ATTRIBUTE-BASED ADMINISTRATION OF ROLE-BASED ACCESS CONTROL

by

JIWAN LIMBU NINGLEKHU, M.S.

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 ELECTRICAL ENGINEERING

COMMITTEE MEMBERS:
Ram Krishnan, Ph.D., Chair
Eugene John, Ph.D.
Wonjun Lee, Ph.D.
Ravi Sandhu, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Engineering
Department of Electrical and Computer Engineering
December 2017

DEDICATION

To Manju

ACKNOWLEDGEMENTS

I would like to express my most sincere gratitude towards Professor Dr. Ram Krishnan. It is my great privilege to have him as my dissertation advisor and mentor. During a course of past six years with him, I have learned to love my work, and had an opportunity to understand some of the key elements of creativity in technology. I shall always learn from his philosophy.

I would like to express my gratitude to Professor Dr. Ravi Sandhu, not only for providing me with constructive feedback and agreeing to be in my dissertation committee, but also for helping build the realm of computer security the way it stands today. In addition, for leading Institute for Cyber Security (I.C.S.) at UTSA in a world-class manner, of which, I am a proud member.

Professor Dr. Eugene John has been a great inspiration for me since the time I took one of his courses. I also had an opportunity to work with him previously during my masters thesis. I would like to thank Dr. John for his constructive feedback, for believing in me, and encouraging me to move forward.

I would like to thank Professor Dr. Wonjun Lee for his thought provoking questions and valuable feedbacks.

I would also like to thank Mr. Farhan Patwa for sharing his expertise on cloud computing and software architecture. I want to thank Mr. James Benson for his support on integration of my work in the OpenStack Cloud.

I would like to thank our College of Engineering staffs Ms. LiPing Bein and Ms. Monique Wimberley for their help. I would also like to express my gratitude to Institute for Cyber Security staff member, Ms. Suzanne Tanaka for her world-class help.

I would like to thank brother Indra P. Limbu for his constant encouragement and support.

I would like to thank my family members, Dad Gopal Bahadur Ninglekhu, Mom Krishna Kumari Limbu, Little sisters Sita Limbu and Gita Limbu for their love and support.

I would like to express special gratitude towards my elder brother Bhumiraj Ninglekhu for his continuous encouragement, support and love.

I would like to thank my wife Manju for being with me on this adventure while maintaining her patience, trust, support and love. I would like to send my love and gratitude to my daughter Coraline. *Thank you, Sanu! for your cheers and laughs which helped me regain courage when the times were low.*

Finally, I would like to express my sincere gratitude to Sushil Karki. Without him, I wouldn't have gathered enough courage to pursue my Ph.D. *Thank you for always being there, your continuous and dedicated support, and specially, for always believing in me. Sushil ! from you, I shall always learn what a friend can do.*

This research was made possible by Nation Science Foundation (NSF) research grants: CNS-1423481 and CNS-1553696.

December 2017

ATTRIBUTE-BASED ADMINISTRATION OF ROLE-BASED ACCESS CONTROL

Jiwan Limbu Ninglekhu, Ph.D.
The University of Texas at San Antonio, 2017

Supervising Professor: Ram Krishnan, Ph.D.

Role-Based Access Control (RBAC) is an operational model in which if a user wants to access an object, she does it by activating roles that are assigned to her, which in turn activates the permissions that are associated with that role. This indirection allows an easy designation of permissions to users.

Administrative Role-Based Access Control (ARBAC) models deal with the administration of RBAC. ARBAC model primarily involves how to manage user-role assignments (URA), permission-role assignments (PRA), and role-role assignments (RRA). A wide variety of approaches have been proposed in the literature for URA, PRA, and RRA. In each of these models, only one or two static properties of involved entities such as users and permissions have been used in making assignment decisions. For example, in one of the prior models, a user's initial membership or non-membership on a role qualifies that user for further role assignment. In another case, a permission's association on either a role or an organizational unit in an organizational structure allows that permission to be assigned to another role. These models make plausible arguments for URA, PRA or RRA assignments. However, a unified approach that allows checking for all or a combination such policies, while allowing the administrator to introduce new policies remains to be explored.

In this dissertation, a thorough study on developing administrative models that allow a unified approach that allows us to dynamically incorporate properties that can be used to make assignment decisions is conducted. An attribute-based access control (ABAC) approach is taken to develop each model for enhanced URA, PRA and PRA. There is significant prior work done in the ARBAC domain. A set of such models namely, Administrative RBAC '97 (ARBAC97), Administrative RBAC '99 (ARBAC99), Administrative RBAC '02 (ARBAC02), A Unified Administrative Model for Role-Based Access Control (Uni-ARBAC) and Unnamed ARBAC (UARBAC) are studied. From each of these models, URA, PRA and RRA techniques are studied and, corresponding assignment models that yield a family of models for Attribute-Based Administration of RBAC (AARBAC) are developed. They are called attribute-based user-role assignment (AURA), attribute-based permission-role assignment (ARPA) and, attribute-based role-role assignment (ARRA), respectively. These models are sufficient enough to unify URA, PRA and RRA approach exhibited in prior models. For each attribute-based model, a translation algorithm is developed, which can take any instance from the prior model as its input and map it into the corresponding instance of attribute-based assignment approach.

Finally, among all the theoretical attribute-based administration models that are developed, AURA is considered to demonstrate the advantage of attribute-based approach in the user-role assignment, by applying it as a proof-of-concept in OpenStack Infrastructure as a Service (IaaS) cloud's identity service. This implementation shall demonstrate flexibility and policy specification power brought-forward by the attribute-based approach. A performance evaluation is conducted to compare the time variation with and without attributes using different test cases.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: INTRODUCTION

The term ‘protection’ was defined for all the mechanisms that control the access of a program to resource in the system [36]. Since computer’s early usage, a primary concern about protection has been the ability to share resources (in the computer) while having a mechanism to protect them from unauthorized access [22, 36]. The rules by which a decision on whether a request to access resources are granted or denied are known as access control policies. The components such as subject and objects involved in the access control process can be referred to as access control entities. These entities can be used to build access control models, which provides a structure for specifying and introducing security policies.

The entity that requests access of resources is called the subject. A subject can be a human user or a machine (logic) that generates access requests on behalf of humans. The resources are commonly known as objects. A high-level access control scenario is shown in Figure 1.1. In the diagram,the subject’s access to object is controlled by the access control block, where mechanism and the access control policies are defined. An authenticated subject is either granted or denied access based on the policies specified by an access control model. Access control models are desired to be relatively flexible to incorporate security policies, be policy neutral and simple to administer.

Role-based access control (RBAC) [18, 53, 58] is apparently, one of the most well-adopted access control model in enterprise settings [17, 43]. In addition, it is very well-studied access control model in the academic community [20]. The 2010 Economic Analysis of Role-Based Access Control [43] reported by NIST shows the rate by which RBAC has been adopted in IT intensive organizations. It estimates that over 50% of users at organizations with more than 500 employees are expected to have at least some of their permissions managed via roles [43]. A chart in Figure 1.2 shows an increasing RBAC adoption rate since 1995 to 2010. This suggests that the adoption rate is expected to grow.

