user-attribute reachability problem in GURA.

In the proof of concept part, we demonstrate the advantage of ABAC by applying it as an access control model in Infrastructure as a Service (IaaS) cloud, building upon the theoretical models enumerated above. We show the flexibility of our ABAC model by comparing it with existing IaaS models, which are primarily role-based. We design operational and administrative models for cloud IaaS. We define different enforcement models and implement them on a widely deployed open-source cloud platform OpenStack. Performance evaluation is provided to reflect the cost incurred by enforcing ABAC.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# Chapter 1: INTRODUCTION AND MOTIVATION

Access control is one of the earliest problems in computer security and remains a continuing challenge. Access control component determines whether requests to access resources are granted. The entity making the request is typically called a subject, which is usually a program or process operating on behalf of a user. A user is an entity who interacts with the system and accesses resources. As shown in figure 1.1, a subject requests to access objects which are resources (e.g., mp3 file, documents, networks) on behalf of users. Objects are protected by access control. After authenticating the user and receiving the request information from the subject, the access control component either grants or denies the request based on the provided information and the authorization policy. Depending on the information required for authorization and the process of making decisions, different access control models can be implemented for various purposes.

## 1.1 Motivation

Starting with Lampson's access matrix in the late 1960's, dozens of access control models have been proposed. Only three have achieved success in practice: discretionary access control (DAC) [123], mandatory access control (MAC, also known as lattice based access control or multilevel security) [121] and role-based access control (RBAC) [57, 122] (see figure 1.2). DAC controls access based on the identity of subjects and MAC makes access control decision based on the security level of subjects and objects. In RBAC, permissions are encapsulated in roles which are further assigned to users. Users activate assigned roles to get the permissions associated with the roles. These three models have deep conceptual, theoretical and intuitive foundations, and demonstrably address real-world practitioners' concerns.

In DAC, information may be accessed by unauthorized users because there is no control on copies of objects. MAC deals with information flow and solves this problem by attaching security levels on both users and objects. All users are required to obtain certain clearance to access objects. Security labels propagate to derivative objects, including copies. However, the policies in DAC and

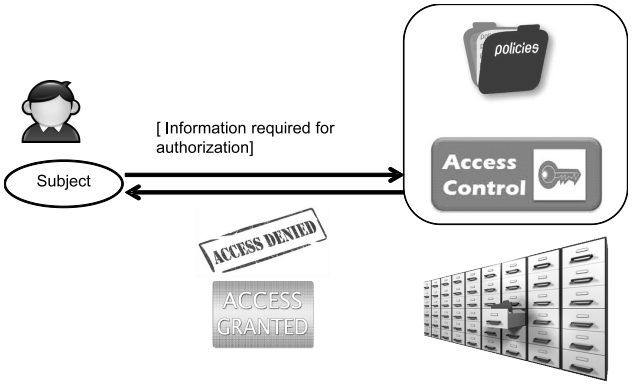



**Figure 1.1**: Access Control Example

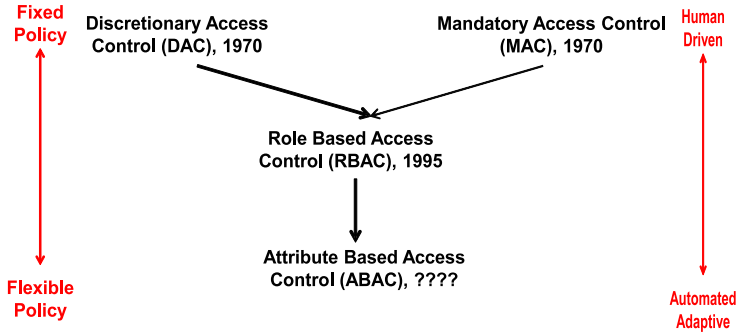



**Figure 1.2**: *Timeline of Classical Access Control Models [120]*

MAC are fixed and there is no room for flexible access control. RBAC emerged due to increasing practitioner dissatisfaction with the then dominant DAC and MAC paradigms, inspiring academic research on RBAC. Since then RBAC has become the dominant form of access control in practice. While DAC and MAC emerged in the early 1970's it took another quarter century for RBAC to develop robust foundations and flourish.

Recently there has been growing practitioner concern with the limitations of RBAC. For example, role explosion is caused by the situation where each role requires different sets of permissions and large number of roles have to be defined. Role engineering also delays the deployment of RBAC as it is the most costly process before deployment. Those limitations have been met by researchers in two different ways. On one hand researchers have diligently and creatively extended RBAC in numerous directions. For examples, role activation process has been extended to be constrained by contextual information such as time and location [67, 81], prerequisite roles [17] and

so on. Users are associated with additional information besides role, such as organization [151], group [96] and team [11, 137]. Permission is also extended to include purpose and conditions to support privacy aware RBAC [103]. More examples can be found in section 2.2. However, those extensions are proposed for specific purpose where customized RBAC is required. There is no framework which can combine the advantages of these extensions. Further extension to this model is not possible unless a new model is designed. The lack of inherent extendability of RBAC discourages deployment of its extensions since they are not general purpose. Beside extensions to RBAC, other models are proposed to overcome the limitations of traditional models. Examples are organization based access control [83], task based access control [136], and relationship based access control [58]. However, they too are developed for specific application context such as social networks and organizations, rather than for general purpose.

There is growing appreciation that a more general model, specifically attribute-based access control (ABAC), could encompass the demonstrated benefits of DAC, MAC and RBAC while transcending their limitations. Intuitively, an attribute is a property expressed as a name:value pair associated with any entity in the system, including users, subjects and objects and even attributes themselves. Appropriate attributes can capture identities and access control lists (DAC), security labels, clearances and classifications (MAC) and roles (RBAC). Languages for specifying permitted accesses based on the values and relationships among these attributes provide policy flexibility and customization. As such ABAC supplements and subsumes rather than supplants these currently dominant models. Moreover any number of additional attributes such as location, time of day, strength of authentication, departmental affiliation, qualification, and frequent flyer status, can be brought into consideration within the same extensible framework of attributes. Thus the proliferation of RBAC extensions might be unified by adding appropriate attributes within a uniform framework, solving many of these shortcomings of core RBAC. At the same time we should recognize that ABAC with its flexibility may further confound the problem of role design and engineering. Attribute engineering is likely to be a more complex activity, and a price we may need to pay for added flexibility. As shown on the left hand side of figure 1.2, ABAC provides

richer policy expressive power compared with traditional models. However, the proliferation and flexibility of policy configuration points in ABAC leads to greater difficulty in policy expression and comprehension relative to the simplicity of DAC, MAC and RBAC. It will require strong and comprehensive foundations for ABAC to flourish. As shown on the right hand side of figure 1.2, ABAC provides automation in access control compared to human administration in classical models. For example, in RBAC role has to be manually assigned to users before authorization while in ABAC, once the authorization policy is composed, authorization can be computed at the time of the request and permissions do not need to be pre-assigned to users.

Although considerable related research has been published and even formal models have been proposed for ABAC, there is still lack of a comprehensive model which precisely describes the constituent components and operations of ABAC. The related work mainly falls into the following four categories: authorization process, policy specification language, enforcement models and implementations. For example, UCON [111] and [141] mainly focus on the rich features in authorization process. The problem addressed by UCON is how to process authorization given the attributes and values from subject[1] and object. UCON specifies mutable attributes and continuous enforcement. [149] focuses on how to enforce ABAC based on the web infrastructure. This work is not at policy level rather based on certain ABAC enforcement models. [51, 75] focus on languages for specifying interesting and useful policies and propose features of languages such as richer decision (e.g., grant, deny, not applicable, error) and policy compliance (e.g., HIPPA). This addresses only one component of ABAC. XACML [2] provides a standardized mechanism to specify ABAC authorization policy, request and policy evaluation. Many authors following XACML have focused on conflict resolution (e.g., deny-override), policy integration and redundant resolution. Further discussion of related work can be found in section 2.

While it is generally accepted that ABAC provides flexible access control and supplements the limitations of traditional models, there is yet no agreement on a formal ABAC model. Fundamental questions such as components of core models lack authoritative answers, let alone a widely

[1] User and subject are not explicitly distinguished in UCON. We later demonstrate this distinction is needed.

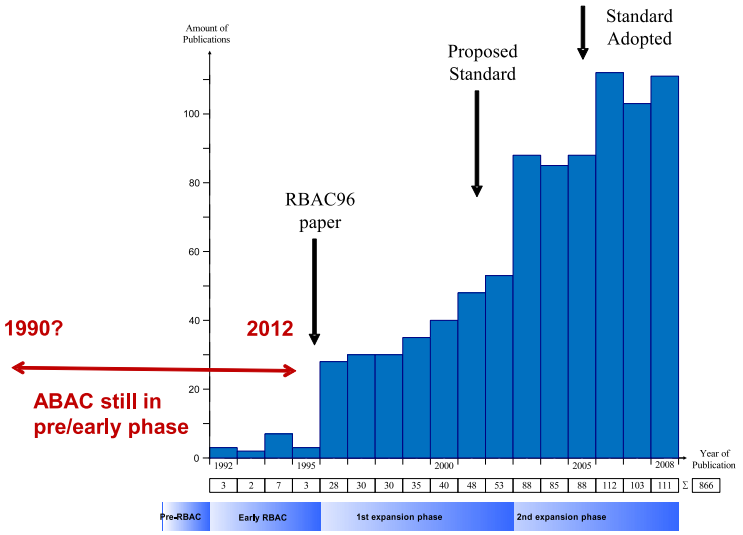



**Figure 1.3**: *Current Status of ABAC [120]*

accepted ABAC model. Formal and systematic study are needed to enhance the concept of ABAC. The ABAC situation today is analogous to RBAC in its pre-1992 pre-RBAC and 1992-1996 early-RBAC periods [61]. The development of RBAC model is illustrated in figure 1.3 which shows the state of RBAC model during the years 1992 to 2008. This figure is adopted from a survey paper of RBAC [61]. The y-axis shows the number of the related publications and the x-axis shows the year and publication numbers of RBAC-related paper. Much as RBAC concepts were around for decades (from early 1970s to 1996) before their formalization as shown in figure 1.3, nascent ABAC notions have been around for a while (see related work). This dissertation focuses on the formal model of ABAC, providing foundations towards a widely agreed formal ABAC model.

## 1.2 Research Challenges

In order to define a formal ABAC model, we start by looking at the development path of the successful RBAC model from 1992 to date. Firstly, what should a formal model look like and what are the core features. For this purpose, we review the successful RBAC model in figure 1.4. This formal model defines the core components and configuration points.

- **Core components.** This part defines the basic sets and function of the model. Those include

the required information for all authorization decisions. In the case of RBAC, user, roles, objects and operations are defined.

- **Model configuration.** The green box represents security architects who can configure the model using configuration points. In RBAC, roles are the only configuration point (other than defining the basic sets).

In this dissertation we define the similar parts for ABAC models.

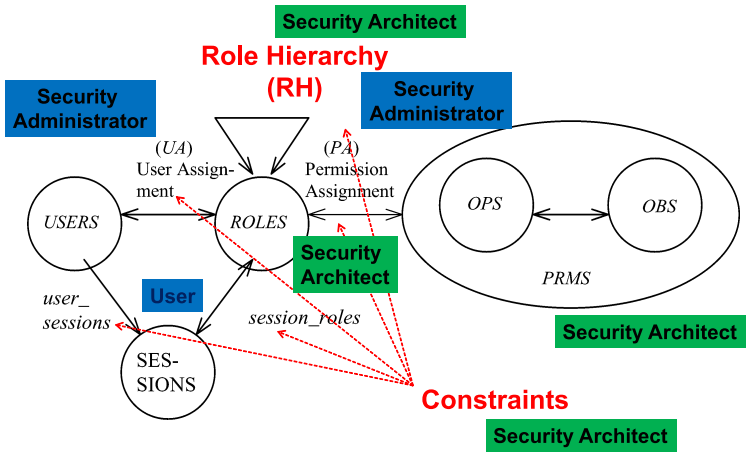



**Figure 1.4**: *Structure of RBAC Model*

Secondly, administrative model needs to be defined to manage the operational model. In the context of RBAC, this deals with issues such as user-role assignment, role-permission assignment, role hierarchy design and so on. The blue box in figure 1.4 represents security administrators who can decide user-role assignment and role-permission assignment. Similarly, a corresponding model is required to deal with attribute administration in ABAC.

The above motivation focuses on the need to develop a systematic investigation on formal ABAC models. However, another challenge is to demonstrate that the ABAC model defined in this dissertation is well suited to practical applications. Thus, we need to present a concrete example where an ABAC model satisfies the specific requirements of that application.

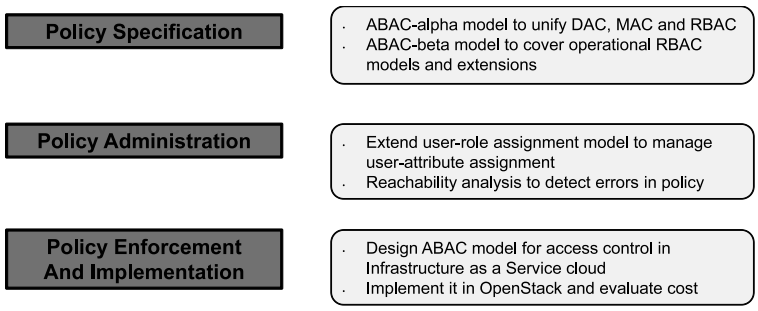



**Figure 1.5**: *Dissertation Contribution*

## 1.3 Contribution

We summarize the contribution in figure 1.5 and we explain them as follows:

- **ABAC Models**. Two ABAC models are proposed with increasing expressive power.
  - **ABAC$_\alpha$ Model**. With the widely deployment of the classical models in many different areas, an ABAC model that cannot configure these models would not be acceptable as it will discourage users from adopting ABAC. Thus, an initial ABAC$_\alpha$ model is defined with the purpose of establishing the connections between classical models and ABAC. ABAC$_\alpha$ is designed with the "least" features to cover DAC, MAC and RBAC. It provides four configuration points. With proper specifications, this framework can simulate those classical models. This basic ABAC model serves as the starting point of our ABAC research.
  - **ABAC$_\beta$ Model**. To further enhance the expressive power of ABAC model, we build an extended model by exploring the required features of ABAC to express RBAC-related extensions. We choose this direction because of the wide adoption of RBAC and proliferation of RBAC-extended models.
- – **User attribute administrative model**. The assumption in ABAC models is that user attributes are administered by separate administrative models. In this part, we define an administrative model GURA (generalized user-role assignment) by extending user-role administration model. It provides ARBAC97 style user attributes administration. It

takes advantages of RBAC to manage user attributes. The core idea is that permissions to modify user attributes are associated with administrative roles. Administrators are made members of these roles, thus obtaining associated permissions. It is also possible to use ABAC to administer ABAC. However, the model we provide here is sufficient for this purpose and serves as the starting point for future research.

  - **User attribute reachability analysis**. This topic studies the user-attribute reachability problems in a restricted version of the GURA model called rGURA. For this purpose, we formalize the model as a state transition system. We show that the reachability problems for general cases of rGURA are PSPACE-complete. However, we do find polynomial-time solutions to reachability problems for limited versions of rGURA that are still useful in practice. The algorithms not only answer reachability problem but also provide a "plan" of sequential attribute updates by different administrators in order to reach particular values for user attributes. Some open questions and future research directions are discussed.

- **ABAC model for access control in cloud infrastructure as a service**. We demonstrate proof of concept by utilizing ABAC in the area of access control for Infrastructure as a Service cloud. We analyze the requirement of access control in IaaS cloud and motivate the advantages of ABAC by showing that ABAC satisfies these requirements. More specifically, we design operational and administrative model for IaaS cloud and implement proof-of-concept based on widely deployed open-source platform OpenStack. Performance evaluation is carried out to evaluate the cost of enforcing ABAC.

## 1.4 Organization

Related work is discussed in chapter 2. We introduce formal ABAC models in chapter 3. Chapter 4 introduces administrative model for ABAC. These chapters complete our formal policy level contributions. Chapter 5 provide proof-of-concept by motivating ABAC as a suitable access control model for single tenants in cloud IaaS. Our implementation use the widely deployed cloud platform

OpenStack. We provide an evaluation of the performance. Chapter 6 concludes the dissertation and discusses future work.

# Chapter 2: RELATED WORK

## 2.1 Classical Access Control Models

### 2.1.1 Discretionary Access Control

We understand DAC as user-discretionary access control [123]. DAC governs the access of users to the information on the basis of the user's identity and authorizations that specify, for each user and object in the system, the access modes (e.g., read, write, execute) the user is allowed on the object, i.e., access list for each object. Each request of a user to access an object is checked against the specified authorization. If there is an authorization specifying that the user can access the object in the specific mode, the request is granted, otherwise, denied. In user-discretionary access control, the users become the owner of an object created by them and they are the only one who can specify the authorization policy on those objects and destroy the object. While some models allow transfer and sharing of ownership, for the most part ownership cannot be transferred to another user in DAC models. For example, Alice creates object A in the system and thus becomes the owner. If user Alice requests to read object A, it is authorized because Alice is the creator of object A. Alice is the only one who can grant and revoke the access on object A for other users. Alice then can grant Bob to read and write object A.

DAC has been widely used in industrial and commercial environments. One drawback of DAC is that the information flow is not enforced. For example, a user who is authorized to read the data can bypass access controls by creating a copy and sharing it with other users without the cognizance of the owner. The reason is that DAC does not impose any restriction on copies of objects.

### 2.1.2 Mandatory Access Control

Mandatory access control (MAC) [121] governs access based on the classification of subjects and objects in the system. Each user, subject and object in the system is assigned a security level. The

security level associated with an object reflects the sensitivity of the information contained in the objects, i.e., the potential damage that could result from unauthorized disclosure of the information. The security level associated with a user also called clearance reflects the user's trustworthiness not to disclosure sensitive information to users not cleared to see it. Request from users to object is granted only if some relationship is satisfied between the security levels associated with the two. In particular, the following properties hold, where $\gamma$ signifies the security label of the indicated subject or object.

- **Simple Security Property.** Subject $s$ can read object $o$ if the security level of s is equal or higher than that of the object $o$, i.e., $\gamma(s) \geq \gamma(o)$.
- **Liberal $\star$-Property.** Subject $s$ can write object $o$ if the security level of subject is equal or lower than that of the object, i.e., $\gamma(s) \leq \gamma(o)$.
- An alternative to liberal star property is **Strict $\star$-Property.** Subject can write object if the security level of subject is the same of that of the object, i.e., $\gamma(s) = \gamma(o)$.

However, many practical requirements are not covered by MAC as it rises from rigid environment (e.g., military, national intelligence, civilian government) and it does not satisfy requirements of commercial enterprises.

### 2.1.3 Role based Access Control Model

The basic structure of role based access control (RBAC) [57] is shown in figure 2.1. It regulates user access control on the basis of the activities the users execute in the system. Roles define the meanings of the activities and are associated with a set of permissions which are operations on objects. Users are then assigned to certain set of roles and get the permissions associated with the roles. Users can activate any subset of the assigned roles in any sessions. A request made by users with certain roles is authorized if the user has currently activated a role which contains the permission. Advanced features of RBAC include role hierarchy and constraints. Role hierarchy defines partial order between roles. Senior roles inherit all permissions from junior roles. This

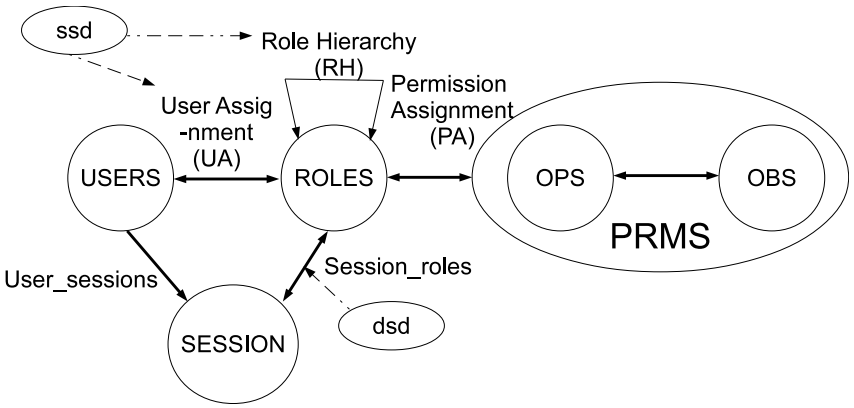



**Figure 2.1**: *The Structure of NIST-RBAC Model [57]*

feature in RBAC provides ease in system administration. Many security assurance can be achieved using RBAC such as least privilege, static and dynamic separation of duties and so on.

## 2.2 Related Work of ABAC Model

There are a large number of related research and they can be categorized into four classes discussed as follows.

### 2.2.1 Formal Model

The first category is theoretical access control models. Role based trust management (RT) [95] provides a set of role assignment credentials. The only attribute is role and the authorization policy is the same as that in role based access control and is not configurable. Policy Machine (PM) [70] is proposed to provide a unified framework to support a wide range of attribute-based policies or policy combinations through a single mechanism that requires changes only in its data configuration. The National Institute of Standards and Technology (NIST) recently released a first draft ABAC model [69]. This draft provides detailed guidelines in various aspects of enterprise ABAC while no formal model is provided. The UCON usage control model [111] focuses on usage control where authorizations are based on the attributes of the involved components. It is attribute-based but, rather than dealing with core ABAC concepts, it focuses on advanced access control features such as mutable attributes, continuous enforcement, obligations and conditions.

UCON more or less assumes that an ABAC model is in place on top of which the UCON model is constructed. [141] models authorization policy of access control using logic programming with set constraints of a computable set theory. Similarly with UCON, this work focuses only on one component in ABAC which is authorization. It is based on the assumption that users and objects are associated with sets of attributes.

### 2.2.2 Policy Specification Language

The second category is on authorization policy specification languages. SecPAL [21, 24] has a concrete syntax consisting of simple statements close to natural language. DYNPAL [23] and SMP [22] enable specifying modifications to the system state in authorization. [141] proposes to specify authorization policies using set theory to ensure consistency and completeness. Binder [54] is an extension to the datalog logical-programming language. In Soutei [112] policies and credentials are written in a declarative logic-based security language. [28] proposes a formal framework based on C-Datalog language. Rule-based policy specification [12] enables authorization policy specification based on system behavior. Other examples are SPKI/SDSI [55], extensible access control markup language (XACML) [2] and enterprise privacy authorization language (EPAL) [85]. There has been considerable research based on XACML. [114] simplifies XACML policies by providing an ontology-based attribute management facility. Other examples are policy integration [99, 115], policy evaluation [97], conformance checking [71] and policy derivation [143]. Xu *et al* [146] propose authorization policy mining. PolicyMorth [92] proposes a framework to support interactive ABAC policy specification and maintenance. In summary, all these work focus on how authorization policy can be specified and evaluated. While authorization policy is an important component of ABAC, these authors do not present comprehensive formal models for ABAC.

### 2.2.3 Enforcement Model

The third category is about concerns on enforcement of ABAC systems. This class deals with problems such as how to represent, store, transfer and authenticate attributes. In credential based access control [91], attribute assertions of subject and environment are encoded in verifiable digital credentials issued by trusted third-party certifiers. [46] demonstrates that ABAC can be used as a primary authorization and authentication mechanism for legacy or modern enterprise systems, but it is based only on RT model. Akenti [138] is an authorization service based on X.509. Information communication between service requester and service provider are investigated. PolicyMaker [32] proposes an approach to trust management. KeyNote [31] and SPKI/SDSI [47] use credentials to delegate permissions. Automated trust negotiation [34, 35, 142] deals with credential disclosure before authorization evaluation. [60] presents an efficient protocol that protects both sensitive credentials and policies. [13,37,128] deal with privacy concerns in requester attribute release. [86,149] discusses general ABAC implementation architecture for web service. Although these works provide tools for enforcing ABAC, they do not present the full picture of theoretical aspect of ABAC such as subject and user distinction.

### 2.2.4 Attribute-based Encryption

The fourth category is attribute based encryption (ABE). It is proposed to support fine-grained sharing of encrypted data. In this kind of systems, cipher-texts are labeled with sets of attributes and private keys are associated with access structures. The decryption of a ciphertext thus is possible only if the set of attributes of the user key matches the attributes of the ciphertext. [117] allowed for decryption when at least $k$ attributes overlap between a cipher-text and a private key. Key-Policy Attribute-Based Encryption (KP-ABE) [65] extends the above work to associate policy tree instead of lists of attributes with private keys. Ciphertext-Policy Attribute-Based Encryption (CP-ABE) [30] on the other hand enables encryptor to associate policy trees with cipher-text. [109] extends KP-ABE so that private keys can represent non-monotone access formula over attributes.

[45] enables multiple authorities to monitor attributes and distribute secret keys. In summary, ABE is a method for securely enforcing ABAC policies on data sharing and access control. Due to the complexity and cost, the access control policies (logical **AND**, **OR**, **NOT**, etc.) included in these techniques are very limited in expressive power.

## 2.3 Related Work for Attribute Administrative Model

Firstly, we introduce the related work to user-attribute administration model. There are very limited number of models proposed for this purpose. However, we find that the administrative model for user-role assignment is highly related. Administrative role based access control (ARBAC97) [118] manages user-role assignment, permission-role assignment and role-hierarchy for RBAC. The major concept for user-role assignment is that users need to be previously assigned to certain roles in order to get new roles. Role based trust management [95] defines credentials for delegating user-role assignment to third parties to support distributed user-role assignment. However, these work focus on a single attribute which is role and the connection between attributes of the same user is not covered.

Secondly, we introduce the related work for reachability analysis for user attributes. The closest category of this work is the role reachability analysis on the role based administrative model ARBAC97. Li et al [94] presented algorithms and complexity results for analysis of two restricted versions of ARBAC97–AATU and AAR. However, this work did not consider negative preconditions. Sasturkar et al [124] and Jha et al [76] presented algorithms and complexity results for analysis of ARBAC policies subject to a variety of restrictions on how the policy can be specified. Stoller et al [129] proposed the first fixed-parameter-tractable algorithm for analyzing ARBAC policies. However, the algorithm only applies to rules with one positive precondition and unconditional role revocation. Stoller et al [130, 131] analyzed security on parameterized RBAC and ARBAC97. Although the parameters of role can be considered as user attributes, all parameters are treated as atomic-valued and are only changed together with the modification of role. Similar works are [9, 14] which presented symbolic analysis for attribute RBAC models. Our work is

fundamentally different from these in consideration of administration of multiple attributes including atomic valued attributes, whereas the ARBAC97 analysis only deals with a single set-valued attribute called role. The second category is policy analysis in attribute-based models. Gupta et al [66] proposed rule-based administrative policy model that controls addition and removal of both rules and facts, and a symbolic analysis algorithm for answering reachability queries. The facts of users may be termed as attributes. However, the model does not distinguish atomic and set valued attributes and the current version of the algorithm is incomplete. Li et al [93] discussed security analysis in role based trust management (RT). It is different from our work in that the focus is on delegation and trust. In addition, only one attribute, i.e. role, is considered. Jajodia et al [75] proposed a policy language to express positive and negative authorizations and derived authorizations, and they give polynomial-time algorithms to check consistency and completeness of a policy. [20] showed how to eliminate policy mis-configurations using data mining. [53] presented security constraint patterns for modelling security system architecture and verifying whether required security constraints are correctly enforced. However, this framework facilitates design and deployment of security polices rather than run-time security analysis. [74] developed a graphical constraint expression model to simplify constraints specification and make safety verification practical, but does not ensure polynomial time safety checking.

# Chapter 3: ATTRIBUTE BASED ACCESS CONTROL MODELS

## 3.1 Scope

Our eventual goal is to develop an authoritative family of foundational models for attribute-based access control. We believe this goal is best pursued by means of incremental steps that advance our understanding. ABAC is a rich platform. Addressing it in its full scope from the beginning is infeasible. There are simply too many moving parts. A reasonable first step is to develop a formal ABAC model, which we call $\text{ABAC}_\alpha$, that is just sufficiently expressive to capture DAC, MAC and RBAC. This provides us a well-defined scope while ensuring that the resulting model has practical relevance. There have been informal demonstrations, such as [41, 111], of the classical models using attributes. Our goal is to develop more complete and formal constructions.

In the second step, we develop the $\text{ABAC}_\beta$ model based on the observation that RBAC has been dominant in access control in both industry and academia. Extensions of RBAC have been proposed in many directions to meet the different requirements of various systems. These systems are potential applications of ABAC because their practical applications are well-documented in the literature. $\text{ABAC}_\beta$ helps understand the capability of ABAC in expressing different policies. It also unifies numerous RBAC extensions in a single model.

$\text{ABAC}_\alpha$ and $\text{ABAC}_\beta$ are developed based on extensive prior research where practical motivations are well-defined. We believe that $\text{ABAC}_\alpha$ model contains the core features of a future standard ABAC model. Many extensions can be proposed based on $\text{ABAC}_\alpha$ with $\text{ABAC}_\beta$ being one important example. These models strongly indicate what needs to be provided as core features of ABAC and the method to make extensions for ABAC. A standard family of ABAC models analogous to the NIST standard family of RBAC models [57, 122] must include $\text{ABAC}_\alpha$ and $\text{ABAC}_\beta$ as particular instances. Moreover it is likely to incorporate the policy configuration and administration points identified in these two models.

To make our goals for $\text{ABAC}_\alpha$ precise we characterize DAC, MAC and RBAC as follows.

**DAC**

There are many variations of DAC models [68, 123]. The DAC we define for our purpose is user-discretionary access control and features the following properties.

- Each user in the system is assigned a unique $userID$ which can uniquely distinguish the user in the system.
- The user can create a subject which can only be assigned with the same $userID$ as the user. The creating user becomes the only owner of the subject and only the owner can terminate the subject. Each subject can create objects in the system. The owner of the creating subject becomes the only owner of an object and ownership cannot be transferred.
- Each object is associated with access control lists which contain the $userID$s who can access the object in the corresponding access mode. For example, the $readlist$ and $writelist$ of objects contain lists of users who can read and write the object respectively.
- Only the owner of an object can modify the access control list. The owner can add or remove any $userID$s from the access control list of the objects he or she owns. Only the owner can destroy the object.

**MAC**

We understand MAC as lattice based access control (LBAC) [121] with tranquility. More specifically, we have the following characterizations.

- Each user and subject are labelled with a clearance level. Each object is labelled with a sensitivity level. Clearance and sensitivity are from the same partially ordered set of security levels.
- A user can create subjects at his own or lower levels than his assigned clearance and access resources using the subject. The creating user becomes the only owner of the created subject and is the only one who can destroy the subject.

- The sensitivity of objects cannot be changed after creation and users cannot modify the clearance level of the subjects they own.

As for authorization policy, we will discuss simple security property, liberal star property and strict star property in developing $ABAC_\alpha$.

**RBAC**

The RBAC models we consider are $RBAC_0$ and $RBAC_1$ from NIST definition [57] and the RBAC96 model [122]. $RBAC_0$ has the following features.

- There exists a set of roles defined by system architects and there is no partial order among those roles. Each user is associated with a set of roles which are assigned by administrators. Each role is associated with a set of permissions which are in the format of operations on objects.

- Users can create sessions in the system and activate any subset of the assigned roles in a session. The creating user thus becomes the owner of the session and is the only one who can delete the session. When a user deletes a session, the association between the session and activated role is also deleted. A session is essentially equivalent to a subject.

- The owner of a session can activate and deactivate the roles that he is assigned in that session. User permissions in each session is determined by the set of activated roles within that session.

The only difference between $RBAC_1$ and $RBAC_0$ is that the role set in $RBAC_1$ is a partially ordered where the senior roles inherit all permissions from the junior roles. In addition, users can activate junior roles of their assigned roles in their sessions.

## 3.2 $ABAC_\alpha$ Model

In this section, we present the $ABAC_\alpha$ model developed to have the "least" features to cover DAC, MAC and RBAC as defined above.

### 3.2.1 Model Requirement

The intrinsic features of $ABAC_\alpha$ that follow from the above interpretation of DAC, MAC and RBAC are highlighted in table 3.1. This table recognizes three kinds of familiar entities: users, subjects (or sessions in RBAC) and objects. Each user, subject and object has attributes associated with it. The range of each attribute is either atomic valued or set valued, with atomic values partially ordered or unordered and set values ordered by subset.

**Table 3.1**: $ABAC_\alpha$ Intrinsic Requirements

| | Subject attribute values constrained by creating user? | Object attribute values constrained by creating subject? | Attribute range ordered? | Attribute functions return set value? | Object attributes modification? | Subject attribute modification by creating user? |
|---|---|---|---|---|---|---|
| DAC | YES | YES | NO | YES | YES | NO |
| MAC | YES | YES | YES | NO | NO | NO |
| $RBAC_0$ | YES | NA | NO | YES | NA | YES |
| $RBAC_1$ | YES | NA | YES | YES | NA | YES |
| $ABAC_\alpha$ | YES | YES | YES | YES | YES | YES |

Let us consider each column in turn.

**Column 1.** In all cases subject attribute values are constrained by attributes of the creating user. In MAC, users can only create subjects whose clearance is dominated by that of the user. In RBAC, subjects can only be assigned roles assigned to or inherited by the creating user. In DAC, MAC and RBAC, the subject's creator is set to be the creating user. Interestingly this is the only column with YES values for all rows.

**Column 2.** For object attributes in MAC a subject can only create objects with the same or higher sensitivity as the subject's clearance. In DAC there is no constraint on the access control list associated with a newly created object. It is up to the creator's discretion. However, we recognize that DAC has a constraint on newly created objects in that root user usually has all access rights to every object and the owner cannot forbid this. RBAC does not speak to object creation.

**Column 3.** In MAC clearances are values from a lattice of security labels. In $RBAC_1$ roles are partially ordered by permission inheritance. DAC and $RBAC_0$ do not require ordered attribute

values.

**Column 4.** In MAC the clearance attribute is atomic valued as a single label from a lattice.[1] In $\text{RBAC}_0$ and $\text{RBAC}_1$ attributes are sets of roles, and in DAC each access control list is a set of user identities.

**Column 5.** In DAC the user who created an object can modify its access control lists. MAC (with tranquility) does not permit modification of an object's classification. $\text{RBAC}_0$ and $\text{RBAC}_1$ do not speak to this issue.

**Column 6.** Modification of subject attributes by the creating user is explicitly permitted in $\text{RBAC}_0$ and $\text{RBAC}_1$ to allow dynamic activation and deactivation of roles. DAC and MAC do not require this feature.

Each column imposes requirements on $\text{ABAC}_\alpha$ so we have YES across the entire row. Table 3.1 is, of course, not a complete list of all required features to configure the classical models, but rather highlights the salient requirements that stem from each classical model.

### 3.2.2 Model Overview

The structure of the $\text{ABAC}_\alpha$ model is shown in Figure 3.1. The core components of this model are: users ($U$), subjects ($S$), objects ($O$), user attributes ($UA$), subject attributes ($SA$), object attributes ($OA$), permissions ($P$), authorization policies, and constraint checking policies for creating and modifying subject and object attributes.

An **attribute** is a function which takes an entity such as a user and returns a specific value from its range. An attribute range is determined by its scope and type. The scope of an attribute is given by a finite set of atomic values. An atomic valued attribute will return one value from the scope, while a set valued attribute will return a subset of the scope. In other words, the range of an atomic valued attribute is equal to its scope while for a set valued attribute the range is the powerset of the scope. Each **user** is associated with a finite set of **user attribute** functions whose values are assigned by security administrators (outside the scope of the $\text{ABAC}_\alpha$ model). These attributes

[1] In the military lattice the lattice labels are constructed as a pair of hierarchical levels and a set of categories. If the pair is represented as two attributes then MAC attributes can also be regarded as set valued.

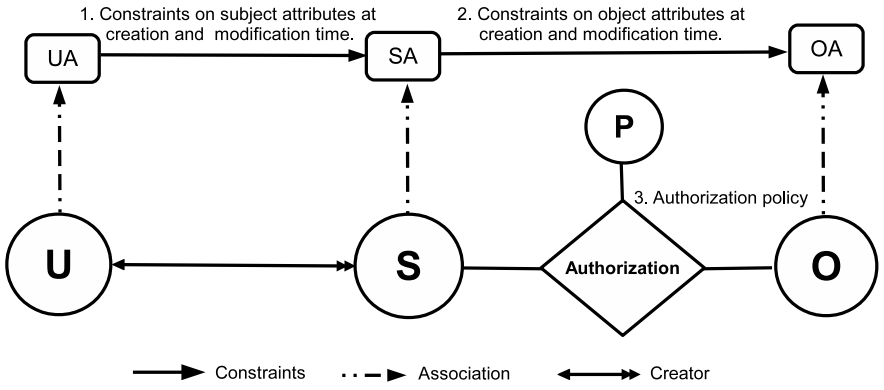



**Figure 3.1**: *Unified ABAC Model Structure*

represent user properties, such as name, clearance, roles and gender. **Subjects** are created by users to perform some actions in the system. For the purpose of this model, subjects can only be created by a user and are not allowed to create other subjects. The creating user is the only one who can terminate a subject. Each subject is associated with a finite set of **subject attribute** functions which require an initial value at creation time. Subject attributes are set by the creating user and are constrained by policies established by security architects. For example, a subject attribute value may be inherited from a corresponding user attribute. This is shown in Figure 3.3 as an arrow from user attributes to subject attributes. **Objects** are resources that need to be protected. Objects are associated with a finite set of **object attribute** functions. Objects may be created by a subject on behalf of its user. At creation, the object's attribute values may be set by the user via the subject. The values may be constrained by the corresponding subject's attributes. For example, the new object may inherit values from corresponding subject attributes. In Figure 3.3, the arrow from subject attributes to object attributes indicates this relationship.

**Constraints** are functions which return true when conditions are satisfied and false otherwise. Security architects configure constraints via policy languages. Constraints can apply at subject and object creation time, and subsequently at subject and object attribute modification time.

**Permissions** are privileges that a user can hold on objects and exercise via a subject. Permissions enable access of a subject to an object in a particular mode, such as read or write. Permissions definition is dependent on specific systems built using this model.

**Table 3.2**: *Basic Sets and Functions of ABAC$_\alpha$*

U, S and O represent finite sets of existing users, subjects and objects respectively.

UA, SA and OA represent finite sets of user, subject and object attribute functions respectively. (Henceforth also referred to as simply attributes.)

P represents a finite set of permissions.

For each $att$ in UA $\cup$ SA $\cup$ OA, SCOPE$_{att}$ is a constant finite set of atomic values.

SubCreator: S $\rightarrow$ U. For each subject SubCreator gives its creator.

$\mathrm{attType}$: UA $\cup$ SA $\cup$ OA $\rightarrow$ {set, atomic}. Specifies attributes as set or atomic valued.

Each attribute function maps elements in U, S and O to atomic or set values.

$$\forall ua \in \text{UA}.\ ua : \text{U} \rightarrow \begin{cases} \text{SCOPE}_{ua} & \text{if attType}(ua) = \text{atomic} \\ 2^{\text{SCOPE}_{ua}} & \text{if attType}(ua) = \text{set} \end{cases}$$

$$\forall sa \in \text{SA}.\ sa : \text{S} \rightarrow \begin{cases} \text{SCOPE}_{sa} & \text{if attType}(sa) = \text{atomic} \\ 2^{\text{SCOPE}_{sa}} & \text{if attType}(sa) = \text{set} \end{cases}$$

$$\forall oa \in \text{OA}.\ oa : \text{O} \rightarrow \begin{cases} \text{SCOPE}_{oa} & \text{if attType}(oa) = \text{atomic} \\ 2^{\text{SCOPE}_{oa}} & \text{if attType}(oa) = \text{set} \end{cases}$$

**Authorization policies** are two-valued boolean functions which are evaluated for each access decision. An authorization policy for a specific permission takes a subject, an object and returns true or false based on attribute values. More generally, access decision may be three-valued, possibly returning "don't know" in addition to true and false. This is appropriate in multi-policy systems. It suffices for our purpose to consider just two values. Security architects are able to specify different authorization policies using the languages offered in this model.

### 3.2.3 Formal Model

The basic sets and functions in ABAC$_\alpha$ are given in Table 3.2. U is the set of existing users and UA is a set of attribute function names for the users in U. Each attribute function in UA maps a user in U to a specific value. This could be atomic or set valued as determined by the type of the attribute function ($\mathrm{attType}$). We specify similar sets and functions for subjects and objects. SubCreator is a distinguished attribute that maps each subject to the user who creates it (an alternate would be to treat this attribute as a function in SA). Finally, P is a set of permissions.

We assume the attribute functions for a given user, subject or object. can all be evaluated in

constant time independent of the size of U, S and O. Since the sets S and O are unbounded due to creation of subjects and objects respectively, the overall storage for maintaining attribute values will also grow in proportion to the number of subjects and objects. Note that the SCOPE of an attribute must be a constant finite set, so it cannot be S or O but is permitted to be U since the set of users remains fixed in $\text{ABAC}_\alpha$. The latter capability is required to accommodate DAC.

We define four policy configuration points as shown in Table 3.3. The first is for authorization policies (item 1 in table 3.3). The security architect specifies one authorization policy for each permission. The authorization function returns true or false based on attributes of the involved subject and object. The second configuration point is constraints for subject attribute assignment (item 2 in table 3.3). The third is constraints for object attributes assignment at the time of object creation (item 3 in table 3.3). The fourth is constraints for object attribute modification after the object has been created (item 4 in table 3.3). Note that we have not provided separate configuration points for subject attribute assignment at creation and at modification after it has been created. Instead a single constraint covers both cases. For objects, however, we have found it is necessary to have separate constraints in these two cases.

**Policy Configuration Languages.** Each policy configuration point is expressed using a specific language. The languages specify what information is available for the functions that configure the four points discussed above. For example, in $\mathrm{ConstrSub}$ function, only attributes from the user who wants to create the subject as well as the proposed subject attribute values are allowed. Since all specification languages share the same format of logical structure while differing only in the values they can use for comparison, we define a template called Common Policy Language (CPL). CPL is not a complete language since the symbols $set$ and $atomic$ are left unspecified. It can be instantiated differently for each configuration point by further specifying these two symbols. CPL is defined in table 3.4.

$\mathrm{LAuthorization}$ is a CPL instantiation for specifying authorization policies in which $set$ and $atomic$ are specified as follows.

set ::= $\mathrm{setsa(s)} \mid \mathrm{setoa(o)}$

**Table 3.3**: *Policy Configuration Points and Languages of $ABAC_\alpha$*

**1. Authorization policies.**
For each p ∈ P, Authorization$_p$(s:S,o:O) returns true or false.
Language $\mathrm{LAuthorization}$ is used to define the above functions (one per permission), where s and o are formal parameters.

**2. Subject attribute assignment constraints.**
Language $\mathrm{LConstrSub}$ is used to specify ConstrSub(u:U,s:S,saset:SASET), where u, s and saset are formal parameters. The variable saset represents proposed attribute name and value pairs for each subject attribute. Thus $\mathrm{SASET}$ is a set defined as follows:

$$\mathrm{SASET} = \bigcup_{\forall sa \in SA} \text{OneElement}(\mathrm{SASET}_{sa}) \text{ where}$$

$$\mathrm{SASET}_{sa} = \begin{cases} \{sa\} \times \mathrm{SCOPE}_{sa} \text{ if } \mathrm{attType}(sa) = \mathrm{atomic} \\ \{sa\} \times 2^{\mathrm{SCOPE}_{sa}} \text{ if } \mathrm{attType}(sa) = \mathrm{set} \end{cases}$$

We define OneElement to non-deterministically return a singleton subset from its input set.

**3. Object attribute assignment constraints at object creation time.**
Language $\mathrm{LConstrObj}$ is used to specify $\mathrm{ConstrObj}$(s:S,o:O,oaset:OASET), where s, o and oaset are formal parameters. The variable oaset represents proposed attribute name and value pairs for each object attribute. Thus $\mathrm{OASET}$ is a set defined as follows:

$$\mathrm{OASET} = \bigcup_{\forall oa \in OA} \text{OneElement}(\mathrm{OASET}_{oa}) \text{ where}$$

$$\mathrm{OASET}_{oa} = \begin{cases} \{oa\} \times \mathrm{SCOPE}_{oa} \text{ if } \mathrm{attType}(oa) = \mathrm{atomic} \\ \{oa\} \times 2^{\mathrm{SCOPE}_{oa}} \text{ if } \mathrm{attType}(oa) = \mathrm{set} \end{cases}$$

**4. Object attribute modification constraints.**
Language $\mathrm{LConstrObjMod}$ is used to specify $\mathrm{ConstrObjMod}$(s:S,o:O,oaset:OASET), where s, o and oaset are formal parameters.

**Table 3.4**: *Definition of* CPL

$\varphi$ ::= $\varphi \wedge \varphi$ | $\varphi \vee \varphi$ | ( $\varphi$ ) | $\neg \varphi$ | $\exists$ x ∈ set.$\varphi$ | $\forall$ x ∈ set. $\varphi$ | set setcompare set | atomic ∈ set
| atomic atomiccompare atomic
setcompare ::= $\subset$ | $\subseteq$ | $\nsubseteq$
atomiccompare ::= $<$ | = | $\leq$

atomic ::= $\mathrm{atomicsa(s)} \mid \mathrm{atomicoa(o)}$

$\mathrm{setsa} \in \{\mathrm{sa} \mid \mathrm{sa} \in \mathrm{SA} \wedge \mathrm{attType(sa)} = \mathrm{set}\}$

$\mathrm{setoa} \in \{\mathrm{oa} \mid \mathrm{oa} \in \mathrm{OA} \wedge \mathrm{attType(oa)} = \mathrm{set}\}$

$\mathrm{atomicoa} \in \{\mathrm{oa} \mid \mathrm{oa} \in \mathrm{OA} \wedge \mathrm{attType(oa)} = \mathrm{atomic}\}$

$\mathrm{atomicsa} \in \{\mathrm{sa} \mid \mathrm{sa} \in \mathrm{SA} \wedge \mathrm{attType(sa)} = \mathrm{atomic}\}$

$\mathrm{LAuthorization}$ allows one to specify policies based only on the attribute values of the involved subject and object. Parameters such as $s$ and $o$ in this and following languages are formal parameters as introduced in table 3.3.

$\mathrm{LConstrSub}$ is a CPL instantiation for specifying $\mathrm{ConstrSub}$ where:

set ::= $\mathrm{setua(u)} \mid$ value

atomic ::= $\mathrm{atomicua(u)} \mid$ value

$\mathrm{setua} \in \{\mathrm{ua} \mid \mathrm{ua} \in \mathrm{UA} \wedge \mathrm{attType(ua)} = \mathrm{set}\}$

$\mathrm{atomicua} \in \{\mathrm{ua} \mid \mathrm{ua} \in \mathrm{UA} \wedge \mathrm{attType(ua)} = \mathrm{atomic}\}$

$\mathrm{value} \in \{\mathrm{val} \mid (\mathrm{sa}, \mathrm{val}) \in \mathrm{saset} \wedge \mathrm{sa} \in \mathrm{SA}\}$

In this case in the constraint function for subject attributes, only the attributes of the user who wants to create the subject and the proposed values for subject attributes are allowed.

$\mathrm{LConstrObj}$ is a CPL instantiation for specifying $\mathrm{ConstrObj}$ where:

set ::= $\mathrm{setsa(s)} \mid$ value

atomic ::= $\mathrm{atomicsa(s)} \mid$ value

$\mathrm{setsa} \in \{\mathrm{sa} \mid \mathrm{sa} \in \mathrm{SA} \wedge \mathrm{attType(sa)} = \mathrm{set}\}$

$\mathrm{atomicsa} \in \{\mathrm{sa} \mid \mathrm{sa} \in \mathrm{SA} \wedge \mathrm{attType(sa)} = \mathrm{atomic}\}$

$\mathrm{value} \in \{\mathrm{val} \mid (\mathrm{oa}, \mathrm{val}) \in \mathrm{oaset} \wedge \mathrm{oa} \in \mathrm{OA}\}$

Here we use subject attributes instead of user attributes.

$\mathrm{LConstrObjMod}$, used to specify $\mathrm{ConstrObjMod}$, is a instance of CPL where the symbols set and atomic are defined as follows:

set ::= $\mathrm{setsa(s)} \mid \mathrm{setoa(o)} \mid$ value

atomic ::= $\mathrm{atomicsa(s)} \mid \mathrm{atomicoa(o)} \mid$ value

**Table 3.5**: *Functional Specification*

| Functions | Conditions | Updates |
|---|---|---|
| **Administrative functions:** Creation and maintenance of user and their attributes.<br>UASET is a set containing name and value pairs for each user attribute.<br>uaset = $\bigcup_{\forall ua \in UA}$ OneElement(UASET$_{ua}$) where<br>$\text{UASET}_{ua} = \begin{cases} \{ua\} \times \text{SCOPE}_{ua} \text{ if } \text{attType}(ua) = \text{atomic} \\ \{ua\} \times 2^{\text{SCOPE}_{ua}} \text{ if } \text{attType}(ua) = \text{set} \end{cases}$ | | |
| **AddUser**<br>($u$:NAME,$uaset$:UASET) | $u \notin$U | U′=U$\cup\{u\}$<br>**forall** (ua,$va$)$\in uaset$ **do**<br>ua($u$)=$va$ |
| **DeleteUser**($u$:NAME)<br>/*delete all u's subjects*/ | $u \in$U | S′=S$\setminus\{s$\|SubCreator($s$)=$u\}$<br>U′=U$\setminus\{u\}$ |
| **ModifyUserAtt**<br>($u$:NAME,$uaset$:UASET)<br>/*delete all u's subjects*/ | $u \in$U | **forall** (ua,$va$)$\in uaset$ **do**<br>ua($u$)=$va$<br>S′=S$\setminus\{s$\|SubCreator($s$)=$u\}$ |
| **System functions:** User level operations. | | |
| **CreateSubject**<br>($u$: NAME,$s$:NAME,$saset$:SASET) | $u \in$U $\wedge$ $s \notin$S$\wedge$<br>ConstrSub($u, s, saset$) | S′=S$\cup\{s\}$;SubCreator(s)=u<br>**forall** (sa,$va$)$\in saset$ **do**<br>sa($s$)=$va$ |
| **DeleteSubject**<br>($u$:NAME,$s$:NAME) | $s \in$S $\wedge$ $u \in$U $\wedge$<br>SubCreator(s)=u | S′=S$\setminus\{s\}$ |
| **ModifySubjectAtt**<br>($u$:NAME,$s$:NAME,$saset$:SASET) | $s \in$S $\wedge$ $u \in$U $\wedge$<br>SubCreator($s$)=$u$ $\wedge$<br>ConstrSub($u, s, saset$) | **forall** (sa,$va$)$\in saset$ **do**<br>sa($s$)=$va$ |
| **CreateObject**<br>($s$:NAME,$o$:NAME,$oaset$:OASET) | $s \in$S $\wedge$ $o \notin$O $\wedge$<br>ConstrObj($s, o, oaset$) | O′=O$\cup\{o\}$<br>**forall** (oa,$va$)$\in$oaset **do**<br>oa($o$)=$va$ |
| **ModifyObjectAtt**<br>($s$:NAME,$o$:NAME,$oaset$:OASET) | $s \in$S $\wedge$ $o \in$O $\wedge$<br>ConstrObjMod($s, o, oaset$) | **forall** (oa,$va$)$\in$oaset **do**<br>oa($o$)=$va$ |
| $\forall\, p \in$ **P. Authorization$_p$; ConstrSub; ConstrObj; ConstrObjMod** | /*Left to be specified by security architects*/ | |

Note that this language allows one to compare proposed new attribute values with current attribute values of an object unlike LConstrObj, where the newly created object has no current attribute values as such.

**Functional Specifications.** The ABAC$_\alpha$ functional specification, as shown in Table 3.5, outlines the semantics of various functions that are required for creation and maintenance of the ABAC$_\alpha$ model components. The first column lists all the function names as well as required parameters. The second column represents the conditions which need to be satisfied before the updates, which are listed in the third column, can be executed. $NAME$ refers to set of all names

for various entities in the system.

The first kind of functions are administrative in nature which are designed to be invoked only by security administrators. We do not specify the authorization conditions for administrative functions which are outside the scope of $\text{ABAC}_\alpha$. They mainly deal with user and user attribute management and will be discussed in chapter 4. One important issue with user management is that the subjects created by the user are affected because of the user attributes' change or deletion of the user. We discuss the various options as follows.

- All subjects created by the user are forced to be terminated whenever user attributes are modified or the user is deleted. This is required in security critical applications where the modification to user attributes reflect urgent administrative response to the user's future permissions in the system. Keeping all subjects of the user would be dangerous because all permissions are actually not authorized any more while the existence of these subjects grants the permission anyhow.

- A relaxed response is that all subjects are kept if the user is deleted. At this time, the user who does not exist in the system lost control over the subjects and will not be able to terminate them. They are kept until the system starts a garbage collection-like process where they are terminated by the system process. They can be resumed if the user is added again and the attributes constraints is satisfied. If the user's attributes are modified, the subjects are still under control of the user. While at the same time, the permission sets of the subjects are restricted by the new sets of permissions of the user.

- An alternative of the above response is that the active subjects of the user are kept in the system till the completion of the task (or for certain length of time after the change) if the user is not deleted, i.e., only the attributes are changed. Otherwise, they are forced to be destroyed if the user is deleted.

We understand that the above options are not a complete list and it's up to the system to choose an appropriate response. The second kind of functions are system functions which can be invoked

**Table 3.6**: DAC (Owner-controlled Access Control Lists) Configuration

| |
|---|
| **Basic sets and functions** |
| UA={}, SA={}, OA={$reader$, $writer$, $createdby$} |
| $P$={$read$, $write$} |
| SCOPE$_{reader}$ = SCOPE$_{writer}$ = SCOPE$_{createdby}$=U |
| attType($reader$)=attType($writer$)=set |
| attType($createdby$)=atomic |
| Thus, $reader$ : O $\rightarrow 2^U$, $writer$ : O $\rightarrow 2^U$, $createdby$ : O $\rightarrow$ U |
| The function SubCreator is defined in Table 2. |
| **Configuration points** |
| **1. Authorization policy** |
| $\mathrm{Authorization}_{read}(s\ o) \equiv$SubCreator($s$)$\in reader(o)$ |
| $\mathrm{Authorization}_{write}(s, o) \equiv$SubCreator($s$)$\in writer(o)$ |
| **2. Constraint for subject attribute is not required** |
| Note that SubCreator is implicitly captured in function CreateSubject in table 3.2. |
| $\mathrm{ConstrSub}(u, s, \{\}) \equiv$ return true. |
| **3. Constraint for object attribute at creation time** |
| $\mathrm{ConstrObj}(s, o, \{(reader,\mathrm{val}_1), (writer,\mathrm{val}_2), (createdby,\mathrm{val}_3)\}) \equiv \mathrm{val}_3$=SubCreator($s$) |
| **4. Constraint for object attribute at modification time** |
| $\mathrm{ConstrObjMod}(s, o, \{(reader,\mathrm{val}_1), (writer,\mathrm{val}_2), (createdby,\mathrm{val}_3)\}) \equiv createdby(o)$=SubCreator($s$) $\wedge$ $\mathrm{val}_3 = createdby(o)$ |

by subjects and users. By default, the first function parameter is the invoker of each function. For example, CreateSubject is invoked by user $u$ and ModifyObjectAtt is invoked by subject $s$. The third kind of functions are authorization policies and subject and object attribute constraint functions which are left to be configured by security architects.

### 3.2.4 Configurations for Classical Models

In this section, we show the capability of ABAC$_\alpha$ in configuring DAC, MAC and RBAC. For this illustration, we set P = $\{read, write\}$.

- **DAC (Table 3.6)**. Each object is associated with the same number of set-valued attributes as the number of permissions and there is a one-to-one semantic mapping between them. An object attribute returns the list of users that hold the permission indicated by the object attribute name. Object attribute $createdby$ is set to be the owner of this object.

**Table 3.7**: MAC Configuration

| |
|---|
| **Basic sets and functions** |
| UA={$uclearance$}, SA={$sclearance$}, OA={$sensitivity$} |
| P={$read, write$} |
| SCOPE$_{uclearance}$ = SCOPE$_{sclearance}$ = SCOPE$_{sensitivity}$ = $L$ |
| $L$ is a lattice defined by the system. |
| attType($uclearance$)=attType($sclearance$)=attType($sensitivity$)= atomic |
| Thus, $uclearance$ : U $\rightarrow L$, $sclearance$ : S $\rightarrow L$, $sensitivity$ : O $\rightarrow L$. |
| **Configuration points** |
| **1. Authorization policies** |
| $\mathrm{Authorization}_{read}(s, o) \equiv sensitivity(o) \leq sclearance(s)$ |
| Liberal Star: $\mathrm{Authorization}_{write}(s, o) \equiv sclearance(s) \leq sensitivity(o)$ |
| Strict Star: $\mathrm{Authorization}_{write}(s, o) \equiv sclearance(s) = sensitivity(o)$ |
| **2.** $\mathrm{ConstrSub}(u, s, \{(sclearance, \text{value})\}) \equiv \text{value} \leq uclearance(u)$ |
| **3.** $\mathrm{ConstrObj}(s, o, \{(sensitivity, \text{value})\}) \equiv sclearance(s) \leq \text{value}$ |
| **4.** $\mathrm{ConstrObjMod}(s, o, \{(sensitivity, \text{value})\})$ returns false. |

- **MAC (Table 3.7)**. Each user is associated with an atomic-valued attribute $uclearance$. Each subject is also associated with an atomic-valued attribute $sclearance$. Each object is associated with an atomic-valued attribute $sensitivity$. Similar to MAC, the user and subject attributes represent their clearance in the system. The $sensitivity$ attribute of the object represents the object's classification in MAC. The three attributes share the same range which is represented by a system maintained lattice L.

- **RBAC (Table 3.8)**. Each user and subject is associated with set-valued attributes $urole$ and $srole$ respectively. Each object is associated with the same number of set-valued attributes as the number of permissions and there is a one-to-one semantic mapping between them. Each attribute returns the roles that are assigned the corresponding permission on this specific object. For example, $rrole$ of object $obj$ returns the roles which are assigned the permission of reading $obj$. The ranges of all attributes are the same as that of a system defined set of role names $R$ which are unordered for $\mathrm{RBAC}_0$ and partially ordered for $\mathrm{RBAC}_1$. Note that subjects model sessions in RBAC.

**Table 3.8**: RBAC Configurations

| |
|---|
| $RBAC_0$ configuration |
| **Basic sets and functions** |
| UA={$urole$}, SA={$srole$}, OA={$rrole, wrole$} |
| P={$read, write$} |
| $SCOPE_{urole}$= $SCOPE_{srole}$= $SCOPE_{rrole}$ = $SCOPE_{wrole}$=*R* |
| *R* is a set of atomic roles defined by the system. |
| attType($urole$)=attType($srole$)=attType($rrole$)=attType($wrole$)=set |
| Thus, $urole : U \rightarrow 2^R$, $srole : S \rightarrow 2^R$, $rrole : O \rightarrow 2^R$, $wrole : O \rightarrow 2^R$ |
| **Configuration points** |
| **1. Authorization policy** |
| $\text{Authorization}_{read}(s, o) \equiv \exists r \in srole(s).r \in rrole(o)$ |
| $\text{Authorization}_{write}(s, o) \equiv \exists r \in srole(s).r \in wrole(o)$ (same as above) |
| **2.** $\text{ConstrSub}(u, s, \{(srole, val_1)\}) \equiv val_1 \subseteq urole(u)$ |
| **3.** $\text{ConstrObj}(s, o, \{(rrole, val_1), (wrole, val_2)\})$ returns false. |
| **4.** $\text{ConstrObjMod}(s, o, \{(rrole, val_1), (wrole, val_2)\})$ returns false. |

| |
|---|
| $RBAC_1$ configuration |
| **Basic sets and functions** |
| The basic sets and functions are the same as $RBAC_0$ except: |
| *R* is a partially ordered set defined by the system. |
| **Configuration points** |
| **1. Authorization policy** |
| $\text{Authorization}_{read}(s, o) \equiv \exists r_1 \in srole(s).\ \exists r_2 \in rrole(o).r_2 \leq r_1$ |
| $\text{Authorization}_{write}(s, o) \equiv \exists r_1 \in srole(s).\ \exists r_2 \in wrole(o).r_2 \leq r_1$ (same as above) |
| **2.** $\text{ConstrSub}(u, s, \{(srole, val_1)\}) \equiv \forall r_1 \in val_1.\exists r_2 \in urole(u).r_1 \leq r_2$ |
| **3.** $\text{ConstrObj}(s, o, \{(rrole, val_1), (wrole, val_2)\})$ returns false. |
| **4.** $\text{ConstrObjMod}(s, o, \{(rrole, val_1), (wrole, val_2)\})$ returns false. |

### 3.2.5 Formal Proof of Equivalence

In this section, we provide formal proof for the equivalence of the original model and its $ABAC_\alpha$ configuration. We only show the formal proof for $RBAC_0$ here. The proofs for other models can be similarly developed. To show equivalence, we adopt the state-matching reduction method [139] which was developed to formally compare the expressive power of various access control models. Before we formally present the proof, we define the following schemes.

**Scheme $RBAC_0$.**

Although RBAC has been analyzed in [139], the focus there is on state changes caused by

user-role assignment and thus the administrative model for $RBAC_0$. In this dissertation, we model the changes caused by user activating and deactivating their roles in a session. Thus, the scheme for $RBAC_0$ is formally defined as follows.

- State $\Gamma$. They are summarized as the following well-known sets of $RBAC_0$: U (users), R (roles), URA (user-role assignment), S (sessions), SRA (session-role assignment), USA (user-session assignment), PRA (permission-role assignment), OP (operations) and O (objects).

- State transition $\Psi$. The state can only change by users creating and deleting sessions, and activating and deactivating roles in a session. The precondition for activating roles is that those roles are assigned to the user, i.e., there exists appropriate user-role assignment. Formally, the possible operations, their preconditions and their effects are defined as follows.

  $CreateSession(u, s)$

  Precondition: None

  Effect: $S' = S \cup \{s\},\ USA' = USA \cup \{(u, s)\}$

  $DeleteSession(u, s)$

  Precondition: $(u, s) \in USA$

  Effect: $\forall (s, r) \in SRA, SRA' = SRA \backslash \{(s, r)\},\ S' = S \backslash \{s\},$
  $USA' = USA \backslash \{(u, s)\}$

  $ActivateRole(u, s, r)$

  Precondition: $(u, s) \in USA \wedge (u, r) \in URA$

  Effect: $SRA' = SRA \cup \{(s, r)\}$

  $DeactivateRole(u, s, r)$

  Precondition: $(u, s) \in USA \wedge (s, r) \in SRA$

  Effect: $SRA' = SRA \backslash \{(s, r)\}$

- Query Q. On each state in this scheme, the query is whether their exists a session s with role r. We write a query as r. Other more general queries can be similarly considered.

- Entailment relation $\vdash$. It is defined as follows for a state $\gamma$ and query r.
  - $\gamma \vdash$ r = true iff $\exists$s.(s, r) $\in$ SRA

**Scheme ABAC-RBAC$_0$.** The ABAC instance for RBAC$_0$ is shown in table 3.8. It is expressed in the notation of [139] as follows.

- State $\Gamma$. They are the basic sets and functions shown in table 3.8.
- State transition $\Psi$. The only state changes are a user creating and deleting a subject or modifying subject attributes. Formally, these are defined as follows.

  $CreateSub(u, s, (srole, val))$

  | | |
  |---|---|
  | Precondition: | ConstrSub(u, s, (srole, val))= true |
  | Effect: | S′ = S $\cup$ {s}, srole(s) = val |

  $DeleteSub(u, s)$

  | | |
  |---|---|
  | Precondition: | SubCreator(s) =u |
  | Effect: | S′ = S $\setminus$ {s} |

  $ModifySubAttr(u, s, (srole, val))$

  | | |
  |---|---|
  | Precondition: | ConstrSub(u, s, (srole, val))= true |
  | Effect: | srole(s) = val |

- Query Q. On each state in this scheme, the query is whether their exists a session s with r $\in$ srole(s). We continue to write a query as r.
- Entailment relation $\vdash$. It is defined as:
  - $\gamma \vdash$ r = true if r$\in$srole(s)
  - $\gamma \vdash$ r = false if r$\notin$srole(s)

**Theorem 1.** *There exists a state-matching reduction from RBAC$_0$ to ABAC-RBAC$_0$.*

*Proof.* By construction. We present the mappings $ReduceState$ in algorithm 1 and $ReduceTransition$ in algorithm 2 to map the states and transition rules in RBAC$_0$ to those in ABAC-RBAC$_0$. The

**Algorithm 1** $ReduceState$ from $RBAC_0$ state ABAC-$RBAC_0$ state

**Input:** $RBAC_0$ state
**Output:** ABAC-$RBAC_0$ state
Given any $RBAC_0$ state: $U_{RBAC_0}$, $S_{RBAC_0}$, $O_{RBAC_0}$, R, URA, SRA, USA, PRA.
We have the following sets for ABAC-$RBAC_0$:
$U_{ABAC-RBAC_0} = U_{RBAC_0}$, $S_{ABAC-RBAC_0} = S_{RBAC_0}$, $O_{ABAC-RBAC_0} = O_{RBAC_0}$
UA = {urole}, SA = {srole}, OA = {readrole, writerole}
For each (u, s) $\in$ USA, SubCreator(s) = u.
SCOPE$_{urole}$ = SCOPE$_{srole}$ = R.
$\mathrm{attType}$(urole) = $\mathrm{attType}$(srole) = set
For each subject s $\in S_{ABAC-RBAC_0}$, srole(s) = { r | (s, r) $\in$ SRA}
For each user u $\in U_{ABAC-RBAC_0}$, urole(u) = {r | (u, r) $\in$ URA}
For each obj $\in O_{ABAC-RBAC_0}$, readrole(obj) = { r | (r, (read, obj)) $\in$ PRA}
For each obj $\in O_{ABAC-RBAC_0}$, writerole(obj) = { r | (r, (write, obj)) $\in$ PRA}

**Algorithm 2** $ReduceTransition$ from $RBAC_0$ transition rule to ABAC-$RBAC_0$ transition rule

**Input:** Transition Rule in $RBAC_0$
**Output:** Transition rule in ABAC-$RBAC_0$
CreateSession(u, s) is mapped to CreateSubject(u, s, {(srole, $\emptyset$)})
ActivateRole(u, s, r) is mapped to ModifySubAttr(u, s, {(srole, $val$)}) where the proposed value $val$ is srole(s) $\cup$ {r} and the constraints is: val $\in$ urole(u) $\wedge$ SubCreator(s) = u.
DeactivateRole(u, s, r) is mapped to ModifySubAttr(u, s, {(srole, $val$)}) where the proposed value $val$ is srole(s) $\setminus$ {r} and the constraints is: SubCreator(s) = u.
DeleteSession(u, s) is mapped to DeleteSubject(u,s) and the constraints is: SubCreator(s) = u.

query in $RBAC_0$ : (s, r) $\in$ SRA is mapped to a corresponding query: r $\in$ srole(s) in ABAC-$RBAC_0$. We show that they satisfy the properties for a state-matching reduction.

Given any initial state in RBAC as $\gamma_{RBAC_0}$, the corresponding state in ABAC is $\gamma_{ABAC-RBAC_0}$.

- If there is no transition. Given any state $\gamma_{RBAC_0}$ and query $q_{RBAC_0}$ in $RBAC_0$ scheme, if $\gamma_{RBAC_0} \vdash q_{RBAC_0}$ = true, then MAP($\gamma_{RBAC_0}$) $\vdash$ MAP($q_{RBAC_0}$) = true, where MAP is the process of mapping state and query in $RBAC_0$ scheme to ABAC-$RBAC_0$ scheme. The reason is that for any subject s and role r, if (s, r) $\in$ SRA, then srole(s) = { r | (s, r) $\in$ SRA} and thus, r $\in$ srole(s) = true. Similarly, if $\gamma_{RBAC_0} \vdash q_{RBAC_0}$ = false, i.e., (s, r) $\notin$ SRA, then r $\notin$ srole(s). Thus, $\gamma_{RBAC_0} \vdash q_{RBAC_0} = \gamma_{ABAC-RBAC_0} \vdash q_{ABAC-RBAC_0}$.

- Assume that after k transitions ($0 \leq k \leq \infty$) from the initial state , the two states $\gamma'_{RBAC_0}$ and $\gamma'_{ABAC-RBAC_0}$ is equivalent which means that all queries $\gamma'_{RBAC_0} \vdash q_{RBAC_0} = \gamma'_{ABAC-RBAC_0} \vdash q_{ABAC-RBAC_0}$. We want to prove that at state k+1, the result for all queries are also equivalent. There are the following possible transitions and we consider them one by one:

– $CreateSession(u, s)$. After this transition, in $\gamma'_{RBAC_0}$, USA′ = USA ∪ {(u, s)}. Correspondingly, the operation in ABAC-RBAC$_0$ scheme is $CreateSubject(u, s, \{(srole, \emptyset)\})$. In $\gamma'_{ABAC-RBAC_0}$, S′ = S ∪ {s} and SubCreator(s) = $u$, where (u, s) ∈ USA. The only change in query set is that new queries (s, r) where r ∈ R could be evaluated against the new state. For any r ∈ R, in RBAC$_0$ , $\gamma_{RBAC_0} \vdash$ (s, r) = false and in ABAC-RBAC$_0$ scheme, $MAP(\gamma_{ABAC-RBAC_0}) \vdash$ (s, r) = false.

– $DeleteSession(u, s)$. After this transition, in $\gamma'_{RBAC_0}$, ∀ (s,r) ∈ SRA, SRA′= SRA \ {(s,r)}, and USA′ = USA \ {(u, s)}. Thus, all queries regarding session s will return false. Queries on other sessions will not change compared with the previous state. Correspondingly, the operation in ABAC-RBAC$_0$ scheme is $DeleteSubject(u, s\})$. In $\gamma'_{ABAC-RBAC_0}$, S′ = S \ {s}. This transition does not affect existing queries regarding other subjects.

– $AcivateRole(u, s, r)$. The corresponding transition in ABAC-RBAC$_0$ is $ModifySubject$($u$, $s$, {($srole$, $val$)}), where val = srole(s) ∪ {r}. There are two situations after this transition:

  ∗ The request is not authorized because the precondition is not valid, then the state does not change. The precondition in $ModifySubject$ is also false. Thus, $\gamma_{ABAC-RBAC_0}{}'$ is the same as $\gamma_{ABAC-RBAC_0}$. The result for all queries are also the same as previous state.

  ∗ In the next state, SRA′ = SRA ∪ {(s, r)}. The only change to all queries is that the query $\gamma'_{RBAC_0} \vdash$ (s,r ) = true. In ABAC-RBAC$_0$ scheme, srole(s) = val, where val = srole(s) ∪ {r}. Thus, the only change to all queries is that r ∈ srole(s) = true.

– $DeactivateRole(u, s, r)$. Similarly, there are two possibilities:

  ∗ The state does not change because the precondition is not satisfied. The results for all queries do not change compared with the state at transition k.

  ∗ In the next state, SRA′ = SRA \ {(s, r)}. The only change to all queries is that the

**Algorithm 3** $ReduceState$ from ABAC-RBAC$_0$ state to RBAC$_0$ state

**Input:** ABAC-RBAC$_0$ state
**Output:** RBAC$_0$ state
Given any ABAC-RBAC$_0$ state:
We have the following sets for RBAC$_0$:
$U_{RBAC_0} = U_{ABAC-RBAC_0}$, $S_{RABC_0} = S_{ABAC-RBAC_0}$, $O_{RBAC_0} = O_{ABAC-RBAC_0}$
$R_{RBAC_0}$ = SCOPE$_{urole}$
URA = {(u, r) | r ∈ urole(u)}
SRA = {(s, r) | r ∈ srole(s)}
USA = {(u, s) | SubCreator(s) = u}
PRA = {((read, obj), r) | r∈readrole(obj)} ∪ {((write, obj), r) | r∈writerole(obj)}

**Algorithm 4** $ReduceTransition$ from ABAC-RBAC$_0$ to RBAC$_0$ transition rule

**Input:** Transition rules in RBAC$_0$
**Output:** Transition rules in ABAC-RBAC$_0$
CreateSubject(u, s, {(srole, $\emptyset$)}) is mapped to CreateSession(u, s)
ModifySubAttr(u, s, {(srole, $val$)}) is mapped to a sequence of ActivateRole(u, s, r) and DeactivateRole(u, s, r) operations. For all roles r ∈ (val - srole), ActivateRole(u,s,r) and for all roles r ∈ (srole- val), DeactivateRole(u, s, r).
DeleteSubject(u, s) is mapped to DeleteSession(u, s).

query $\gamma'_{RBAC_0} \vdash$ (s, r) = false. In ABAC-RBAC$_0$ scheme, srole(s) = val, where val = srole(s) $\setminus$ {r}. Thus, the only change to all queries is that r ∈ srole(s) = false.

In summary, the mapping is a state matching reduction from RBAC$_0$ to ABAC-RBAC$_0$. □

**Theorem 2.** *There exists a state-matching reduction from ABAC-RBAC$_0$ to RBAC$_0$.*

*Proof.* By construction. We present the mappings $ReduceState$ in algorithm 3 and $ReduceTransition$ in in algorithm 4 from ABAC-RBAC$_0$ scheme's state and transition rule to RBAC$_0$ scheme's state and transition rule. Similarly, the query in ABAC-RBAC$_0$ scheme r is mapped to query r in RBAC$_0$ scheme.

We show that the reduction shown above is a state matching reduction.

- No transition. Similar as in the proof for theorem 1, given any state in ABAC-RBAC$_0$, there exists a corresponding state in RBAC$_0$ such that all queries are with the same result.
- We assume that after k transition, the states $\gamma_{ABAC-RBAC_0}$ and $\gamma_{RBAC_0}$ are equivalent. We want to prove that after the next transition, the states are also equivalent. There are the following possible transitions and we consider them one by one.

- $CreateSubejct(u, s)$. Similar as the proof in theorem 1.
- $DeleteSubejct(u, s)$. Similar as the proof in theorem 1.
- $MidifySubject(u, s, \{(srole, val)\})$. There are two possibilities after this transition:
  - The request is not authorized and the state stays the same. The proof thus is the same as those in theorem 1.
  - The request is authorized, then srole(s) = val in $\gamma'_{ABAC-RBAC_0}$. After the corresponding transition in $RBAC_0$ scheme, the role of session s is equal to ((val $\setminus$ (srole(s) - val)) $\cup$ (val - srole(s)) (the current role excludes the deactivated role and plus the newly activated roles) which is val. Thus, in the next state, all queries are with the same result.

In summary, the mapping is a state matching reduction. □

**Theorem 3.** *The $RBAC_0$ model and ABAC instance of $RBAC_0$ (ABAC-$RBAC_0$) are equivalent in expressive power.*

*Proof.* The proof is based on theorem 1, theorem 2 and definition 9 in [139]. □

## 3.3 ABAC$_\beta$ Model

In this section, we develop the ABAC$_\beta$ model for the purpose of unifying numerous extensions proposed for the RBAC96 model [119]. We first discuss the scope of RBAC extensions that are covered by ABAC$_\beta$ model and then summarize the required features of ABAC$_\beta$. Based on this analysis, we present the formal model and show configurations for selected RBAC extensions.

### 3.3.1 Scope of RBAC Models

Generally, extensions to the RBAC model itself falls in two categories: administrative model and operational model. We include extensions to the operational RBAC model only. More specifically, we start from the NIST RBAC core model and cover advanced features as well as extensions to the authorization process of the model. Examples are role hierarchy, dynamic separation of duty (DSD), role structure, users associated with additional attributes and the structure of permissions. We exclude extensions on administrative models regarding user-role and permission-role assignment. Examples are static separation of duty (SSD), constraints on user-role assignment [6, 43, 72, 87, 88, 95, 110, 134], and constraints on role-permission assignment [6, 27, 100]. Another example is role-based trust management [95] which uses trust credentials to assign users to roles in different organizations. The central idea to this model is to assign roles to users. It does not discuss how role is used for authorization.

In addition, some extensions require features such as users modifying attributes of subjects created by other users, mutable attributes, continuous enforcement and obligations in authorization process. For example, [26, 27, 33, 44, 81, 116, 145] require mutable attributes and continuous enforcement. Workflow enhanced RBAC [29, 84, 140] also requires mutable attributes. These features are already available in existing attribute-based models such as UCON [111]. Role delegation [18, 59, 150] requires subject attributes to be modified by users other than the creator of the subject. This needs changes to built-in rules of ABAC$_\alpha$. Although these features are important and interesting, we exclude them and focus on enhancing ABAC$_\alpha$.

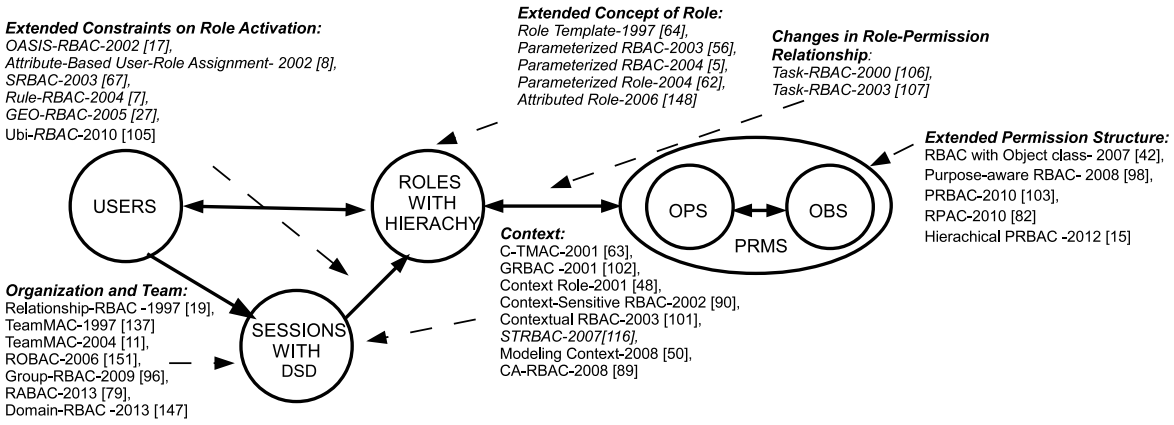



**Figure 3.2**: *RBAC Extesions Covered by* $\mathrm{ABAC}_\beta$

### 3.3.2 Brief Overview of Covered RBAC Extensions

To further analyze the required features for $\mathrm{ABAC}_\beta$, we categorize RBAC extensions that are covered in $\mathrm{ABAC}_\beta$ in figure 3.2. It shows the original RBAC model with advanced features as well as different groups of extensions marked in association to each related component as indicated by dashed arrows. For the extension to each component, we will see that they require the same set of new features for $\mathrm{ABAC}_\beta$. Note that some models do belong to multiple categories. For example, spatial and temporal RBAC [116] require extensions to not only context but also role activation process because the context information will be integrated into role activation rules. Privacy aware RBAC [103] belongs to context and extended permission structure categories. Here we discuss them only in one category, as shown in figure 3.2, and summarize the required features for each model at the end of this section.