Administrative Role-Based Access Control (ARBAC) is an approach developed for adminis-

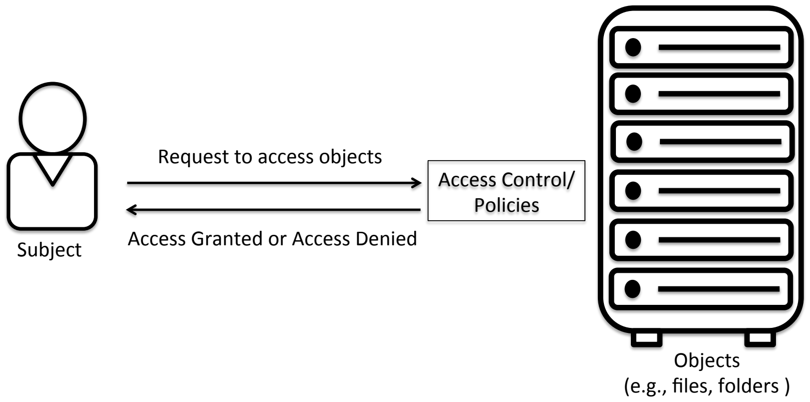

Figure 1.1: Basic Access Control

tering RBAC. It primarily involves administration of user-role assignment (URA), permission-role assignment (PRA) and, role-role assignment (RRA) among other operations such as creating and updating RBAC entities like users and permissions. It is evident that ARBAC is developed based on RBAC philosophy. Because like RBAC, ARBAC were designed to adapt simplicity, flexibility, and ease-of-use. The development of ARBAC models starting with ARBAC97 model [55] seems to have incorporated those qualities to their best extent. It will be later seen that the development of these models have evolved, and thus, ARBAC models that have been published in the literature cover many different important aspects necessary for controlled administration that obey rules of access control like least privilege and separation of duties.

Administrative RBAC is considered both critical and challenging task [55]. ARBAC focusses on assigning/revoking users to/from roles, permission to/from roles, etc. Many approaches have been proposed in the literature for ARBAC [10, 33, 37, 44, 55, 56, 64]. But among all different ARBAC models, five ARBAC models have been considered in this research work, as a foundation and motivation in developing an ARBAC model with a new approach. They are ARBAC97, ARBAC99, ARBAC02, Uni-ARBAC and UARBAC. Most of these models incorporate URA and PRA, while only few cover RRA [37, 55]. Moreover, most of these approaches are role-driven. For example, in URA97 [55], user-role assignment is determined based on prerequisite roles of the target user, in URA99 [56], it is determined based on the target users' mobile or immobile membership on roles. At the same time, these models are making use of few fixed set of properties of

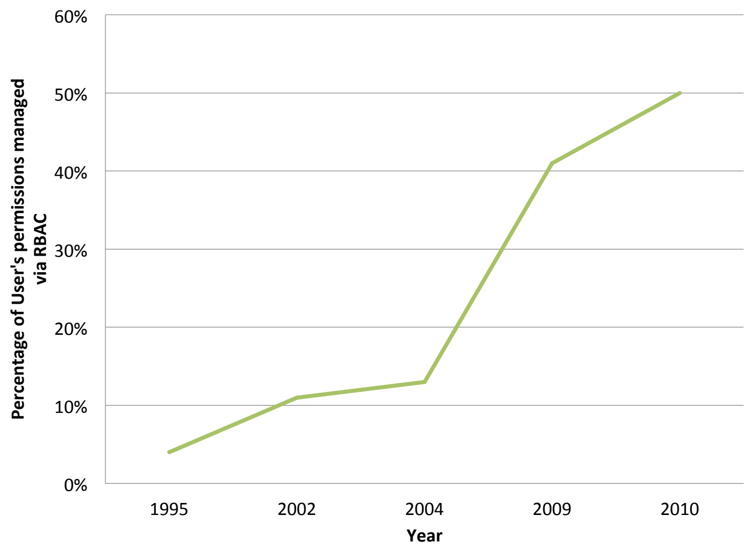

Figure 1.2: RBAC Adoption in IT Intensive Organizations [43]

RBAC in making assignment decisions. For instance, RRA97 presents a notion of authority range in RRA97 [55]. A user with admin role is given an authority range, over which that admin role can conduct administrative operations such as assigning a role to a role. In addition, it can be observed that latter models were developed either to incorporate or overcome (or both) the strengths of prior models. It can also be observed that every time a new feature was created and adopted, a new model had to be built. Therefore, there exists a need for an approach, in which features from previously developed models can be dynamically incorporated while at the same time this new approach would also allow the user to introduce new features.

Attribute-Based Access Control (ABAC) has recently gained a significant amount of attention because of the flexibility it offers [6, 11, 23, 24, 29, 35]. It has proven to have the ability to represent different access control models [29]. Moreover, ABAC has been applied in different technology domains such as the Cloud and the Internet of Things (IoT) [2, 4, 7–9, 30, 52]. However, the use cases so far are towards the operational aspect of access control and, the usage of ABAC for administrative purposes has not been thoroughly explored. This dissertation is an investigation of an attribute-based approach for administration of RBAC. In the context of ARBAC, attributes allow for more flexibility in specifying the conditions under which users and permissions can be assigned to roles. For instance, the notions of prerequisite roles in ARBAC97 [55], mobility of roles in ARBAC99 [56], and organization unit in ARBAC02 [44] can be captured as user attributes. The notion of administrative roles in the above models and the notion of administrative unit in Uni-ARBAC [10] can be captured as attributes of administrative users. Similarly, the notion of authority range in RRA97 [55] can be captured using admin role attributes.

This allows for the attribute-based models developed in this research to express any of these ARBAC models and beyond. That is, it allows the attribute-based models presented in this literature to express any combination of features from prior models, and new features that are not *intuitively* expressible in those prior models. Thus, this work is motivated largely by two critical factors: (a) since administrative RBAC has been fairly explored in the literature, it is appropriate to explore unification of these works into building a coherent model that can be configured to express prior models and beyond, and (b) a unified model can be analyzed *once* for various desirable security properties, and a *single* codebase can be generated to express prior models and beyond.

1.1 Problem Statement

Although administration of role-based access control has been well explored, almost all the prior models are fairly static and limited in what they can express. Any new property, if required to be incorporated in an administrative decision process, a new model had to be developed. In the real world scenarios, there may be a need for more than few properties of participating entities in making access control decisions. In addition, there may also be cases where new properties would be needed to be added at later time. Therefore, there exists a need for a platform where one can intuitively select or create features as needed in developing security policies.

ABAC's potential originates from its expressive power and flexibility. It has recently gained great attention, and thus has been explored well in the operational access control realm. However, it hasn't been explored well in administrative access control domain.

To this end, this work shall be driven with a confidence that access control models can be administered in more dynamic manner, and that ABAC properties can be leveraged to achieve flexibility in RBAC administration.

1.2 Scope and Assumption

The scope and assumptions of attribute based administrative models developed in this work are as follows:

- Attribute-based administration is a need for addressing present-day administrative access control problems.
- RBAC, ABAC and ARBAC models can be composed well for better access control.
- Minimum objective of this work is to build family of attribute-based administrative access control models that are sufficient to express prior models, and in addition, they can express new features for access control. Five different prior models, each for PRA and URA, and two prior models for RRA are considered as foundation for this study. This is one of the reasons why only few attributes from RBAC entities such as regular user and admin user are developed.
- RBAC is the foundation for ARBAC model. In RBAC roles are central to its design. Therefore, in the models developed in this research, roles are assumed to exist, and therefore system maintains a mapping between user and assigned roles.
- Expressing prior models in terms of proposed attribute-based models allows one with flexibility to use combination of one or more prior ARBAC models.

1.3 Thesis Statement

Thesis of this dissertation are as follows:

- Attribute-based approach can bring flexibility by its power of expression into RBAC administration.

- A unified and dynamic model that is sufficient to express prior ARBAC models, and that has potential to express many other properties can be developed using attribute-based approach.

1.4 Summary of Contribution

The contributions of this dissertation includes the following:

AURA

A model known as attribute-based user-role assignment (AURA) that unifies prior URA models (URA97, URA99, URA02, Uni-ARBAC's URA and UARBAC's URA) has been developed. To demonstrate that AURA is capable of expressing prior approaches to URA, a manual translation of an example instance from each prior model is first translated into equivalent AURA instance. Finally, a formal mapping algorithm to translate any instance from prior URA models to equivalent AURA instance is presented.

ARPA

A model known as attribute-based permission-role assignment (ARPA) that unifies prior PRA models (PRA97, PRA99, PRA02, Uni-ARBAC's PRA and UARBAC's PRA) has been developed. To demonstrate that ARPA model is capable of expressing prior approaches to PRA, a manual translation of an example instance from each prior model is first translated into equivalent ARPA instance. Then, a formal mapping algorithm that can translate any instance from prior PRA models to equivalent ARPA instance is presented.