We first consider the advanced features of core RBAC. These include hierarchical role and dynamic separation of duty (DSD) [57]. Since we model role as an attribute of user and subject, role hierarchy can be expressed using comparison between attribute values. DSD specifies the policy on whether a set of roles could be activated at the same time in a session. As we model session as subject in $\mathrm{ABAC}_\beta$, it is straight forward to express DSD constraints using subject attribute constraints policy where the conflict roles are represented using constant sets in the policy.

**Extended Constraints on Role Activation**

This type of extension puts constraints on role activation process. Examples are the order in which roles are activated, contextual factors (e.g., location, time) and prerequisite roles. In OASIS-RBAC [17], role activation is constrained by predefined activation rules. Activation rules are specified based on active roles, appointment certificate and environmental constraints. Appointment certificates are credentials which are assigned by other valid users (we skip the details of appointment certificate and assume that each user is associated with a set of appointment certificates). This model is concerned with subject attributes. It is straight forward to configure role activation rules using subject attribute constraints. Appointment certificate can be viewed as one of the user attributes. An example activation rule could be $r_1 \vdash r_2$ meaning that users can only activate$r_2$ in a session if role $r_1$ is already activated in that same session. Note that this model requires subject attributes constraint policy to be different at creation time and modification time because the policy for activating a role in a new session could be different from that at modification time. This is the first additional feature to ABAC$_\alpha$ model where subject attributes constraint policy is the same at subject creation and modification time. In addition, contextual attribute is required to capture environmental constriants. This is a different kind of attributes as they are not associated with user, subject or object. Instead, they are associated with the system and thus managed by the system, contrasted with user attributes being administered by administrators (discussed in chapter 4), subject attributes being managed by users, and object attributes being managed by subjects. Examples are risk level, time and information of the server. This is the second extension to the ABAC$_\alpha$ model.

Ubi-RBAC [105] includes context information in session. The model introduced the concept of spaces which are a partially ordered set. Users can activate certain roles in each space. If the user does not specify the roles when creating a space, the session inherits roles from a space which is the child of the current space and is created by the same user. This requires a new feature from ABAC$_\alpha$ which is that subject attributes are also constrained by attributes of subjects created by

the same user compared with subject attributes being constrained only by attributes of the creating users. This feature presents connections between the subjects created by the same user. This is the third kind of extension to ABAC$_\alpha$ model. In addition, in order to capture the child of any spaces in the set, a fourth new feature needs to be supported. This is called meta-attributes. This kind of attribute is defined as attributes of attributes and it reflects the properties of other attributes. Here the child of each space is represented as an attributes of the space.

[7, 8] provide models to specify policies which automatically assign users to roles based on their attributes. This feature can be configured as subject attributes constraints as the mapping between user attributes and roles can be captured in subject attribute constraints policy. [27, 67] put geographical constraints on role activation. Roles are only allowed to be activated at certain locations. The attributes can be captured by modeling location as a user and subject attribute.

Finally, in order to specify the above policies, the policy specification language for each configuration point should be extended. For example, in ABAC$_\alpha$, the authorization policy only allows attributes of the involved subject and object. However, in order to configure authorization policies for the above models, contextual attributes, meta-attributes and constant values are also allowed to be used in the policy specification. This is the fifth extension to the ABAC$_\alpha$ model.

In summary, this category requires five extensions to ABAC$_\alpha$ model and those extensions will also be required by the following RBAC extensions.

**Extended Concept of Role**

The idea of role structure extension is that roles are also associated with a set of parameters [5, 56, 62]. Permissions are then parameterized and associated with roles. The actual set of permissions associated with the parameterized role are determined by the actual role attributes values which are assigned explicitly when the roles are assigned to users. For example, student(department) represents a parameterized role. The department parameter is used in an example parameterized permission: "read any document whose major is the same as the role parameter department". If Alice is assigned with student($Business$) role, then Alice obtains the permissions of reading

documents from $Business$ department. Similar extensions are role template [64] and attributed role [148]. To configure this kind of extension, an intuitive method is to treat role parameters as user, subject and object attributes. Parameterized permissions are configured in authorization policy. When roles are assigned to users, their corresponding attributes are also assigned with specific values.

**Changes in Role-Permission Relationship**

Task and Role Based Access Control (Task-RBAC) [106,107] is a model where tasks are associated with a set of permissions (the same as the permissions in RBAC) and then associated with roles. Users are made members of roles and thus obtain the permissions. To configure these models, the connection between role and task are captured by a meta-attribute of role which represents the tasks that are associated with the role. Tasks are further associated with permissions (i.e., operation and object pair) the same way role is associated with permissions in $\text{ABAC}_\alpha$ instance of RBAC (shown in table 3.8)).

**Organization and Team**

This type of model extends RBAC for collaboration and consolidation. The general idea is to integrate organizations, groups, teams and so on, into RBAC. In role and organization based access control (ROBAC) [151], users are assigned to organization and role pair. Objects are assigned to certain types and organizations. In addition, permission in this model are defined as $(op, otyoe)$ where $op$ is an operator (the same as that in RBAC) and $otype$ is an object types. Access decisions are made based on both organizations and roles. Thus, the connection between role and object type is captured by meta-attributes, i.e., attributes of object type which returns the list of roles which can access the type of objects. Similar work include domain role based access control [147] and TeaM based Access Control (TMAC) [11, 137]. Group based RBAC [96] proposes two level user-role assignment. User can be assigned to roles either through direct assignment (same as user-role assignment in RBAC) or through groups. Groups are associated with roles and users

can activate those corresponding roles if made members of the groups. In role centric attribute based access control (RABAC) [79], the filtering policy restricts the available permissions in each session by specifying policies based on user and object attributes. The policy can be expressed in authorization policies. [19] adds relationship between requester and resource owner to control access. We model the relationship as another user attribute. For example, doctor and patient relationship is expressed as an attribute called $attendantdoc$ for each patient user.

**Context**

In this kind of extension, context information is used in access decision. Examples are [50, 63, 90, 101]. [89] propose context aware RBAC (CA-RBAC). [48] proposes to associate context information with environmental role. [116] adds spatial and temporal factors to each component of RBAC. Contextual attributes are required to configure the above models. Generalized RBAC (GRBAC) [102] extends the traditional RBAC by incorporating subject roles, object roles and environment roles. Subject roles are like traditional RBAC roles. Object roles abstract the various properties of objects, such as object type (e.g., text, JPEG, executable) or sensitivity level (e.g., classified, top secret) into categories. Environment roles capture environmental information, such as time of day or system load so it can be used to mediate access control. The policy is specified in the format of ($srole$, $orole$, $erole$, $op$), which represents that a subject with role $srole$ is authorized to perform operation $op$ on a resource with object role $orole$ under environment role $erole$. To configure this model, object roles are treated in the same way as subject roles and represented as an object attribute. In this way, all polices specified in [102] can be configured in authorization policy.

**Extended Permission Structure**

In RBAC, permissions are operation and object pairs $(op, obj)$ where $op$ represents an operation and $obj$ represents an object. Extension to the structure of permission requires additional information besides operations and objects. [103] proposed the core privacy aware RBAC (PRBAC)

where privacy sensitive data permission (PDP) are associated with roles. PDP is defined as a tuple $(dp, pu, c, o)$, where $dp$ represents operations on objects (same as $(op, obj)$ in RBAC), $pu$ represents purpose picked from a predefined finite set, $c$ represents contextual conditions specified using a language provided in the model, and $o$ represents a subset of predefined finite set of obligations. In this model, we cover PRBAC excluding its obligations model since that require mutable attributes (which has been previously developed in UCON [111] and can be incorporated in subsequent extensions of ABAC$_\beta$). Similarly, [98] proposes purpose-aware RBAC. [15] proposes hierarchical PRBAC. [82] proposes role involved purpose based access control RPAC. In these models, purpose is modeled as user and subjects attributes. [42] classifies objects into different types and permissions are defined as operations on different types of objects. Object classes can be modeled as an object attribute and the permissions can be configured using meta-attribute for role. This meta-attribute represents the types of objects that the role can access.

### 3.3.3 Summary of Required Features

In this section, we summarize and show some details on the required features of ABAC$_\beta$ to cover all RBAC models mentioned in section 3.3.2. There are generally five extensions described as follows.

- ***Extension 1: Context attributes.*** A new and separate component called "context" is defined to manage a finite set of contextual attributes. Examples of such attributes are $time$, $location$, and $machine\_type$. Context is a separate component because this type of attribute is not associated with any specific users, subjects or objects. Rather, these attributes are global and are managed by the system. For example, the current time of the system is returned from a system API (e.g., sys.getTime()) and is changed automatically. To accommodate $time$ as an attribute with finite scope we can reduce clock time to a suitable granularity such as *normal-business-hours* (say 9am to 5pm), *extended-business-hours* (say 7am to 7pm) and *non-business-hours* (7pm to 7am).

- ***Extension 2: Subject attributes constraint policy at creation time is different from modification time.*** Subject attributes constraints policy at creation and modification time is differently specified and it is not limited to be constrained by the creating user attributes. For example, in OASIS-RBAC, the constraint policy for creating a subject is different from that for modifying subjects. Thus, different policies need to be specified for these constraints.

- ***Extension 3: Subject attribute constrained by attributes of subjects created by the same user.*** In $\text{ABAC}_\alpha$, subject attributes are only constrained by the creating users. To cover Ubi-RBAC model, attributes of one subject can also be constrained by attributes of other subjects created by the same user. The roles in one space can be inherited by a senior space belonging to the same user.

- ***Extension 4: Enhanced policy specification Language.*** In $\text{ABAC}_\alpha$, policy specification only contains attributes of the involved entities including users, subjects and objects and it contains conjunctions and disjunctions of value comparison. It should be extended to cover meta-attributes, contextual attributes as well as constant values.

- ***Extension 5: Meta-attributes.*** The concept of meta-attributes deals with the fact that attributes can not only be associated with users, subjects and objects but also with other sets. For example, users can be associated with roles and role may have properties that are represented by attributes of roles. In task and role based access control, roles are associated with tasks and tasks are associated with permissions. Thus, the relationship between role and task can be captured by attribute of role, that is a meta-attribute.

**Table 3.9**: *Additional Features Required for* $\mathrm{ABAC}_\beta$ *to Cover RBAC and Extended Models*

| Model | Context Attribute | Subject creation and modification time constraints | Subject constrained by other subjects of the same user | Policy language | Meta-Attribute |
|---|---|---|---|---|---|
| **Extended Constraints on Role Activation** | | | | | |
| RBAC-DSD-96 [122] | NO | NO | NO | YES | NO |
| Attribute-RBAC-2002 [8] | YES | NO | NO | YES | NO |
| OASIS-RBAC-2002 [17] | YES | YES | NO | YES | YES |
| SRBAC-2003 [67] | YES | NO | NO | YES | NO |
| Rule-RBAC-2004 [7] | YES | YES | NO | YES | NO |
| GEO-RBAC-2005 [27] | YES | NO | NO | YES | NO |
| Ubi-RBAC-2010 [105] | YES | YES | YES | YES | YES |
| **Extended Concept of Role** | | | | | |
| Role Template-1997 [64] | YES | NO | NO | YES | NO |
| Parameterized-RBAC-2003 [56] | NO | NO | NO | YES | NO |
| Parameterized-RBAC-2004 [62] | NO | NO | NO | YES | NO |
| Parameterized-Role-2004 [5] | NO | NO | NO | YES | NO |
| Attributed Role-2006 [148] | NO | NO | NO | YES | NO |
| **Change in Role-Permission Relationship** | | | | | |
| Task-RBAC-2000 [106] | YES | NO | NO | YES | YES |
| Task-RBAC-2003 [107] | YES | NO | NO | YES | YES |
| **Organization and Team** | | | | | |
| Relationship-RBAC-1999 [19] | NO | NO | NO | YES | NO |
| TeamMAC-1997 [137] | NO | NO | NO | YES | NO |
| TeamMAC-2004 [11] | NO | NO | NO | YES | NO |
| ROBAC-2006 [151] | NO | NO | NO | YES | YES |
| Group-RBAC-2009 [96] | NO | NO | NO | YES | YES |
| RABAC-2013 [79] | NO | NO | NO | YES | NO |
| Domain-RBAC-2013 [147] | NO | NO | NO | YES | NO |
| **Context** | | | | | |
| C-TMAC-2001 [63] | YES | NO | NO | YES | YES |
| GRBAC-2001 [102] | YES | NO | NO | YES | NO |
| Context-Role-2001 [48] | YES | NO | NO | YES | NO |
| Context Sensitive RBAC 2002 [90] | YES | NO | NO | YES | NO |
| Contextual RBAC-2003 [101] | YES | NO | NO | YES | NO |
| STRBAC-2007 [116] | YES | NO | NO | YES | YES |
| Modeling Context-2008 [50] | YES | NO | NO | YES | NO |
| CA-RBAC-2008 [89] | YES | NO | NO | YES | NO |
| **Extended Permission Structure** | | | | | |
| RBAC with Object Class-2007 [42] | NO | NO | NO | YES | YES |
| Purpose Aware RBAC-2008 [98] | YES | NO | NO | YES | NO |
| PRBAC-2010 [103] | YES | NO | NO | YES | NO |
| RPAC-2010 [82] | YES | NO | NO | YES | NO |
| Hierarchical PRBAC-2012 [15] | YES | NO | NO | YES | NO |

Based on our analysis in the last section, we summarize the required features for each RBAC-related model that is covered by $\mathrm{ABAC}_\beta$ in table 3.9. This table specifies the required features of $\mathrm{ABAC}_\beta$ to configure each RBAC-related model. Based on the above analysis, we build an $\mathrm{ABAC}_\beta$

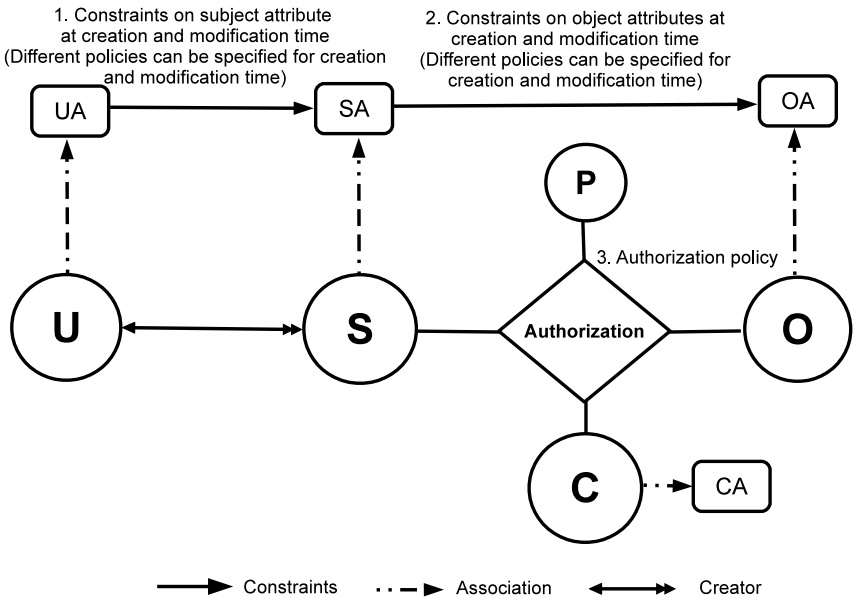



**Figure 3.3**: $\mathrm{ABAC}_\beta$ *Model Structure*

model with all five extensions.

### 3.3.4 Model Overview

As we extend the model based on $\mathrm{ABAC}_\alpha$, the basic components are the same as in $\mathrm{ABAC}_\alpha$. The model structure is shown in figure 3.3. We only elaborate on the extended components as follows.

- **Context and context attributes**. Context is a single entity. The properties of this component is represented by a finite set of contextual attributes which are system dependent. This kind of attribute is different from user, subject or object attributes in that they cannot be modified by users or subjects. Rather, they are updated automatically by system or system administrators. Examples are $time$, $CPU\ usage$, $free\ disk\ space$ and $risk\ level$. Due to the nature of these attributes, policies may require continuous enforcement and response to attribute changes. For example, a policy may require that access is only authorized during *normal-business-hours*. An application may require all authorized permissions to be revoked automatically outside *normal-business-hours*, or applications may allow access that are granted within the time range to be authorized until discretionarily released or suspended

by users. In this model, we support such attributes in policy specification but do not deal with the detail of policy enforcement of such attributes.

- **Meta-attributes**. We define attributes as functions which contain domain and scope. Domain is the set of values this function can take and scope represents the values that the function can return. In $\text{ABAC}_\alpha$, attributes are only associated with user, subjects and objects. Meta-attribute defines attributes whose domain comes from the scope of existing attributes. Meta-attribute carries the similar properties as other attributes such as they are either set or atomic-valued. Example of such attributes are the risk level of general attributes, the task which are associated with roles and so on.

- **Configuration points**. We add a new configuration point and it is subject attributes constraint policy at modification time. In this policy, subject attributes can be constrained by attributes of the creating user and attributes of subjects created by the same user.

### 3.3.5 Formal Model

The **basic sets** and **functions** in $\text{ABAC}_\beta$ are the same as in $\text{ABAC}_\alpha$ model except that context and contextual attributes and meta-attributes are introduced. We only introduce the additional concepts here. The symbol $\text{c}$ represents the context entity and CA represents a finite set of context attributes associated with the context $\text{c}$. MA represents a finite sets of meta-attributes. The domain of these attributes must be from the scope of existing attributes.

**Policy Configuration Points.** We define five policy configuration points as shown in table 3.11. The first configuration point (item 1 in table 3.11) is authorization policies. The security architect specifies one authorization policy for each operation. The authorization function returns **true** or **false** based on attributes of the involved subject, object, context and meta attributes. The second configuration point (item 2 in table 3.11) is constraints for subject attributes at creation time. The third configuration point (item 3 in table 3.11) is constraints for subject attributes at modification time. The fourth configuration point (item 4 in table 3.11) is constraints for object

**Table 3.10**: *Basic Sets and Functions of* $\mathrm{ABAC}_\beta$

U, S and O represent finite sets of *existing* users, subjects and objects respectively. Symbol c represents the context entity.

UA, SA OA, CA and MA represent finite sets of user, subject, object, context attribute and meta-attributes functions respectively. (Henceforth referred to as simply attributes.)

P represents a finite set of permissions.

For each $att$ in UA $\cup$ SA $\cup$ OA $\cup$ CA$\cup$ MA, $\mathrm{SCOPE}_{att}$ is a constant finite set of *atomic* values.

attType: UA $\cup$ SA $\cup$ OA $\cup$ CA $\cup$ MA$\rightarrow$ {set, atomic}, specifies attributes as set or atomic valued.

SubCreator: S $\rightarrow$ U. For each subject SubCreator gives its creator.

Each attribute in MA, its domain should come from the range of existing attributes. That is, for each attribute attr $\in$ MA, the domain of attribute, domain(attr) $\in$ {SCOPE$_{att}$ | att $\in$ UA $\cup$ SA $\cup$ OA $\cup$ CA $\cup$ MA}.

$$\forall ua \in \mathrm{UA}.\, ua : \mathrm{U} \rightarrow \begin{cases} \mathrm{SCOPE}_{ua} & \text{if attType}(ua) = \text{atomic} \\ 2^{\mathrm{SCOPE}_{ua}} & \text{if attType}(ua) = \text{set} \end{cases}$$

$$\forall sa \in \mathrm{SA}.\, sa : \mathrm{S} \rightarrow \begin{cases} \mathrm{SCOPE}_{sa} & \text{if attType}(sa) = \text{atomic} \\ 2^{\mathrm{SCOPE}_{sa}} & \text{if attType}(sa) = \text{set} \end{cases}$$

$$\forall oa \in \mathrm{OA}.\, oa : \mathrm{O} \rightarrow \begin{cases} \mathrm{SCOPE}_{oa} & \text{if attType}(oa) = \text{atomic} \\ 2^{\mathrm{SCOPE}_{oa}} & \text{if attType}(oa) = \text{set} \end{cases}$$

$$\forall ca \in \mathrm{CA}.\, ca : \mathrm{c} \rightarrow \begin{cases} \mathrm{SCOPE}_{ca} & \text{if attType}(ca) = \text{atomic} \\ 2^{\mathrm{SCOPE}_{ca}} & \text{if attType}(ca) = \text{set} \end{cases}$$

$$\forall ma \in \mathrm{MA}.\, ma : domain(ma) \rightarrow \begin{cases} \mathrm{SCOPE}_{ma} & \text{if attType}(ma) = \text{atomic} \\ 2^{\mathrm{SCOPE}_{ma}} & \text{if attType}(ma) = \text{set} \end{cases}$$

attributes at creation time. The fifth configuration point (item 5 in table 3.11) is constraints for object attributes at modification time. Constraints are boolean expressions specified using policy specification languages presented below.

**Policy Configuration Languages.** Each policy configuration point is specified using a specific language. Although some of the RBAC extensions covered in $\mathrm{ABAC}_\beta$ provide concrete syntax for specifying their policies, we provide a level of abstraction and unification such that all models can be covered. The languages specify what information (e.g., user attribute, subject attribute) is allowed to be used in the five configuration points. Since all specification languages share the same logical structure while differing only in the values they can use for comparison, we extend the common policy language CPL (table 3.4) defined for $\mathrm{ABAC}_\alpha$ to define a similar template CPL for $\mathrm{ABAC}_\beta$. The enhanced CPL is defined in table 3.12.

Language LAuthorization is used to specify the authorization policy. It is a CPL instance

**Table 3.11**: *Policy Configuration Points of* $\mathrm{ABAC}_\beta$

| |
|---|
| **1. Authorization policies.**<br>For each p ∈ P, $\mathrm{Authorization}_p$(s:S, o:O) returns true or false.<br>Language LAuthorization is used to define the above functions (one per permission), where s and o are formal parameters. c is also usable as a global context for this policy. |
| **2. Subject attribute constraints at creation time.**<br>Language LConstrSub is used to specify ConstrSub(u:U, s:S, saset:SASET), where u, s and saset are formal parameters. c is also usable as a global context for this policy. The variable saset represents proposed attribute name and value pairs for each subject attribute (defined in table 3.3). |
| **3. Subject attribute constraints at modification time.**<br>Language LConstrSubMod is used to specify ConstrSubMod(u:U,s:S,saset:SASET), where s, o and saset are formal parameters. c is also usable as a global context for this policy. |
| **4. Object attribute assignment constraints at creation time.**<br>Language LConstrObj is used to specify ConstrObj(s:S, o:O, oaset:OASET), where s, o and oaset are formal parameters. The variable oaset represents proposed attribute name and value pairs for each object attribute (defined in table 3.3). Note that the global context c can not be used here. |
| **5. Object attribute modification constraints.**<br>Language LConstrObjMod is used to specify ConstrObjMod(s:S, o:O, oaset:OASET), where s, o and oaset are formal parameters. Note that the global context c can not be used here. |

**Table 3.12**: *Definition of Enhanced* CPL

| |
|---|
| $\varphi ::= \varphi \wedge \varphi \mid \varphi \vee \varphi \mid (\varphi) \mid \neg\varphi \mid \exists x \in set.\varphi \mid \forall x \in set.\ \varphi \mid \varphi \rightarrow \varphi \mid expr$ |
| $expr ::= eatomic\ copa\ eatomic \mid eatomic\ copm\ eset \mid eset\ cops\ eset$ |
| $eatomic ::= eatomic\ atomic\_op\ eatomic \mid atomic \mid \mid eset \mid$ |
| $eset ::= eset\ set\_op\ eset \mid set$ |
| $atomic\_op ::= + \mid - \mid \times \mid \setminus$ |
| $set\_op ::= \cap \mid \cup \mid - \mid \Delta$ |
| $copa ::= \leq \mid \geq \mid < \mid > \mid = \mid != $ |
| $cops ::= \subset \mid \subseteq \mid \nsubseteq$ |
| $copm ::= \in \mid \notin$ |

where the symbols $set$ and $atomic$ are specified as follows:

$$set ::= \mathrm{setsa(s)} \mid \mathrm{setoa(o)} \mid \mathrm{constantSet} \mid \mathrm{setca(c)} \mid \mathrm{setma}(entity)$$

$$atomic ::= \mathrm{atomicsa(s)} \mid \mathrm{atomicoa(o)} \mid \mathrm{constantAtomic} \mid \mathrm{atomicca(c)} \mid \mathrm{atomicma}(entity)$$

$$\mathrm{setca} ::= \{\mathrm{ca} \mid \mathrm{ca} \in \mathrm{CA} \wedge \mathrm{attType(ca)} = \mathrm{set}\}$$

$$\mathrm{setma} ::= \{\mathrm{ma} \mid \mathrm{ma} \in \mathrm{MA} \wedge \mathrm{attType(ma)} = \mathrm{set}\}$$

$$\mathrm{atomicca} ::= \{\mathrm{ca} \mid \mathrm{ca} \in \mathrm{CA} \wedge \mathrm{attType(ca)} = \mathrm{atomic}\}$$

$$\mathrm{atomicma} ::= \{\mathrm{ma} \mid \mathrm{ma} \in \mathrm{MA} \wedge \mathrm{attType(ma)} = \mathrm{atomic}\}$$

LAuthorization allows one to specify policies based on the value of involved subject, object, context attributes and meta attributes. $entity$ represents the domain of meta-attributes. Parameters such as $s$ and $o$ in this language are formal parameters as introduced in item 1 in table 3.11. We adopt the same definition of setsa, setoa, atomicsa and atomicoa from section 3.2. constantSet and constantAtomic are constant sets and atomic values picked by the policy composer. Example constantSet is a set of roles $\{r_1, r_2, \ldots, r_n\}$ or organization $\{org_1, org_2, \ldots, org_n\}$ and example constantAtomic is a clearance level $top\ secret$ or a logic time $first\_Day\_of\_Month$. An example authorization policy for context aware RBAC [48, 63, 89, 90, 101] is "manager is authorized to edit salary data only on the first Monday of each month during business hour in the company". The rule can be specified using the following policy:

$\text{Authorization}_{modify}$(s, salary) $\equiv$ manager $\in$ $role$(s) $\wedge$ $location$(s) = company $\wedge$ $is\_Business\_Hour$(c) = true $\wedge$ $is\_First\_Monday$(c) = true

In this policy, a user is associated with $location$ and $role$ attributes, the context attribute $is\_Business\_Hour$ defines whether the current time is in business hour and attribute $is\_First\_Monday$ represents whether it is the first Monday of the month.

Language LConstrSub is used to specify subject attribute constraints policy ConstrSub at creation time. It is a CPL instance where the symbols $set$ and $atomic$ are specified as follows:

set ::= setua(u) | value | constantSet | setca(c) | setsa(sub) | setma(entity)

atomic ::= atomicua(u) | value | constantAtomic | atomicca(c) | atomicsa(sub) | atomicma($entity$)

sub :: = SameCreator(u)

value $\in$ {val | (sa, val) $\in$ saset $\wedge$ sa $\in$ SA}

LConstrSub allows one to specify policies based on the attribute value for involved user, other subjects created by the same user, context attributes, meta-attributes and the suggested attributes values for the subject to be created. We adopt setua, atomicua from the definition from section 3.2. As the policy can refer to attributes of subjects created by the same user, we define SameCreator(s) which returns the set of subjects created by the same user. That is, $\forall$ s $\in$ S, SameCreator(s) repre-

sents the set of subjects that are created by the same user as the creator of s, i.e., $\mathrm{SameCreator}$(s) = {sub | sub $\in$ S $\wedge$ $\mathrm{SubCreator}$(sub) = $\mathrm{SubCreator}$(s)}. where s is the formal parameter.

Language $\mathrm{LConstrSubMod}$ is used to specify subject attribute constraint policy $\mathrm{ConstrSubMod}$ at modification time. It is different from $\mathrm{LConstrSub}$ in that $\mathrm{LConstrSubMod}$ is allowed to refer to the current value of existing subjects. It is an instance of CPL where the symbols $set$ and $atomic$ are defined as follows:

set ::= setua(u) | value | $\mathrm{constantSet}$ | $\mathrm{setca}$(c) | $\mathrm{setsa}$(sub) | $\mathrm{setma}$(entity) | setsa(s)

atomic ::= atomicua(u) | value | $\mathrm{constantAtomic}$ | $\mathrm{atomicca}$ | $\mathrm{atomicsa}$(sub) | $\mathrm{atomicma}$(entity) |atomicsa(s)

Language $\mathrm{LConstrObj}$ is used to specify object attribute constrains policy $\mathrm{ConstrObj}$ at creation time. It is a CPL instance where the symbols $set$ and $atomic$ are specified as follows:

set ::= setsa(s) | value

atomic ::= atomicsa(s) | value

value $\in$ {val | (oa, val) $\in$ oaset $\wedge$ oa $\in$ OA}

Language $\mathrm{LConstrObj}$ allows one to specify policies using attributes of the involved subject and suggested attribute values for the object to be created. Note that different from the above languages, this policy does not allow context attributes nor meta-attributes. Language $\mathrm{LConstrObjMod}$ is used to specify object attribute constrains policy $\mathrm{ConstrObjMod}$ at modification time. It is a CPL instance where the symbols $set$ and $atomic$ are specified as follows:

set ::= setsa(s) | value | setoa(o)

atomic ::= atomicsa(s) | value | atomicoa(o)

value $\in$ {val | (oa, val) $\in$ oaset $\wedge$ oa $\in$ OA}

$\mathrm{LConstrObjMod}$ is different from $\mathrm{LConstrObj}$ in that $\mathrm{LConstrObjMod}$ can refer to attributes of the objects whose attributes are to be modified.

**Table 3.13**: $\mathrm{ABAC}_\beta$ *Configuration for OASIS-RBAC Without Role Membership Rule*

| |
|---|
| 1. **Basic sets and functions** |
| UA={uap}, SA={srole, sap}, OA={readrole, writerole}, CA = {Mapped from environment constraints}, MA = { } |
| P={read, write} |
| $\mathrm{attType}$(srole) = $\mathrm{attType}$(readrole) = $\mathrm{attType}$(writerole)= $\mathrm{attType}$(uap) = $\mathrm{attType}$(sap) = $\mathrm{attType}$(condition) = set |
| $\mathrm{SCOPE}_{srole}$= $\mathrm{SCOPE}_{readrole}$ = $\mathrm{SCOPE}_{writerole}$ = ROLE, where ROLE is a finite set of roles. |
| $\mathrm{SCOPE}_{uap}$ = $\mathrm{SCOPE}_{sap}$ = AP, where AP is finite set of appointment certificates in the system. |
| 2. **Configuration Points** |
| (1)Authorization policy |
| $\mathrm{Authorization}_{read}$(s, o)$\equiv\exists$ $r_1 \in$ srole(s).$\exists r_2 \in$ readrole(o). $r_1 >= r_2$. |
| $\mathrm{Authorization}_{write}$(s, o)$\equiv\exists$ $r_1 \in$ srole(s).$\exists r_2 \in$ writerole(o). $r_1 >= r_2$. |
| (2)Subject attribute constraints at creation time |
| $\mathrm{ConstrSub}$(u, s, {(srole, v), (sap, vap)}) $\equiv$ Policy mapped from initial role activation rule (*No prerequisite role is included in this policy*). |
| (3)Subject attribute constraints at modification time. |
| $\mathrm{ConstrSubMod}$(u, s , {(srole, v), (sap, vap)})$\equiv$ Policy mapped from role activation rule (*Prerequisite role is allowed to be included in this policy*). |
| (4)Object attributes constraints at creation time. |
| $\mathrm{ConstrObj}$(u ,s , {(readrole, read), (writerole, write)})$\equiv$ false |
| (5) Object attributes constraints at modification time. |
| $\mathrm{ConstrObjMod}$(u ,s , {(readrole, read), (writerole, write)})$\equiv$ false |

### 3.3.6 Configuration Examples

We now show the configuration for a sample of RBAC extensions in $\mathrm{ABAC}_\beta$. For each category of RBAC extensions shown in figure 3.3.2, they share the similar set of extensions in $\mathrm{ABAC}_\beta$. Thus, we pick one model from each category and show the detailed configuration.

#### OASIS-RBAC Without Role Membership Rule

Central to the OASIS model is the idea of credential-based role activation. Role can be activated in a certain session when the user satisfies conditions. There is no changes on other components in RBAC compared with the original RBAC model. The condition can be specified based on the following factors associated with the users:

- Appointment certificate. This is assigned by other users specifying that the user can have special permissions.

**Table 3.14**: $\mathrm{ABAC}_\beta$ *Configuration for ROBAC*

**1.Basic Sets and functions**
UA = {uorgrole}. SA = {sorgrole}, OA = {org, type}, MA = {readtype, writetype}
P = {read}
attType(uorgrole) = attType(sorgrole) = attType(readtype) = attType(writetype) = $\mathrm{set}$, attType(org) = attType(type) = $\mathrm{atomic}$
SCOPE$_{uorgrole}$ = SCOPE$_{sorgrole}$ = ORG$\times$ROLE, SCOPE$_{org}$ = ORG, SCOPE$_{type}$ = TYPE
ORG is finite set of organizations, TYPE is finite set of object types, ROLE is finite set of roles.
readtype: ROLE $\rightarrow 2^{TYPE}$, writetype: ROLE $\rightarrow 2^{TYPE}$
**2. Configuration Points**
(1) Authorization policy
Authorization$_{read}$(s, o)$\equiv \exists$(org, role) $\in$ sorgrole(s).(org $>=$ org(o) $\wedge$ type(o) $\in$ readtype(role)).
Authorization$_{write}$(s, o)$\equiv \exists$(org, role) $\in$ sorgrole(s).(org $>=$ org(o) $\wedge$ type(o) $\in$ writetype(role)).
(2) Subject attribute constraints at creation time
$\mathrm{ConstrSub}$(u, s, {(sorgrole, val)})$\equiv \forall$ (org, role)$\in$ val.$\exists$(org$_1$, role$_1$) $\in$ uorgrole(u). org $<=$ org$_1$ $\wedge$ role $<=$ role$_1$.
(3) Subject attribute constraints at modification time
$\mathrm{ConstrSubMod}$(u, s, {(sorgrole,val)})$\equiv \forall$ (org, role)$\in$ val.$\exists$(org$_1$, role$_1$) $\in$ uorgrole(u). org $<=$ org$_1$ $\wedge$ role $<=$ role$_1$.
(4) Object attribute constraints at creation time.
$\mathrm{ConstrObj}$(s, o, {(org, o), (type, t)})$\equiv$ false
(5) Object attribute constraints at modification time.
$\mathrm{ConstrObjMod}$(s, o, {(org, o), (type, t)})$\equiv$ false

- Prerequisite role. The roles that the user has activated in a session.
- Environmental constraints. This represents the context information such as time, risk level and so on.

Role activation rule is specified using conjunctions of comparisons of the above factors and return boolean result stating the role activation is allowed or not. Role activation rules are different for session creation and modification time because at creation time, the policy does not contain prerequisite roles and at modification time, the prerequisite roles can be included.

In order to configure this model, the first two factors are modeled as user attribute. The environment condition is modeled as context attributes. As the OASIS-RBAC model does not specify a specific set of environment attributes, we assume there exists a set of contextual attributes. The (initial) role activation rule is configured in subject attribute constraints at creation and nidification time. Formal configuration is shown in table 3.13. We describe the mapping process from role activation rule to subject attribute constraints policy as follows:

**Table 3.15**: $\mathrm{ABAC}_\beta$ *Configuration for Role Template*

| |
|---|
| **1. Basic Sets and Functions** |
| UA={$\mathrm{ua}_1$, $\mathrm{ua}_2$, ... $\mathrm{ua}_n$, urole}, SA={$\mathrm{sa}_1$, $\mathrm{sa}_2$, ... $\mathrm{sa}_n$, srole}, OA={$\mathrm{oa}_1$, $\mathrm{oa}_2$, ... $\mathrm{oa}_n$, readrole}, MA = {} |
| The type and range of other attributes depend on specific applications. |
| P={read} |
| attType(urole) = attType(srole) = set |
| $\mathrm{SCOPE}_{urole}$= $\mathrm{SCOPE}_{srole}$ =ROLES, ROLES is finite set of roles in the systems. |
| **2. Configuration Points** |
| (1) A. Authorization policy for flat role template. |
| $\mathrm{Authorization}_{read}$(s, o)$\equiv$srole(s)$\cap$readrole(o)$\neq \emptyset \wedge$ (comparison between subject attribute and object attribute) |
| B. Authorization policy for nested role template. |
| $\mathrm{Authorization}_{read}$(s, o)$\equiv \exists$ role1$\in$ srole(s).$\exists$role2$\in$readrole(o). role1>=role2$\wedge$ (comparison between subject attribute and object attribute) |
| (2) Subject attribute constraints at creation time. |
| $\mathrm{ConstrSub}$(u, s, {(srole, val), ($\mathrm{sa}_1$, $\mathrm{v}_1$) ... ($\mathrm{sa}_n$, $\mathrm{v}_n$)})$\equiv$ val $\subseteq$ urole(u) $\wedge$ $\mathrm{v}_1$ = $\mathrm{ua}_1$(u) $\wedge$ $\mathrm{v}_n$ = $\mathrm{ua}_n$(u) |
| (3) Subject attribute constraints at modification time. |
| $\mathrm{ConstrSubMod}$(u, s, {(srole, val),(other attributes)})$\equiv$ val $\subseteq$ urole(u) $\wedge$ $\mathrm{v}_1$ = $\mathrm{ua}_1$(u) $\wedge$ $\mathrm{v}_n$ = $\mathrm{ua}_n$(u) |
| (4)Object attribute constraints at creation time. |
| $\mathrm{ConstrObj}$(s, o, {(readrole, role), ($\mathrm{oa}_1$, $\mathrm{v}_1$) ... ($\mathrm{oa}_n$, $\mathrm{v}_n$) })$\equiv$ false |
| (5)Object attribute constraints at modification time. |
| $\mathrm{ConstrObjMod}$(s, o, {(readrole, role), ($\mathrm{oa}_1$, $\mathrm{v}_1$) ... ($\mathrm{oa}_n$, $\mathrm{v}_n$) })$\equiv$ false |

- For each x from R, we map it into x $\in$ R. (No need to map this factor in initial role activating rules.)
- For each x from appointment certificate, we map it to x $\in$ $uap$(u).
- For each x from environment constraints, we map it to x $\in$ $ca$(c) where ca is mapping to a specific contextual attributes.

They are connected using conjunction. For different rules defined for the same role, we connect them using disjunction in the subject attribute constraint policy.

### Role and Organization based Access Control

In ROBAC, users are assigned with role and organization pair and objects are associated with organizations. In order for a user to access an object, there are two preconditions: (1) there exists a role which can access the object and (2) the role is activated in the same organization as the objects. The only change to the configuration of RBAC model is that users and objects are now

**Table 3.16**: $\mathrm{ABAC}_{\beta}$ *Configuration for Spatial and Temporal RBAC*

**1.Basic Sets and functions**
UA = {urole, ulocation}, SA = {srole, slocation}, OA = {readrole, writerole}, CA = {time}, MA = {acTime, acLoc}
P={read}
attType(urole) = attType(srole) = attType(acTime) = attType(acLoc) = set
attType(ulocation) = attType(slocation) = attType(time) = atomic
$\mathrm{SCOPE}_{urole}$ = $\mathrm{SCOPE}_{srole}$ = ROLE, a finite set of roles
$\mathrm{SCOPE}_{time}$ = TIME, a finite set of logical times.
$\mathrm{SCOPE}_{ulocation}$ = $\mathrm{SCOPE}_{slocation}$ = LOC, a set of locations.
acTime: ROLE$\rightarrow$TIME.
acLoc: ROLE $\rightarrow$ LOC.
**2. Configuration Points**
(1) Authorization policy
$\mathrm{Authorization}_{read}$(s, o)$\equiv$ $\exists$(role, loc, time) $\in$ readrole(o).(role$\in$ srole(s)$\wedge$ slocation(s) = loc $\wedge$ time(c) = time).
$\mathrm{Authorization}_{write}$(s, o)$\equiv$ $\exists$(role, loc, time) $\in$ writerole(o).(role$\in$ srole(s)$\wedge$ slocation(s) = loc $\wedge$ time(c) = time).
(2) Subject attribute constraints at creation time
$\mathrm{ConstrSub}$(u, s, {(srole,role), (slocation, proloc)})$\equiv$ proloc = ulocation(u) $\wedge$ $\forall$ r$\in$ srole.(time(c) $\in$ acTime(r) $\wedge$ proloc $\in$ acLoc(r))
(3) Subject attribute constraints at modification time
$\mathrm{ConstrSubMod}$(u, s, {(srole,role), (slocation, proloc)})$\equiv$ proloc = ulocation(u) $\wedge$ $\forall$ r$\in$ srole.(time(c) $\in$ acTime(r) $\wedge$ proloc $\in$ acLoc(r))
***Note:*** The constraints between user and subject location can be either the same or different.
(4) Object attribute constraints at creation time.
$\mathrm{ConstrObj}$(s, o, {(readrole, read), (writerole, write)})$\equiv$ false.
(5) Object attribute constraints at modification time.
$\mathrm{ConstrObjMod}$(s, o, {(readrole, read), (writerole, write)})$\equiv$ false.

associated with organization attributes which further is included in subject attribute constraints and authorization policy. The formal configuration is shown in table 3.14.

### Role Template-1997

The basic concept of role template is that roles are associated with parameters and parameterized permissions. Role parameters are modeled as user attributes and object attributes. The permission further is specified using authorization policy. As the model does not specify a specific set of role parameters, it is not possible to configure the detailed user attributes in this configuration. Instead, we show the equivalent scheme and leave the authorization policy to be configured by specific instance. We show the formal configuration in table 3.15.

**Table 3.17**: $\mathrm{ABAC}_\beta$ *Configuration for Task-RBAC 2003*

**1. Basic Sets and functions.**
UA={urole}, SA={srole}, OA={readtask, writetask}, MA ={roletask}
P={read, write}
SCOPE$_{readtask}$= SCOPE$_{writetask}$= SCOPE$_{roletask}$ =TASK
SCOPE$_{urole}$ = SCOPE$_{srole}$= ROLES
attType(urole)=attType(srole)=attType(readtask)=attType(readtask)= $\mathrm{attType}$(roletask) = set
roletask: ROLES $\rightarrow 2^{TASK}$
**2. Configuration Point.**
(1) Authorization policy.
Authorization$_{read}$(s, o)$\equiv \exists$ role $\in$ srole(s).roletask(role) $\cap$ readtask(o) ! $= \emptyset$
Authorization$_{write}$(s, o)$\equiv \exists$ role $\in$ srole(s).roletask(role) $\cap$ writetask(o) ! $= \emptyset$
(2)Subject attribute constraints at creation time.
$\mathrm{ConstrSub}$(u,s {(srole, val)})$\equiv$ val$\subset$ urole(u)
(3)Subject attribute constraints at modification time.
$\mathrm{ConstrSubMod}$(u,s {(srole, val)})$\equiv$ val$\subset$ urole(u)
(4)Object attribute constraints at creation time.
$\mathrm{ConstrObj}$(s, o, {(readrole, read), (writerole, write)})$\equiv$ false
(5)Object attribute constraints at modification time.
$\mathrm{ConstrObjMod}$(s, o, {(readrole, read), (writerole, write)})$\equiv$ false

### Spatial and Temporal RBAC

In this model, the only extension is that roles, permissions can be activated within certain location and time. Subject attribute constraints check whether the role can be activated. Authorization policy further checked whether permissions are allowed at certain time and location. To configure this model, users and subjects are associated with location attributes besides role. Roles carries meta-attributes which reflect the time and location to activate. For each object, it carries attributes which returns the role, time and location to access it. The formal configuration is shown in table 3.16.

### Task-RBAC-2003

This model adds an additional level of permission assignment. Permissions on object is associated with certain tasks which are further associated with roles. Other than that, this model is the same as the original RBAC model. In order to capture the relationship between task and role, a meta-attribute needs to be specified. This attribute returns the set of tasks which are associated with

**Table 3.18**: $\mathrm{ABAC}_\beta$ *Configuration for Ubi-RBAC*

**1. Basic Sets and functions.**
UA ={urolespace}, OA = {readrole, writerole}, SA = {srole, space}
CA = {mapped from contextual constraints}, MA = {child}
SCOPE$_{space}$ = SPACE, is a partially ordered set of space
$\mathrm{attType}$(urolespace) = $\mathrm{attType}$(readrole) = $\mathrm{attType}$(writerole) = $\mathrm{attType}$(srole) = set
SCOPE$_{readrole}$ = SCOPE$_{writerole}$ = SCOPE$_{srole}$ = ROLE, a finite set of roles.
SCOPE$_{urolespace}$= ROLE $\times$ SPACE
$\mathrm{attType}$(space) = $\mathrm{attType}$(child) = atomic
child: SPACE $\rightarrow$SPACE
**2. Configuration Point.**
(1) Authorization policy.
Authorization$_{read}$(s, o)$\equiv \exists$ (r, c) $\in$ readrole(o). r $\in$ srole(s) $\wedge$ *computation on c*.
Authorization$_{write}$(s,o)$\equiv \exists$ (r, c) $\in$ writerole(o). r $\in$ srole(s) $\wedge$ *computation on c*.
(2)Subject attribute constraints at creation time.
$\mathrm{ConstrSub}$(u,s {(srole, prorole),(space, prospace)})$\equiv$ ($\neg(\exists$ s $\in$ S.space(s) = prospace) $\wedge \forall$ r $\in$ prorole. ( ($\exists$ sub $\in$ SameCreator(s).(space(s) = child(prospace) $\wedge$ r $\in$ srole(s))) $\vee$ ($\exists$(role, valspace) $\in$ urolespace(u). (role=r $\wedge$ valspace = prospace)))
(3)Subject attribute constraints at modification time.
$\mathrm{ConstrSubMod}$(u, s, {(srole, prorole),(space, prospace)})$\equiv$ ($\neg(\exists$ s $\in$ S.space(s) = prospace) $\wedge \forall$ r $\in$ prorole. ( ($\exists$ sub $\in$ SameCreator(s). (space(s) = child(prospace) $\wedge$ r $\in$ srole(s))) $\vee$ ($\exists$(role, valspace) $\in$ urolespace(u). (role=r $\wedge$ valspace = prospace)))
(4)Object attribute constraints at creation time.
$\mathrm{ConstrObj}$(s, o, {(readrole, read), (writerole, write)})$\equiv$ false
(5)Object attribute constraints at modification time.
$\mathrm{ConstrObjMod}$(s, o, {(readrole, read), (writerole, write)})$\equiv$ false

each role. Each object carries the attribute returning the set of tasks that have certain permission on them. The formal configuration is shown in table 3.17.

### Ubi-RBAC-2010

The core extensions of this model from $\mathrm{RBAC}_1$ [122] are in two aspects:

- Each session carries space information. Space is a partially ordered set. In each space, users can only activate a subset of their assigned roles. If a user activate a session in a space without specifying the roles, this space, by default will inherit roles from the sessions activated in the nearest child space.

- Context information. In the structure of permission, contextual constraints are included.

Thus, we model space as an attribute of subjects and model contextual constraints in authorization policy. The formal configuration is shown in table 3.18.

### 3.3.7 Expressive Power Discussion

$\mathrm{ABAC}_\beta$ is designed to cover RBAC extensions in different directions and provide a uniform framework to combine those advantages in scope. We've shown that this model is capable to cover a wide variety of extensions to RBAC model. The benefit of using $\mathrm{ABAC}_\beta$ is not limited to expressing previously proposed RBAC extensions. Rather, it can be used to discover interesting extensions to RBAC that have not been proposed. In this section, we present an example RBAC extension in the format of $\mathrm{ABAC}_\beta$ configuration.

**RBAC with Automatic Role Enhancing (RBAC-ARE)**

In RBAC, permissions are associated with roles by system architects when deploying the system. The advantage of RBAC is based on the assumption that the set of roles and permissions do not change frequently. Problem arises when new permissions on newly created objects need to be associated with roles. The only way to authorize those permissions to users is to redesign roles and then assign roles to users. Whether new roles are created or permissions are associated with the existence roles, the complexity increases with the amount and dynamic nature of object creation.

In this section, we extend RBAC by allowing creation of new objects. And at the same time, we associate the permissions on those newly created objects with existing roles automatically. In this way, system architects do not need to redesign roles and RBAC scale in dynamic systems. For this purpose, we allow users to suggest values to be assigned to attributes of objects when they create new objects. As in $\mathrm{ABAC}_\alpha$ instance of RBAC, each object is associated with the same number of attributes as permissions. Each attribute is mapped to the permission and represents the set of roles which can have the permission on that object. For example, readrole is an object attribute and readrole($file$) represents the roles that can read this $file$. In RBAC configuration, those attribute values are set at model deployment time. In RBAC_ARE,

**Table 3.19**: $\mathrm{ABAC}_\beta$ Configuration for RBAC_ARE

**1. Basic sets and functions**

UA ={$urole$}, SA ={$srole$}, OA ={$rrole, wrole, owner$}, CA = { }, MA = { }

P ={$read, write$}

$\mathrm{SCOPE}_{urole}$ = $\mathrm{SCOPE}_{srole}$ = $\mathrm{SCOPE}_{rrole}$ = $\mathrm{SCOPE}_{wrole}$ = *R*, *R* is a set of atomic roles define by the system.

$\mathrm{SCOPE}_{owner}$ = U.

attType($urole$) = attType($srole$) = attType($rrole$) = attType($wrole$) = set.

attType($owner$) = atomic.

**2. Configuration Points**

(1). Authorization policy

$\mathrm{Authorization}_{read}(s, o) \equiv \exists \mathrm{r}_1 \in srole(s).\ \exists \mathrm{r}_2 \in rrole(o).\mathrm{r}_2 \leq \mathrm{r}_1$

$\mathrm{Authorization}_{write}(s, o) \equiv \exists \mathrm{r}_1 \in srole(s).\ \exists \mathrm{r}_2 \in wrole(o).\mathrm{r}_2 \leq \mathrm{r}_1$

(2). Subject attribute constraints at creation time

$\mathrm{ConstrSub}(u, s, \{(srole, \mathrm{val}_1)\}) \equiv \forall \mathrm{r}_1 \in \mathrm{val}_1.\exists \mathrm{r}_2 \in urole(u).\mathrm{r}_1 \leq \mathrm{r}_2$

(3). Subject attribute constraints at modification time

$\mathrm{ConstrSubMod}(u, s, \{(srole, \mathrm{val}_1)\}) \equiv \forall \mathrm{r}_1 \in \mathrm{val}_1.\exists \mathrm{r}_2 \in urole(u).\mathrm{r}_1 \leq \mathrm{r}_2$

(4). Object attribute constraints at creation time

$\mathrm{ConstrObj}(s, o, \{(\mathrm{rrole}, \mathrm{val}_1), (\mathrm{wrole}, \mathrm{val}_2), (\mathrm{owner}, \mathrm{name})\}) \equiv$ name = SubCreator(s) $\wedge$ *specified by system architects.*

(5). Object attribute constraints at modifcation time

$\mathrm{ConstrObjMod}(s, o, \{(\mathrm{rrole}, \mathrm{val}_1), (\mathrm{wrole}, \mathrm{val}_2), (\mathrm{owner}, \mathrm{name})\}) \equiv$ owner(o) = SubCreator(s) $\wedge$ name = owner(o) $\wedge$*specified by system architects.*

- When users create new objects, the attributes of the newly created object are assigned and modified by its creator. That is, the creator can specify the list of roles for each permission on the objects.
- Only the creator can modify the attributes of the objects.
- Different object attributes constraints result in $\mathrm{ABAC}_\beta$ instances with different features.

### $\mathrm{ABAC}_\beta$ Configuration for RBAC_ARE

In order to configure RBAC_ARE, one change needs to be made to the original $\mathrm{ABAC}_\beta$ configuration for RBAC [80]. Instead of returning false, the object attribute constraints is specified by system architects. This change can be applied to both $\mathrm{RBAC}_0$ and $\mathrm{RBAC}_1$ model. We shows the hierarchical RBAC configuration in table 3.19.

**Example Constraints**

We show several examples of interesting and practical object attributes constraints and illustrate them in this section. The examples are as follow:

1. $\mathrm{ConstrObj}(s, o, \{(\text{rrole}, \text{val}_1), (\text{wrole}, \text{val}_2)\}) \equiv$ name = SubCreator(s) $\wedge$ $\text{val}_1 = \{\text{manager}\} \wedge \text{val}_2 = \{\text{manager}\}$.
2. $\mathrm{ConstrObj}(s, o, \{(\text{rrole}, \text{val}_1), (\text{wrole}, \text{val}_2)\}) \equiv$ name = SubCreator(s) $\wedge$ $\text{val}_1 \subseteq$ srole(s) $\wedge$ $\text{val}_2 \subseteq$ srole(s).
3. $\mathrm{ConstrObj}(s, o, \{(\text{rrole}, \text{val}_1), (\text{wrole}, \text{val}_2)\}) \equiv$ name = SubCreator(s) $\wedge \forall$ r $\in \text{val}_1.\forall$ $\text{r}_1 \in$ srole(s). r$>\text{r}_1 \wedge \forall$ r $\in \text{val}_2.\forall$ $\text{r}_2 \in$ srole(s). r$<\text{r}_2$.

The first constraint allows users with "manager" role to automatically grow with permissions on newly created objects by all other users. This is useful in companies where managers need to review, audit and modify documents from the rest of employees. The second constraint allows the creator to grant permissions on the object to users with the same roles as the creator. This enables information sharing among users with the same roles. For example, multiple manager users may want to share their report with each other within only manager users. The third constraint specifies information flow in roles with hierarchy. For each newly created objects, only users with roles which are senior than the object creator can read it and users with roles which are junior than the object creator can write it.

## 3.4 Conclusion

In this chapter, we develop $\text{ABAC}_\alpha$ to establish the connections between classical models and ABAC. We define core components and configuration points of ABAC model. With the formal configuration of ABAC for classical models, we demonstrate the expressive power of $\text{ABAC}_\alpha$ model. In addition, we develop $\text{ABAC}_\beta$ based on the observation that RBAC has been dominant in access control in both industry and academia. This model provides richer configuration languages and covers a wide range of RBAC extensions. Formal configuration for sample RBAC extensions

in ABAC$_\beta$ is provided.

# Chapter 4: ROLE BASED USER ATTRIBUTE ADMINISTRATIVE MODEL AND POLICY ANALYSIS

## 4.1 Scope

The ABAC models proposed in the previous section are based on the assumption that all users are associated with user attributes and each attribute is assigned with certain values. An administration model is needed to manage user attribute assignment. It's crucial to regulate the assignment of user attributes as these attributes are further used in authorization policy and thus determine the permissions a user may obtain. The central contribution in this chapter is to study administrative issues of user attribute management in ABAC. For this purpose, we use the well-known administrative role based access control model (ARBAC97) which to our knowledge has not been previously applied in this domain. Our motivation for choosing ARBAC97 includes its ease of administration and sizable literature. Our principle finding is that ARBAC97, with proper generalization, is suitable in large measure to address user attribute assignment administration. In particular, we generalize the user role assignment model (URA) which is part of ARBAC97. Since role is just one type of user attribute, this generalization is straight forward yet efficient.

It is straight forward to extend this work to user attribute administration based on attributes where administrator also has attributes and their permissions to modify attributes is computed by their attributes and policies. However, in this dissertation, we mainly focus on GURA as it is sufficient to reflect the context of attributes. In addition, by proposing this initial model, further work which extends this model will inherit many properties researched here. For example, we conduct reachability analysis later in this chapter for the purpose of policy analysis based on GURA. Clearly, any extensions based on GURA will inherit the systematical study of complexities for GURA model.

## 4.2 User-Role Assignment Model

Administrative role based access control (ARBAC97) [118] is designed for user-role assignment, role-permission assignment and role hierarchy specification in RBAC [119]. In this chapter, we deal with user attribute assignment, and hence discuss the use of user-role assignment (URA) which is part of ARBAC97. The URA97 model is defined in two steps: granting a user membership in a role and revoking a user's membership. The goal of URA97 is to impose restrictions on which users can be added to a role by whom, as well as to clearly separate the ability to add and remove users from other operations on the role. The notion of prerequisite condition is a key part of URA97. All examples included in the following are according to figure 4.1 adapted from [118].

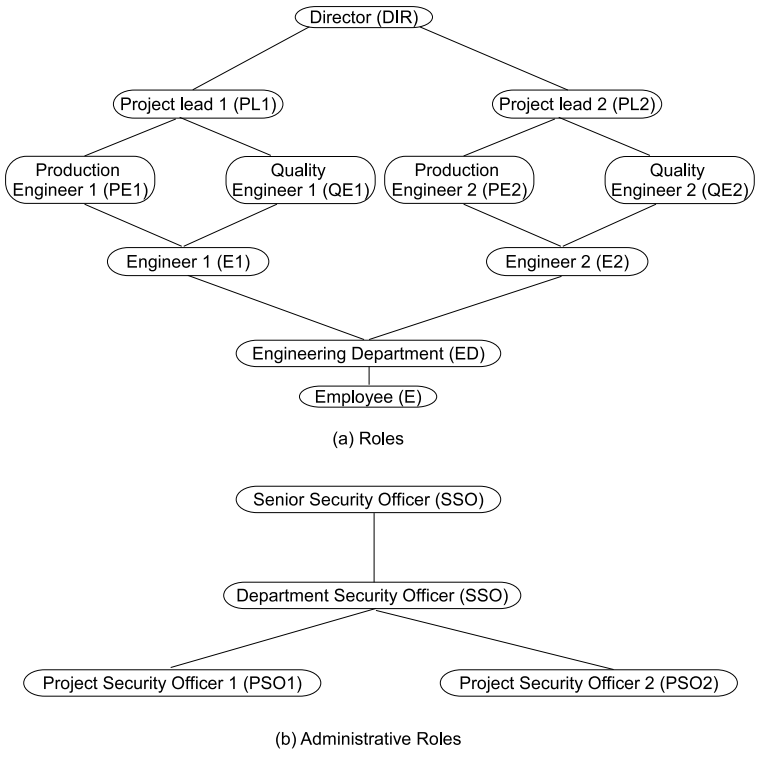



**Figure 4.1**: *Example Role and Administrative Role Hierarchies*

**Table 4.1**: *Role Range Notation*

| |
|---|
| [x,y]={r∈R \| r ≥ x ∧ y ≥ r} |
| [x,y)={r∈R \| r ≥ x ∧ y > r} |
| (x,y]={r∈R \| r > x ∧ y ≥ r} |
| (x,y)={r∈R \| r > x ∧ y > r} |

### 4.2.1 The URA97 Grant Model

User-role assignment is authorized by means of the following relation .

$$can\_assign \subseteq AR \times CR \times 2^R \tag{4.1}$$

The meaning of $can\_assign(x, y, \{a, b, c\})$ is that a member of the administrative role $x$ (or a member of an administrative role that is senior to $x$) can assign a user whose current membership, or nonmembership, in regular roles satisfies the prerequisite condition $y$ to be a member of regular roles $a$, $b$, or $c$. In relation (4.1), AR stands for specific administrative roles such as Project Security Officer 1 (PSO1). CR is a prerequisite condition which is a boolean expression using the usual $\vee$ and $\wedge$ operators on terms of the form x and $\bar{x}$ where x is a regular role. A prerequisite condition is evaluated for a user $u$ by interpreting x to be true if $(\exists x' \geq x)(u, x') \in$ URA and $\overline{x}$ to be true if $(\forall x' \geq x)(u, x') \notin$ URA, where URA is the user-role assignment relation of RBAC. For a given set of roles R, we let CR denote all possible prerequisite conditions that can be formed using the roles in R. $2^R$ represents the roles which can be assigned to the users who satisfy the prerequisite condition. Here the role sets are specified using the range notation given in table 4.1.

Examples are shown in table 4.2. The first tuple authorizes PSO1 role (and its seniors) to assign users with prerequisite role ED into roles {E1, QE1, PE1, PL1}. The second tuple authorizes PSO1 role to assign users with prerequisite condition ED $\wedge$ $\overline{\text{QE1}}$ to PE1. The third tuple authorizes PSO1 to assign users with prerequisite condition ED $\wedge$ $\overline{\text{PE1}}$ to QE1. The second and third together authorize PSO1 to put a user who is a member of ED into one but not both of PE1 and QE1.

**Table 4.2**: *can_assign with Prerequisite Roles*

| Admin. Role | Prereq. Condition | Role Range |
|---|---|---|
| PSO1 | ED | [PL1, E1] |
| PSO1 | ED $\wedge$ $\overline{\text{QE1}}$ | [PE1, PE1] |
| PSO1 | ED $\wedge$ $\overline{\text{PE1}}$ | [QE1, QE1] |

**Table 4.3**: *Examples of can_revoke*

| Aadmin. Role | Role Range |
|---|---|
| PSO1 | [E1, PL1) |
| PSO2 | [E2, PL2) |

### 4.2.2 The URA97 Revoke Model

In URA, the role assignment and revoke permissions are authorized separately. The URA97 model controls user-role revocation by means of the following relation.

$$can\_revoke \subseteq AR \times 2^R \tag{4.2}$$

In relation (4.2), the meaning of $can\_revoke(x, Y)$ is that a member of the administrative role $x$ (or a member of an administrative role senior to $x$) can revoke membership of a user from any regular role $y \in Y$. $Y$ is also specified using range notation. Examples are shown in table 4.3. The first tuple authorizes PSO1 to revoke membership from the roles {E1, PE1, QE1} (represented by the role range notation). Suppose Alice is member of PSO1 and Bob is member of PE1. Then Alice is authorized to remove Bob's membership of PE1. The second tuple authorizes PSO2 to remove membership from the roles {E2, PE2, QE2}.

## 4.3 Generalized User-Role Assignment Model (GURA)

The main purpose of the model is to provide policy to specify the permissions for each administrative role in managing user attributes. We first introduce several concepts and then present the formal model.

### 4.3.1 Preliminaries

**Administrative Requests.** In GURA, *administrative requests* are made by members of administrative roles to modify attributes of users in $\mathrm{U}$. A request takes effect only if it is authorized by an *administrative rule* introduced later in this section. An authorized request is called an *action*. In this model we are concerned with the collective power of the administrative roles. We therefore do not distinguish specific members, and simply ascribe a request to an administrative role rather than to one of its members. Management of membership in administrative roles is discussed in existing models such as [118]. Thus, we assume a finite set of administrative roles $\mathrm{AR}$ ( it may be flat or hierarchical).

Atomic-valued attributes are modified via an *assign* action which replaces the current value with a new value. For set-valued attributes, an *add* action is used to add a single atomic value to an existing attribute set, while a *delete* action is used to remove a specific atomic value from an existing set. More formally, for each $ar \in \mathrm{AR}$, $u \in \mathrm{U}$, $att \in \mathrm{UA}$ and $val \in \mathrm{SCOPE}_{att}$,

- assign$(ar,\ u,\ att,\ val)$ is a request made by (a member of administrative role) $ar$ to assign value $val$ to the atomic-valued attribute $att$ of user $u$. Suppose $\mathrm{AR}$ ={gameleader, manager} and Alice's attribute assignments are as in example 1. The request assign(manager, Alice, Dept, hardware) is made by (a member of administrative role) manager to assign Alice to hardware department. If this request is authorized, Alice's Dept attribute will change from software to hardware.

- add$(ar,\ u,\ att,\ val)$ is a request made by $ar$ to add value $val$ to the set-valued attribute $att$ of user $u$. For instance, add(manager, Alice, Proj, game) is a request made by manager to add Alice to the game project. If authorized, Proj(Alice) becomes {mobile, social, search, game}.

- delete$(ar,\ u,\ att,\ val)$ is a request made by $ar$ to delete the value $val$ from the set-valued attribute $att$ of user $u$. For instance, delete(manager, Alice, Proj, game) is a request by

manager to delete Alice from game project. If authorized, Proj(Alice) would revert back to {mobile, social, search}.

**Definition 1** (**Administrative Requests**). Administrative requests *are made by members of administrative roles to modify user attributes.* $Request = \text{ASSIGN} \cup \text{ADD} \cup \text{DELETE}$ *denotes the set of all possible administrative requests where,*

- $\text{ASSIGN} = \{\mathsf{assign}(ar,\ u,\ att,\ val) \mid ar \in \text{AR} \ \wedge\ u \in \text{U} \ \wedge\ att \in \text{UA} \ \wedge\ \text{attType}(att) = \text{atomic} \ \wedge\ val \in \text{SCOPE}_{att}\}$
- $\text{ADD} = \{\mathsf{add}(ar,\ u,\ att,\ val) \mid ar \in \text{AR} \ \wedge\ u \in \text{U} \ \wedge\ att \in \text{UA} \ \wedge\ \text{attType}(att) = \text{set} \wedge\ val \in \text{SCOPE}_{att}\}$
- $\text{DELETE} = \{\mathsf{delete}(ar,\ u,\ att,\ val) \mid ar \in \text{AR} \wedge u \in \text{U} \wedge att \in \text{UA} \wedge \text{attType}(att) = \text{set} \ \wedge\ val \in \text{SCOPE}_{att}\}$

**Administrative Rules.** An administrative request takes effect only if it is authorized by an *administrative rule*. Administrative rules specify the necessary preconditions for authorizing administrative requests. A precondition is a logical formula expressed over user attributes that evaluates to **true** or **false** (*e.g.*, Clr(Alice) $\geq$ classified $\wedge$ game $\in$ Proj(Alice)). The power of the administrative model lies in the expressive power of the preconditions, which can be specified by different grammars for defining preconditions. We will define the grammar for the two GURA models defined in section 4.3.2.

**Definition 2** (**Administrative Rules**). *An* administrative rule *specifies the authorization for administrative requests. Administrative rules are tuples in the following three relations where* C *represents a set of preconditions (specified by a formal grammar for each instance of a GURA model).*

*For each atomic-valued attribute* $aua \in \text{UA}$*,*

$$\mathsf{can\_assign}_{aua} \subseteq \text{AR} \times \text{C} \times \text{SCOPE}_{aua}$$

*The rule* $\langle ar,\ c,\ val\rangle \in \mathsf{can\_assign}_{aua}$ *authorizes the requests* $\mathsf{assign}(ar,\ u,\ aua,\ val)$ *if user* $u$ *satisfies precondition* $c$*.*

*For each set-valued attribute* $sua \in \mathrm{UA}$*,*

$$\mathsf{can\_add}_{sua} \subseteq \mathrm{AR} \times \mathrm{C} \times \mathrm{SCOPE}_{sua}$$

$$\mathsf{can\_delete}_{sua} \subseteq \mathrm{AR} \times \mathrm{C} \times \mathrm{SCOPE}_{sua}$$

*The rule* $\langle ar,\ c,\ val\rangle \in \mathsf{can\_add}_{sua}$ *authorizes the requests* $\mathsf{add}\ (ar,\ u,\ sua,\ val)$ *if user* $u$ *satisfies the precondition* $c$*, and the rule* $\langle ar,\ c,\ val\rangle \in \mathsf{can\_delete}_{sua}$ *similarly authorizes requests* $\mathsf{delete}(ar,\ u,\ sua,\ val)$ *if user* $u$ *satisfies precondition* $c$*.*

Administrative rules are specified using different relations. A can_assign relation is specified for each atomic-valued attribute. Similarly, can_add and can_delete relations are specified for each set-valued attribute. We use can_assign to denote the collection of $\mathsf{can\_assign}_{aua}$ relations for all atomic-valued attributes. That is,

$$\mathsf{can\_assign} = \langle \mathsf{can\_assign}_{\mathsf{att}_1}, \ldots, \mathsf{can\_assign}_{\mathsf{att}_m} \rangle,$$

where $\mathrm{att}_1$, $\mathrm{att}_2$, ..., $\mathrm{att}_m$ are atomic-valued attributes. Similarly, we use can_add and can_delete as follows:

$$\mathsf{can\_add} = \langle \mathsf{can\_add}_{\mathsf{att}'_1}, \ldots, \mathsf{can\_add}_{\mathsf{att}'_n} \rangle$$

$$\mathsf{can\_delete} = \langle \mathsf{can\_delete}_{\mathsf{att}'_1}, \ldots, \mathsf{can\_delete}_{\mathsf{att}'_n} \rangle$$

where $\mathrm{att}'_1$, $\mathrm{att}'_2$, ..., $\mathrm{att}'_n$ are set-valued attributes. We give examples when introducing the two models.

### 4.3.2 GURA Models

We introduce two GURA models with incremental expressive power in precondition specification in the policy.

#### GURA$_0$

This model contains administrative rules defined as above and provide the language for specifying the preconditions in administrative rules. We first adopt the common structure called CPL shown in table 3.4 from chapter 3.2.

The language for specifying the preconditions in rules for a set-valued attribute $sua$ is an instance of CPL where the symbols $set$ and $atomic$ are defined as follows:

set ::= sua(u) | $\mathrm{constantSet}$

atomic ::= $\mathrm{constantAtomic}$

This language allows one to specify precondition using only the set-valued attribute $sua$ which is to be modified and constant set and atomic values. $\mathrm{constantSet}$ and $\mathrm{constantAtomic}$ are adopted from chapter 3 and they represent constant set and atomic values respectively. The language for specifying the preconditions in rules for an atomic-valued attribute $aua$ is an instance of CPL where the symbols $set$ and $atomic$ are defined as follows:

set ::= $\mathrm{constantSet}$

atomic ::= aua(u) | $\mathrm{constantAtomic}$

This language allows one to specify precondition using only the atomic-valued attribute $aua$ which is to be modified and constant set and atomic values.

**Table 4.4**: Example User Attributes

| Attribute | Type | Scope | Range |
|---|---|---|---|
| Clr | atomic | $\mathrm{SCOPE_{Clr}}$ = {unclassified, classified, secret, topsecret} | $\mathrm{Range(Clr) = SCOPE_{Clr}}$ |
| Dept | atomic | $\mathrm{SCOPE_{Dept}}$ ={software, hardware, finance, market} | $\mathrm{Range(Dept) = SCOPE_{Dept}}$ |
| Proj | set | $\mathrm{SCOPE_{Proj}}$ ={search, game, mobile, social, cloud} | $\mathrm{Range(Proj) = \mathcal{P}(SCOPE_{Proj})}$ |
| Skill | set | $\mathrm{SCOPE_{Skill}}$ ={web, system, server, win, linux, security} | $\mathrm{Range(Skill) = \mathcal{P}(SCOPE_{Skill})}$ |

***Example policy for GURA$_0$.*** We use the attributes information shown in table 4.4 where Clr

**Table 4.5**: Example Rules in GURA Schemes

| **Example Rules in GURA$_0$ Scheme** | | | | |
|---|---|---|---|---|
| N. | Relation | Admin Role | Precondition | Value |
| 1 | $\mathsf{can\_add}_{\mathrm{Proj}}$ | gameleader | mobile ∈ Proj(u) ∧ social ∈ Proj(u) ∧ ¬(cloud ∈ Proj(u)) | game |
| 2 | $\mathsf{can\_delete}_{\mathrm{Proj}}$ | gameleader | game ∈ Proj(u) ∧ social ∈ Proj(u) | game |
| 3 | $\mathsf{can\_assign}_{\mathrm{Dept}}$ | manager | Dept(u) = software | market |
| 4 | $\mathsf{can\_assign}_{\mathrm{Dept}}$ | manager | Dept(u) = hardware | market |
| **Example Rules in GURA$_1$ Scheme** | | | | |
| 5 | $\mathsf{can\_add}_{\mathrm{Proj}}$ | gameleader | mobile ∈ Proj(u) ∧ social ∈ Proj(u) ∧ ¬(cloud ∈ Proj(u)) ∧ Dept(u) = software ∧ Clr(u) = unclassified ∧ web ∈ Skill(u) ∧ security ∈ Skill(u) | game |
| 6 | $\mathsf{can\_delete}_{\mathrm{Proj}}$ | gameleader | game ∈ Proj(u) ∧ ¬(Clr(u) = topsecret) | game |
| 7 | $\mathsf{can\_assign}_{\mathrm{Dept}}$ | manager | Dept(u) = software ∧ ¬(Clr(u) = unclassified) ∧ server ∈ Skill(u) ∧ win∈ Skill(u) | market |
| 8 | $\mathsf{can\_assign}_{\mathrm{Dept}}$ | manager | Dept(u) = hardware ∧ ¬(Clr(u) = unclassified) ∧ server ∈ Skill(u) ∧ win∈ Skill(u) | market |

(Clearance) and Dept (Department) are atomic-valued attributes, while Prj (Project) and Skill are set-valued. We use the administrative rules specified in table 4.5. The first part shows example administrative rules in GURA$_0$. We assume that Bob is a member of administrative role "gameleader" and the attribute assignment for user Alice is: Clr(Alice) = unclassified, Dept(Alice) = software, Proj(Alice) = {mobile, social, search} and Skill(Alice) = {web, security}. Rule 1 authorizes Bob to add "game" to the Proj attribute of Alice. Bob is also authorized to delete "game" from Proj attribute of Alice based on rule 2. However, if Proj(Alice) = {mobile, social, cloud}, the add request will not take effect. Rules 3 and 4 are specified for the attribute Dept and they allow members of "manager" role to assign users who are currently in "software" or "hardware" department to "market" department. Note that multiple rules can be specified for the same attribute name and value combinations.

### GURA$_1$

Similarly, GURA$_1$ defines the language for specifying preconditions in administrative rules. The language is an instance of CPL where the symbols $set$ and $atomic$ are defined as follows:

$$\text{set} ::= \text{setua(u)} \mid \text{constantSet}$$

atomic ::= $\mathrm{atomicua(u)} \mid \mathrm{constantAtomic}$

This language allows one to specify policies using any attributes values of the same user as well as constant values. The symbols $\mathrm{setua}$ and $\mathrm{atomicua}$ are adopted from chapter 3 and they represent all possible set-valued and atomic-valued user attributes respectively.

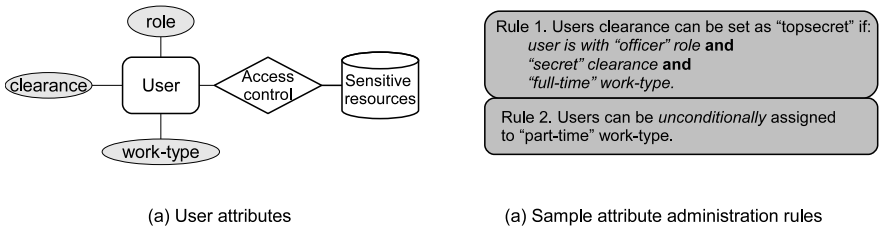
***Example policy for GURA*$_1$.** The second part in table 4.5 shows example administration rules in $\mathrm{GURA}_1$ based on attributes in table 4.4. Unlike $\mathrm{GURA}_0$, the preconditions of administrative rules in $\mathrm{GURA}_1$ can be specified using any attributes of the user. We assume that Alice has the same attribute assignment as in example 1. Bob is a member of "gameleader" role. Rule 5 authorizes Bob to add value "game" to Alice's Proj attribute. If Clr(Alice) = topsecret, the above request will not be authorized. Similarly, according to rule 7, Bob is not authorized to assign Alice to "market" department since Clr(Alice) = unclassfied. Other rules are self-explanatory.

## 4.4 User Attribute Reachability Analysis

We have defined GURA for user attribute administration. In this section, we study the user-attribute reachability problem.

### 4.4.1 Motivation for Reachability Analysis

User attributes are often used in sensitive activities such as authorization, authentication and audit. For example, in attribute-based access control (ABAC) [69, 80, 95], access decisions are made based on various user attributes compared to identity in discretionary access control (DAC) [123], clearance in mandatory access control (MAC) [121] and role in role based access control (RBAC) [119]. A critical question regarding access control policies is whether they ensure certain security properties. In context of GURA, as user attributes are further utilized for security-sensitive activities, it is important to ensure that every user can only be assigned appropriately valid attribute values. Although administrators might be trusted and expected to exercise due diligence in attribute assignments, it is nevertheless desirable to determine exactly what values of attributes can




(a) User attributes (a) Sample attribute administration rules

**Figure 4.2**: *Example User Attribute Reachability Problem*

be assigned by a collection of administrators acting cooperatively. Such analysis can also provide guidance on the sequence of attributes modifications to achieve specific attribute assignments for specific users. It is not straightforward to understand administrative policies by simple inspection. Large number of attributes and policies and unexpected actions of administrators complicate the analysis. Take the scenario in figure 4.2 as an example. Figure 4.2(a) shows that users can only access sensitive resources when the three attributes reach certain values simultaneously. Suppose an administrative user in a manager role can assign a user to "topsecret" clearance if the user is with "officer" role, with "secret" clearance and not "part time" in work-type. It might seem that a user can never be "part time" and with "topsecret" clearance at the same time. The policy is summarized in figure 4.2(b). However, the policy may inadvertently allow that. A user may be "full time" and then assigned to "topsecret" clearance according to rule 1. After that, he can be assigned to "part time" work-type according to rule 2. Reachability analysis can reveal such anomalies which may not be explicitly considered or immediately obvious in policy design.

We explore attribute reachability analysis in this work. Policy analysis, which allows policy designers to check whether their policies meet their security goals, has been recognized as important in access control [93, 94, 124]. The closest work is role reachability analysis in ARBAC97 [66, 94, 124, 129, 131]. Since GURA is an extension to ARBAC which contains more than one attribute, it is not sufficient to directly use the results from ARBAC97 analysis. In ARBAC97 there is a single set-valued attribute called role. In GURA on the other hand there are multiple attributes, some set-valued and some atomic-valued with possible constraints across multiple attributes. We study reachability analysis on a restricted version of the GURA model called rGURA. rGURA is different from GURA in the precondition specification language. It has a restricted ver-

sion of the languages provided in GURA models. We start by studying rGURA because even the reachability of this model, which is not as expressive as GURA model, is challenging. Our main contributions are as follows.

- The reachability problem asks whether a user can be assigned to specific attributes values by the actions of a set of administrators. We formally define user attribute reachability by abstracting the rGURA model as a state transition system. We consider different variations of queries for set-valued attributes than that in ARBAC study. In ARBAC97 analysis, a $goal$ (a set of roles) is reached if the user is assigned the roles in $goal$ or a superset of these. This is because the additional roles cannot reduce authorization in RBAC systems. With general attributes it is not clear how their values can impact authorization. Additional values in a set-valued attribute may possibly reduce authorization, depending on how the authorization policies are specified. Therefore, we also consider the case where set-valued attributes should be assigned exactly the same values as in $goal$ but not a superset.

- By reductions from the role reachability problem in ARBAC97 and the planning problem in artificial intelligence, we show that general attribute reachability problems are PSPACE-complete, along with identified special cases. We also consider special NP-complete or NP-hard cases.

- We identify polynomial time solvable cases which are useful in practice. As the input to the algorithms may be large and the analysis may need to be performed frequently, polynomial performance is always desirable. We further evaluate performance through simulations.