ARRA

A model for attribute-based role-role assignment (ARRA) that unifies prior approaches to RRA has been developed.

To demonstrate that ARRA is capable of expressing prior RRA approaches such as RRA97 and UARBAC's RRA that are part of ARBAC97 and UARBAC, respectively, a manual translation of example instance from each prior model to an equivalent ARRA instance is presented. Then finally, a formal mapping algorithm to map any instance from prior RRA model to its equivalent ARRA model is developed.

Implementation of AURA

AURA model is implemented as a proof-of-concept into identity service of OpenStack Infrastructure as a Service (IaaS) cloud. Any time-overhead introduced due to newly added attributes and values, and its enforcement code is then analyzed.

1.5 Organization of Dissertation

This dissertation is organized as follows. In chapter 2, background and related work is discussed. It discusses all the administrative RBAC models that have been taken into account for study. It is followed by a brief compare-and-contrast among relevant ABAC works that have been explored.

AURA, ARPA and ARRA are members of AARBAC design. In Chapter 3, AURA model is presented. This includes a formal specification of AURA model along with supportive examples in which example instances from each of the prior URA model is manually translated into equivalent AURA instances. For the same process, formal translation (mapping) algorithms for each prior model is exhibited.

In Chapter 4, formal specification of attribute-based permission-role assignment (ARPA) model is presented. It is followed by example instances from prior models and their manual translation into equivalent ARPA instances. Finally, formal translation algorithms to translate instances from prior PRA models into equivalent ARPA instances is presented.

Similarly, in Chapter 5, a model for attribute-based role-role assignment (ARRA) model is discussed. Again, previously published RRA models are considered that can be realized with ARRA. Example instances for each cited prior RRA models are developed, which are then converted into equivalent ARRA instances based on attributes. Finally, translation algorithms that maps instance from prior model to equivalent ARRA instance are presented.

In Chapter 6, AURA model is integrated into the identity service of OpenStack IaaS cloud. In that chapter, general overview of OpenStack architecture is discussed and contrasted with the attribute-based approach. In the end, time overhead for various use cases that involves attribute-based user-role assignment and revocation is analyzed.

Chapter 7 concludes this dissertation with future research direction.

Chapter 2: BACKGROUND AND LITERATURE REVIEW

2.1 Related Work

This dissertation work focuses on attribute-based user-role assignment (AURA), attribute-based permission-role assignment (ARPA), attribute-based role-role assignment (ARRA) and, implementation of AURA in the identity service of OpenStack Cloud IaaS.

The very basis behind this research is to integrate attributes into existent RBAC model, and hence the foundation of this work is tied to RBAC. Literature beginning with RBAC96 model [19, 57], which is also a NIST stardard for role-based access control are reviewed in the following segments.

2.2 Role-Based Access Control Model

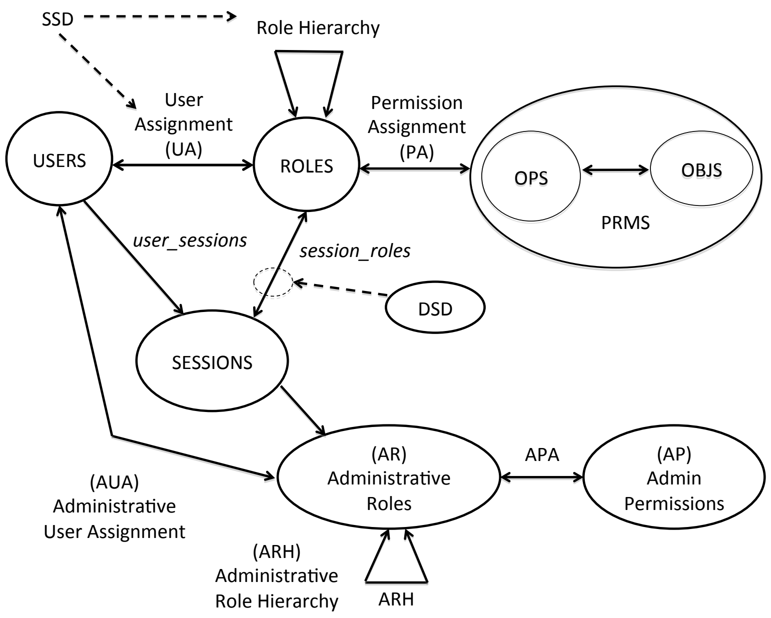

Figure 2.1: Role Based Access Control Model

Figure 2.1 presents RBAC96 model along with administrative components. In RBAC model, users are assigned to roles, permissions are assigned to roles, and roles are assigned to roles. A user is able to execute a permission by activating a subset of roles assigned to her, and hence activating permissions that has been assigned to those roles. A user initiates a session to activate one or more roles. RBAC model is also known as operational RBAC model. More advanced features of RBAC involves static separation of duties (SSD), dynamic separation of duties (DOD) and constraints. Role hierarchy defines how a senior role can inherit all the permissions from its junior roles, and how a user with a role senior to another role can become an implicit member of junior roles.

A user with administrative role can conduct administrative operations to assign or revoke a user to/from a role, assign or revoke a permission to/from a role, and assign or revoke a role to/from a role among other administrative authorities such as to create or delete users and roles. A model dedicated and designed for administration of these entities can be referred to as Administrative RBAC model (ARBAC).

2.3 Administrative RBAC Models

ARBAC97 [54, 55], ARBAC99 [56], ARBAC02 [44], Uni-ARBAC [10] and UARBAC [37] are some of the prominent past works in administrative RBAC. All these models deal with user-role and permission-role assignments, and some of them include role-role assignments.

In all these models, the policies for assigning users or permissions to roles are specified based on explicit and fixed set of properties of the relevant entities that are involved in the decision-making process. Some of them are the administrative user, the target role, the regular user or the permission (that is assigned to the role), and the administrative role.

2.3.1 ARBAC97

ARBAC97 [54, 55] model comprises of URA97, PRA97 and RRA97 models. In URA97, the properties that are used to make user-role assignment include the *admin role* of the administrative user and the *current set of roles* of the regular user. A user with an admin role can assign a regular user to a regular role based on the regular user's membership or non-membership on one or more roles.

Similarly, in PRA97 a permission is assigned to a role based on the permission's membership or non-membership on role(s).

RRA97 model deals with role-role assignments. Assigning a regular role to another regular role creates a role hierarchy. RRA97 assumes a lattice structure of role-hierarchy for regular roles and, a set of admin roles with hierarchy. An example of role hierarchy from RRA97 is depicted in Figure 5.1. In RRA97, each admin role is given an authority range over which that admin role has authority to either assign or revoke a role to/from a role, or create or delete a role. ARBAC97 is also driven by decentralization of tasks. In RRA7, it furthers the concept of autonomy of roles over their tasks. This is asserted by the notion of encapsulated ranges. An authority range must be an encapsulated range. A general idea about encapsulated range is that any admin role must be responsible for role administration within their authority range and, any operation should not violate the encapsulation of any authority range.

2.3.2 SARBAC

Crampton et. al [13, 14] present models for RRA with administrative scopes, which is a semi-response to the role-hierarchy intensive RRA [55]. Admin scopes are plausible approach for role hierarchy operations in RBAC. However, admin scope may not be intuitive to express as an attribute with models presented in this dissertation. Thus, administrative scope has been scoped out at this point.

2.3.3 ARBAC99

ARBAC99 includes two models namely, URA99 and PRA99. URA99 presents a notion of mobile and immobile memberships on a role. Mobile or immobile property of a user may be necessary when a user is assigned with temporary position (e.g., a contractor). In those scenarios, an immobile user cannot be assigned with any further roles. A decision on whether a regular user can be assigned/revoked to/from a role is based in user's mobile or immobile membership on some pre-assigned role.

Similarly in PRA99, decisions on whether a permissions can be assigned to a role is based on its mobile or immobile membership or non-membership on one or more roles.

2.3.4 ARBAC02

Like ARBAC99, ARBAC02 [45] is also composed of two different models for URA and PRA. In ARBAC02, a concept of organization units is explored. It distinguishes the structure and importance of user-pool and permission-pool by proposing organizational structure for user-pool and, organizational structure for permission-pool. Users in a user-pool form a hierarchy of organizational units. Similarly, permissions in permission-pool form a hierarchy of organizational units. However, their hierarchy is opposite to each other.

It comes from a real-world scenario where users are put into some organization unit, and based on that organization unit necessary roles can be assigned to users. However, URA02 also brings the property from URA97 along with it as an option. Thus, in URA02, if a user is a member or a non-member of one or more organization units then that user qualifies or disqualifies to be assigned to a given set of roles or, user's membership or non-membership on a role qualifies or disqualifies the user for role assignment to a given set of roles. Similar to PRA97 and PRA99, PRA02 is a dual of URA02.

2.3.5 Uni-ARBAC

In Uni-ARBAC [10], a set of users are put into a user-pools. The notion of user-pools and user-pool hierarchy are adopted from ARBAC02 [44]. Similarly, set of permissions are put together as tasks. Tasks are designed with a concept that a job may take different steps/permissions to complete. Uni-ARBAC provides a decentralized approach for separate user administration, and task administration. In which, an admin user is assigned to an administrative unit, either for user administration or permission administration. Regular roles, user-pools and tasks are mapped to admin units. If users (via user pools) and roles are mapped to the an admin unit where an admin user is assigned then that admin user can assign a user from users set to a role from role set mapped to same admin unit. PRA in Uni-ARBAC follows a similar approach where permissions are grouped as tasks, and then mapped to an admin unit. Any admin user that has been given task assignment authority with assignment to an admin unit can assign or revoke any role to/from the task that has been mapped to the admin unit. Other features include flow of authority according to the definition of hierarchy. For example, an admin user dedicated for user-role administration can conduct user-role assignment in an admin unit that she has been assigned to, and in addition, can perform similar actions in all the admin units that are junior to the admin unit (that she has been assigned to).

2.3.6 UARBAC

In UARBAC [37], the properties on the basis of which user-role assignment is done include a relationship based on *access modes* between the admin user, the target role and the target regular user. One of the driving principles of this model is principle of economy of mechanism. This allows the model to treat all the entities as objects. Hence, treated as the same. For example, a user, a file or a role are considered as objects.

One (an admin user) can grant a particular role to a target user, if she has access modes, grant on target role and empower on target user. Permission-role assignment follows similar approach with different sets of *access modes* towards permissions and roles.

In addition to URA and PRA, UARBAC also includes role-role assignment (RRA). Role assignment is done based on admin user's *access mode* towards each role or role class.

But because AURA, ARPA and ARRA are designed based on non-explicit and varying set of properties (attributes) of the relevant entities involved in the decision-making process, these models tend to be more flexible.

2.3.7 A Model for Attribute-Based User-Role Assignment

A closely related work is that of Al-Kahtani et. al [1], which presents a family of models for automated assignment of users to roles based on user attributes. The primary focus of their work is user-role assignment based on user attributes. Their model is also called Rule-Based Access

Control (RB-RBAC) model, as there are set of rules that decide what roles a user must take based on attributes of a user. Furthermore, those rules exercise hierarchy among them. Our models take a more holistic approach to RBAC administration based on attributes of various RBAC entities such as regular users, admin users and permissions. The major advantage of taking such an approach is that our models both subsume prior approaches to RBAC administration, and allow for specification of new features.

2.3.8 Other Administrative Models

Work by Kern et. al [34] presents a concept for administration of enterprise role-based access control (A-ERBAC). They make a plausible argument over how tasks for administrators may vary according to their association about specific sectors of an organization and, hence a need for decentralization of administrative tasks. They also define their own meaning of *scope* and, how it can help decentralize and ease administrative tasks. However, their work does not include criteria and constraints for administrative operations such as user-role assignment and revoking users from roles.

Organization-based administration of RBAC is explored in [15]. It discusses URA, PRA and UPA. It treats these assignments as views and, provides an abstraction for authority over these views. Although this model may be plausible to address scenarios in an organization, the notion doesn't intuitively fit to meet some concrete assignment criteria. This model is not taken into consideration for designing AARBAC at this time.

2.4 ABAC Models and Benefits of Using Attributes

Attribute-based access control has been well-studied [5, 24, 25, 29, 59, 62, 63]. Growing interest in Attribute Based Access Control (ABAC) has been because of its flexibility and expressive-power. ABAC has demonstrated that attributes are not limited to roles, subjects, users and objects but can incorporate many other properties including identities, affiliation, time of day, qualification, age, locations, etc. ABAC_α [29] has shown that ABAC model is able to demonstrate MAC, DAC and

RBAC and do much more.

ABAC has been popular because its simplicity and flexibility. For example, developing attribute-based rules for allowing or denying a user from getting a permission is relatively simple. Furthermore, rules can be flexible in that additional properties can be added within a rule, a rule can be pruned or combined with other rules.

ABAC is expressive. Assuming a computational language, ABAC's expressive power is only limited by the power of computational language. ABAC can provide high flexibly in that attributes can be used to identify large sets of users or system entities, and thus can be precisely controlled. On the other hand, rules for fine-grained control of system resources can also be developed.

The benefits of integrating attributes into an RBAC operational model has been investigated in the literature with wide range of applications [31,32,35]. However, our work focuses on advantages of using an attribute-based approach for RBAC administration. On the other hand, Jin et. al explore administration of attributes using RBAC [28]

Prior ABAC works primarily focus on operational aspects of access control—that is, making decisions when a user requests access to an object.

Integrating attribute based policies with RBAC can bring many powerful advantages from both ABAC and RBAC [48,49]. It is these advantages that are leveraged towards making secure assignment decision in administration of RBAC.

Chapter 3: AURA: ATTRIBUTE-BASED USER-ROLE ASSIGNMENT

Portion of materials in this chapter are published in the following venue [39]:

- Jiwan Ninglekhu and Ram Krishnan. AARBAC: Attribute-Based Administration of Role-Based Access Control. In 2017 IEEE 3nd International Conference on Collaboration and Internet Computing (CIC). IEEE, 2017.

In this section, an attribute-based administrative model for user-role assignment (AURA) is presented. Figure 3.1 illustrates a conceptual model for AURA and ARPA in the same diagram. Left hand side of the diagram in Figure 3.1 shows attribute-based user-role assignment while the right hand side of the diagram shows attribute-based permission-role assignment. It shows that the entities collectively involved in AURA include admin users and their associated attributes, regular users and their attributes, roles with a hierarchy, and the administrative operations. Similarly, it shows that permission-role assignment (PRA) decision is based on the rules written based on the admin user and their attributes, permissions and their attributes, and the target role.

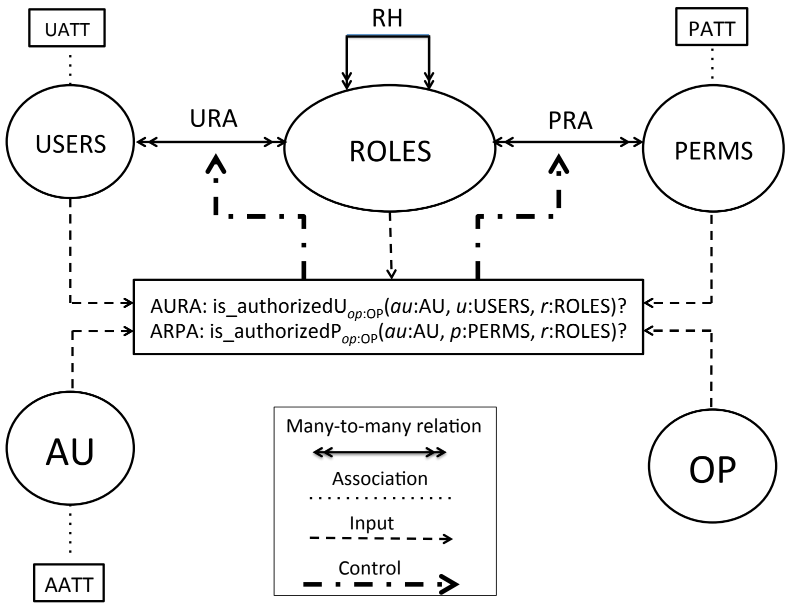
UATT
RH
PATT
USERS
URA
ROLES
PRA
PERMS
AURA: is_authorizedU$_{op:OP}$(*au*:AU, *u*:USERS, *r*:ROLES)?
ARPA: is_authorizedP$_{op:OP}$(*au*:AU, *p*:PERMS, *r*:ROLES)?
AU
OP
AATT
Many-to-many relation
Association
Input
Control

Figure 3.1: Attribute-Based Administration of RBAC

Note that this dissertation also includes a model for attribute-based role-role assignment (ARRA).