We focus on user attributes. In general, attributes are also associated with other entities such as subjects, objects and environment in ABAC systems [69]. Our reachability analysis results apply to attributes of any such entity so long as they are managed using the rGURA style, that is each attribute is managed by administrative users who can modify it provided the entities' attributes satisfy specified preconditions. However, rGURA-style models may not be appropriate for attribute administration in all possible scenarios. For instance, one may allow subjects of regular users to

modify object attributes, especially objects that they create. While the rGURA style is likely best suited to user attributes it does have broader applicability beyond the scope studied here. For example, in a cloud computing scenario, the attributes of the same user may be administered distributively by different attribute providers and users may use these attributes to request the same online service. These attributes may also be constrained by each other as modeled in GURA.

### 4.4.2 rGURA Scheme

An rGURA scheme is a state transition system where a state is an attribute assignment for each user and each attribute. State transitions are caused by authorized administrative requests to modify user attributes. We give the general definition for an rGURA scheme below, followed by two specific instantiations $\text{rGURA}_0$ and $\text{rGURA}_1$ with specific formal grammars for preconditions.

**Definition 3** (rGURA **Scheme**). *An rGURA scheme is a state transition system* $\langle \text{U}, \text{UA}, \text{AR}, \text{Range}[], \text{attType}[], \text{SCOPE}, \Psi, \Gamma, \delta \rangle$ *where,*

- $\text{U}, \text{UA}, \text{AR}, \text{Range}[]$ *and* $\text{attType}[]$ *are defined above.*
- $\text{SCOPE} = \langle \text{SCOPE}_{att_1} \ldots \text{SCOPE}_{att_n} \rangle$ *where* $att_i \in \text{UA}$*, is the collection of the scopes of all attributes.*
- $\Psi = \langle \mathsf{can\_assign}, \mathsf{can\_add}, \mathsf{can\_delete} \rangle$*, is the collection of administrative rules for all attributes.*
- $\Gamma$ *is the finite set of states. A* state $\gamma \in \Gamma$ *records assigned attribute values for each user. The user attribute assignment in state* $\gamma$*, denoted* $\text{UAA}_\gamma$*, contains tuples of the form* $\langle u, att, val \rangle$ *for every* $u \in \text{U}$*,* $att \in \text{UA}$ *such that* $att(u) = val$*. Formally,*

$$\begin{aligned}\text{UAA}_\gamma = \{ & \langle u, att, val_1 \rangle \mid u \in \text{U} \wedge att \in \text{UA} \wedge \\ & val_1 \in \text{Range}(att) \ \wedge (\forall val_2 \in \text{Range}(att). \\ & val_2 \neq val_1 \rightarrow \langle u, att, val_2 \rangle \notin \text{UAA}_\gamma)\}\end{aligned}$$

**Table 4.6**: State Transition Function $\delta : \Gamma \times Request \rightarrow \Gamma$

| (1) Let the source state be $\gamma_1$. In all requests in the first column, $ar \in$ AR, $u \in$ U, $att \in$ UA, $val' \in \text{SCOPE}_{att}$.<br>(2) Satisfy: U $\times$ C $\times$ $\Gamma \rightarrow \{\mathbf{true}, \mathbf{false}\}$, returns $\mathbf{true}$ if user $u \in U$ satisfies precondition $c \in$ C in state $\gamma \in \Gamma$, else **false**. | |
|---|---|
| **Request** | **Target State** |
| assign($ar$, $u$, $att$, $val'$) | **if** $\exists \langle ar,\ c,\ val\rangle \in \mathsf{can\_assign}_{att} \wedge \text{Satisfy}(u,\ c,\ \gamma_1)$<br>**then** transition to target state $\gamma_2$ where: $\text{UAA}_{\gamma_2} = \text{UAA}_{\gamma_1} \setminus \langle u, att, val\rangle \cup \langle u, att, val'\rangle$<br>**else** remain in $\gamma_1$ |
| add($ar$, $u$, $att$, $val'$) | **if** $\exists \langle ar,\ c,\ val\rangle \in \mathsf{can\_add}_{att} \wedge \text{Satisfy}(u,\ c,\ \gamma_1)$<br>**then** transition to target state $\gamma_2$ such that $\text{UAA}_{\gamma_2} = \text{UAA}_{\gamma_1} \setminus \langle u, att\ , setval\rangle \cup \langle u, att\ , setval'\rangle$<br>where $att(u) = setval$ in state $\gamma_1$ and $setval' = setval \cup \{val'\}$,<br>**else** remain in $\gamma_1$ |
| delete($ar$, $u$, $att$, $val'$) | **if** $\exists \langle ar,\ c,\ val\rangle \in \mathsf{can\_delete}_{att} \wedge \text{Satisfy}(u,\ c,\ \gamma_1)$<br>**then** transition to target state $\gamma_2$ where: $\text{UAA}_{\gamma_2} = \text{UAA}_{\gamma_1} \setminus \langle u, att\ , setval\rangle \cup \langle u, att\ , setval'\rangle$<br>where $att(u) = setval$ in state $\gamma_1$ and $setval' = setval \setminus \{val'\}$,<br>**else** remain in $\gamma_1$ |

- $\delta : \Gamma \times Request \rightarrow \Gamma$ *is the transition function, where Requestis the set of all possible administrative requests defined above. Function $\delta$ is formally defined in table 4.6.*

### The rGURA$_0$ Scheme

An rGURA$_0$ scheme is an instance of rGURA scheme where the grammar for specifying preconditions is specified as follows. In each $\mathsf{can\_assign}_{aua}$ relation for each atomic-valued attribute, the preconditions in all rules are generated by the following grammar,

$$\varphi ::= \neg\ \varphi \mid \varphi \wedge \varphi \mid aua(u) = avalue$$

$$avalue ::= aval_1 \mid aval_2 \ \ \ldots \mid aval_n$$

where $\text{SCOPE}_{aua} = \{aval_1,\ aval_2,\ \ldots,\ aval_n\}$. In all rules in $\mathsf{can\_add}_{sua}$ and $\mathsf{can\_delete}_{sua}$ relations, the preconditions are formulas generated by the following grammar,

$$\varphi ::= \neg\ \varphi \mid \varphi \wedge \varphi \mid svalue \in sua(u)$$
$$svalue ::= sval_1 \mid sval_2 \mid\ \ldots\ \mid sval_m$$

where $\text{SCOPE}_{sua} = \{sval_1,\ sval_2,\ \ldots,\ sval_m\}$.

**The rGURA$_1$ Scheme**

An rGURA$_1$ scheme is an instance of rGURA scheme where the preconditions in all rules can be specified using the grammar,

$$\varphi ::= \neg\varphi \mid \varphi \wedge \varphi \mid aua(u) = avalue \mid svalue \in sua(u)$$

where $avalue$ and $svalue$ could be any value from the scope of any atomic-valued and set-valued attribute respectively. Similarly, the symbols $aua$ and $sua$ can be any atomic-valued and set-valued attribute respectively of the user $u$. Note that each attribute should be compared with a value from its respective scope, otherwise, the formulas return **false**.

### 4.4.3 User Attribute Reachability Problem Definition

The attribute reachability problem (or simply the reachability problem) is informally defined as follows. Given an initial state with an assignment of each attribute for a particular user, can members of a set of administrative roles issue one or more administrative requests that transition the system to a target state with specific attribute assignments for that user? Before formally defining the reachability problems in the context of the rGURA$_0$ and rGURA$_1$ schemes defined earlier, we note two simplifications. Firstly, reachability analysis questions are about one user. Since modifications to attributes of one user have no impact on potential changes to the attributes of other users,

we only consider the attribute assignment of a single user of interest in a state. That is, we assume $\mathrm{U} = \{\mathbf{u}\}$ in all of our analysis. Secondly, reachability problems ask about the power of members of a set of administrative roles $\mathrm{SUBAR} \subseteq \mathrm{AR}$. In this case, the administrative rules specified for roles not in $\mathrm{SUBAR}$ need not be considered for the analysis. We assume that the scheme is provided with $\Psi$ which only contains administrative rules for roles in $\mathrm{SUBAR}$, that is, $\mathrm{AR} = \mathrm{SUBAR}$.

The above simplification eases our presentation without loss of generality. We now define the notion of a *query* which is concerned about whether a particular state "satisfies" specific attribute assignments for a given user. A query can be satisfied at varying levels. We discuss two levels here.

A query can be satisfied in a "strict" fashion if every attribute assignment specified in the query is exactly the same as that in the concerned state. A query can be satisfied in a "relaxed" fashion if in the concerned state every atomic-valued attribute assignment specified in the query is exactly the same but if every set-valued attribute assignment in the concerned state is a superset of the corresponding set-valued attribute assignment specified in the query. Note that both in the strict and relaxed levels of satisfaction of a query, the atomic-valued attributes in the concerned state should exactly match the query. The distinction arises in the values of set-valued attributes since the values in the concerned state can either exactly equal or simply contain (superset) the value specified in the query. Since atomic-valued attributes do not affect query satisfaction levels, we illustrate the difference on set-valued attributes. For instance, let $\mathrm{UA} = \{\text{Proj}\}$ and $\mathrm{U} = \{\text{Alice}\}$. An example query is a user attribute assignment: $q = \langle$ Alice, Proj, {cloud, game}$\rangle$. In $\mathrm{RP}_{=}$ query type, $q$ can be satisfied only by states $\gamma$ where $\mathrm{UAA}_{\gamma} = \{\langle$ Alice, Proj, {cloud, game}$\rangle\}$. While in $\mathrm{RP}_{\supseteq}$ query type, $q$ can be satisfied by any state $\gamma'$ where $\mathrm{UAA}_{\gamma'} = \{\langle$Alice, Proj, $setval\rangle\}$ and $\{\text{cloud, game}\} \subseteq setval$.

Our analysis is confined to these two levels of query satisfaction. We formally specify the queries and the satisfaction levels below.

**Definition 4 (Reachability Query).** *A* Reachability Query *specifies value-assignments for selected attributes of a user in the target state. Let* $\mathrm{Q}$ *denote the set of queries. Each query* $q \in \mathrm{Q}$ *is a subset*

*of* $\mathrm{UAA}_\gamma$.

**Definition 5** (**Reachability Query Types**). *Given a scheme* $\langle \mathrm{U},\ \mathrm{UA},\ \mathrm{AR},\ \mathrm{Range}[],\ \mathrm{attType}[],\ \mathrm{SCOPE},\ \Psi,\ \Gamma,\ \delta\rangle$, *we define two* Reachability Query Types*:*

- $\mathrm{RP}_=$ *queries have the entailment function* $\vdash_{\mathrm{RP}_=}: \Gamma \times \mathrm{Q} \to \{\mathbf{true}, \mathbf{false}\}$ *which returns* **true** *(i.e.,* $\gamma \vdash_{\mathrm{RP}_=} q$*) if* $\forall\ \langle u, att, val\rangle \in q.\ \langle u, att, val\rangle \in \mathrm{UAA}_\gamma$.

- $\mathrm{RP}_\supseteq$ *queries have the entailment function* $\vdash_{\mathrm{RP}_\supseteq}: \Gamma \times \mathrm{Q} \to \{\mathbf{true}, \mathbf{false}\}$ *which returns* **true** *(i.e.,* $\gamma \vdash_{\mathrm{RP}_\supseteq} q$*) if* $\forall\ \langle u, att, val\rangle \in q$*: (1)* $\langle u,\ att,\ val\rangle \in \mathrm{UAA}_\gamma$ *if* $\mathrm{attType}(att) = \mathrm{atomic}$ *and (2)* $\exists \langle u,\ att,\ val'\rangle \in \mathrm{UAA}_\gamma$ *where* $val' \supseteq val$ *if* $\mathrm{attType}(att) = \mathrm{set}$.

**Definition 6** (**Plan**). *A* plan *is a sequence of authorized administrative requests that causes a transition from one state to another. Given a scheme* $\langle \mathrm{U},\ \mathrm{UA},\ \mathrm{AR},\ \mathrm{Range}[],\ \mathrm{attType}[],\ \mathrm{SCOPE},\ \Psi,\ \Gamma,\ \delta\rangle$ *and states* $\gamma, \gamma' \in \Gamma$, *a sequence of authorized requests* $\langle\ req_1, req_2, \ldots, req_n\rangle$ *where* $req_i \in$ *Request*$(1 \leq i \leq n)$ *is called a* plan *to transition from an initial state* $\gamma$ *to the target state* $\gamma'$ *if:* $\gamma \overset{req_1}{\to} \gamma_1 \overset{req_2}{\to} \gamma_2 \ldots \overset{req_n}{\to} \gamma'$. *The arrow denotes a successful transition from one state to another in response to an administrative request* $req_i$ *that is authorized by rules in* $\Psi$. *For convenience, we also write* $\gamma \overset{plan_\Psi}{\rightsquigarrow} \gamma'$.

The reachability problem is concerned about whether it is possible to transition an initial state to some target state where the attribute-value assignments satisfy a particular query.

**Definition 7** (**Reachability Problems**). *Given a scheme* $\langle \mathrm{U},\ \mathrm{UA},\ \mathrm{AR},\ \mathrm{Range}[],\ \mathrm{attType}[],\ \mathrm{SCOPE},\ \Psi,\ \Gamma,\ \delta\rangle$*:*

- *An* $\mathrm{RP}_=$ Reachability Problem *instance* $\mathrm{I}$ *is of the form* $\langle \gamma,\ q\rangle$ *where* $\gamma \in \Gamma$ *and* $q \in \mathrm{Q}$ *and asks does there exist a plan* $P$ *for problem instance* $\mathrm{I}$ *such that* $\gamma \overset{P_\Psi}{\rightsquigarrow} \gamma'$ *and* $\gamma' \vdash_{\mathrm{RP}_=} q$.

- *An* $\mathrm{RP}_\supseteq$ Reachability Problem *instance* $\mathrm{I}$ *is of the form* $\langle \gamma,\ q\rangle$ *where* $\gamma \in \Gamma$ *and* $q \in \mathrm{Q}$ *and asks does there exist a plan* $P$ *for problem instance* $\mathrm{I}$ *such that* $\gamma \overset{P_\Psi}{\rightsquigarrow} \gamma'$ *and* $\gamma' \vdash_{\mathrm{RP}_\supseteq} q$.

### 4.4.4 Analysis Result

It is evident from the definitions, given the same scheme and problem instance, if the $\mathrm{RP}_{=}$ problem has a positive answer then so does the $\mathrm{RP}_{\supseteq}$ problem, but not vice versa. The size of input for each problem instance $I$ is the sum of size of each set in $I$. Our analysis proves the complexity of attribute reachability problems for rGURA schemes in general is PSPACE-complete. However, we have identified instances of the rGURA scheme with some restrictions on the precondition specification in administrative rules that have more practical time complexity. Moreover, these instances have many practical applications as will be discussed later. The following restrictions are considered:

- **No negation ($\overline{N}$)**: $\Psi$ satisfies $\overline{N}$ if no rules in $\Psi$ use negation in preconditions.

- **No deletion ($\overline{D}$)**: $\Psi$ satisfies $\overline{D}$ if for all set-valued attributes $sua$ in $\mathrm{UA}$, $\mathsf{can\_delete}_{sua}$ is *empty*. This restriction applies only to problem instances containing set-valued attributes. It implies that once a value is added, it can never be deleted.

- **Single rule** ($\mathrm{SR}$): $\Psi$ satisfies $\mathrm{SR}$ if: (1) for each atomic-valued attribute $aua \in \mathrm{UA}$, there is at most one precondition associated with a particular value assignment in the $\mathsf{can\_assign}_{\mathsf{aua}}$ relations and (2) for each set-valued attribute $sua \in \mathrm{UA}$, there is at most one precondition associated with a particular value assignment in each of the $\mathsf{can\_add}_{\mathsf{sua}}$ and $\mathsf{can\_delete}_{\mathsf{sua}}$ relations. That is, $\Psi$ satisfies $\mathrm{SR}$ if (1) $\forall att \in \mathrm{UA} \wedge \mathrm{attType}(att) = \mathrm{atomic}$. $\forall val \in \mathrm{SCOPE}_{att}$, $|\{c \mid \langle ar, c, val\rangle \in \mathsf{can\_assign}_{\mathsf{att}}\}| \leq 1$ and (2) $\forall\ att \in \mathrm{UA} \wedge \mathrm{attType}(att) = \mathrm{set}$. $\forall\ val \in \mathrm{SCOPE}_{att}$, $|\{c \mid \langle ar, c, val\rangle \in \mathsf{can\_add}_{att}\}| \leq 1$ and $|\{c \mid \langle ar, c, val\rangle \in \mathsf{can\_delete}_{att}\}| \leq 1$.

  The $\mathrm{SR}$ restriction means that the preconditions in can_assign, can_add or can_delete rules specified for each attribute-value pairs are unique. However, the corresponding rules could still be assigned to different administrative roles. For instance, if an additional rule is speci-

fied in the $\mathsf{can\_add}_{\mathrm{Proj}}$ relation in item 1 in table 4.5 as follows:

$$(\mathrm{manager}, \mathrm{mobile} \in \mathrm{Proj(u)} \wedge \mathrm{social} \in \mathrm{Proj(u)} \wedge \neg(\mathrm{cloud} \in \mathrm{Proj(u)}), \mathrm{game})$$

The $\mathrm{SR}$ restriction is still satisfied since given an attribute and value pair (the attribute being "Proj" and the value being "game"), the preconditions remain the same even though there are multiple rules that allow adding the same value to that attribute. Here, the two rules allow members of different administrative roles to add "game" to the attribute "Proj". If the above rule were to be specified as follows:

$$(\mathrm{manager}, \mathrm{mobile} \in \mathrm{Proj(u)} \wedge \neg(\mathrm{cloud} \in \mathrm{Proj(u)}), \mathrm{game})$$

the $\mathrm{SR}$ restriction is no longer satisfied. Similar considerations apply to rules in $\mathsf{can\_delete}_{\mathsf{att}}$ and $\mathsf{can\_assign}_{\mathsf{att}}$.

Another restriction is the type of attributes contained in the system. It's possible that a system deal only with atomic-valued or set-valued attributes. These restrictions are also considered.

Many attribute semantics and applications work well with the above restrictions to be of practical use. The positive results are that in such situations reachability analysis can be performed efficiently. For instance, consider a scenario to express necessary prerequisites to register for a course in a university. Let a "course" attribute keep track of the set of courses a student has earned credits for. The preconditions to register for a course in this scenario are mostly positive which commonly check whether the student has successfully completed all the prerequisite courses. This would satisfy the $\overline{N}$ restriction. Consider a "Skill" attribute that keeps track of user skills (*e.g.,* web, system, etc. as mentioned in table 4.4) which may never need to be deleted after add and hence satisfies the $\overline{D}$ restriction. The $\mathrm{SR}$ restriction can be satisfied in situations where there are no alternative preconditions that allow a particular value to be assigned/added/deleted to/from an attribute. That is, there is exactly one and only way an attribute can obtain a particular value.

We use [rGURA$_x$-[atomic, set], `Restriction`] to denote a specialized rGURA scheme on which we perform reachability analysis. The subscript $x$ takes a value of 0 or 1 representing an rGURA$_0$ or rGURA$_1$ scheme and [atomic, set] means that the scheme contains only the specified type of attributes (if not specified, it represents the general case where both types of attributes are included). `Restriction` represents possible combinations of SR, $\overline{D}$ and $\overline{N}$ meaning that the rules in the scheme satisfies those restrictions. Thus [rGURA$_1$-atomic, $\overline{N}$] denotes an rGURA$_1$ scheme $\langle$U, UA, AR, Range[], attType[], SCOPE, $\Psi$, $\Gamma$, $\delta\rangle$ where UA contains only atomic-valued attributes and $\Psi$ satisfies $\overline{N}$.

Figure 4.3 summarizes our analysis using the above notation. The left column shows the results for rGURA$_0$ variations and the right column shows the results for rGURA$_1$ variations. Each scheme is specified in a box which includes theorem and corollary numbers (abbreviated Th and C respectively) in the paper that provides the proofs for that specific scheme. An arrow from a box to another shows that the restrictions specified on the arrow which applied to the former box lead to the latter. Note that for schemes that contain only atomic-valued attributes, only RP$_=$ queries are considered. For schemes that only contain set-valued attributes or both types of attributes, both RP$_=$ and RP$_\supseteq$ are considered. Reachability analysis in general are intractable except RP$_=$ in [rGURA$_0$-atomic] Some observations are given below.

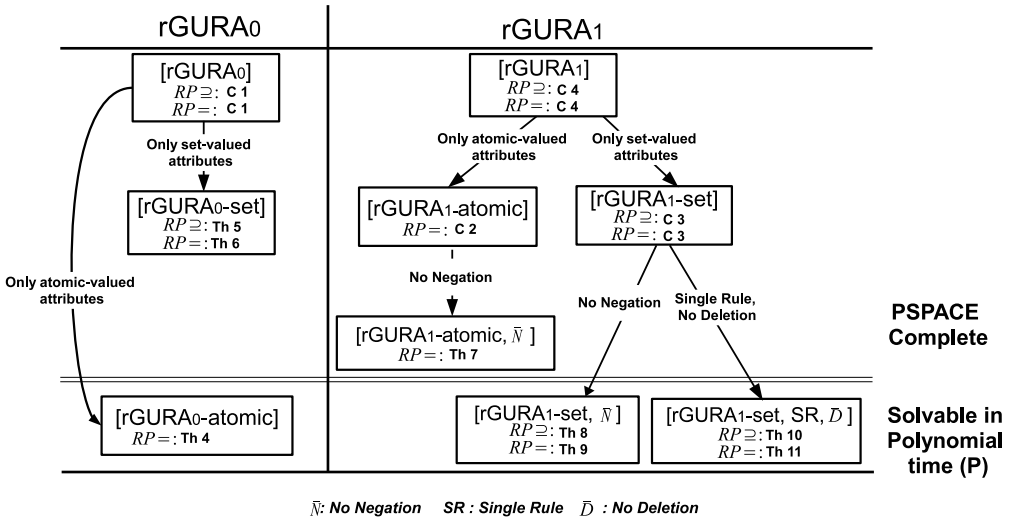



**Figure 4.3**: *Complexity Results for Different Classes of Reachability Problems*

- In the rGURA$_0$ column, [rGURA$_0$-atomic] is the only scheme that has a polynomial-time algorithm without further restrictions. Since this scheme only has atomic-valued attributes, only the $\mathrm{RP}_{=}$ query is considered. However, $\mathrm{RP}_{\supseteq}$ and $\mathrm{RP}_{=}$ in [rGURA$_0$-set] abruptly escalate to PSPACE-complete. This big jump is caused by negative preconditions and delete operations. Since [rGURA$_0$-set] is similar to ARBAC97, many of the results from prior work on reachability analysis in ARBAC97 can be adapted here [124]. Thus if $\overline{N}$ is enforced, $\mathrm{RP}_{\supseteq}$ is polynomial-time solvable. If $\overline{D}$ is enforced, $\mathrm{RP}_{\supseteq}$ for [rGURA$_1$-set, $\overline{D}$] is in NP.

- In rGURA$_1$, the complexity for reachability problems differs sharply from rGURA$_0$ for systems containing only atomic-valued attributes (from polynomial-time to intractable) while not so much for systems containing only set-valued attributes (both are PSPACE-complete). The huge increase in complexity for schemes containing only atomic-valued attributes is caused by mutual constraints of attributes on each other in the preconditions for rGURA$_1$. In order to satisfy a query, attribute values may need to be re-assigned a large number of times. In addition, each attribute needs to be satisfied simultaneously which also increases the complexity.

- Also interestingly, similar restrictions work fairly differently on atomic-valued and set-valued attributes in rGURA$_1$. A notable example is $\overline{N}$. $\mathrm{RP}_{=}$ in [rGURA$_1$-atomic, $\overline{N}$] is surprisingly intractable while $\mathrm{RP}_{=}$ and $\mathrm{RP}_{\supseteq}$ for [rGURA$_1$-set, $\overline{N}$] are in P.

- The [rGURA$_1$-set] scheme is intractable for both $\mathrm{RP}_{=}$ and $\mathrm{RP}_{\supseteq}$. However, two types of restrictions yield polynomial-time analysis results: [rGURA$_1$-set, $\overline{N}$] and [rGURA$_1$-set, $\mathrm{SR}$, $\overline{D}$].

### 4.4.5 Formal Proofs

In this section, we walk through proofs for the results in figure 4.3. For schemes with PSPACE-complete complexity, we show reduction to a known problem in that class. For schemes in the polynomial-time solvable class, we provide polynomial-time algorithms with correctness proofs.

There are two parts to a PSPACE-complete proof: a proof to show that the scheme is solvable in PSPACE and a proof to show that the scheme is PSPACE-hard. The first part is the same for all schemes in figure 4.3.

**Lemma 1.** *Every scheme in figure 4.3 is in PSPACE.*

*Proof.* Given any problem instance, a Non-deterministic Turing Machine can simulate the following algorithm. In each state, the Turing machine stores the current user-attribute assignments, attributes scopes, query and administrative policies. These are needed to guess the next possible states. In each step, it guesses the next possible states it can enter (there maybe more than one possible next states) by running the administrative policies against the current user-attribute assignments. For each next state, the Turing machine checks it against the query. If the query is satisfied, the process stops. Otherwise, it repeats the same process. The space needed for each state is polynomial to the input size which includes the initial user-attribute assignments, attributes scopes[1], query and administrative policies. Thus, all problems are in NPSPACE and thus also in PSPACE as implied by Savitch's theorem [125].

□

### rGURA$_0$ Schemes

Consider the [rGURA$_0$-atomic] scheme which only contains atomic-valued attributes. In an rGURA$_0$ scheme recall that modifications to one attribute have no impact on future modifications to other attributes (see section 4.4.2). Hence it is sufficient to confine our analysis to the set of rules specified for a single attribute, repeating this process for each attribute in turn. In other words, we can analyze the reachability of each attribute in the query independently and the combine the results. Note that this strategy does not work for rGURA$_1$ schemes.

**Theorem 4.** $\mathrm{RP}_{=}$ *for* [*rGURA*$_0$*-atomic*] *is solvable in P.*

[1] Although some of the attributes ranges may be encoded with smaller space than the number of values it may take (e.g., a integer with a range of [1,n] can be encoded by O(1) rather than O(n)), the space needed in the input is still polynomial because the administrative policies cannot be encoded in the same method.

*Proof.* The reachability query for each single attribute is transformed to a path search problem between two nodes in a directed graph. In this graph, the nodes are the values from the scope of this attribute. A directed edge from node $n_1$ to node $n_2$ represents an action of assigning $n_2$ to this attribute if the current value of that attribute is $n_1$.

The following simple algorithm can be used for solving the $\mathrm{RP}_{=}$ query (recall that $\mathrm{RP}_{\supseteq}$ query does not apply to this scheme). The plan to the original query on all attributes can be obtained by combining the plan for satisfying each attribute. Based on this observation, the reachability problem instance I = $\langle \gamma, q \rangle$ for the scheme [rGURA$_0$-atomic] can be reduced to finding whether, for an attribute $att \in \mathrm{UA}$, it is possible to reach a state $\gamma'$ from $\gamma$ that satisfies the condition that the value of $att$ is the same as the corresponding value specified in the query $q$. A directed graph *TG* = $\langle$V, E$\rangle$ is constructed based on the rules in $\mathsf{can\_assign}_{att}$. In this graph, $V = \mathrm{Range}(att)$. For each val$_1$ and val$_2$ in $V$, an edge $\langle$val$_1$, val$_2\rangle$ is added to E if $\langle ar, c, val_2 \rangle \in \mathsf{can\_assign}_{att}$ and $att(u) = val_1$ is a conjunct in $c$. A query on $att$ is equivalent to a path search problem between the corresponding two nodes in the graph which can be solved using well-known search algorithms such as depth first search (DFS). It is straight forward to generate a plan if the target value is reachable.

We assume that DFS is used. (1) **Correctness.** The algorithm is well-known to be correct. (2) **Complexity.** We first discuss the complexity for querying a single attribute $att \in \mathrm{UA}$. The graph can be created by traversing each rule once. Each rule adds at most $|\mathrm{Range}(att)|$ edges to the graph. The complexity of DFS on a graph $\langle$V, E$\rangle$ is O(|V| + |E|). Thus, the complexity for solving $\mathrm{RP}_{=}$ for attribute $att$ is O($|\mathrm{Range}(att)| \times |\mathsf{can\_assign}_{att}|$). For problem instances containing $|\mathrm{UA}|$ attributes, the overall complexity is O($|\mathrm{UA}| \times |\mathrm{Range}(att)| \times |\mathsf{can\_assign}_{att}|$), where $att$ has the largest $|\mathrm{Range}(att)| \times |\mathsf{can\_assign}_{att}|$ amongst all attributes. □

**Theorem 5.** $\mathrm{RP}_{\supseteq}$ *for* [*rGURA$_0$-set*] *is PSPACE-complete.*

*Proof.* Given lemma 1, it suffices to show PSPACE-hardness. As earlier, it is feasible to analyze the complexity of each attribute independent of each other. Our proof is to show the reduction from the role reachability problem. Combining the above observation, the fact that role can be

treated as simply another set-valued user attribute and [rGURA$_0$-set] has the same expressive as miniARBAC97 [124], the reduction is straight forward.

Our proof shows the reduction that the $\mathrm{RP}_{\supseteq}$ problem for [rGURA$_0$-set] is at least as hard as the role reachability problem in miniARBAC97 which is PSPACE-complete [124]. Its problem instance can be understood as a 3-tuple $\langle \gamma, goal, \psi \rangle$ where $\gamma$ is an initial state with role assignments for a particular user, $goal$ is the desired set of roles for that user in some future state and $\psi$ is a set of administrative rules that guides user-role assignments by a set of administrative roles. The reachability question asks whether a given set of administrative roles $AR$ can act with the permissions associated with their roles in $\psi$ and transition $\gamma$ to a future state $\gamma'$ such that the desired role assignments specified in $goal$ for a particular user is satisfied in $\gamma'$. In addition to can_assign and can_revoke relations for roles, the miniARBAC97 also considers SMER (Static Mutually Exclusive Roles) constraints which represents a set of conflicting roles that cannot be assigned to the same user at any time.

The core idea of our construction is to treat roles in miniARBAC97 as a user attribute called role. We assume that SMER = $\emptyset$ since it can be expressed in can_add rules using negative pre-conditions [124]. The scope of the role attribute is the same as the set of roles in miniARBAC97. It is straight-forward to specify each of the can_assign and can_revoke rules in miniARBAC97 using corresponding can_add$_{att}$ and can_delete$_{att}$ rules in [rGURA$_0$-set] since the pre-condition grammar of rGURA$_0$ is similar to that of miniARBAC97. User-role assignment in the initial state in miniARBAC97 can be mapped to attribute assignment in [rGURA$_0$-set] and the query can be specified. The reduction process takes O($|\gamma| + |goal| + |\psi|$). For a problem instance containing $|\mathrm{UA}|$ number of attributes, the total complexity is the sum of complexity of reduction for each attribute which is polynomial. This establishes that $\mathrm{RP}_{\supseteq}$ for [rGURA$_0$-set] is PSPACE-hard.

□

The miniARBAC97 work is useful for reduction with respect to $\mathrm{RP}_{\supseteq}$ queries. However, it does not deal with $\mathrm{RP}_{=}$ for which we utilize the SAS planning problem [16] from artificial intelligence.

**Theorem 6.** $\mathrm{RP}_{=}$ *for* [*rGURA*$_0$*-set*] *is PSPACE-complete.*

*Proof.* Per lemma 1, it suffices to show PSPACE-hardness. We use the result from SAS planning problem [16]. An instance of SAS planning problem is a tuple $\langle \mathcal{V}, \mathcal{O}, \mathrm{s}_0, \mathrm{s}_* \rangle$, where $\mathcal{V}$ represents a finite set of state variables with pre-specified domains for each variable, $\mathcal{O}$ represents a finite set of operators, $\mathrm{s}_0$ and $\mathrm{s}_*$ represent initial and goal states and they are both total states (i.e., each variable is assigned with a value from its domain). An operator $\langle \mathsf{pre}, \mathsf{prv}, \mathsf{post} \rangle$ updates state variables in post if the conditions pre and prv are satisfied in the current state. The conditions pre, prv and post are members of partial state space (state variables are allowed to be unspecified). The problem is given an initial state $\mathrm{s}_0$, does there exist a sequence of operators (a plan) which transition $\mathrm{s}_0$ to $\mathrm{s}_*$? The plan-existence for the SAS planning problem under U (each operator changes only a single state variable) and B (boolean domain for state variables) restrictions is PSPACE-complete [16].

We show that $\mathrm{RP}_{=}$ for [rGURA$_0$-set] is at least as hard as the [SAS planning, U, B] problem. As earlier, we consider the complexity of reachability of one attribute $att \in \mathrm{UA}$ independent of others. The reduction is as follows, given any SAS planning problem satisfying U and B. (1) Each state variable is mapped to one value in the scope of $att$. Thus, the scope of $att$ is a set of values whose size is the same as $\mathcal{V}$. In each state, if a state variable is set to **true**, the corresponding value is added to the attribute $att$. Thus, $\mathrm{s}_0$ is specified using attribute assignment of $att$ and $\mathrm{s}_*$ is specified as a query. (2) The operator which updates a state variable to **true** is mapped to one rule in $\mathsf{can\_add}_{att}$ and the operator which sets a state variable to **false** is mapped to one rule in $\mathsf{can\_delete}_{att}$ ($att$ is mapped to the state variable in the operator). A precondition in an operator can be specified as the precondition in each administrative rule. The complexity of the reduction process is $\mathrm{O}(|\mathcal{V}| + |\mathcal{O}|)$. This establishes that $\mathrm{RP}_{=}$ for [rGURA$_0$-set] is PSPACE-hard.

□

**Corollary 1.** $\mathrm{RP}_{\supseteq}$ *and* $\mathrm{RP}_{=}$ *for* [*rGURA*$_0$] *are PSPACE-complete.*

*Proof.* Since $\mathrm{RP}_{\supseteq}$ and $\mathrm{RP}_{=}$ in [rGURA$_0$] can also be answered by querying each attribute separately and this scheme supports both atomic and set-valued attributes, this result follows from

Lemma 1 and Theorems 4, 5 and 6.

□

This completes the left hand side of figure 4.3.

### rGURA$_1$ Schemes

For rGURA$_1$-atomic schemes we have the following results for $\mathrm{RP}_{=}$. (Recall that $\mathrm{RP}_{\supseteq}$ does not apply to rGURA$_1$-atomic schemes.)

**Theorem 7.** $\mathrm{RP}_{=}$ *for* [*rGURA*$_1$*-atomic,* $\overline{N}$] *is PSPACE-complete.*

*Proof.* By lemma 1, it suffices to show PSPACE-hardness. The SAS planning problem under the U restriction is PSPACE-complete [16]. We give a reduction from [SAS planning, U] to [rGURA$_1$-atomic, $\overline{N}$]. In the former, only positive conditions are allowed in the operators (pre and post) which is accommodated by the $\overline{N}$ restriction. Consider an instance of the SAS planning problem $\langle \mathcal{V}, \mathcal{O}, \mathrm{s}_0, \mathrm{s}_* \rangle$. (1) Map each state variable to one user attribute whose scope corresponds to the domain of the variable. Thus, $\mathrm{s}_0$ and $\mathrm{s}_*$ map to two different attribute assignments. (2) Since each operator updates only one variable, it can be mapped to one rule in can_assign$_{att}$ where $att$ is the corresponding user attribute of the variable updated in the operator. The value to be assigned is the same as that in post in the operator. The preconditions pre and prv only specifies positive precondition (no negative comparisons between state variables and values from their domains) and they can be specified using conjunctions. Negation is not required. The reduction process takes $\mathrm{O}(|\mathcal{V}| + |\mathcal{O}|)$. This establishes that $\mathrm{RP}_{=}$ for [rGURA$_0$-atomic, $\overline{N}$] is PSPACE-hard. □

**Corollary 2.** $\mathrm{RP}_{=}$ *for* [*rGURA*$_1$*-atomic*] *is PSPACE-complete.*

*Proof.* Follows from Theorem 7 and the fact that [rGURA$_1$-atomic, $\overline{N}$] is a sub-problem of [rGURA$_1$-atomic]. □

For rGURA$_1$-set we begin with the following observations.

**Corollary 3.** $\mathrm{RP}_{\supseteq}$ *and* $\mathrm{RP}_{=}$ *for* [*rGURA*$_1$*-set*] *are PSPACE-complete.*

*Proof.* This follows from Theorems 5 and 6 and the fact that $\mathrm{RP}_{\supseteq}$ and $\mathrm{RP}_{=}$ for [rGURA$_0$-set] are sub-problems of $\mathrm{RP}_{\supseteq}$ and $\mathrm{RP}_{=}$ for [rGURA$_1$-set] respectively. □

**Corollary 4.** $\mathrm{RP}_{\supseteq}$ *and* $\mathrm{RP}_{=}$ *for* [*rGURA*$_1$] *are PSPACE-complete.*

*Proof.* This follows from Corollaries 2 and 3. □

We now consider the $\mathrm{RP}_{\supseteq}$ problem for [rGURA$_1$-set, $\overline{N}$] which can be solved in polynomial-time by Algorithm 5. The algorithm works as follows. For $\mathrm{RP}_{\supseteq}$, we only need to consider issuing add requests to the set-valued attributes. This is because only positive preconditions are allowed, the rules cannot specify addition of new values to the set-valued attribute based on absence of certain values in the existing set. So existing values need not be removed in order to successfully add a new value. Thus, we only need to investigate can_add rules and completely ignore can_delete rules. Furthermore, additional values can be added to the attribute even if the desired set value is reached because it deals with the $\mathrm{RP}_{\supseteq}$ problem. Starting from the given state, we traverse all rules in can_add and try the add requests allowed by any rule if the corresponding attribute value is not yet in the current set. Algorithm 5 terminates either when (1) the current set can no longer be augmented by the rules in can_add, or (2) all attributes are assigned with all values from their scope. The outer while loop is required because a value added to one or more of the attributes in an earlier round can potentially enable new additions in subsequent rounds. This is due to rGURA$_1$ preconditions where attributes can constrain each other.

**Theorem 8.** $\mathrm{RP}_{\supseteq}$ *for* [*rGURA*$_1$*-set,* $\overline{N}$] *is in P.*

*Proof.* Algorithm 5 provides a polynomial-time solution.