Assigning a role to another role is assumed to form a role hierarchy. Although AURA and ARPA models are presented prior to ARRA, both models assume roles with a hierarchy present.

3.1 AURA Model

AURA adapts a notion of admin users who are responsible for controlling the user-role assignment (URA) relation. Thus, in AURA, authorization decisions for assigning a regular user to a role is made based on attributes of the admin user and that of the regular user.

Table 3.1 presents the formal AURA model. As illustrated in Figure 3.1, the entities involved in AURA include regular users (USERS), admin users (AU), roles (ROLES) with a role hierarchy (RH), and admin operations (AOP). The goal of AURA is to allow for an admin user in AU to perform an admin operation such as assign and revoke in AOP between a regular user in USERS and a role in ROLES, by using attributes of various entities. To meet this goal, a set of attribute functions for the regular users (UATT) and admin users (AATT) are defined. One of the motivations for developing AURA is to provide AURA with the ability to capture the features of prior URA models such as URA97, URA99, and URA02. To this end, the only required attributes are the attributes of regular users and admin users. While one can envision attributes for other entities in AURA such as attributes for AOP, the scope of the model is limited based on the above-mentioned motivations. In addition, a system maintained user attribute function called *assigned_roles* is assumed to exist, which maps each user to set of roles currently assigned to them. Although the notion of roles can be captured as a user attribute function in UATT, this design choice was made in order to reflect the fact that role is not an optional attribute in the context of AURA.

The attribute functions (or simply attributes) are defined as a mapping from its domain (USERS or AU as the case may be) to its range. The range of an attribute *att*, which can be atomic or set valued, is derived from a specified set of scope of atomic values, denoted Scope(*att*). Whether an attribute is atomic or set valued is specified by a function called attType. Also, the scope of an attribute can be either ordered or unordered, which is specified by a function called is_ordered. If an attribute *att* is ordered, a corresponding hierarchy, denoted H_{att}, should be specified on its scope

Table 3.1: AURA Model

– USERS is a finite set of regular users.

– AU is a finite set of administrative users.

– AOP is a finite set of admin operations such as assign and revoke.

– ROLES is a finite set of regular roles.

– RH $\subseteq$ ROLES $\times$ ROLES, a partial ordering on the set ROLES.

We assume a system maintained user attribute function called *assigned_roles* that specifies the roles assigned to various regular users as follows:
– *assigned_roles* : USERS $\rightarrow 2^{\text{ROLES}}$

– UATT is a finite set of regular user attribute functions.

– AATT is a finite set of administrative user attribute functions.

– For each *att* in UATT $\cup$ AATT, Scope(*att*) is a finite set of atomic values from which the range of the attribute function *att* is derived.

– attType : UATT $\cup$ AATT $\rightarrow$ {set, atomic}, which specifies whether the range of a given attribute is atomic or set valued.

– Each attribute function maps elements in USERS and AU to atomic or set values.

$$\forall uatt \in \text{UATT}.\ uatt : \text{USERS} \rightarrow \begin{cases} \text{Scope}(uatt) \text{ if attType}(uatt) = \text{atomic} \\ 2^{\text{Scope}(uatt)} \text{ if attType}(uatt) = \text{set} \end{cases}$$

$$\forall aatt \in \text{AATT}.\ aatt : \text{AU} \rightarrow \begin{cases} \text{Scope}(aatt) \text{ if attType}(aatt) = \text{atomic} \\ 2^{\text{Scope}(aatt)} \text{ if attType}(aatt) = \text{set} \end{cases}$$

– is_ordered : UATT $\cup$ AATT $\rightarrow$ {True, False}, specifies if the scope is ordered for each of the attributes.
– For each *att* $\in$ UATT $\cup$ AATT,
if is_ordered(*att*) = True, $\text{H}_{att} \subseteq$ Scope(*att*) $\times$ Scope(*att*), a partially ordered attribute hierarchy, and $\text{H}_{att} \neq \phi$,
else, if is_ordered(*att*) = False, $\text{H}_{att} = \phi$

(For some $att \in$ UATT $\cup$ AATT for which attType(*att*) = set and is_ordered(*att*) = True, if $\{a, b\}$, $\{c, d\} \in 2^{\text{Scope}(att)}$ (where $a, b, c, d \in$ Scope(att)), we infer $\{a, b\} \geq \{c, d\}$ if (a, c), (a, d), (b, c), $(b, d) \in \text{H}^*_{att}$.)

AURA model allows an administrator to perform an operation on a single user or a set of users at a time. The authorization rule for performing an operation on a single user is as follows:
For each *op* in AOP, **is_authorizedU$_{op}$**(*au*: AU, *u* : USERS, *r* : ROLES) specifies if the admin user *au* is allowed to perform the operation *op* (e.g. assign, revoke, etc.) between the regular user *u* and the role *r*. This rule is written as a logical expression using attributes of the admin user *au* and attributes of the regular user *u*.

The authorization rule for performing an operation on a set of users is as follows:

For each *op* in AOP, **is_authorizedU$_{op}$**(*au*: AU, χ : 2^{USERS}, *r* : ROLES) specifies if the admin user *au* is allowed to perform the operation *op* (e.g. assign, revoke, etc.) between the users in the set χ and the role *r*.
Here χ is a set of users that can be specified using a set-builder notation, whose rule is written using user attributes.

Scope(*att*). H_{att} is a partial ordering on Scope(*att*). Note that, even in the case of a set valued attribute *att*, the hierarchy H_{att} is specified on Scope(*att*) instead of $2^{\text{Scope}(att)}$. It is inferred that the ordering between two set values given as ordering on atomic values as explained in Table 3.1. Note that H^{*}_{att} denotes the reflexive transitive closure of H_{att}.

AURA supports two ways to select a set of regular users for assigning a role. The first one allows an admin user to identify a single regular user, a role and perform an operation such as assign. The second one allows an admin user to identify a set of regular users, a role and perform an operation such as assign for all those regular users. In this case, the selection criteria for the set of regular users can be specified using a set-builder notation whose rule is stated using the regular users' attributes. For example, is_authorized$\text{U}_{\textbf{assign}}$(**au**, $\{u \mid u \in \text{USERS} \wedge \textbf{aunit\_1} \in$ *admin_unit*$(u)\}$, **r**) would specify a policy for an admin user **au** who identifies the set of all users who belong to the admin unit **aunit_1** in order to assign a role **r** to all those users. Finally, the authorization rule is specified as a usual logical expression on the attributes of admin users and those of regular users in question.

3.2 Mapping Prior URA Models in AURA

In the sections that immediately follow, the idea that AURA can intuitively simulate the features of prior URA models is demonstrated. In particular, these sections first present example instances for each of the prior URA models which are then mapped into their corresponding equivalent instances in AURA model. Finally, for each prior model, concrete algorithms that can convert any instance of URA97, URA99, URA02, the URA model in UARBAC, and the URA model in Uni-ARBAC into their corresponding equivalent instance of AURA are exhibited.

3.3 URA97 in AURA

One of the earliest user-role assignment models is URA97. The following sections present a detail approach on translating an instance of URA97 and the way by which we can produce its equivalent representation as an AURA instance. Summary of URA97 model is presented first.

3.3.1 Summary of URA97 Model

– USERS is a finite set of regular users.

– AR is a finite set of administrative roles.

– ROLES is a finite set of regular roles.

– RH $\subseteq$ ROLES $\times$ ROLES a partial order on roles.

– CR is a finite set of prerequisite conditions.

A *prerequisite condition* is a boolean expression using the usual $\wedge$ and $\vee$ operators on terms of form *x* and $\bar{x}$ where *x* is a regular role (i.e., $x \in$ ROLES).

URA97 Grant Model:

The URA97 model controls the assignment by means of the relation,

$$can\_assign \subseteq \text{AR} \times \text{CR} \times 2^{\text{ROLES}}.$$

The meaning of *can_assign*(*x, y,* {*a, b, c*}) is that a member of the administrative role *x* (or an administrative role that is senior to *x*) can assign a user whose current membership, or non-membership in a regular role satisfies the prerequisite condition *y* to be a member of regular roles *a, b,* or *c*.

A prerequisite condition is evaluated for a user *u* by interpreting *x* to be true if ($\exists x' \geq x$) (*u, x'*) $\in$ UA and $\bar{x}$ to be true if ($\forall x' \geq x$) (*u, x'*) $\notin$ UA.

URA97 Revoke Model:

The URA97 model controls user-role revocation by means of the following relation:

$$can\_revoke \subseteq \text{AR} \times 2^{\text{ROLES}}.$$

The meaning of *can_revoke*(*x, Y*) is that a member of the administrative role *x* (or a member of an administrative role that is senior to *x*) can revoke membership of a user from any regular role $y \in Y$. *Y* defines the range of revocation.

3.3.2 URA97 Instance

An example instance for URA97 can be expressed as the following:

Sets and Functions:

- USERS = $\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}\}$
- ROLES = $\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \mathbf{x_4}, \mathbf{x_5}, \mathbf{x_6}\}$
- AR = $\{\mathbf{ar_1}, \mathbf{ar_2}\}$
- UA = $\{(\mathbf{u_1}, \mathbf{x_1}), (\mathbf{u_1}, \mathbf{x_2}), (\mathbf{u_2}, \mathbf{x_3}), (\mathbf{u_2}, \mathbf{x_4})\}$
- AUA = $\{(\mathbf{u_3}, \mathbf{ar_1}), (\mathbf{u_4}, \mathbf{ar_2})\}$
- RH = $\{<\mathbf{x_1}, \mathbf{x_2}>, <\mathbf{x_2}, \mathbf{x_3}>, <\mathbf{x_3}, \mathbf{x_4}>, <\mathbf{x_4}, \mathbf{x_5}>, <\mathbf{x_5}, \mathbf{x_6}>\}$
- ARH = $\{<\mathbf{ar_1}, \mathbf{ar_2}>\}$
- CR = $\{\mathbf{x_1} \wedge \mathbf{x_2}, \bar{\mathbf{x}}_\mathbf{1} \vee (\bar{\mathbf{x}}_\mathbf{2} \wedge \mathbf{x_3})\}$

Let cr_1 = $\mathbf{x_1} \wedge \mathbf{x_2}$ and, cr_2 = $\bar{\mathbf{x}}_\mathbf{1} \vee (\bar{\mathbf{x}}_\mathbf{2} \wedge \mathbf{x_3})$ be two prerequisite conditions. Prerequisite condition cr_1 is evaluated as follows:

For any u in USERS undertaken for assignment,

$(\exists x \geq \mathbf{x_1}).\ (u, x) \in \text{UA} \wedge (\exists x \geq \mathbf{x_2}).\ (u, x) \in \text{UA}$

cr_2 is evaluated as follows:

For any u in USERS undertaken for assignment,

$(\forall x \geq \mathbf{x_1}).\ (u, x) \notin \text{UA} \vee ((\forall x \geq \mathbf{x_2}).\ (u, x) \notin \text{UA} \wedge (\exists x \geq \mathbf{x_3}).\ (u, x) \in \text{UA})$

Let *can_assign* and *can_revoke* be as follows:

can_assign = $\{(\mathbf{ar_1}, cr_1, \{\mathbf{x_4}, \mathbf{x_5}\}), (\mathbf{ar_1}, cr_2, \{\mathbf{x_6}\})\}$

can_revoke = $\{(\mathbf{ar_1}, \{\mathbf{x_4}, \mathbf{x_5}, \mathbf{x_6}\})\}$

3.3.3 Equivalent URA97 Instance in AURA

In this segment, AURA instance equivalent to aforementioned URA97 instance in presented based on the AURA model depicted in Table 3.1.

Map sets and functions from URA97 to AURA

- USERS $=\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}\}$
- AU $= \{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}\}$
- AOP = {**assign, revoke**}
- ROLES $= \{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \mathbf{x_4}, \mathbf{x_5}, \mathbf{x_6}\}$
- RH $= \{<\mathbf{x_1}, \mathbf{x_2}>, <\mathbf{x_2}, \mathbf{x_3}>, <\mathbf{x_3}, \mathbf{x_4}>, <\mathbf{x_4}, \mathbf{x_5}>, <\mathbf{x_5}, \mathbf{x_6}>\}$
- $assigned\_roles(\mathbf{u_1}) = \{\mathbf{x_1}, \mathbf{x_2}\}$, $assigned\_roles(\mathbf{u_2}) = \{\mathbf{x_3}, \mathbf{x_4}\}$

Define attributes and values

- UATT = { }
- AATT = {*aroles*}

  Scope(*aroles*) $= \{\mathbf{ar_1}, \mathbf{ar_2}\}$,

  attType(*aroles*) = set,

  is_ordered(*aroles*) = True,

  $H_{aroles} = \{<\mathbf{ar_1}, \mathbf{ar_2}>\}$
- $aroles(\mathbf{u_3}) = \{\mathbf{ar_1}\}$, $aroles(\mathbf{u_4}) = \{\mathbf{ar_2}\}$

Construct authorization function for user-role assignment:

Authorization rule for user-role assignment for the given instance can be expressed as follows:

– $\text{is\_authorizedU}_{\mathbf{assign}}(au : \text{AU}, u : \text{USERS}, r : \text{ROLES}) \equiv ((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(au) \land r \in \{\mathbf{x_4}, \mathbf{x_5}\} \land ((\exists x \geq \mathbf{x_1}).\ x \in assigned\_roles(u) \land (\exists x \geq \mathbf{x_2}).\ x \in assigned\_roles(u)) \lor (\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(au) \land r \in \{\mathbf{x_6}\} \land ((\exists x \geq \mathbf{x_1}).\ x \notin assigned\_roles(u) \lor ((\exists x \geq \mathbf{x_2}).\ x \notin assigned\_roles(u) \land (\exists x \geq \mathbf{x_3}).\ x \in assigned\_roles(u)))$

Construct authorization function for revoking user from role:

Authorization rule for user-role revocation for the given instance can be expressed as follows:

– is_authorizedU$_{\mathbf{revoke}}$(au : AU, u : USERS, r : ROLES) $\equiv (\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(au) \wedge$ $r \in \{\mathbf{x_4, x_5, x_6}\}$

The manual translation process primarily involves four steps. First, the sets from entities involved in URA97 instance is mapped equivalently into sets in AURA instance. The notion of admin users doesn't exist in ARBAC97 model. Hence, the user pool for admin user and regular user is the same set, equivalent to $\{\mathbf{u_1, u_2, u_3, u_4}\}$. To map the assign and revoke in *can_assign* and *can_revoke*, administrative operations **assign** and **revoke** are used. ROLES and RH represent set of roles and role hierarchy, respectively. *assigned_roles*(u) function provides information on UA relation from URA97. This function is assumed to exist because RBAC model is certain to have roles. In second step, appropriate attributes are engineered that intuitively represent features from URA97. In particular user attribute set UATT is kept empty as user attributes are not required to represent feature from URA97 and, the relation between admin user and admin roles are captured by admin user attribute *aroles*(au). This attribute yields admin roles that are assigned to a given admin user. For example, *aroles*($\mathbf{u_3}$) yields admin role $\mathbf{ar_1}$, which means admin user $\mathbf{u_3}$ has role $\mathbf{ar_1}$. For attribute *aroles*, its corresponding attribute type, scope and order are defined. H_{aroles} represents partial order hierarchy on admin roles. In third step, authorization rules that is equivalent to URA97 instance's user-role assignment authorization function based on attributes is constructed. Finally, authorization function for revoking user from role based on attributes is constructed.