**Correctness**. (1) Assume that Algorithm 5 returns a $plan$. If the $plan$ is empty, the query $q$ is trivially satisfied in $\gamma$. Otherwise, it is ensured that if executed sequentially, there exists at least one rule in can_add that authorizes each action in the $plan$. Thus, the $plan$ takes $\gamma$ to another state that satisfies $q$. (2) When Algorithm 5 returns $false$, there does not exist a $plan$. We use contradiction. Assume $plan$ = $\langle \mathrm{a}_1, \mathrm{a}_2, \ldots, \mathrm{a}_n \rangle$ is a valid plan of length n and is not detected

**Algorithm 5** Plan Generation for $\mathrm{RP}_{\supseteq}$ in [rGURA$_1$-set, $\overline{N}$]

```
1: Input: problem instance I = ⟨γ, q⟩ Output: plan or false
2: plan = ⟨⟩;
3: Begin with state s=γ;
4: if s ⊢_{RP⊇} q then return plan
5: while true do
6:     Let Save = UAA_s;
7:     for each att ∈ ATTR and ⟨ar, c, val⟩ ∈ can_add_att do
8:         if s satisfies precondition c ∧ (∄ ⟨u, att, sv⟩
9:         ∈ UAA_s such that val ∈ sv) then
10:            Suppose that ⟨u, att, setval⟩ ∈ UAA_γ;
11:            Go to state t such that
12:                UAA_t == UAA_s \{⟨u, att, setval⟩} ∪ {⟨u, att, setval ∪ {val}⟩};
13:            s = t;
14:            plan = plan.append(add(ar, u, att, val));
15:        end if
16:    end for
17:    if UAA_s == Save then break; else Save = UAA_s;
18:     end if
19: end while
20: if s ⊢_{RP⊇} q then return plan else return false end if
```

by algorithm 5. Without loss of generality, we assume $a_k$ ($1 \leq k \leq n$) is not detected and the state before $a_k$ is cur′. Thus, according to line 8 in algorithm 5, either (a) there does not exist a rule whose preconditions are satisfied in cur′ to authorize $a_k$, or (b) cur′ already contains the attribute value to be added in request $a_k$. In either case such an $a_k$ cannot exist. (3) Algorithm 5 always terminates because the number of attributes and the attribute values to be added to $\gamma$ are bounded. **Complexity**. The complexity is determined by the number of times all the rules in can_add are traversed. In the worst case, one value is added to one attribute during each round of rule traversing. Thus, the complexity of Algorithm 5 is O(($\sum_{att \in \mathrm{UA}} |\mathrm{SCOPE}_{att}|$) $\times$ |can_add|) and it is polynomial. |can_add| represents the size of all rules for all attributes. □ □

Next we consider the $\mathrm{RP}_{=}$ problem for [rGURA$_1$-set, $\overline{N}$].

**Theorem 9.** $\mathrm{RP}_{=}$ *for* [*rGURA*$_1$*-set,* $\overline{N}$] *is in P.*

*Proof.* The proof is by reduction to the STRIPS planning problem [36].

We use the result from STRIPS planning problem [36]. An instance of STRIPS planning problem is a tuple $\langle \mathcal{P}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{P}$ is a finite set of ground atomic formulas called conditions

(each take the value **true** or **false**), $\mathcal{O}$ is a finite set of operators pre $\Rightarrow$ post, where post updates the conditions to either positive or negative if pre is satisfied. The pre and post are satisfiable conjunctions of positive and negative conditions. Any state can be specified by a subset of $\mathcal{P}$, indicating that each element in the subset is **true** and all others are **false** in the state. $\mathcal{G}$ called $goal$ is a satisfiable conjunction of positive and negative conditions. $\mathcal{S}$ is a goal state if $\mathcal{S}$ all positive conditions in goal is in $\mathcal{S}$ and none of the negative conditions in goal appears in $\mathcal{S}$. STRIPS planing explores a sequence of operators which transition the initial state $\mathcal{I}$ to a state in which $\mathcal{G}$ is satisfied. PLANSAT is defined as determining whether an instance of STRIPS planing is satisfiable.

[36] shows that [PLANSAT, $*$+preconds, 1 postcond] is polynomial time solvable . Here, only positive preconditions are allowed and each operator only modifies one condition, setting it as either positive or negative. We show the reduction: plan existence in [PLANSAT, $*+$ preconds, 1 postcond] is at least as hard as $\mathrm{RP}_{=}$ in [rGURA$_1$-Set, $\overline{N}$]. The reduction is as follows. Given any [rGURA$_1$-set, $\overline{N}$] scheme: (1) each attribute and value pair $(att, value)$ is mapped to a corresponding condition; (2) to specify a state in [rGURA$_1$-set], for each attribute and value pair, the corresponding condition is set to true. To specify a query in [rGURA$_1$-set], for each attribute and value pair in the query, the corresponding condition is set to true. For all other attribute and value pairs not in the query, their corresponding conditions are set to false. This ensures the query is only satisfiable with exact the same value for each attribute; (3) each rule in can_add$_{att}$ is specified as a positive operator which updates the corresponding condition for the specified attribute and value pair. The precondition is specified as pre, the value to be added is specified in post; and finally (4) each rule in can_delete$_{att}$ rule is specified as a negative operator. The reduction process takes O(($\sum_{att \in \mathrm{UA}} |\mathrm{SCOPE}_{att}|$)+$|\Psi|$) where $|\Psi|$ is the number of all administrative rules ($\Psi$ contains only can_add and can_delete relations) which is polynomial. □

We now consider $\mathrm{RP}_{\supseteq}$ for [rGURA$_1$-set, SR, $\overline{D}$] for which Algorithm 6 provides a polynomial-time solution. The $\overline{D}$ restriction obviates the need to include the rules from can_delete relations. The SR restriction provides a critical advantage in our analysis. Since a unique precondition facilitates addition of a value to an attribute, the number of paths in the search space is greatly

reduced. The algorithm works by traversing backwards from the target state as follows. Assume that the problem instance is $\langle \gamma, q \rangle$. For at least one attribute, $q$ requires a value which is a superset of that in $\gamma$ (otherwise, $q$ is already satisfied by $\gamma$). For those attribute values in $q$ not in $\gamma$, there must be a corresponding add action in the plan if it exists. In addition, in order to add those values, attribute values which appear as positive preconditions in the administrative rules which authorize those add actions must also be added and so on. Thus, the basic idea is to use backward search to compute all attribute values that are required to be added in order to satisfy $q$. This is done by tracing rules in can_add for attributes and values that need to be added and recursively for those values required in the previous state (line 5 - line 9). Till now, only positive preconditions have been considered. If negative preconditions of any add request for values to be added are not satisfied in $\gamma$, $q$ cannot be satisfied (since they can never be deleted). Otherwise, negative preconditions can only be introduced during each step of adding new values. Thus, those add actions need to be ordered based on mutual dependencies. This is achieved by creating a directed graph which reflects dependencies of attribute values (line 16 - line 21). A plan is a topological ordering of the graph (line 22). If there are cycles in the graph, $q$ can never be satisfied.

**Theorem 10.** $\mathrm{RP}_{\supseteq}$ *for* [*rGURA*$_1$*-set,* SR*,* $\overline{D}$] *is in P.*

*Proof.* Algorithm 6 provides a polynomial-time solution.

**Correctness**. (1) *If algorithm 6 returns a* $plan$ and it is empty, query q is satisfied in $\gamma$. If the $plan$ is not empty, we assume $plan$ = $\langle \mathrm{a}_1, \mathrm{a}_2, \ldots \mathrm{a}_n \rangle$. We prove that the plan is valid. We first prove that the first action $\mathrm{a}_1$ is allowed to be executed. Firstly, its positive precondition is satisfied in $\gamma$. The $repeat$ loop (line 5) only stops when $toadd$ does not change. It means positive preconditions for all vertices already in $toadd$ are either satisfied by $ppre$ or $cur$. Since $\mathrm{a}_1$ is the first action, its positive precondition is satisfied in $cur$. Secondly, its negative precondition is satisfied in $\gamma$ (otherwise, line 14 returns false). If request $\mathrm{a}_k$ ($1 \leq k \leq n$) is authorized, then $\mathrm{a}_{k+1}$ is also allowed because $\mathrm{a}_{k+1}$'s positive precondition may be satisfied in $cur$ or action $\mathrm{a}_k$ (and its negative precondition is satisfied by both $cur$ and $\mathrm{a}_k$). Thus, sequentially executing the plan leads to a set of reachable states. $q$ is satisfiable because in the first round of $repeat$, $toadd$ contains attribute values in $q$

**Algorithm 6** Plan Generation for $\mathrm{RP}_{\supseteq}$ in [rGURA$_1$-set, SR, $\overline{D}$]

1: **Input:** problem instance I = $\langle \gamma, q \rangle$
2: **Output:** plan or false
3: $toadd$ = $\emptyset$; $npre$ = $\emptyset$; $cur$ = $\emptyset$; $plan$ = $\langle\rangle$;
4: **if** $\gamma \vdash_{\mathrm{RP}_{\supseteq}} q$ **then return** $\langle\rangle$
5: $\forall$ ⟨u, att, vset⟩ $\in \mathrm{UAA}_\gamma$. $\forall$ val $\in$ vset.
6: $cur'$ = $cur \cup$ {(att, val)};
7: $\forall$ ⟨u, att, vset⟩ $\in q$. $\forall$ val $\in$ vset.
8: $toadd'$ = $toadd \cup$ {(att, val)};
9: **Repeat**:
10: $\forall$ (att, val) $\in toadd$
11: $ppre$ = {(att$_1$, val$_1$) | $\exists \langle ar, c, val \rangle \in can\_add_{att}$ such that $val_1 \in att_1(u)$ is a conjunct in $c$};
12: $toadd'$ = $toadd \cup ppre \setminus cur$;
13: **Until** $toadd$ **does not change**
14: **if** $\exists$ (att, val) $\in toadd$ such that $\nexists \langle ar, c, val \rangle \in can\_add_{att}$
15: **then return false end if**
16: $\forall$ $(att, val) \in toadd$
17: $npre'$ = $npre \cup$ {att$_1$, val$_1$) | $\exists \langle ar, c, val \rangle \in can\_add_{att}$
18: such that $\neg(val_1 \in att_1(u))$ is a conjunct in $c$};
19: **if** $npre \cap cur$ != $\emptyset$ **then return false end if**
20: Construct a directed graph $G = \langle V, E \rangle$;
21: V = $toadd$; E = $\emptyset$;
22: **for** each pair of nodes $((att, val), (att_1, val_1)) \in$ V **do**
23: **if** ($\exists \langle ar, c, val_1 \rangle \in can\_add_{att_1}$ such that $val \in att(u)$ is a conjunct in c) $\vee$($\exists \langle ar, c, val \rangle \in can\_add_{att}$ such
24: that $\neg(val_1 \in att_1(u))$ is a conjunct in c)
25: **then** E$'$ = E $\cup$ {$\langle (att, val), (att_1, val_1) \rangle$}; **end if**
26: **end for**
27: $plan$ = Topological ordering on graph G;
28: **if** topological ordering is successful **then return** $plan$;
29: **else return false**; **end if**

while not in $\gamma$. (2) *If algorithm 6 returns false*, there are several reasons: (a) no $can\_add$ rule for some of the attribute values in $toadd$; (b) some of the negative preconditions are not satisfied in $\gamma$; (c) there are loops in the graph created by lines 17-21. We show that at the least all attribute values in $toadd$ should be added to reach $q$. Assume $toadd$ = {(a$_1$, v$_1$), (a$_2$, v$_2$) ... (a$_n$, v$_n$)}. Without loss of generality, we let $vset$ = $toadd \setminus$ {(a$_k$, v$_k$)} which if added, transitions $\gamma$ to a state in which $q$ is satisfied. Thus (a$_k$, v$_k$) is in $ppre$ and is not in $cur$ in some round of $repeat$ through lines 5 to 9. If it is not added, it means $\langle ar, c, val \rangle \notin \mathsf{can\_add}_{a_{k-1}}$ and $(a_{k-1}, val)$ cannot be added. Thus, other attribute values which depend on these attribute values are unreachable. Hence, $q$ will not be satisfied. This suffices to prove that situations (a), (b) and (c) are all correct. (3) *Algorithm 6 always terminates.* The only loop is from line 5 to line 9. It always ends because the number of rules and conjuncts in preconditions is finite. **Complexity**. The graph can be created

in polynomial time and the topological sort also takes polynomial time. The total complexity is O(($\sum_{att \in \mathrm{UA}} |\mathrm{SCOPE}_{att}|$) $\times$ $|\Psi|$). □

A minor extension to algorithm 6 can solve $\mathrm{RP}_{=}$ for [rGURA$_1$-set, SR, $\overline{D}$]. Since in this problem, $q$ requires that a state has exactly the same values for each attribute, adding attribute values which are not specified in $q$ is not allowed (attribute values in $q$ should be superset of corresponding attribute values in $\gamma$, otherwise, $q$ is not satisfiable). Before topologically sorting the graph, we do the following preprocessing: (1) `vset` is a set of attribute values which is in $q$ while not in $\gamma$ and (2) detect in the created graph whether there exists a vertex in `vset` which contains incoming edges from vertices not in `vset`. If yes, return false. Otherwise, remove all other vertices not in `vset`. A topological ordering of the vertex in `vset` is a valid plan for the problem.

**Theorem 11.** $\mathrm{RP}_{=}$ *for* [*rGURA*$_1$*-set,* SR*,* $\overline{D}$] *is in P.*

*Proof.* **Correctness**. The only change to algorithm 6 is in the last step (line 22). Because of the nature of $\mathrm{RP}_{=}$, if there does not exist such a `vset` as explained earlier, $q$ is never satisfiable. **Complexity**. As described above, there is only one additional process compared to algorithm 6: to detect `vset` which takes O($\sum_{att \in \mathrm{UA}} |\mathrm{SCOPE}_{att}|$). Thus, the total complexity is O(($\sum_{att \in \mathrm{UA}} |\mathrm{SCOPE}_{att}|$) $\times$ $|\Psi|$) which is polynomial. □

Earlier we've shown rGURA schemes for which reachability problems are either PSPACE-complete or P. Here we briefly go over additional schemes for which the $\mathrm{RP}_{\supseteq}$ is NP-complete and NP. Firstly we look at additional results for $\mathrm{RP}_{\supseteq}$ for [rGURA$_{\mathbf{0}}$-set]. Sizeable results on role reachability in miniARBAC97 can be borrowed directly and utilized for $\mathrm{RP}_{\supseteq}$ for [rGURA$_{\mathbf{0}}$-set]. The reason is that [rGURA$_0$-set] is designed with the same expressive power as ARBAC97 (considering role as one user attribute). Even though there are multiple attributes in [rGURA$_0$-set], their management is independent of each other [94, 124, 129].

For $\mathrm{RP}_{\supseteq}$ in [rGURA$_1$-set, $\overline{N}$], we look at a relaxed restriction compared to $\overline{N}$, PosCanAdd which is defined as: in all rules in can_add, only positive preconditions are allowed. Thus [rGURA$_1$-

set, PosCanAdd] is solvable in P (follows trivially from Theorem 8). We discuss positive precondition in can_add (PosCanAdd) which is defined as: in all rules in can_add, only positive preconditions are allowed.

**Theorem 12.** $\mathrm{RP}_{\supseteq}$ *for problem class* [*rGURA*$_1$*-set, PosCanAdd*] *is solvable in polynomial time.*

*Proof.* Proof follows from theorem 8. □

For $\mathrm{RP}_{\supseteq}$ in [rGURA$_1$-set, SR, $\overline{D}$], if we take out SR restriction, the complexity increases to NP-complete.

**Theorem 13.** $\mathrm{RP}_{\supseteq}$ *for* [*rGURA*$_1$*-set,* $\overline{D}$] *is NP-complete.*

*Proof.* NP-hardness is proved through a reduction from role reachability problem in miniARBAC97 policies without revocation which is NP-complete. In addition, The length of any plan is bounded by $\sum_{att \in \mathrm{UA}} |\mathrm{SCOPE}_{att}|$ as each attribute value can be added at most once. Any plan can be verified in polynomial time. □

Interestingly, even if we further loosen the $\overline{D}$ restriction to $\overline{CD}$ (No Conditional Deletion), the complexity remains to be NP-complete. $\overline{CD}$ is defined as follows: the can_delete relation is empty except for a certain set of values for which the delete rules are unconditionally **true** for some administrative roles in AR.

**Theorem 14.** $\mathrm{RP}_{\supseteq}$ *for* [ *rGURA*$_1$*-set,* $\overline{\mathrm{CD}}$ ] *is NP-complete.*

*Proof.* The proof is borrowed from earlier results of other schemes as shown in [rGURA$_1$-set, $\overline{D}$].

We assume that DV = $\{(att, val) \mid att \in \mathrm{UA} \wedge val \in \mathrm{SCOPE}_{att}\}$ is a set of attribute values that can be deleted without preconditions. If any attribute value appears in any of the preconditions as negative conjuncts, it is safe to remove it from the precondition of those rules if it is also in DV for the purpose of our analysis. Members of administrative roles can delete the values at any time for all users, it is no need to specify them in preconditions. We pre-process all rules in can_add. There are two situations when the pre-process is finished: (1) if there are no negation in can_add rules,

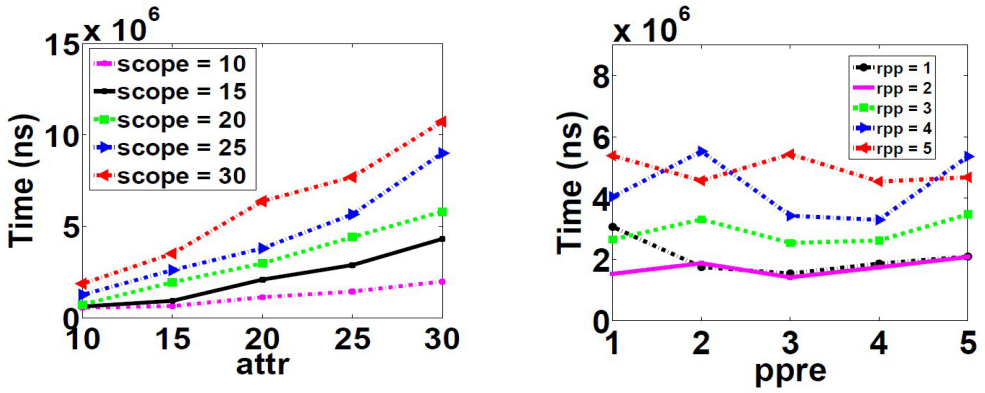
the problem is equivalent to $\mathrm{RP}_{\supseteq}$ in [rGURA$_1$-Set, PosCanAdd] and it has been shown earlier to be solvable in P; (2) if negation exists in the preconditions of some of the rules in can_add, the problem is then equivalent to $\mathrm{RP}_{\supseteq}$ in [rGURA$_1$-set, $\overline{D}$]. The complexity is proved to be NP-complete in theorem 13. □

Recall that we assumed the roles are flat in $\mathrm{AR}$. However, the analysis results also apply in hierarchies $\mathrm{AR}$. The rules specified for each administrative role $ar \in \mathrm{AR}_h$ are prorogated to roles which are senior to $ar$ as they are implicitly assigned with $ar$. The restrictions defined will not be violated by the above process because no new preconditions are introduced.

### 4.4.6 Experimental Results

This section presents the experiments to evaluate the performance of algorithms 5 and 6. We automate administrative rule generation as follows. There are several parameters: $attr$ represents the number of attributes, $scope$ represents the size of each attribute scope, $ppre$ and $npre$ represent the number of positive and negative conjuncts in a precondition respectively, $rpp$ represents the fixed number of can_add rules for each attribute-value pair. For each randomly generated query, $d$ represents the total number of desired attribute and value pairs specified in the query where the desired values are not already available in the initial state. For instance, suppose that $\mathrm{UA}$ = {Prj, Skill} and $\mathrm{U}$ ={Alice}. In the initial state, Prj(Alice) ={search} and Skill(Alice) ={web}. If the query requires Prj(Alice) ={game, mobile}, Skill(Alice) ={web, system, server}, then $d$ would be 4 which is the number of attribute and value pairs not reached in the initial state. We vary all these parameters to generate instances for both algorithms except that $rpp$ applies to only algorithm 5 and $npre$ applies to only algorithm 6. We generate at least one rule for each attribute and value pair (In practice, it is possible that no rule is specified for some attribute and value pairs). Each data reported is an average over 16 instances generated using the same parameter values. In all 16 problem instances, the query is satisfiable and a plan is returned. Times were measured on a 2.53 GHz dual-core CPU with 2 GB RAM.

**Results for Algorithm 5.** The results are in figure 4.5 . Figure 4.5 (a) shows the impact of




(a) Varying attr and scope (Alg 2)

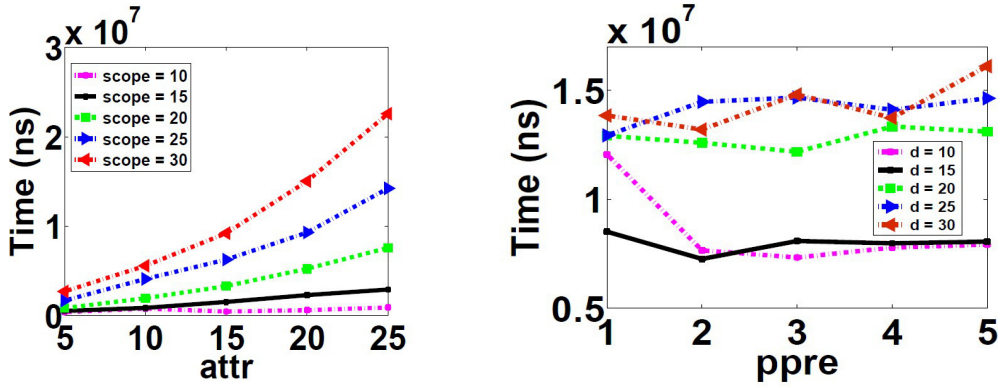
(b) Varying ppre and d (Alg 2)

**Figure 4.5**: *Performance Evaluation of Algorithm 1 With Various Parameters*

the sizes of $attr$ and $scope$ on execution time. We plot the number of attributes on the $x$-axis and time consumed on $y$-axis. We plot a curve for different values of $scope$. In all instances, we use $ppre$ = 5, $rpp$ = 3, $d$ = 20. As expected, the general trend is that the execution time increases with the increase in attribute number and scope size and the change is faster as they become larger. For instance, the execution time for $attr$ = 30 and $scope$ = 30 is nearly 6 times that of when $attr$ = 10 and $scope$ = 30. The major reason is that the total number of rules are increased (recall that we generate at least one rule for each attribute and value pair). However, we believe that the number of the parameters are reasonably small (for example, we do not expect a user to carry 100 attributes) in practical systems and hence the reachability problems can be solved in a very reasonable amount of time.

Figure 4.5(b) evaluates the impact of $ppre$ and $rpp$ parameters. We plot $ppre$ on $x$-axis and time consumed on $y$-axis. We plot multiple curves for different $rpp$ values. In all problem instances, we use $attr$ = 20, $scope$ = 50 and $d$ = 10. Our result shows that there is no trend of time increase with the size of $ppre$ given the same size of $rpp$. However, the total time increases with $rpp$ given the same $ppre$. The major reason is that the total number of rules affects the algorithm complexity and it stays the same when $rpp$ remains the same. (The distance between initial state and the final state satisfying the query is also a factor. Since we set $d$ to be constant, its impact is not visible here.) Note that since the problem instances are randomly generated, given the same $ppre$, the time for

solving the case of a bigger $rpp$ may be lesser than when the $rpp$ is smaller.




(a) Varying attr and scope (Alg 1)

(b) Varying ppre and rpp (Alg 1)

**Figure 4.6**: *Performance Evaluation of Algorithm 2 With Various Parameters*

**Results for Algorithm 6.** The results are in figure 4.6. Figure 4.6(a) shows the impact of $attr$ and $scope$ on the execution time. The $x$-axis shows the number of attributes and we plot multiple curves for different values of scope. In all problem instances, we use $npre$ = 1, $d$ = 5 and $ppre$ = 5. The time consumption increases with attribute and scope size and it increase faster with larger $attr$ and $scope$. The major reason is that a larger number of attribute and values pairs may need to be added to satisfy the query.

Figure 4.6(b) evaluates the impact of $ppre$ and $d$. We plot the positive precondition size on $x$-axis and consider $d$ varying from 10 to 30. In all problem instances, we use $npre$ = 1, $attr$ = 20 and $scope$ = 30. We can see from the figure that the time does not increase significantly with $ppre$ given the same $d$. However, the time increases with $d$ given the same $ppre$. The major reason is that when the difference between the expected attribute values in the query and the values in the initial state is larger, more attribute and value pairs need to be added (recursive back tracking of can_add relations results in more attribute and value pairs to be added). Again, note that in some instances (e.g. black and pink curves in the figure), the time for a bigger $d$ value is lesser than that when $d$ is higher due to the randomness in generation of our administrative rules.

## 4.5 Conclusion

This section presents user attribute administrative models $GURA_0$ and $GURA_1$. They take advantage of the ease of administration in RBAC and are based on existing role based administrative model. We then study the user attribute reachability analysis problem in a restricted GURA model rGURA. We formally define the problem and prove that it is in general intractable. We provide restrictions on the precondition specification in administrative rules to show polynomial time solutions. We provide algorithms which determines reachability as well as generates plans for the query.

# Chapter 5: ABAC FOR CLOUD INFRASTRUCTURE AS A SERVICE IN SINGLE TENANT

In this chapter, we demonstrate the advantage of our proposed ABAC model over existing models in the context of access control in cloud infrastructure as a service (IaaS).

## 5.1 Motivation

Cloud computing is revolutionizing the way businesses and governments manage their information technology (IT) assets. Infrastructure as a Service (IaaS) cloud, where traditional IT infrastructure such as compute resources complemented by storage and networking capabilities are owned and operated by a cloud service provider (CSP) and offered as a service to its customers, is being rapidly adopted by many organizations [1, 4]. Many newer companies such as Netflix, Dropbox and Instagram have eschewed development of proprietary IT infrastructure in favor of CSPs. Established companies such as SAP, GE, Adobe and Domino's Pizza, to name a few, are increasingly migrating to the cloud. In this work, we use the following terminology. We have CSP, organizations and tenants/customers. An *organization* becomes a *tenant* of a particular CSP when it signs up for services with that CSP. We use the terms tenant and customer interchangeably.

Although the functional aspects of IaaS are maturing, the security issues involving this technology are not yet fully understood. Security is often cited as a leading concern in moving to cloud [39, 49, 132, 133], for reasons including uncertainty in continued control over a customer's assets and lack of interoperability between the customer and CSP, and across different CSPs. In particular, when an organization uses the cloud, it faces unfamiliar and non-standard abstractions of access control facilities provided by the CSP over its virtualized resources (compute, storage, networking, etc.). Several challenges arise when an IT infrastructure is outsourced to the cloud. We illustrate them through figures 5.1(a) and 5.1(b). Figure 5.1(a) shows that CSP#1 has multiple tenants. Figure 5.1(b) shows that tenant A is a customer of CSP#1 to CSP#N. What access control requirements arise in this scenario? Consider the resources in IaaS including virtual machines

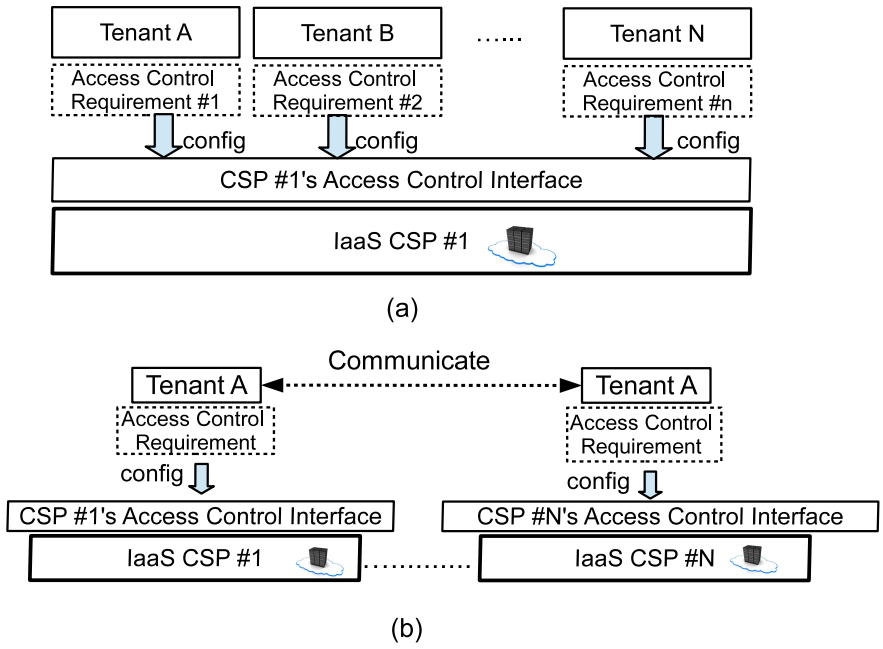



**Figure 5.1**: *Access Control in IaaS Cloud*

(VM), storage and network. Before moving to cloud, an organization specifies policies for its IT personnel over its assets including who can access server rooms, maintain servers, add or remove server capacity, start, stop or take a snapshot of the server, establish a network, modify network configurations, add storage, backup, connect a storage volume to a server, etc. When moving to IaaS cloud, these resources become virtual and remote. Access control policies in the physical world, in part achieved via physical keys, access cards and fingerprints, will need to be comparably specified and enforced in the cloud. That is, in IaaS, there is a need to mimic policies enforcing both physical and digital controls in traditional IT operations.

Two major issues emerge in context of figure 5.1(a). Before moving to cloud, each tenant has its own in-house access control policies. However, when they become a tenant of CSP #1, they are forced to re-think their native policies in terms of access control facilities offered by the CSP. A different problem manifests for the CSP. Each customer will want to configure their own access control policies which may be vastly different from that of others. With an unknown number of potential customers, it is unrealistic to pre-design and implement all kinds of access control models in the cloud or design one by one on demand [133]. The cloud platform should provide a flexible

and intuitive access control framework such that customers can easily configure their own access control policies. Furthermore, in order to distribute their resources (for availability reasons, for example), some organizations may be tenants of multiple CSPs as in figure 5.1(b). In this case, such an organization encounters an additional problem of dealing with multiple different access control interfaces and integrating them.

Current access control models for IaaS in the academic literature and industry are mostly built on role-based access control (RBAC), and fail to tackle the above challenges. RBAC is designed more for easy management as opposed to flexibility and fine-grained control. As explained earlier, the tenants may need to configure different access control policies in a CSP. Consider the following scenarios. Bob, an IT person in tenant A creates a VM. As a creator, he has complete rights over this VM including the ability to start and stop. However, he may wish to grant selected rights over this VM to other IT users in tenant A in the form of an access control list to ease management burden. Consider Alice, an IT person in Tenant B who creates a set of VMs that need to be highly available. First, she wants to ensure that these VMs are managed only by users with a role of "networkOperator". Next, she wants to ensure that not all VMs are in the "stopped" or "underMaintenance" or "underMigration" state simultaneously to guarantee availability of at least 1 VM to serve user requests. Alice also wants to create a storage volume that will be used to store sensitive information. Hence she wants to ensure that this storage volume can only be attached to VMs with an image with the right patches and security updates. That is, a "sensitive" volume can only be attached to "hardened" VMs. Following the scenarios above, one can appreciate the diverse access control needs that may arise in IaaS. Virtualized resources offer functional flexibility and hence to realize the true potential of IaaS there is a need for flexibility in access control.

In spite of its theoretical flexibility [108], RBAC is not appropriate for the scenarios illustrated above since the number of roles will increase significantly. Attribute-based access control (ABAC) is a more natural and intuitive candidate. In ABAC, access control decisions are based on attributes of various entities such as users, subjects and objects. Sufficient abstractions can be built on top of ABAC in order to closely mimic the access control abstractions expected by each tenant. A

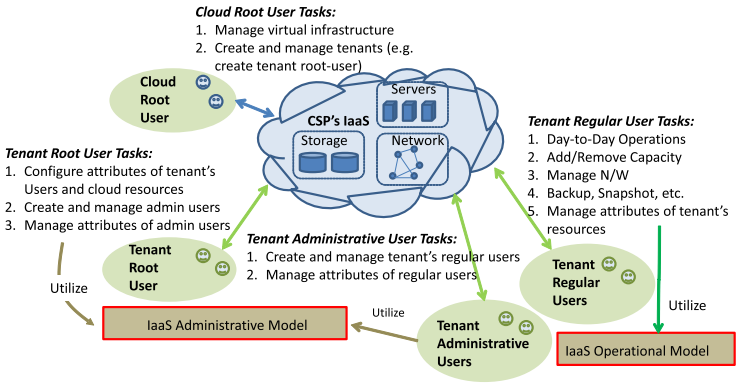



**Figure 5.2**: *Access Control Challenges In IaaS Cloud*

sufficiently flexible ABAC engine at the CSP-side can be configured to enforce each tenant's access control expectations. Thus the requirements of figure 5.1(a) can be met. Figure 5.1(b) requires standardization of the ABAC IaaS capabilities supported by different CSPs, which may emerge over time [69].

Although there has been considerable work in ABAC , what is lacking today is an ABAC framework for IaaS that is intuitive and easy to administer and use, yet with formal foundations. This is vital for successful adoption of ABAC in IaaS given the complex real-world nature of the domain. Our contribution in this chapter is to present a comprehensive ABAC approach for single tenant access control in IaaS cloud. Cross/multi-tenant access control can be achieved by extending our work. We systematically evaluate major access control models in IaaS cloud in academia and industry and show limitations of these models. We then present a discussion on the unique requirements of access control in cloud IaaS and motivate the possibility of ABAC as a candidate model for cloud IaaS. We present a formal framework of ABAC for cloud IaaS. To demonstrate practicality, we conduct implementation and evaluation of the models in the prominent IaaS platform OpenStack [3]. OpenStack is a robust open-source IaaS software for building public, private, community or hybrid clouds that is developed and maintained by a vibrant community with participation from more than 200 world-leading organizations.

## 5.2 Access Control Approach for Cloud IaaS for Single Tenant

Our general approach is illustrated in figure 5.2. The users who interact with cloud IaaS in a management or administrative capacity are categorized into different types (shown in four ovals). A **cloud root user** is a user who manages cloud resources for the CSP. Consideration of policies for this purpose is out of scope. In this model, we focus on tenant-side resource management in the cloud. For ease of presentation we assume there is a single cloud root user.

On the tenant side, we have there three types of IT users. A **tenant root user** represents an IT user who has root access to the tenant. For ease of presentation, we assume that for each tenant, there is only one root user who has full permissions in the tenant. The tenant root user is created by the cloud root user. A **tenant admin user** represents an administrative IT user with administrative permissions in the tenant. Administrative permissions allow management of regular IT users (discussed below) and their attributes in a tenant. A **tenant regular user** is a regular IT user with permissions for standard IT operations such as creating and deleting virtual machines, storage volumes, networks, etc.,on the tenant's behalf. Note that in figure 5.2 an administrative ABAC model is necessary to guide the tasks of tenant root and administrative IT users while an operational ABAC model is necessary for managing the tasks of regular IT users. The administrative model facilitates creating and updating attributes while the operational model facilitates specifying authorization policies that control the actions of regular IT users. We emphasize that non-IT users of a tenant who only interact with VMs and services are not considered in figure 5.2. They do not manage any cloud IaaS resources and are controlled by access control mechanisms within the VMs and within applications running in VMs. These users are beyond the scope.

The general process is as follows. In order to use cloud services, the first step is that an organization's representative (say Alice) obtains an account from the CSP typically via some automated process which is a surrogate for the cloud root user. Thereby the organization becomes a tenant of the CSP with Alice as that tenant's root user. Now it is not practical for Alice to create and manage all the resources herself. Instead, in the second step Alice sets up tenant specific access control and

administrative policies using the the CSP-provided facilities, and creates some number of tenant admin users. Then, the tenant admin users create regular IT users and administer their attributes. Finally, regular IT users can then create and manage virtual resources as per the policies specified by the tenant root user and attribute values administered by tenant admin users. Each tenant goes through a similar process.

## 5.3 Related Work

In this section we first review access control models in two leading cloud IaaS platforms: OpenStack (the Grizzly release) and Amazon Web Services (AWS). We also review models discussed in the IaaS access control literature. Since our scope is restricted to access control issues within a single tenant, we omit discussion of features in platforms such as AWS that concern multiple tenants as well as research papers on cross-tenant issues. Note that the concept of an "account" in AWS and that of a "project" in OpenStack are the same as the concept of a "tenant" for our purpose. In our discussion below, we uniformly use the term tenant.

### 5.3.1 Access Control Models in Cloud IaaS

#### Access Control in Amazon Web Service

AWS is the commercially dominant cloud IaaS platform. Example services include elastic compute cloud (EC2), simple storage service (S3) and elastic block storage (EBS). Related reviews of AWS can be found in [113, 144]. We discuss the AWS Identity and Access Management (IAM) component which concerns access control as related to the above cloud services.

The access control model structure is shown in figure 5.3(a). The major components in each tenant are *Users*, *Objects*, *Groups*, *Policies*, *Actions*, *Conditions*, *User-Group-Assignment*, *User-Policy-Assignment* and *Policy-Group-Assignment*. A *permission* is defined in the format of *Action* on *Object* under certain *Conditions*. *Conditions* are in the form of key-value pairs. Each key-value pair can be one of following types including String, Numeric, Date & Time, Boolean,

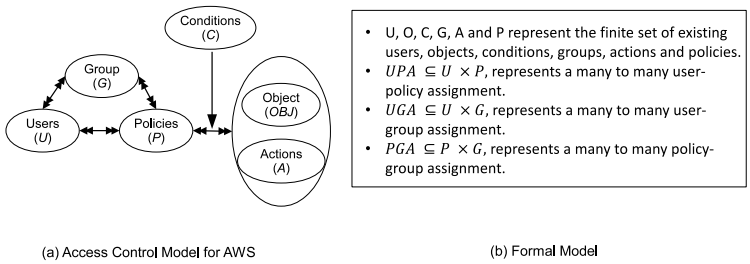



**Figure 5.3**: *Amazon Web Service Access Control in Single Tenant*

and IP-address. For example, the condition "DateLessThan: {aws : CurrentTime : "2013-09-01T00:00:00Z"}" uses the Date & Time type DateLessThan condition restricting the requests to be made before Sep 1, 2013 [144]. A *group* is a similar concept as the $role$ in RBAC. It is associated with a set of policies which define a set of permissions. Users can either be assigned to groups or directly to policies. Each policy is specified using the policy specification language provided by AWS. Each policy consists of a number of $statements$ which contain a description of the requests they apply to, plus an effect, which may be allow or deny. Each $statement$ contains lists of $actions$, lists of $resources$ and lists of $principals$, plus a number of $conditions$ which must be met. If multiple statements match a request, then deny effects take precedence over permit effects. If no statements match, then the effect is referred to as a soft deny: that is to say that final effect will be deny unless another policy has an effect of permit. The same evaluation logic applies to multiple policies.

In summary, IAM supports an RBAC-style model as permissions are grouped and assigned to users. The only configuration point is to specify groups and define permissions for the group. The formal model is summarized in figure 5.3(b). While this model is slightly more flexible than RBAC (by extending permissions with policy and condition) it lacks the flexibility necessary to handle scenarios discussed in the previous sections. The number of groups can increase dramatically for fine-grained access control policies.