3.3.4 MAP$_{\mathbf{URA97}}$ Algorithm

Map$_{\text{URA97}}$ is an algorithm that maps a URA97 instance into its equivalent AURA instance. For brevity, sets and functions from URA97 and AURA are marked with superscripts `97` and `A`, respectively. Map$_{\text{URA97}}$ takes URA97 instance as its input. In particular, inputs for Map$_{\text{URA97}}$ fundamentally consists of USERS^{97}, ROLES^{97}, AR^{97}, UA^{97} and AUA^{97}, RH^{97}, ARH^{97}, *can_assign*^{97} and *can_revoke*^{97}.

The output of Map$_{\text{URA97}}$ algorithm is an equivalent AURA instance, with primarily consisting of USERS^{A}, AU^{A}, AOP^{A}, ROLES^{A}, RH^{A}, For each $u \in$ USERS^{A}, *roles*^{A}(u), UATT^{A}, AATT^{A}, For

Algorithm 3.1 Map$_{\text{URA97}}$

```
    Input: URA97 instance

    Output: AURA instance

Step 1:    /* Map basic sets and functions in AURA */

    a. USERS ← $\text{USERS}^{97}$ ; AU ← $\text{USERS}^{97}$ ; $\text{AOP}^{\mathtt{A}}$ ← {assign, revoke}

    b. $\text{ROLES}^{\mathtt{A}}$ ← $\text{ROLES}^{97}$ ; $\text{RH}^{\mathtt{A}}$ ← $\text{RH}^{97}$

    c. For each $u \in \text{USERS}^{\mathtt{A}}$, $roles(u) = \phi$

    d. For each $(u, r) \in \text{UA}^{\mathtt{A}}$, $roles(u) = roles(u) \cup r$

Step 2:    /* Map attribute functions in AURA */

    a. $\text{UATT}^{\mathtt{A}} \leftarrow \phi$ ; $\text{AATT}^{\mathtt{A}} \leftarrow \{aroles\}$

    b. Range($aroles$) = $\text{AR}^{97}$ ; attType($aroles$) = set ; is_ordered($aroles$) = True ; $\text{H}_{aroles} \leftarrow \text{ARH}^{97}$

    c. For each $u \in \text{AU}^{\mathtt{A}}$, $aroles(u) = \phi$ ; For each $(u, ar)$ in $\text{AUA}^{97}$, $aroles(u) = aroles(u) \cup ar$

Step 3:    /* Construct assign rule in AURA */

    a. assign_formula = $\phi$

    b. For each $(ar, cr, Z) \in can\_assign^{97}$,

        assign_formula' = assign_formula $\vee$ $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z \wedge$

        $(translate_{97}(cr)))$

    c. auth_assign = is_authorizedU$_{\textbf{assign}}$($au : \text{AU}^{\mathtt{A}}, u : \text{USERS}^{\mathtt{A}}, r : \text{ROLES}^{\mathtt{A}}$) $\equiv$ assign_formula'

Step 4:    /* Construct revoke rule for AURA */

    a. revoke_formula = $\phi$

    b. For each $(ar, cr, Z) \in can\_revoke^{97}$

        revoke_formula' = revoke_formula $\vee$ $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z)$

    c. auth_revoke = is_authorizedU$_{\textbf{revoke}}$($au : \text{AU}^{\mathtt{A}}, u : \text{USERS}^{\mathtt{A}}, r : \text{ROLES}^{\mathtt{A}}$) $\equiv$ revoke_formula'
```

Support routine for algorithm 3.1 *translate*$_{97}$

```
Input: A URA97 prerequisite condition, $cr$
Output: An equivalent sub-rule for AURA authorization assign rule.
1: $rule\_string = \phi$
2: For each $symbol$ in $cr$,
3:     if $symbol$ is a role and in the form $x$ (i.e., the user holds role $x$)
4:         $rule\_string' = rule\_string + (\exists x' \geq x).\ x' \in roles(u)$
5:     else if $symbol$ is a role and in the form $\bar{x}$ (i.e., the user doesn't hold role $x$)
6:         $rule\_string' = rule\_string + (\exists x' \geq x).\ x' \notin roles(u)$
7:     else
8:         $rule\_string' = rule\_string + symbol$    /* where a $symbol$ is a $\wedge$ or $\vee$ logical operator */
9:     end if
```

each attribute $att \in \text{UATT}^{\mathbb{A}} \cup \text{AATT}^{\mathbb{A}}$, $\text{Scope}^{\mathbb{A}}(att)$, $\text{attType}^{\mathbb{A}}(att)$, $\text{is\_ordered}^{\mathbb{A}}(att)$ and $\text{H}^{\mathbb{A}}{}_{att}$, For each user $u \in \text{USERS}^{\mathbb{A}}$, and for each $att \in \text{UATT}^{\mathbb{A}} \cup \text{AATT}^{\mathbb{A}}$, $att(u)$, Authorization rule for assign (auth_assign) and Authorization rule for revoke (auth_revoke).

As depicted in $\text{Map}_{\text{URA97}}$, there are four main steps in the traslation process. In Step 1, sets and functions from URA97 are mapped into equivalent AURA sets and functions. In Step 2, user attributes and administrative user attribute functions are expressed. It can be observed that necessity for user attributes doesn't exist in representing an equivalent AURA instance for URA97. Hence, UATT set is empty. Admin user attribute *aroles* captures the notion of admin roles in URA97. *aroles* is a function that takes admin user as an input and yields all the roles that admin user has. Step 3 involves constructing assign_formula in AURA that is equivalent to *can_assign*^{97} in URA97 but are based on the entities that apply and their related attributes. *can_assign*^{97} is a set of triples. Each triple bears an access control policy on whether an admin user with admin role can assign a candidate user to a set of roles. Equivalent translation of *can_assign*^{97} in AURA is given by $\text{is\_authorizedU}_{\textbf{assign}}(au : \text{AU}^{\mathbb{A}}, u : \text{USERS}^{\mathbb{A}}, r : \text{ROLES}^{\mathbb{A}})$. Similarly, In Step 4, revoke_formula equivalent to *can_revoke*^{97} is presented.

translate$_{97}$ is a piece of algorithm considered as a support routine for Map$_{\text{URA97}}$ which translates prerequisite condition (*cr*) from URA97 into its AURA equivalent.

3.4 URA99 in AURA

Second, URA99 model was studied. This section presents a summary of URA99 model followed by a translation process of a URA99 model instance to its equivalent represenation in AURA model. It is first described in detail as a manual translation, and then a formal algorithm for mapping URA99 to AURA is presented.

3.4.1 Summary of URA99 Model

Sets and functions:

– USERS is a finite set of regular users.

– ROLES is finite set of regular roles.

– AR is a finite set of administrative roles.

– CR is set of prerequisite conditions.

A *prerequisite condition* is a boolean expression using the usual $\wedge$ and $\vee$ operators in terms of form x and $\bar{x}$ where x is a regular role (i.e. $x \in$ ROLES).

URA99 assumes two sub-roles of $x \in$ ROLES to be *Mx and IMx*. Membership of a user *u* in *Mx* and *IMx* are called mobile membership and immobile membership, respectively.