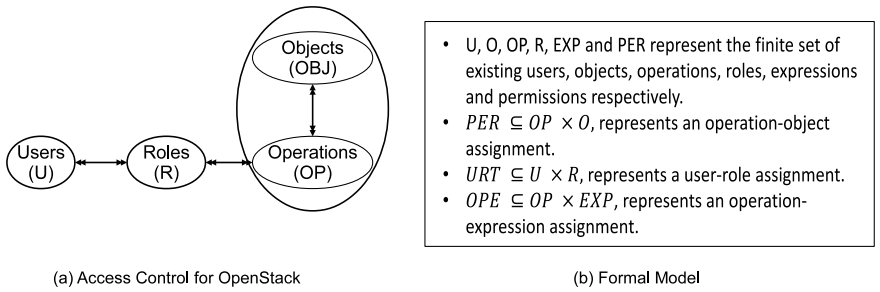



**Figure 5.4**: *OpenStack Access Control in Single Tenant*

### Access Control in OpenStack

OpenStack is an open source IaaS software adopted by many cloud providers such as Rackspace, IBM, Dell and RedHat. The structure of OpenStack access control model is shown in figure 5.4(a), where we omit cross-tenant and multi-tenant features as mentioned earlier. The major components in each tenant include *Users*, *Objects*, *Roles*, *Operations*, *Permissions*, and *Expressions*. A role is a string such as "professor" or "manager". An operation and object are respectively the same as the action and object in the AWS model. A permission is an operation on an object. Each user may be assigned to multiple roles. Each operation is associated with a boolean expression specified using the usual $\wedge$ and $\vee$ operators on terms of the form r and $\bar{\text{r}}$ where r is a role. The expression is evaluated for a user by interpreting r to be true if the user is assigned with role r and $\bar{\text{r}}$ to be true if user is not assigned with role r. For example, consider "compute : create_instance: $\text{r}_1 \wedge \text{r}_2 \wedge \bar{\text{r}}_3$". It says that the user is authorized to perform the compute : create_instance operation if he is assigned with roles $\text{r}_1$ and $\text{r}_2$ and is not assigned with role $\text{r}_3$. If a user tries to operate on an object, the policy check is as follows: the user's roles in the same tenant as the object should satisfy the expression associated with the operation. For example, a user is assigned with roles $\{\text{r}_1, \text{r}_2\}$ in tenant $\text{t}_1$ and roles $\{\text{r}_1, \text{r}_3\}$ in tenant $\text{t}_2$. According to the above policy, he is authorized to perform compute : create_instance operation in $\text{t}_1$ but not in $\text{t}_2$. Access control in OpenStack is coarse grained because only operation level authorization is supported. If an operation such as compute : stop_instance is authorized to a user, that user can stop any VM instance in the tenant. Authorization of this operation for particular tenant VMs and not others cannot be specified. The

formal model is summarized in figure 5.4(b).

As this model is also RBAC-based, it has some of the same drawbacks as AWS. In addition, current OpenStack access control facilities have two major issues. Firstly, all tenants share the same policy for all OpenStack components such as Nova (compute), Keystone (identity and access management) and Glance (VM image repository), and these policies can only be configured by CSP root users. There is no mechanism to support customized access control policies for individual tenants. This means that all tenants should use the same set of roles and role-operation assignment, which is unrealistic. Secondly, certain administrative operations can only be assigned across all tenants rather than on a tenant-by-tenant basis. Operations such as create user, delete user and assign a role to a user can only be done by CSP root users rather than being delegated to tenant root users so that each tenant can have autonomy of control over its resources and users.

### 5.3.2 Other IaaS Models in the Literature

Most of the access control models for IaaS in the literature are based on RBAC. Wu *et al.* [144] designed and implemented access control as a service (ACaaS) based on RBAC to extend the AWS access control model. ACaaS$_{RBAC}$ introduced role hierarchies, sessions, constraints and an administration model. Domain based access controls ($dCloud$) [126, 127] were proposed based on the original RBAC model. The general idea is to group related resources and users in the same domain such that administration can be delegated to each domain. In this way, distributed administration can be achieved. Daniel *et al.* provided an authorization system to control the execution of virtual machines (VMs) to ensure that only administrators and owners could access them [104]. Berger *et al.* [25] proposed an authorization model based on both RBAC and security labels to control access to shared data, VMs, and network resources. Almutairi *et al.* [10] proposed a distributed access control architecture for cloud computing. Chadwick *et al.* [40] proposed fine-grained access control in private cloud. This work mainly focus on federated identity management and is specific to cloud storage systems. Takabi *et al.* [135] proposed a comprehensive security framework for cloud computing environments. The work provides a big picture of security requirements in cloud.

**Table 5.1**: *Requirements for Cloud IaaS Access Control*

| Models | Flexibility | Least Privilege | Fine-grained Access Control | Automation | Tenant Management of Users |
|---|---|---|---|---|---|
| OpenStack | NO | NO | NO | NO | NO |
| AWS | NO | NO | YES | NO | YES |
| RBAC | YES | YES | NO | NO | NA |
| $\text{IaaS}_{\text{op}}$ | YES | YES | YES | YES | YES |

A common drawback of RBAC-related models is that they do not tackle the challenge of flexible and heterogeneous access control requirements from different tenants in IaaS.

There are limited number of initial works on using ABAC for cloud in general as opposed to specifically for IaaS. Cha *et al.* [38] proposed ABAC in cloud computing environment. Iqbal *et al.* [73] proposed semantic-enhanced ABAC for cloud services. Danwei *et al.* [52] proposed access control for cloud service based on UCON. However, these works are neither focused on IaaS access control nor present a formal ABAC model. In addition, they did not present the functionalities for the entire lifecycle of tenant management (e.g., tenant creation, policy configuration, administration, etc.). There is, of course, a considerable body of literature on ABAC in general beyond cloud IaaS [80, 149]. The ABAC literature focuses on issues such as enforcement architectures and policy languages. Policy languages, while important, by themselves are not sufficient. We need models that guide policy specification using these languages.

## 5.4 Requirements of Access Control in IaaS Cloud

In this section, we discuss the requirements of access control model in cloud IaaS and motivate the usage of ABAC. We first list the requirements of access control in IaaS and then compare the capability of existing models in satisfying these requirements. For this purpose, we summarize the requirements in table 5.1. The columns show the list of requirements and the rows show the list of existing models. This table shows whether each requirement can be satisfied by existing models. We discuss them below.

- **Flexibility**. Among all requirements, flexibility is critical for access control in IaaS cloud.

As we discussed in section 5.1, the access control engine in cloud should be able to provide a wide variety of access control features for customers. Since different customers may demand different access control requirements, the cloud provider should provide access control model that is flexible and intuitive. Hence classical models such as DAC and MAC are fundamentally limited in this regard. While RBAC has been successful in enterprises, most of the existing cloud systems use some augmented version of RBAC. It is reasonable to expect that RBAC will be further enriched for usage in cloud IaaS in the future. Since ABAC is a generalized version of RBAC (multiple attributes instead of a single attribute "role"), we expect ABAC to satisfy the flexibility requirements of cloud IaaS.

- **Least Privilege**. Users manage resources by logging in to the system. To be secure, they want to be associated with only the necessary permissions instead of full permissions depending on their tasks during that specific session. For example, to list active virtual resources through public network, they don't want to be associated with the permissions to change the status of the virtual machines such as turn off and restart virtual machines. In this way, even if the session is hijacked, the VMs cannot be modified. This requires the concept of session and the configuration points to configure session policy. In both AWS and OpenStack, users cannot choose the permissions in a session. Instead, they always have full permissions in all sessions.

- **Fine-grained access control**. Different users have different sets of permissions and this requires fine-grained control over the resources. However, the advantages of RBAC is based on the assumption that the number of roles is much smaller than the number of users. This will cause role-explosion problem if each user has different sets of permissions. OpenStack only provides coarse-grained access control at the operation level. AWS provides fine-grained access control by considering time, location, address and so on and those can be captured by ABAC authorization policies.

- **Automation**. This feature enables users to create objects and set attributes on them. Those

attributes are constrained by the attributes of the user himself and can be used in authorization policy for future accesses from other users. For example, an architect from Email department wants to create a server and label it as Email department. Later on, other architects belonging to the Email department can access this server. No manual assignment of the new permissions (i.e., operations on this new virtual machines) to other authorized users is needed in ABAC. However, in RBAC, these permissions have to be manually associated with certain roles, which is cumbersome because of the number of potentially new permissions (depending on the number of newly created objects) as well as the expensive cost of role engineering. In AWS, users are allowed to tag the objects and those tagged information can be used in authorization policy. However, this tag is not constrained by attributes of the user, i.e., the user can tag the resources using any name and value pairs.

- **Tenants full control of their users.** Cloud service provider should support the ability for tenant to fully manage their own users in the cloud. For this purpose, each tenant should be able to configure an administrative user who has full permissions within the assigned tenant. In OpenStack, only the cloud administrative user from cloud service provider can add users to tenants and assign roles. AWS allows tenant to manage their own users.

We can see that no existing model systems satisfy the critical requirement specified in the table. With our theoretical work in chapter 3 as a foundation, our proposed ABAC in this chapter captures all requirements in a single model and distinguishes itself from other similar ABAC models.

## 5.5 Formal IaaS Models

In this section, we provide a formal specification of two models for IaaS cloud: the operational model $\mathrm{IaaS}_{\mathrm{op}}$ and the administrative model $\mathrm{IaaS}_{\mathrm{ad}}$. $\mathrm{IaaS}_{\mathrm{op}}$ is built on the unified attribute based access control model $\mathrm{ABAC}_{\alpha}$ introduced in chapter 3. It enables specification of authorization policies for day to day operations of the tenant regular IT users (see figure 5.2). The $\mathrm{IaaS}_{\mathrm{ad}}$ model is built on the generalized user-role assignment model GURA. It enables specification of policies

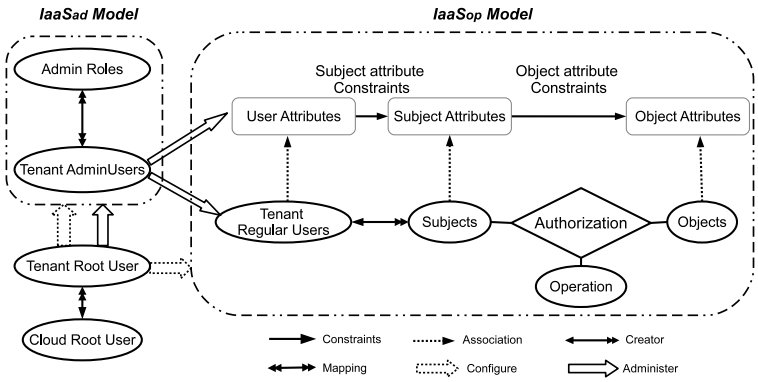



**Figure 5.5**: $\mathrm{IaaS_{op}}$ *and* $\mathrm{IaaS_{ad}}$ *For Single Tenant Access Control*

for modifying attributes of tenant regular IT users (see figure 5.2). The tenant root users utilize the administrative model for specifying policies for attribute updates of tenant regular IT users by tenant administrative IT users.

We choose to build upon $\mathrm{ABAC}_\alpha$ since it provides an intuitive collection of policy configuration points. The general idea of $\mathrm{IaaS_{op}}$ is as follows. Users, subjects and objects are associated with attributes. Subject and object attributes are set and modified by users and subjects respectively. The modification of subject and object attributes are guided by constraint policies specified in the model. A subject is able to perform an action on an object (e.g. stop a VM) as allowed by the authorization policies in the model.

User provisioning and attribute assignment is covered in the GURA model. We build on the GURA model because of the lack of administrative models in the literature for ABAC, GURA being one of the first formally specified models in this regard. In this model, administrative permissions such as add and assign user attributes, are associated with administrative roles. Tenant administrative IT users are then assigned these roles and obtain administrative permissions. In GURA, it does not discuss the permission of adding and deleting users. In this chapter, we allow all administrative users to add and delete users. Note that the $\mathrm{IaaS_{ad}}$ model can be generalized into an ABAC instance, that is ABAC based administrative model for an ABAC operational model. We start by providing a simple and clear approach using $\mathrm{IaaS_{ad}}$ giving us a role-based administrative model for an ABAC operational model. We define the core components and specify operations for

configuring policies in a single tenant in the context of IaaS cloud. We introduce related components first and then present the formal model and operations.

### 5.5.1 The Operational Model $\mathrm{IaaS_{op}}$

#### Components

The structure of $\mathrm{IaaS_{op}}$ is shown in the right part of figure 5.5. The $\mathrm{IaaS_{op}}$ model is configured by the tenant root user. We use "configure" to signify the operation of system architects who design the elements in the system based on formal models. The major components are regular users ($\mathrm{U}$), subjects ($\mathrm{S}$), objects ($\mathrm{O}$), user attributes ($\mathrm{UA}$), subject attributes ($\mathrm{SA}$), object attributes ($\mathrm{OA}$), operations ($\mathrm{P}$), subject ($\mathrm{ConstrSub}$) and object ($\mathrm{ConstrObj}$) attribute constraint and authorization policies ($\mathrm{Authorization}$).

An **attribute** is a function which takes an entity and returns certain properties of the entity. Each attribute is associated with a finite set of atomic values as its scope. There are two types of attributes: $\mathrm{set}$ valued and $\mathrm{atomic}$ valued. The major difference is $\mathrm{set}$ valued attributes can take multiple values from their scope while $\mathrm{atomic}$ valued attributes take a single value from their scope. Example $\mathrm{set}$ valued attributes are `role` and `division` and example $\mathrm{atomic}$ valued attributes are `clearance` and `level`.

A **user** is an entity which interacts with the cloud. We have introduced cloud root user, tenant root user, tenant administrative user and tenant regular user in section 5.2. User attributes reflect the properties of users. In this model, only regular users are associated with attributes since we employ ABAC only for the operational component of IaaS. A **subject** is a program or process created by users to access the resources on behalf of the users. Only the creator can terminate a subject. For example, when a user creates a connection from his mobile phone to the cloud, the connection is a subject. He can also create another concurrent subject from his laptop. A subject carries attributes which can be used for authentication and authorization. Examples are `ip`, `timestamp` and `networktype` (public, private, etc.). Besides those, there may be attributes inherited from

user attributes. In some systems, subjects are associated with a signed credential of the users' information and can be represented as a token (OpenStack [3]) or access key and secret key pair (Amazon Web Service [1]). The cloud authenticates and authorizes all requests submitted by this subject based on information included in the token or access key and secret key pair. Subjects created by a user may take attributes and values that differ from that of its user. **Subject attribute constraint policy** specifies the constraints on subject attributes when users create subjects and set values for subject attributes. For example, if a MAC policy is required, the subject's `clearance` should always be lower or equal to that of the user. A user at $top\ secret$ `clearance` level can log in to the system at either $secret$ or $unclassified$ `clearance`. In this way, user accesses the tenant with least privilege.

**Objects** represent the virtual resources in cloud. Examples are virtual machines, virtual networks, images, volumes and storages. Objects are created by subjects on behalf of users. Objects are also associated with attributes and those attributes are set and modified by their owner who creates them. There is a difference between this model and the original $\text{ABAC}_\alpha$ model in that different types of objects may be associated with different sets of object attributes. For example, volumes may be associated with `size` and `attachedVM` attributes while it is not natural to associate virtual machines with these attributes. When a user sets or modifies the attributes of objects, there are also constraints. **Object attribute constraint policy** specifies the constraints on the values that object attributes may take at and post creation time of the object. An example object attribute constraints policy may require that when a subject creates a volume, the volume should be labelled with the same `division` (or `tenant`) as the subject and the volume's owner is set to the subject's creator.

An **operation** represents an access mode on objects. Operations are defined by the CSP and will vary across different CSPs. For example, operations on virtual machines include create, start, stop and resize. Operations on images include upload and list.

**Authorization policy** specifies policies for evaluating requests made by subjects (on behalf of regular IT users). It is specified based on attribute values of the involved subject and object. It returns either **true** or **false** meaning the request is authorized or rejected. For example, if a

**Table 5.2**: *Basic Sets and Functions for* $\mathrm{IaaS_{op}}$ *Model*

U, S and O represent finite sets of *existing* regular users, subjects and objects respectively.

UA, SA and OA represent finite sets of user, subject and object attribute functions respectively.

$\mathrm{objType}$: $\mathrm{O} \rightarrow \mathrm{OT}$. For each object, $\mathrm{objType}$ gives its type.

$\forall\, t \in \mathrm{OT}$, $\mathrm{O}_t = \{\mathrm{obj} \mid \mathrm{obj} \in \mathrm{O} \wedge \mathrm{objType(obj)} = t\}$, represents objects of type $t$.

oaType: $\mathrm{OA} \rightarrow \mathrm{OT}$. For each object attribute, oaType gives its type.

$\forall\, t \in \mathrm{OT}$, $\mathrm{OA}_t = \{\mathrm{oa} \mid \mathrm{oa} \in \mathrm{OA} \wedge \mathrm{oaType(oa)} = t\}$, represents object attributes of type $t$.

$\mathrm{SubCreator}$: $S \rightarrow U$. For each subject $\mathrm{SubCreator}$ gives its creator.

For each $att$ in $\mathrm{UA} \cup \mathrm{SA} \cup \mathrm{OA}$, SCOPE represents the attribute's scope, a finite set of *atomic* values.

$\mathrm{attType}$: $\mathrm{UA} \cup \mathrm{SA} \cup \mathrm{OA} \rightarrow \{\mathrm{set}, \mathrm{atomic}\}$. Specifies attributes as $\mathrm{set}$ or $\mathrm{atomic}$ valued.

P represents finite set of operations.

Each attribute function maps elements in U, S and O to $\mathrm{atomic}$ or $\mathrm{set}$ values.

$$\forall ua \in \mathrm{UA}.\, ua : \mathrm{U} \rightarrow \begin{cases} \mathrm{SCOPE}_{ua} & \text{if attType}(ua) = \text{atomic} \\ 2^{\mathrm{SCOPE}_{ua}} & \text{if attType}(ua) = \text{set} \end{cases}$$

$$\forall sa \in \mathrm{SA}.\, sa : \mathrm{S} \rightarrow \begin{cases} \mathrm{SCOPE}_{sa} & \text{if attType}(sa) = \text{atomic} \\ 2^{\mathrm{SCOPE}_{sa}} & \text{if attType}(sa) = \text{set} \end{cases}$$

$$\forall t \in \mathrm{OT}. \forall oa \in \mathrm{OA}_t. oa : \mathrm{O}_t \rightarrow \begin{cases} \mathrm{SCOPE}_{oa} & \text{if attType}(oa) = \text{atomic} \\ 2^{\mathrm{SCOPE}_{oa}} & \text{if attType}(oa) = \text{set} \end{cases}$$

user requests to stop a virtual machine, the user and the virtual machine should be of the same `division`.

### Formal Definition

The formal operational model $\mathrm{IaaS_{op}}$ is summarized in table 5.2. This model is configured by tenant root users. The **basic sets and functions** in $\mathrm{IaaS_{op}}$ model are as follows: $\mathrm{U}$, $\mathrm{S}$ and $\mathrm{O}$ represent finite sets of tenant regular users, subjects and objects respectively. There is one distinguished attribute for object, $\mathrm{objType}$, which maps objects to their respective types. $\mathrm{OT}$ represents the scope of this function and thus $\mathrm{O}_t$ represents the set of objects of type $t$. We define a finite set of object types based on the current architecture of cloud IaaS. For example, $\mathrm{OT}$ = {`vm`, `file`, `image`, `network`, `volume`}. $\mathrm{UA}$, $\mathrm{SA}$ and $\mathrm{OA}$ represent finite sets of user, subject and object attributes respectively. oaType is a function mapping each object attribute to the type of objects it applies. For each $t$ in $\mathrm{OT}$, $\mathrm{OA}_t$ represents the attributes defined for objects of type $t$. These could be $\mathrm{atomic}$ or $\mathrm{set}$ valued as determined by the type of the attribute function ($\mathrm{attType}$). For

each attribute, $\mathrm{SCOPE}$ represents the finite set of atomic values it can take. $\mathrm{SubCreator}$ is a distinguished attribute for subject. It maps each subject to the user who creates it (an alternate would be to treat this attribute as a function in $\mathrm{SA}$) and thus the scope of this function is $\mathrm{U}$. Finally, $\mathrm{P}$ represents finite set of operations.

There are three **policy configuration points** in the $\mathrm{IaaS_{op}}$ model. $\mathrm{Authorization}$, $\mathrm{ConstrSub}$, $\mathrm{ConstrObj}$ and $\mathrm{ConstrObjMod}$ represent authorization policy, subject attribute constraint policy at creation and modification time, and object attribute constraints policy at creation time and modification time. We need a grammar to express these policies which we adopt from chapter 3. The structure of subject attribute constraint policy is specified by comparing the proposed value of subject attributes with the attribute values of the creating user. The structure of object attribute constraint policy is conjunction and disjunction connection of comparison between the proposed object attribute value and subject attribute value. Similar to subject and object attribute constraints, authorization policy specifies comparison between attributes of the involved subject and object. We provide examples of all policies in the next section together with operations of this model.

**Operations for tenant regular users**

Operations can be submitted by tenant regular users to the cloud. The cloud system updates state according to the operations if it is authorized. We call the user who submits the operation the "requester". The following operations are authorized to regular users if the evaluation result from authorization policy is $\mathtt{true}$. We adopt the precondition checking for each operation from chapter 3. We briefly introduce the format of each operation in table 5.3. Operation 1 creates a subject, operation 2 creates an object. Operations 3 and 4 modify subject and object attributes. Operation 5 represents any of the regular operation on resources such as starting a *server*, creating a *volume*, etc. In these operations, $\mathrm{OASET}_t$ represent the data type in which each element represents an attribute assignment for all object attributes for object type $t$. Formally, for each t $\in$ OT,

**Table 5.3**: *Complete List of Operations for Tenant Regular Users*

| Operations | Updates |
|---|---|
| 1. createSubject(req:U, sub:NAME, saset:SASET) | $S' = S \cup \{sub\}$, for each (sa, val) $\in$ SASET, sa(sub) = val, $\mathrm{SubCreator}(s) = req$ |
| 2. createObject(sub:S, obj:NAME, oaset:$\mathrm{OASET}_t$, t:OT) | $O' = O \cup \{obj\}$, $\mathrm{objType}(obj) = t$, for each (oa, val) $\in \mathrm{OASET}_t$, oa(obj) = val |
| 3. modifySubAttr(req:U, sub:NAME, saset:SASET) | For each (sa, val) $\in$ SASET, sa(sub) = val |
| 4. modifyObjAttr(sub:S, obj:NAME, oaset:$\mathrm{OASET}_t$, t:OT) | For each (oa, val) $\in \mathrm{OASET}_t$, oa(obj) = val |
| 5. Operation(sub:S, obj:O) | None |

$$\mathrm{OASET}_t = \bigcup_{\forall oa \in OA \wedge objType(oa)=t} \text{OneElement}(\mathrm{OASET}_{oa}) \text{ where}$$

$$\mathrm{OASET}_{oa} = \begin{cases} \{oa\} \times \mathrm{SCOPE}_{oa} \text{ if } \mathrm{attType}(oa) = \text{atomic} \\ \{oa\} \times 2^{\mathrm{SCOPE}_{oa}} \text{ if } \mathrm{attType}(oa) = \text{set} \end{cases}$$

### 5.5.2 The Administrative Model $\mathrm{IaaS}_{\mathrm{ad}}$

#### Components

Recall that cloud root user (CRU) and tenant root user (TRU) have been defined in section 5.2. The structure of $\mathrm{IaaS}_{\mathrm{ad}}$ model is shown in the left part of figure 5.5. This model is configured by tenant root user and administered by tenant root users. Here "administer" signifies operations such as creating users and modifying user role assignment. The major components are tenant administrative users (TAU), administrative roles (AR) and user role assignment (UAR). **Administrative roles** are associated with administrative permissions such as add, delete and assign user attributes. **Administrative users** are associated with administrative roles and thus obtain the associated permissions.

#### Formal Definition

The formal administrative model $\mathrm{IaaS}_{\mathrm{ad}}$ is summarized in part I in table 5.4. The **basic sets and functions** are CRU, TRU, TAU, AR and UAR. CRU and TRU represent the cloud root user and tenant root user respectively. TAU represents the set of administrative users, AR represents a finite

set of administrative roles, and $\mathrm{UAR}$ represents user administrative role assignment.

There is one **configuration point** for $\mathrm{IaaS_{ad}}$ model which is **administrative policies**. They specify the condition under which certain administrative roles can modify user attributes. The precondition is specified based on the attribute value of the user whose attributes are to be modified. For example, users with $manager$ role can assign IT architects to a different $division$ if they are currently assigned with $employee$ role in that $division$. $\mathrm{AdminPolicy}$ represents finite set of administrative policies. Again, we need a grammar for specifying these policies which we adopt from [78]. For each attribute `att` in $\mathrm{UA}$, $\mathsf{can\_add}_{\mathtt{att}}$ is a set containing tuples in the format of $(ar, condition, values)$ where $ar$ is one of the administrative roles, $condition$ is a boolean expression specified using the current values of attributes of the regular IT users, and $values$ represents a set of value that can be added. It means that administrative role $ar$ can add (more operations will be introduced in section 5.5.2) any value from $values$ to the attribute `att` of user whose attributes satisfy the precondition $condition$. can_add is defined for $\mathrm{set}$-valued attributes. Similarly, can_delete is defined for $\mathrm{set}$-valued attributes representing policies for delete permission. Finally, can_assign is defined for $\mathrm{atomic}$-valued attributes.

### Operations

We define a set of operations to maintain the sets and relations defined above and in $\mathrm{IaaS_{op}}$ model. We provide a list of the operations in part II in table 5.4. We illustrate the operations based on a concrete senario of an organization migrating infrastructure to the cloud. The senario is defined below:

**Scenario 1.** A university called **TechEdu** wants to create their data center in the cloud. The university contains certain number of colleges under which there are several departments. The university, each college and each department maintain certain amount of resources of each type. For consistency, all departments, colleges and the university are called entity. IT architects are added as users so that they can manage their resources. They are assigned with one or multiple resource type and entity pairs. Due to security concern, three policies are required to be enforced:

**Table 5.4**: *Formal Definition For* $\mathrm{IaaS_{ad}}$ *Model*

| **Part I. Basic Sets and Functions** | |
|---|---|
| CRU, TRU represent the cloud root user and tenant root user respectively.<br>TAU represents finite set of tenant administrative users.<br>AR represents a set of administrative roles and UAR represent user-role assignment, i.e., UAR $\subseteq$ TAU$\times$ AR. | |
| **Part II. Complete List of Operations** | |
| Operations | Updates |
| **1. Operations for Cloud Root User** | |
| 1.1 createTenant(req:CRU, tenant:NAME) | T′ = T $\cup$ {tenant} |
| 1.2 createRootUser(req:CRU, u:NAME, tenant:T) | TRU = $\emptyset$, TRU = {u} |
| 1.3 removeTenant(req:CRU, tenant:NAME) | T′ = T $\setminus$ {tenant} |
| **2. Operations for Tenant Root User** | |
| 2.1 createUserAttr(req:TRU, ua:NAME, type: {set, atomic}) | UA′ = UA $\cup$ {ua}, attType(ua) = type |
| 2.2 createUserAttrScope(req:TRU, ua:UA, value:NAME) | $\mathrm{SCOPE}'_{ua}$ = $\mathrm{SCOPE}_{ua}$ $\cup$ {value} |
| 2.3 removeUserAttrScope(req:TRU, ua:UA, value:$\mathrm{SCOPE}_{ua}$) | $\mathrm{SCOPE}'_{ua}$ = $\mathrm{SCOPE}_{ua}$ $\setminus$ {value} |
| 2.4 createSubAttr(req:TRU, sa:NAME, type: {set, atomic}) | SA′ = SA $\cup$ {sa}, attType(sa) = type |
| 2.5 createSubAttrScope(req:TRU, sa:SA, value:NAME) | $\mathrm{SCOPE}'_{sa}$ = $\mathrm{SCOPE}_{sa}$ $\cup$ {value} |
| 2.6 removeSubAttrScope(req:TRU, sa:NAME, value:$\mathrm{SCOPE}_{sa}$) | $\mathrm{SCOPE}'_{sa}$ = $\mathrm{SCOPE}_{sa}$ $\setminus$ {value} |
| 2.7 addSubConstr (req:TRU, policy:POLICY) | ConstrSub′ = ConstrSub $\cup$ {policy} |
| 2.8 removeSubConstr (req:TRU, policy:POLICY) | ConstrSub′ = ConstrSub $\setminus$ {policy} |
| 2.9 createObjAttr (req:TRU, oa:NAME, type:{set, atomic}, oat: OT) | OA′ = OA $\cup$ {oa}, attType(oa) = type, oaType(oa) = oat |
| 2.10 createObjAttrScope(req:TRU, oa:OA, value:NAME) | $\mathrm{SCOPE}'_{oa}$ = $\mathrm{SCOPE}_{oa}$ $\cup$ {value} |
| 2.11 removeObjAttrScope(req:TRU, oa:OA, value:NAME) | $\mathrm{SCOPE}'_{sa}$ = $\mathrm{SCOPE}_{sa}$ $\setminus$ {value} |
| 2.12 addObjConstr (req:TRU, policy:POLICY) | ConstrObj′ = ConstrObj $\cup$ {policy} |
| 2.13 removeObjConstr (req:TRU, policy:POLICY) | ConstrObj′ = ConstrObj $\setminus$ {policy} |
| 2.14 addObjConstrMod (req:TRU, policy:POLICY) | ConstrObjMod′ = ConstrObjMod $\cup$ {policy} |
| 2.15 removeObjConstrMod (req:TRU, policy:POLICY) | ConstrObjMod′ = ConstrObjMod $\setminus$ {policy} |
| 2.16 addAuthz (req:TRU, policy:POLICY) | Authorization′ = Authorization $\cup$ {policy} |
| 2.17 removeAuthz (req:TRU, policy:POLICY) | Authorization′ = Authorization $\setminus$ {policy} |
| 2.18 createAdminRole(req:TRU, role:NAME) | AR ′ = AR $\cup$ {role} |
| 2.19 createAdminPolicy(req:TRU, policy:POLICY) | AdminPolicy′ = AdminPolicy $\cup$ {policy} |
| 2.20 removeAdminPolicy(req:TRU, policy:POLICY) | AdminPolicy′ = AdminPolicy $\setminus$ {policy} |
| 2.21 addAminUser(req:TRU, u:NAME) | TAU′ = TAU $\cup$ { u} |
| 2.22 removeAminUser(req:TRU, u:TAU) | TAU′ = TAU $\setminus$ {u} |
| 2.23 addAminUserRole(req:TRU, u:TAU, r:AR) | UAR′ = UAR $\cup$ {(u, r)} |
| 2.24 removeAminUserRole(req:TRU, u:TAU, r:AR) | UAR′ = UAR $\setminus$ {(u, r)} |
| **3. Operations for Tenant Administrative Users (chapter 4)** | |
| 3.1 addUser(req:TAU, user:NAME) | U ′ = U $\cup$ {user} |
| 3.2 removeUser(req:TAU, user:U) | U ′ = U $\setminus$ {user} |
| 3.3 add(req:TAU, tuser:U, att:UA, value:$\mathrm{SCOPE}_{att}$) | att(tuser)′ = att(tuser) $\cup$ {value} |
| 3.4 delete(req:TAU, tuser:U, att:UA, value:$\mathrm{SCOPE}_{att}$) | att(tuser)′ = att(tuser) $\setminus$ {value} |
| 3.5 assign(req:TAU, tuser:U, att:UA, value:$\mathrm{SCOPE}_{att}$) | att(tuser)′ = value |

(1) When users connect to the cloud, they can choose which entity and type pairs to activate in that session; (2) when subjects create new resources, they should be labeled with the entity and type the user is assigned and (3) IT architects are authorized to only access the resources if they are assigned with the type and same entity of the resource.

To start, assume that we have the following initial state: Rack is a cloud service provider and Alice is the cloud root user who manages resources in Rack. Bob is a representative from the organization TechEdu planning to create university servers and services in the cloud.

- **Category I. Operations For Cloud Root User**

Firstly, we define operations for cloud root user. These operations will only be authorized if the requester ($\mathrm{req}$) is the cloud root user. That is, $\mathrm{req}$=$\mathrm{CRU}$, where $\mathrm{req}$ is the formal parameter and represents the actual requester in each operation. These operations are summarized in the first part in table 5.4. $\mathrm{NAME}$ is an abstract data type whose elements represent identifiers of entities (attributes, user name etc.) that are included in the ABAC system.

Operation 1.1 creates a new tenant in the system and operation 1.2 assigns a tenant root user to a tenant. For simplicity, we assume that a tenant can only have one tenant root user. For example, Alice ($\mathrm{CRU}$) is authorized to create a tenant named TechEdu and then Alice adds Bob as the tenant root user. So far Bob has been granted full permissions in tenant TechEdu.

$$\mathsf{createTenant}(\mathrm{Alice}, TechEdu)$$

$$\mathsf{createRootUser}(\mathrm{Alice}, \mathrm{Bob}, TechEdu)$$

- **Category II. Operations For Tenant Root User**

Secondly, we define operations for tenant root user. They are summarized in the second part in table 5.4. They are authorized if and only if the requester is the tenant root user, i.e., $\mathrm{req} \in \mathrm{TRU}$, where $\mathrm{req}$ is a formal parameter and represents the actual requester in each operation. We look at operations for configuring $\mathrm{IaaS}_{\mathrm{op}}$ policy. Operation 2.1 adds a $\mathrm{set}$-valued or $\mathrm{atomic}$-valued user

attribute. In scenario 1, each regular user is associated with one attribute `entity_type`, which represents the entity and type pairs the user is assigned and it can take more than one such pair. For example, `entity_type`(Gary) = {($cs$, $email$), ($ece$, $web$)}. Bob creates attribute `entity_type`:

$$\mathsf{createUserAttr}(\mathrm{Bob}, \texttt{entity\_type}, set)$$

We omit the step for defining scopes for attributes. Operation 2.4 adds a set-valued or atomic-valued subject attribute. Bob creates a set-valued subject attribute `sentity_type`:

$$\mathsf{createSubAttr}(\mathrm{Bob}, \texttt{sentity\_type}, set)$$

Operation 2.7 creates a subject attribute constraints policy. POLICY is an abstract data type whose elements represent identifiers of policies (authorization policy, subject attribute constraints policy, etc.) that may appear in $\mathrm{IaaS}_{\mathrm{op}}$ system. To satisfy policy 1 in scenario 1, Bob creates the following subject attribute constraint policy.

$$\mathsf{addSubConstr}(\mathrm{Bob}, policy)$$

where $policy$ is:

$$\mathrm{ConstrSub}(u, s, \{(\texttt{sentity\_type}, val)\}) \equiv val \subset \texttt{entity\_type}(u)$$

In this policy, $val$ represents the proposed value for subject attribute `sentity_type`. For example, if `entity_type`(Gary) = {($cs$, $email$), ($ece$, $web$)}, then Gary is allowed to create a subject $sub$ such that `sentity_type`($sub$) ={($ece$, $web$)}.

Operation 2.9 adds a set-valued or atomic-valued object attribute. In scenario 1, each object is associated with two attributes: `entity` and `type`. Attribute `entity` represents the entity to which the resource belongs and attribute `type` represents the type of the resource. Thus, each object can

only take one value for `entity` and one value for `type`. For example, a server cannot be assigned with $\{web, email\}$ for `type` attribute. Bob creates the two object attributes.

$$\mathsf{createObjAttr}(\mathrm{Bob}, \texttt{entity}, atomic, \texttt{vm})$$

$$\mathsf{createObjAttr}(\mathrm{Bob}, \texttt{type}, atomic, \texttt{vm})$$

Operation 2.12 adds an object attribute constraints policy at object creation time. To satisfy policy 2 in scenario 1, Bob creates the following object attribute constraints policy.

$$\mathsf{addObjConstr}(\mathrm{Bob}, policy)$$

where $policy$ is:

$$\mathrm{ConstrObj}(s, o, \{(\texttt{entity}, oentity), (\texttt{type}, otype)\}) \equiv$$

$$\exists (entity, type) \in \texttt{entity\_type}(s).entity = oentity \wedge type = otype$$

In this policy, $oentity$ and $otype$ represents the suggested value of object attributes `entity` and `type` respectively. It means that a subject can only assign the attribute of created object to the entity and type it is assigned to. For example, if $\texttt{sentity\_type}(sub) = \{(ece, web)\}$, then $sub$ can only create object $obj$ such that $\texttt{entity}(obj) = ece$ and $\texttt{type}(obj) = web$.

Operation 2.16 creates the authorization policy for regular users. To satisfy the requirement in policy 3 in scenario 1, Bob creates the following policy.

$$\mathsf{addAuthz}(\mathrm{Bob}, policy)$$

where $policy$ is:

$$\text{Authorization}_{create} \equiv \exists(entity, type) \in \texttt{entity\_type}(s) \wedge$$
$$entity = \texttt{entity}(o) \wedge type = \texttt{type}(o)$$

Now Bob has finished configuring ABAC policy for regular users.

The next step is to configure $\text{IaaS}_{\text{ad}}$ policy such that administrators have permissions to create users and modify user attributes. Operation 2.18 creates an administrative role and operation 2.19 creates policies for administrative roles (we adopt the structure and specification language from [78]). Bob defines an administrative role called $CSmanager$ which is authorized to add IT architects to $cs$ department and any server type. He achieves those using the following operation:

$$\mathsf{createAdminRole}(\text{Bob}, CSmanager)$$
$$\mathsf{createAdminPolicy}(\text{Bob}, policy)$$

where $policy$ is:

$$\mathsf{can\_add}_{\texttt{entity\_type}} = \{(CSmanager, ITarchitect \in \texttt{role}(u),$$
$$\{(cs, web), (cs, email), (cs, app)\})\}$$

Operation 2.23 adds a user-role assignment. Bob adds Frank as a $CSmanager$.

$$\mathsf{addAminUserRole}(\text{Bob}, Frank, CSmanager)$$

So far, Bob has finished configuring the policy for tenant TechEdu and Frank can create regular users and assign attributes.

- **Category IV. Operations For Tenant Regular Users**

The following operations are allowed by tenant root user or administrative users if they are

assigned with appropriate administrative roles. We briefly introduce the format and evaluation of each operation. Operation 3.1 adds a regular user. The operation to add a value to an attribute of a user is Operation 3.3 which is add(req, $target\_user$, att, $value$), where req is the requester, $target\_user$ is the user whose attribute is to be added a value, att represents the attribute to be modified, $value$ represents the value to be added. Similarly, operation 3.4 delete(req, $target\_user$, att, $value$) and operation 3.5 assign(req, $target\_user$, att, $value$) are defined. As Frank is already an administrative user with $CSmanager$ role, Frank now can add IT architect Gary to their designated entity and server type.

$$\mathsf{createUser}(\mathrm{Frank}, \mathrm{Gary})$$

$$\mathsf{add}(\mathrm{Frank}, \mathrm{Gary}, \texttt{entity\_type}, (cs, web))$$

- **Category III. Operations For Tenant Administrative Users**

We show a few $\mathrm{IaaS_{op}}$ operation examples below. Gary becomes a tenant regular user and then all his operations are evaluated by authorization policy. Gary creates a subject with the suggested value for sentity_type attribute, then this subject $g\_sub$ creates a server on behalf of Gary. Any subject with $cs$ entity and $web$ type can access $serverA$ and perform operations such as resize.

$$\mathsf{CreateSub}(\mathrm{Gary}, g\_sub, \{(\texttt{sentity\_type}, \{(cs, web)\})\})$$

$$\mathsf{CreateObj}(g\_sub, serverA, \{(\texttt{entity}, cs), (\texttt{type}, web)\}, \texttt{vm})$$

$$\mathsf{resize\_server}(g\_sub, serverA)$$

## 5.6 Openstack Based Proof Of Concept

We demonstrate practicality of the models of the previous section by a proof-of-concept OpenStack implementation. We briefly introduce the authorization and authentication components in OpenStack and then propose three different enforcement models. OpenStack contains the following

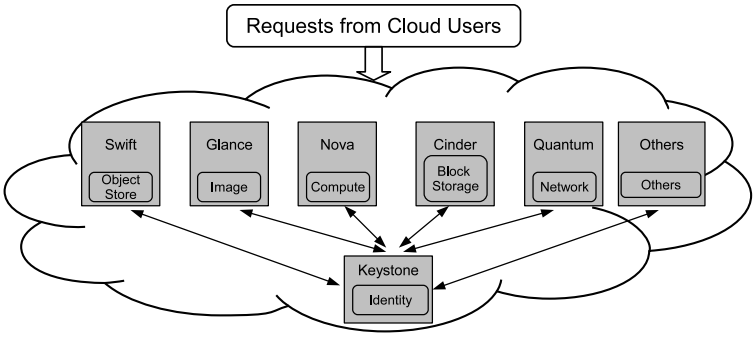



**Figure 5.6**: *Components of OpenStack*

components: Nova, Swift, Glance, Cinder, Keystone, Quantum and Horizon (as shown in figure 5.6). Each component acts as a service which communicate with each other via message queues and hence are loosely-coupled. Nova provides virtual servers upon demand. Swift provides object storage. Glance provides a catalog and repository for virtual disk images. Horizon provides a modular web-based user interface for all OpenStack services. Quantum provides network connectivity as a service between interface devices managed by other OpenStack services. Cinder provides persistent block storage to guest VMs. Keystone provides authentication and authorization for all the OpenStack services. In our discussion, we focus on Keystone and Nova.

### 5.6.1 Access Control in OpenStack

Authorization in OpenStack is enforced by a Policy Enforcement Point in each component. Keystone is the component that stores user information including tenant and role assignments. Keystone provides the user information in the format of token which is signed user data by Keystone using its private key. All other components obtain the public key of Keystone when added as a service. Thus, the public key of Keystone is only distributed to trusted components. They verify the user information by decoding the user's token. Other components then authorize the user based on the user information provided by the token. Generally, Keystone is the policy information point (PIP) where user information is stored and each component has its own policy enforcement point (PEP), policy decision point (PDP), policy administration point (PAP), and a PIP where respective object attributes are stored.

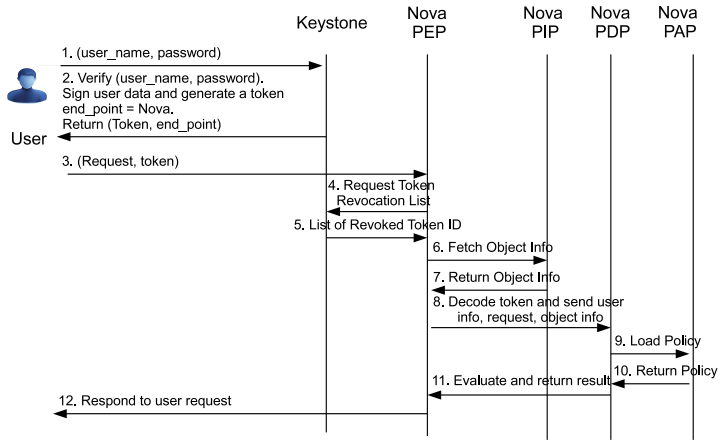



**Figure 5.7**: *OpenStack Authorization Using Asymmetric Keys*

A general authorization process for Nova component is illustrated in figure 5.7. A user sends the user name and password to Keystone for authentication and obtains service end point addresses for various OpenStack services. Keystone then verifies the provided user name and password and generates a token with signed user data. Keystone sends the token back to the user together with the service endpoints (e.g., address for Nova service). The user then sends a request to the Nova service using the token and request details (e.g., operation, arguments). The Nova service's PEP component verifies and validates that the provided token is not revoked by communicating with Keystone. The PEP component then retrieves object data from local PIP and decodes the token with Keystone's public key. User and object data together with the request are sent to the PDP component. The PDP retrieves policy from local files and evaluate the request. A result of `true` or `false` is returned meaning that the request is either authorized or denied.

### 5.6.2 Enforcement Models

We consider three different enforcement models. The structure of the first enforcement model is shown in figure 5.8. This method maintains the original architecture of OpenStack. Keystone stores user attributes definitions, user attribute assignments, subject attributes definitions, subject attribute assignment and subject attribute constraints policy. When a user authenticates through Keystone and tries to create a subject with suggested values for each subject attribute, Keystone

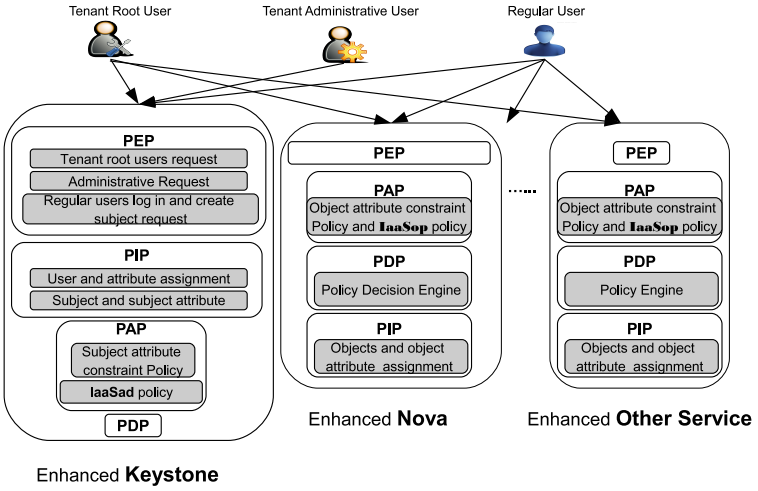



**Figure 5.8**: *Proposed ABAC Enforcement Model I*

verifies the suggested attributes against subject attributes constraints policy and the creating user attributes. Then Keystone generates a token by signing the suggested subject attributes. The administration policy is stored, enforced and decided in Keystone. Components excluding Keystone stores object attributes, object attribute assignments and policies for authorization and object attribute constraints policy.

Enforcement Model II defines a centralized policy engine. The structure is different from that of enforcement model I only in the part shown in figure 5.9(a). We design a separate component called PolicyEngine. It is the central point for policy storage and authorization evaluation. All other components, instead of calling local policy evaluation engine, forward their authorization request (containing details about the request and user token) to this component. Included items in the forwarded request are: subject attributes, object attributes and operation. With the centralized design, all policies for all tenants are stored centrally in a single component. Thereby policy administration is decoupled from the policy enforcement. Object attribute constraints is expressed using authorization policy. However, this enforcement model sacrifices performance for convenience. There is a network latency because each request is sent to the PolicyEngine as a REST call.

We propose a third enforcement model III shown in figure 5.9(b). It is different from Enforce-

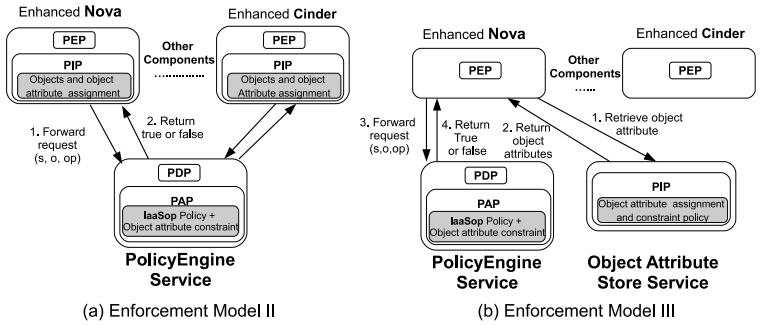



**Figure 5.9**: *Proposed ABAC Enforcement Model II and III*

ment model II only in that a centralized object attribute store is provided. All object attributes are stored. When each component enforces their policies, there are two ways to interact with object attribute store: (1) Each component retrieves object attributes from the object attribute store and forwards the request to the PolicyEngine. (2) The PolicyEngine receives request from other components and retrieves the object attributes from the centralized object attribute store.

## 5.7 Performance Evaluation

We evaluate our ABAC implementation in this section.

### 5.7.1 Experiment Content

We have completed a first stage implementation of ABAC for the Nova and Keystone service components of OpenStack. Our experiments are in two parts.

**Part I.** We evaluate the time increase for token generation in Keystone with or without additional user attributes. As user attributes are included in the token instead of only role information, it requires a longer time for token generation (remember that a token is a signed user credential). For simplicity, we ignore subject attribute constraint policy in this experiment. We test the response time of token generation for cases with 0, 5, 10, 15 and 20 user attributes where "0" means that the "role" is the only attribute as originally included in OpenStack. User attributes in the database are stored as $(\mathsf{attname}, value, tenant)$ tuples. We send concurrent requests to keystone using Keystone Client command and measure the average response time on client side.

**Part II.** We evaluate the network latency introduced by the centralized PolicyEngine in enforcement model II. The latency is introduced by the forwarded data size which contains user token with attributes, object attributes and operation. Thus, we measure the average time taken for the Nova server to send the request to PolicyEngine and receive a result. We change the number of user attributes and concurrent requests.

### 5.7.2 Experiment Environment and Results

Our experiments are based on a private cloud shown in figure 5.10. It is installed on four physical machines. We install two compute nodes, one networking node and one controller node. The configuration of controller node and network node is: 24 cores CPU, 24 GB RAM and 1 TB Disk and the configuration of the two Nova compute node is: 16 cores CPU, 98 GB RAM and 1TB Disk. There are three networks in this installation: (1) the green line on network interface `eth1` shows the administration network which connects different components of OpenStack; (2) the red lines on network interface `eth1 : 1` shows the data network which connects virtual machines with the Internet and (3) the black line on the `eth0` network interface shows the access to the Internet which is only accessible by Controller node and Networking node. In experiment part II, we install the centralized PolicyEngine on another machine which has dual core CPU, 4GB RAM and 10 GB disk.

The result for **Part I** is shown in figure 5.11. A first observation is that given the same concurrent request, the average time for token generation increases with the number of user attributes. This is caused by the increase in the length of user data to be signed by Keystone. As each token contains all user attributes, the signing process and transmission takes a longer to finish as expected. However, we can see that the increase is not significant. The time is increased by 20% when the number of user attribute increases from 0 to 20. Another observation is that if the number of user attributes is the same and we increase the number of concurrent requests, it is not necessary that the average process will time increase. This is due to the internal scheduling mechanism. Keystone accepts request concurrently and processes them sequentially from a queue. Thus each

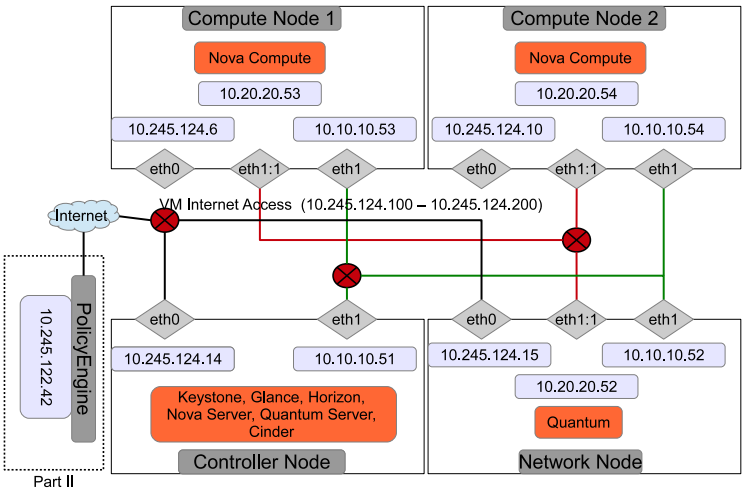



**Figure 5.10**: *OpenStack Installation On Physical Machines*

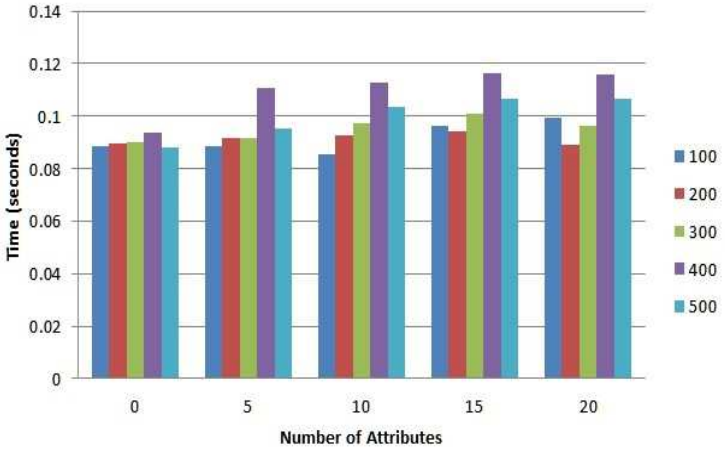



**Figure 5.11**: *Average Time for Token Generation in Keystone*

request got the same time. Note that when the attribute number is 20, the average time for 200 concurrent requests is smaller than that with 100 concurrent requests. This is affected by the current tasks and the state of the system message queue.

The result for **Part II** is shown in figure 5.12. It can be seen that the networking latency increases with the number of user attributes as data to be forwarded to the PolicyEngine component becomes larger. The latency increases with the number of concurrent request even with the same number of user attributes. This is due to the reason that the PolicyEngine is installed on a machine with limited computing power than the machines we installed OpenStack. The waiting time for

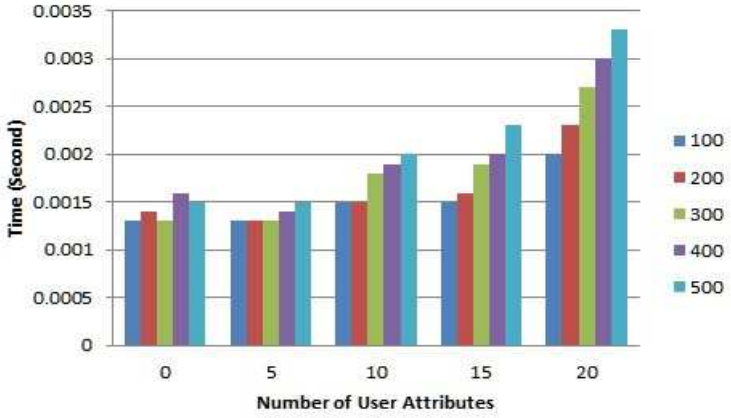



**Figure 5.12**: *Average Time for Nova Communicating with PolicyEngine*

getting a policy decision becomes larger when there are too many requests to be evaluated. The average time for request with 20 attributes in 500 concurrent requests is almost twice the time of that with no attributes. However, our implementation of PolicyEngine is not highly optimized. Furthermore, 20 attributes for access control decisions is a big stretch in practice. This experiment shows that when the user attributes becomes larger, the increase in time for different number of concurrent requests becomes larger. That is , when there are 20 attributes, the average time for 500 concurrent requests is around 16 time of the time for 100 concurrent requests. While for zero attributes, the time for 500 concurrent requests is around 1.7 times of the time for 100 concurrent requests.

### 5.7.3 Conclusion

In this chapter, we proposed an ABAC framework for access control in cloud IaaS. We studied existing models from industry and the academic literature and motivated the need for ABAC by showing practical examples and limitations of the existing models. ABAC is suitable for cloud IaaS. We provided formal models for the operational and administrative aspects of our ABAC framework for cloud IaaS. Based on the proposed ABAC framework, we designed enforcement models based on the open source cloud platform OpenStack. We then evaluated the performance of our proposed enforcement models.

# Chapter 6: CONCLUSION AND FUTURE WORK

## 6.1 Summary

In this work, we propose $ABAC_\alpha$ model to provide "least" features to cover DAC, MAC and RBAC. Further, we extend $ABAC_\alpha$ on each of its configuration points to cover selected various RBAC extensions. Based on this framework, we propose GURA model to manage user attribute assignment. It is an extension to URA component in ARBAC97 model. Further, we analyze attribute reachability problem on a restricted version of GURA model called rGURA. It is restricted from GURA in that it has simplified version of precondition specification language.

One important ongoing work is to deploy ABAC in OpenStack, which is one of the main streams of cloud computing platform. In this dissertation we have described partial progress towards this goal. The ABAC model is applied to regulate the usage of virtual resources such as disk, RAM, network in single tenant with large number of regular users. To achieve it, we design mechanisms for tenant administrators to provision attributes and compose GURA policy to manage user attributes. Further, we provide mechanism for authorization policy composition. Secondly, we extend GURA model to facilitate automatic user attribute update. In this case, user attributes can be updated by administrators, users, and system event. Based on this model, we consider user attribute reachability analysis. Beyond that, objects may be also involved and a more powerful framework which deals with both user and object attributes can be analyzed.

## 6.2 Future Work

- *ABAC model itself.* An example extension is *automatic attribute update.* Attributes can be updated automatically by many factors. For example, in order to configure membership rules in OASIS-RBAC, subject attributes should be able to be automatically updated when the membership rules for activated roles are not satisfied. Mutable attributes should be defined for ABAC and automatic update policies should be specified. Modeling such a feature

requires significant enhancement to the model.

- *Administrative model.* GURA can be extended to ABAC administration of attributes. Attributes can be managed in ad hoc manner where each user can label other users with certain attributes and values which can further be used in access control. An interesting question here is the trustworthiness of the attributes values assigned by other users. Generally, the trustworthiness can be determined by the attributes of the assigned user, relationships between the users and even their relationship with other users.

- *User attribute reachability analysis.* The first direction is additional polynomial time solutions. For instance, tractable solutions for the scheme rGURA$_1$-atomic and rGURA$_1$ remain to be explored. Secondly, the rGURA scheme itself can be extended in many directions. For instance, administrators could be treated as regular users so administrative role is just another user attribute. User attributes are utilized in determining administrative privileges as well as in precondition specification. Precondition in rules could allow specification of other users' attributes thus connecting related users. Thirdly, additional kinds of queries can be defined. Except for the examples introduced in other related work (*e.g.,* existence of length-bounded plan), queries can also be specified on the relationships between the attributes of the same user.

- *ABAC in cloud and possibly other platforms.* In the future, we plan to work on policy analysis of ABAC and administration models. In addition, we plan to study the structure of the policy and improve the throughput of the PolicyEngine component.

- *ABAC with additional issues considered.* Besides the proposed work discussed in the earlier chapter, privacy aware ABAC is an interesting topic in future. As attributes represent information about the users, releasing attributes to the policy evaluating engine is a sensitive activity as the third party may not be trusted. Although trust negotiation and other techniques have been proposed to regulate the release of information, still, least privilege has to

be enforced in ABAC. It means the least set of user attributes are released for the purpose of request evaluating.

# BIBLIOGRAPHY


[1] Amazon web services. `http://aws.amazon.com`.

[2] OASIS, Extensible access control markup language (XACML), v2.0 (2005).

[3] Openstack. `http://www.openstack.org/`.

[4] Rackspace customers. `http://stories.rackspace.com/customers`. Accessed: Dec 2013.

[5] Ali E. Abdallah and Etienne J. Khayat. A formal model for parameterized role-based access control. In *Formal Aspects in Security and Trust*, pages 233–246, 2004.

[6] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur.*, 3:207–226, 2000.

[7] M. A. Al-Kahtani and R. Sandhu. Rule-based RBAC with negative authorization. In *ACSAC*, 2004.

[8] Mohammad A. Al-Kahtani and Ravi S. Sandhu. A model for attribute-based user-role assignment. In *ACSAC*, 2002.

[9] F. Alberti, A. Armando, and S. Ranise. Efficient symbolic automated analysis of administrative attribute-based RBAC-policies. In *ACM ASIACCS*, pages 165–175, 2011.

[10] A. Almutairi, M. Sarfraz, S. Basalamah, W. Aref, and A. Ghafoor. A distributed access control architecture for cloud computing. *IEEE Software*, 2012.

[11] F.T. Alotaiby and J.X. Chen. A model for team-based access control (TMAC ). In *Proceedings. ITCC 2004*, 2004.

[12] Grigoris Antoniou, Matteo Baldoni, Piero A Bonatti, Wolfgang Nejdl, and Daniel Olmedilla. Rule-based policy specification. In *Secure data management in decentralized systems*, pages 169–216. Springer, 2007.

[13] C.A. Ardagna, S. De Capitani di Vimercati, G. Neven, S. Paraboschi, F.-S. Preiss, P. Samarati, and M. Verdicchio. Enabling privacy-preserving credential-based access control with XACML and SAML. In *IEEE CIT*, pages 1090–1095, 2010.

[14] A. Armando and S. Ranise. Automated and efficient analysis of role-based access control with attributes. *Data and Applications Security and Privacy XXVI*, pages 25–40, 2012.

[15] Sanjeev Arora, Eunjee Song, and Yoonjeong Kim. Modified hierarchical privacy-aware role based access control model. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, RACS '12, pages 344–347, 2012.

[16] C. Bäckström and B. Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[17] J. Bacon, K. Moody, and W. Yao. A model of OASIS role-based access control and its support for active security. *TISSEC*, 2002.

[18] E. Barka and R. Sandhu. Role-based delegation model/hierarchical roles (RBDM1). In *Computer Security Applications Conference, 2004. 20th Annual*, 2004.

[19] J. F. Barkley, K. Beznosov, and J. Uppal. Supporting relationships in access control using role based access control. In *ACM Workshop on RBAC*, 1999.

[20] L. Bauer, S. Garriss, and M. K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *ACM SACMAT*, pages 185–194, 2008.

[21] Moritz Y Becker, Cédric Fournet, and Andrew D Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, 2010.

[22] Moritz Y Becker and Sebastian Nanz. A logic for state-modifying authorization policies. In *Computer Security–ESORICS 2007*, pages 203–218. Springer, 2007.

[23] M.Y. Becker. Specification and analysis of dynamic authorisation policies. In *IEEE CSF*, 2009.

[24] M.Y. Becker, C. Fournet, and A.D. Gordon. Design and semantics of a decentralized authorization language. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, 2007.

[25] S. Berger, R. Cáceres, and K. et al Goldman. Security for the cloud infrastructure: Trusted virtual data center implementation. *IBM J. of Res. and Dev.*, 2009.

[26] E. Bertino, P. A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *TISSEC*, 2001.

[27] Elisa Bertino, Barbara Catania, Maria Luisa Damiani, and Paolo Perlasca. GEO-RBAC: a spatially aware RBAC. In *Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 29–37. ACM, 2005.

[28] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, 2003.

[29] Elisa Bertino, Elena Ferrari, and Vijay Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM TISSE*, 1999.

[30] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *IEEE SP'07*, pages 321–334, 2007.

[31] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, pages 185–210. Springer, 1999.

[32] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE, 1996.

[33] Piero Bonatti, Clemente Galdi, and Davide Torres. ERBAC: Event-driven RBAC. In *ACM SACMAT*, pages 125–136, 2013.

[34] Piero A. Bonatti and P. Samarati. Regulating service access and information release on the web. In *ACM CCS*, 2000.

[35] Piero A. Bonatti and P. Samarati. A uniform framework for regulating service access and information release on the web. *J. Comp. Secur.*, 2002.

[36] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, pages 165–204, 1994.

[37] Jan Camenisch, Sebastian Mödersheim, Gregory Neven, Franz-Stefan Preiss, and Dieter Sommer. Credential-based access control extensions to XACML. In *W3C Workshop on Access Control Application Scenarios, Luxembourg*, volume 17, 2009.

[38] J. Cha, B.and Seo and J. Kim. Design of attribute-based access control in cloud computing environment. In *Int. Conf. on IT Conv. and Sec.*, pages 41–50. Springer, 2012.

[39] Eric Chabrow. Overcoming the Apprehension of Cloud Computing: Results from the 2012 Cloud Computing Survey. Technical report, SMG Information Security Media Group, 2012.

[40] D. W. Chadwick, M. Casenove, and K. Siu. My private cloud–granting federated access to cloud resources. *Journal of Cloud Computing*, 2013.

[41] David W. Chadwick, Alexander Otenko, and Edward Ball. Role-based access control with X.509 attribute certificates. *IEEE Internet Computing*, 2003.

[42] Jung Hwa Chae and Nematollaah Shiri. Formalization of RBAC policy with object class hierarchy. ISPEC, 2007.

[43] Sudip Chakraborty and Indrajit Ray. TrustBAC: integrating trust relationships into the RBAC model for access control in open systems. In *SACMAT*, 2006.

[44] Suroop Mohan Chandran and James B. D. Joshi. LoT-RBAC: A Location and time-based RBAC model. In *WISE*, 2005.

[45] Melissa Chase. Multi-authority attribute based encryption. In *Theory of Cryptography*, pages 515–534. Springer, 2007.

[46] David JB Cheperdak. *Attribute-Based Access Control for Distributed Systems*. PhD thesis, University of Victoria, 2012.

[47] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[48] Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind K. Dey, Mustaque Ahamad, and Gregory D. Abowd. Securing context-aware applications using environment roles. In *SACMAT*, 2001.

[49] S. Crago, K. Dunn, and P. et al Eads. Heterogeneous cloud computing. In *2011 IEEE CLUSTER*, pages 378–385.

[50] Frédéric Cuppens and Nora Cuppens-Boulahia. Modeling contextual security policies. *Int. J. Inf. Sec.*, 2008.

[51] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *POLICY*, 2001.

[52] Chen Danwei, Huang Xiuli, and Ren Xunyi. Access control of cloud service based on UCON. In *Cloud Computing*, pages 559–564. Springer, 2009.

[53] Y. Deng, J. Wang, J. JP Tsai, and K. Beznosov. An approach for modeling and analysis of security system architectures. *IEEE TKDE*, 15(5):1099–1119, 2003.

[54] J. DeTreville. Binder, a logic-based security language. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 105–113, 2002.

[55] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI certificate theory. Technical report, IETF RFC 2693, September, 1999.

[56] Mark Evered. Supporting parameterized roles with object-based access control. In *HICSS*, 2003.

[57] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 2001.

[58] Philip W. L. Fong. Relationship-based access control: protection model and policy language. In *CODASPY*, 2011.

[59] Eric Freudenthal, Tracy Pesin, Lawrence Port, Edward Keenan, and Vijay Karamcheti. dRBAC: Distributed role-based access control for dynamic coalition environments. In *ICDCS*, 2002.

[60] Keith Frikken, Mikhail Atallah, and Jiangtao Li. Attribute-based access control with hidden policies and hidden credentials. *IEEE Transactions on Computers*, pages 1259–1270, 2006.

[61] L. Fuchs, G. Pernul, and R. Sandhu. Roles in information security: A survey and classification of the research area. *Comp. and Secur.*, 2011.

[62] Mei Ge and Sylvia L. Osborn. A design for parameterized roles. In *DBSec*, 2004.

[63] Christos K Georgiadis, Ioannis Mavridis, George Pangalos, and Roshan K Thomas. Flexible team-based access control using contexts. In *ACM SACMAT*, pages 21–27, 2001.

[64] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *ACM Workshop on RBAC*, 1997.

[65] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM CCS*, pages 89–98, 2006.

[66] P. Gupta, S. D. Stoller, and Z. Xu. Abductive analysis of administrative policies in rule-based access control. In *ICISS*, volume 7093, pages 116–130. Springer-Verlag, 2011.

[67] Frode Hansen and Vladimir Oleshchuk. Srbac: A spatial role-based access control model for mobile systems. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems (NORDSEC)*, pages 129–141. Citeseer, 2003.

[68] M. A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Comm. of the ACM*, pages 461–471, 1976.

[69] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. Guide to attribute based access control (ABAC) definition and considerations (draft). *NIST Special Publication*, 800:162, 2013.

[70] Vincent C Hu, Deborah A Frincke, and David F Ferraiolo. The policy machine for security policy management. In *Computational Science-ICCS 2001*, pages 494–503. Springer, 2001.

[71] Vincent C Hu, Evan Martin, JeeHyun Hwang, and Tao Xie. Conformance checking of access control policies specified in XACML. In *IEEE COMPSAC*, volume 2, pages 275–280, 2007.

[72] J. Huang, D. Nicol, R. Bobba, and J. H. Huh. A framework integrating attribute-based policies into RBAC. In *ACM SACMAT*, 2012.

[73] Z. Iqbal and J. Noll. Towards semantic-enhanced attribute-based access control for cloud services. In *IEEE TrustCom*, 2012.

[74] T. Jaeger and Jonathon E. Tidswell. Practical safety in flexible access control models. *ACM Trans. Inf. Syst. Secur.*, pages 158–190, 2001.

[75] S. Jajodia, P. Samarati, and VS Subrahmanian. A logical language for expressing authorizations. In *IEEE Symposium on S&P*, pages 31–42, 1997.

[76] S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough. Towards formal verification of role-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 5(4):242–255, 2008.

[77] X. Jin, R. Krishnan, and R. Sandhu. Reachability analysis for role-based administration of attributes. In *ACM DIM 2013*.

[78] X. Jin, R. Krishnan, and R. Sandhu. A role-based administration model for attributes. In *First International Workshop on SRAS*. ACM, 2012.

[79] X. Jin, R. Sandhu, and R. Krishnan. RABAC: Role-centric attribute-based access control. In *MMM-ACNS*, 2012.

[80] Xin Jin, Ram Krishnan, and Ravi Sandhu. A unified attribute-based access control model covering DAC, MAC and RBAC. In *Data and Applications Security and Privacy XXVI*, pages 41–55. Springer, 2012.

[81] J. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. *IEEE Trans. Knowl. Data Eng.*, 2005.

[82] Md Enamul Kabir, Hua Wang, and Elisa Bertino. A role-involved purpose-based access control model. *Information Systems Frontiers*, 14(3):809–822, 2012.

[83] Anas Abou El Kalam, Salem Benferhat, Alexandre Miège, Rania El Baida, Frédéric Cuppens, Claire Saurel, Philippe Balbiani, Yves Deswarte, and Gilles Trouessin. Organization based access control. In *POLICY*, 2003.

[84] S. Kandala and R. Sandhu. Secure role-based workflow models. In *Proceedings of the fifteenth annual working conference on Database and application security*, DAS, 2002.

[85] Paul Ashley Satoshi Hada Günter Karjoth and Calvin Powers Matthias Schunter Enterprise Privacy. Authorization language (EPAL 1.1) Oct. 1, 2003 IBM research mts at zurich. ibm. com. *Source: http://www. zurich. ibm. com/security/enterprise-privacy/epal/Specification*.

[86] Alan H Karp, Harry Haury, and Michael H Davis. From ABAC to ZBAC: the evolution of access control models. *Hewlett-Packard Development Company, LP*, 21, 2009.

[87] A. Kern. Advanced features for enterprise-wide role-based access control. In *ACSAC*, 2002.

[88] Axel Kern and Claudia Walhorn. Rule support for role-based access control. SACMAT, 2005.

[89] Devdatta Kulkarni and Anand Tripathi. Context-aware role-based access control in pervasive computing systems. In *ACM SACMAT*, pages 113–122, 2008.

[90] Arun Kumar, Neeran Karnik, and Girish Chafle. Context sensitivity in role-based access control. *ACM SIGOPS Operating Systems Review*, 36(3):53–66, 2002.

[91] Adam J Lee. Credential-based access control. In *Encyclopedia of Cryptography and Security*, pages 271–272. Springer, 2011.

[92] Michael LeMay, Omid Fatemieh, and Carl A Gunter. Policymorph: interactive policy transformations for a logical attribute-based access control framework. In *ACM SACMAT*, pages 205–214, 2007.

[93] N. Li, J. C. Mitchell, and W. H. Winsborough. Beyond proof-of-compliance: security analysis in trust management. *Journal of the ACM (JACM)*, 52(3), 2005.

[94] N. Li and M.V. Tripunitara. Security analysis in role-based access control. In *ACM SACMAT*, pages 126–135, 2004.

[95] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *2002 IEEE S&P*.

[96] Qi Li, Xinwen Zhang, Mingwei Xu, and Jianping Wu. Towards secure dynamic collaborations with group-based RBAC model. *Computers & Security*, 28(5), 2009.

[97] Alex X Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. Xengine: a fast and scalable XACML policy evaluation engine. In *ACM SIGMETRICS Performance Evaluation Review*, pages 265–276, 2008.

[98] Amirreza Masoumzadeh and James BD Joshi. PuRBAC: Purpose-aware role-based access control. In *On the Move to Meaningful Internet Systems: OTM 2008*, pages 1104–1121. 2008.

[99] Pietro Mazzoleni, Bruno Crispo, Swaminathan Sivasubramanian, and Elisa Bertino. XACML policy integration algorithms. *ACM TISSEC*, page 4, 2008.

[100] C. Moon, D. Park, S. Park, and D. Baik. Symmetric RBAC model that takes the separation of duty and role hierarchies into consideration. *Computers and Security*, pages 126–136, 2004.

[101] G. H M B Motta and S.S. Furuie. A contextual role-based access control authorization model for electronic patient record. *IEEE Transactions on Information Technology in Biomedicine,*, 7(3):202–207, 2003.

[102] M.J. Moyer and M. Abamad. Generalized role-based access control. In *International Conference on Distributed Computing Systems*, 2001.

[103] Qun Ni, Elisa Bertino, Jorge Lobo, Carolyn Brodie, Clare-Marie Karat, John Karat, and Alberto Trombeta. Privacy-aware role-based access control. *ACM TISSEC*, 13(3):24, 2010.

[104] Daniel Nurmi and Richard et al Wolski. The eucalyptus open-source cloud-computing system. In *CCGRID*, pages 124–131. IEEE, 2009.

[105] Sejong Oh. New role-based access control in ubiquitous e-business environment. *Journal of Intelligent Manufacturing*, 21(5):607–612, 2010.

[106] Sejong Oh and Seog Park. Task-role based access control (T-RBAC): An improved access control model for enterprise environment. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, 2000.

[107] Sejong Oh and Seog Park. Task–role-based access control model. *Information Systems*, 28(6):533–562, 2003.

[108] S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM TISSEC*, 2000.

[109] Rafail Ostrovsky, Amit Sahai, and Brent Waters. Attribute-based encryption with non-monotonic access structures. In *ACM CCS*, pages 195–203, 2007.

[110] F. Paci, R. Ferrini, and E. Bertino. Identity attribute-based role provisioning for human WS-BPEL processes. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 535–542, 2009.

[111] Jaehong Park and Ravi Sandhu. The UCON$_{ABC}$ usage control model. *ACM TISSEC*, 2004.

[112] Andrew Pimlott and Oleg Kiselyov. Soutei, a logic-based trust-management system. In *Functional and Logic Programming*, pages 130–145. Springer, 2006.

[113] David Power, Mark Slaymaker, and Andrew Simpson. On the modeling and analysis of Amazon Web Services access policies. Technical Report RR-09-15, Oxford University Computing Laboratory, November 2009.

[114] T. Priebe, W. Dobmeier, and N. Kamprath. Supporting attribute-based access control with ontologies. In *ARES*, pages 8 pp.–, 2006.

[115] Prathima Rao, Dan Lin, Elisa Bertino, Ninghui Li, and Jorge Lobo. An algebra for fine-grained integration of XACML policies. In *ACM SACMAT*, pages 63–72, 2009.

[116] Indrakshi Ray and Manachai Toahchoodee. A spatio-temporal role-based access control model. In *Data and Applications Security XXI*, pages 211–226. Springer, 2007.

[117] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In *Advances in Cryptology–EUROCRYPT 2005*, pages 457–473. Springer, 2005.

[118] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM TISSEC*, 2(1):105–135, 1999.

[119] R. S. Sandhu, E.J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[120] Ravi S. Sandhu. The authorization leap from rights to attributes: Maturation or chaos? `http://profsandhu.com/miscppt/pst_120716.pptx`.

[121] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 1993.

[122] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 1996.

[123] Ravi S. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Com. Mag.*, 1994.

[124] A. Sasturkar, P. Yang, S.D. Stoller, and CR Ramakrishnan. Policy analysis for administrative role based access control. In *IEEE CSFW*, 2006.

[125] W. J Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of computer and system sciences*, 4(2):177–192, 1970.

[126] D. Shin, H. Akkan, W. Claycomb, and K. Kim. Toward role-based provisioning and access control for infrastructure as a service (IaaS). *J. Internet Services and App*, 2011.

[127] Dongwan Shin and Hakan Akkan. Domain-based virtualized resource management in cloud computing. In *IEEE CollaborateCom*, 2010.

[128] Anna Squicciarini, Alberto Trombetta, Abilasha Bhargav-Spantzel, and Elisa Bertino. K-anonymous attribute-based access control.

[129] S. D. Stoller, Ping Yang, C R. Ramakrishnan, and Mikhail I. Gofman. Efficient policy analysis for administrative role based access control. In *ACM CCS*, pages 445–455, 2007.

[130] S.D. Stoller, P. Yang, M.I. Gofman, and CR Ramakrishnan. Symbolic reachability analysis for parameterized administrative role-based access control. *Computers & Security*, 30(2):148–164, 2011.

[131] S.D. Stoller, Ping Yang, Mikhail Gofman, and C. R. Ramakrishnan. Symbolic reachability analysis for parameterized administrative role based access control. In *ACM SACMAT*, pages 165–174, 2009.

[132] S Subashini and V Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 2011.

[133] Hassan T., James BD J., and Gail-Joon A. Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, 2010.

[134] H. Takabi, M. Amini, and R. Jalili. Trust-based user-role assignment in role-based access control. In *IEEE/ACS AICCSA*, pages 807–814, 2007.

[135] H. Takabi, J. BD Joshi, and G. J. Ahn. Securecloud: Towards a comprehensive security framework for cloud computing environments. In *IEEE COMPSACW*, 2010.

[136] R. K. Thomas and R. S. Sandhu. Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented authorization management. In *DBSec*, 1997.

[137] Roshan K. Thomas. Team-based access control (TMAC): a primitive for applying role-based access controls in collaborative environments. In *ACM workshop on RBAC*, 1997.

[138] Mary R. Thompson, Abdelilah Essiari, and Srilekha Mudumbai. Certificate-based authorization policy in a PKI environment. *ACM Trans. Inf. Syst. Secur.*, pages 566–588, 2003.

[139] Mahesh V Tripunitara and Ninghui Li. A theory for comparing the expressive power of access control models. *Journal of Computer Security*, 15(2):231–272, 2007.

[140] Jacques Wainer and Paulo Barthelmess. W-RBAC - a workflow security model incorporating controlled overriding of constraints. *International Journal of Cooperative Information Systems*, 2003.

[141] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. A logic-based framework for attribute based access control. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 45–55. ACM, 2004.

[142] W.H. Winsborough, K.E. Seamons, and V.E. Jones. Automated trust negotiation. In *Proceedings DARPA Information Survivability Conference and Exposition*, volume 1, pages 88 –102 vol.1, 2000.

[143] Christian Wolter, Andreas Schaad, and Christoph Meinel. Deriving XACML policies from business process models. In *Web Information Systems Engineering–WISE 2007 Workshops*, pages 142–153. Springer, 2007.

[144] R. Wu, X. Zhang, G. Ahn, H. Sharifi, and H. Xie. Design and implementation of access control as a service for IaaS cloud. *SCIENCE*, 1(3), 2013.

[145] Z. Xu, D. Feng, L. Li, and H. Chen. UC-RBAC: A usage constrained role-based access control model. In *ICICS*, 2003.

[146] Zhongyuan Xu and Scott D Stoller. Mining attribute-based access control policies from rbac policies. pages 1–6, 2013.

[147] Zan Yang, Jian-xin Wang, Lin Yang, Rui-guang Yang, Bao-sheng Kou, Jie-kun Chen, and Shu-mei Yang. The RBAC model and implementation architecture in multi-domain environment. *Electronic Commerce Research*, pages 1–17, 2013.

[148] Jianming Yong, Elisa Bertino, Mark Toleman, and Dave Roberts. Extended RBAC with role attributes. In *PACIS*, page 8, 2006.

[149] Eric Yuan and Jin Tong. Attributed based access control (ABAC) for web services. In *Intl. ICWS*, 2005.

[150] X. Zhang, S. Oh, and R. Sandhu. PBDM: a flexible delegation model in RBAC. SACMAT, 2003.

[151] Z. Zhang, X. Zhang, and R. Sandhu. ROBAC: Scalable role and organization based access control models. In *IEEE TrustCol*, 2006.

# VITA

Xin Jin was born in Anhui China. He received B.S. in Computer Science at Central South University in Changsha, China in 2009. He got his interim M.Sc degree with a focus on cyber security in University of Texas at San Antonio in 2013. This is his fourth year as a Ph.D student working under the supervision of Dr. Ravi Sandhu and Dr. Ram Krishnan at the University of Texas at San Antonio, and he is currently working in attribute based access control models and implementations in infrastructure as a service cloud.