There are four kinds of user-role membership in URA99 for a given role *x*. They are as follows:

Explicit mobile membership: $EMx \equiv (u, Mx) \in$ UA

Explicit immobile membership: $EIMx \equiv (u, IMx) \in$ UA

Implicit mobile membership: $ImMx \equiv (\exists x' > x)(u, Mx') \in$ UA

implicit immobile membership: $ImIMx \equiv (\exists x' > x)(u, IMx') \in$ UA

URA99 Grant Model:

User-role assignments as mobile or immobile members are authorized by the following relation:

$$\textit{can-assign-M} \subseteq \text{AR} \times \text{CR} \times 2^{\text{ROLES}}$$

The meaning of *can-assign-M*(x, y, $\{a, b, c\}$) is that a member of administrative role x (or a member of administrative role senior to x) can assign a user whose current membership, or non membership in roles satisfies the prerequisite y to a regular roles a, b or c, as a mobile member.

Similar is the definition for assigning a qualifying user as immobile member of regular roles given by the relation:

$$\textit{can-assign-IM} \subseteq \text{AR} \times \text{CR} \times 2^{\text{ROLES}}$$

A prerequisite condition is a boolean expression using the $\wedge$ and $\vee$ operators in terms of the form r and $\bar{x}$.

The prerequisite condition in URA99 grant model is evaluated for a user u, by interpreting x to be true if:

$$u \in EMx \vee (u \in ImMx \wedge u \notin EIMx)$$

and $\bar{x}$ to be true if:

$$u \notin EMx \wedge u \notin EIMx \wedge u \notin ImMx \wedge u \notin ImIMx$$

URA99 Revoke Model:

The URA99 model authorizes revocation of mobile membership by the relation:

$$\textit{can-revoke-M} \subseteq \text{AR} \times \text{CR} \times 2^{\text{ROLES}}$$

and revocation of immobile membership by the relation:

$$\textit{can-revoke-IM} \subseteq \text{AR} \times \text{CR} \times 2^{\text{R}}$$

The meaning of *can-revoke-M*(x, y, $\{a, b, c\}$) is that a member of administrative role x (or a member of a administrative role senior to x) can revoke mobile membership of a user from role a, b or c subject to the prerequisite condition y. Similarly for *can-revoke-IM* with respect to immobile membership.

There exists same interpretation for mobile and immobile membership in URA99's revoke model. A prerequisite condition in URA99 revoke model is evaluated for a user u by interpreting x to be true if:

$$u \in EMx \vee u \in EIMx \vee u \in ImMx \vee u \in ImIMx$$

and $\bar{x}$ to be true if:

$$u \notin EMx \wedge u \notin EIMx \wedge u \notin ImMx \wedge u \notin ImIMx$$

Note that unlike the grant model, x and $\bar{x}$ cannot be false at the same time.

3.4.2 URA99 Instance

In this segment, an example instance of URA99 model is presented as follows:

Sets and functions:

- USERS = $\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}\}$
- ROLES = $\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \mathbf{x_4}, \mathbf{x_5}, \mathbf{x_6}\}$
- AR = $\{\mathbf{ar_1}, \mathbf{ar_2}\}$
- UA = $\{(\mathbf{u_1}, \mathrm{M}\mathbf{x_1}), (\mathbf{u_2}, \mathrm{IM}\mathbf{x_1}), (\mathbf{u_2}, \mathrm{IM}\mathbf{x_2}), (\mathbf{u_1}, \mathrm{IM}\mathbf{x_3})\}$
- AUA = $\{(\mathbf{u_3}, \mathbf{ar_1}), (\mathbf{u_4}, \mathbf{ar_2})\}$
- RH = $\{<\mathbf{x_1}, \mathbf{x_2}>, <\mathbf{x_2}, \mathbf{x_3}>, <\mathbf{x_3}, \mathbf{x_4}>, <\mathbf{x_4}, \mathbf{x_5}>, <\mathbf{x_5}, \mathbf{x_6}>\}$
- ARH = $\{<\mathbf{ar_1}, \mathbf{ar_2}>\}$
- CR = $\{\mathbf{x_1} \wedge \mathbf{x_2}, \bar{\mathbf{x}}_\mathbf{1}\}$

Let $cr_1 = \mathbf{x_1} \wedge \mathbf{x_2}$ and, $cr_2 = \bar{\mathbf{x}}_\mathbf{1}$. cr_1 is evaluated as follows:

For any user $u \in$ USERS undertaken for assignment,

$((u, \mathrm{M}\mathbf{x_1}) \in \mathrm{UA} \vee ((\exists x' \geq \mathbf{x_1}).\ (u, \mathrm{M}x') \in \mathrm{UA}) \wedge (u, \mathrm{IM}\mathbf{x_1}) \notin \mathrm{UA})) \wedge ((u, \mathrm{M}\mathbf{x_2}) \in \mathrm{UA} \vee$

$((\exists x' \geq \mathbf{x_2}).\ (u, \mathrm{M}x') \in \mathrm{UA}) \wedge (u, \mathrm{IM}\mathbf{x_2}) \notin \mathrm{UA})$

cr_2 is evaluated as follows:

For any user $u \in$ USERS undertaken for assignment,

$(u, \mathrm{M}\mathbf{x_1}) \notin \mathrm{UA} \wedge ((\exists x' \geq \mathbf{x_1}).\ (u, \mathrm{M}x') \notin \mathrm{UA}) \wedge (u, \mathrm{IM}\mathbf{x_1}) \notin \mathrm{UA} \wedge ((\exists x' \geq \mathbf{x_1}).\ (u, \mathrm{IM}x') \notin \mathrm{UA})$

Let *can-assign-M* and *can-assign-IM* in URA99 be as follows:

can-assign-M = $\{(\mathbf{ar_1}, cr_1, \{\mathbf{x_4}, \mathbf{x_5}\})\}$

can-assign-IM= $\{(\mathbf{ar_1}, cr_2, \{\mathbf{x_5}, \mathbf{x_6}\})\}$

Unlike URA97, there is a notion of prerequisite condition in URA99 revoke model. Same prerequisite conditions for both grant and revoke models instances have been considered for simplicity. Prerequisite conditions for URA99 revoke model are evaluated as follows:

cr_1 is evaluated as follows:

For any user $u \in$ USERS that needs to be revoked,

$((u, \mathrm{M}\mathbf{x_1}) \in \mathrm{UA} \vee (u, \mathrm{IM}\mathbf{x_1}) \in \mathrm{UA} \vee ((\exists x' \geq \mathbf{x_1}).\ (u, \mathrm{M}x') \in \mathrm{UA}) \vee ((\exists x' \geq \mathbf{x_1}).\ (u, \mathrm{IM}x') \in \mathrm{UA}))$

$\wedge ((u, \mathrm{M}\mathbf{x_2}) \in \mathrm{UA} \vee (u, \mathrm{IM}\mathbf{x_2}) \in \mathrm{UA} \vee ((\exists x' \geq \mathbf{x_2}).\ (u, \mathrm{M}x') \in \mathrm{UA}) \vee$

$((\exists x' \geq \mathbf{x_2}).\ (u, \mathrm{IM}x') \in \mathrm{UA}))$

cr_2 is evaluated as follows:

For any user $u \in$ USERS that needs to be revoked,

$(u, \mathrm{M}\mathbf{x_1}) \notin \mathrm{UA} \wedge (u, \mathrm{IM}\mathbf{x_1}) \notin \mathrm{UA} \wedge ((\exists x' \geq \mathbf{x_1}).\ (u, \mathrm{M}x') \notin \mathrm{UA}) \wedge ((\exists x' \geq \mathbf{x_1}).\ (u, \mathrm{IM}x') \notin \mathrm{UA})$

can-revoke-M and *can-revoke-IM* are as follows:

can-revoke-M = $\{(\mathbf{ar_1}, cr_1, \{\mathbf{x_3}, \mathbf{x_4}, \mathbf{x_5}\})\}$

can-revoke-IM= $\{(\mathbf{ar_1}, cr_2, \{\mathbf{x_5}, \mathbf{x_6}\})\}$

3.4.3 Equivalent URA99 Instance in AURA

An equivalent AURA instance for aforementioned URA99 example instance is presented in this segment.

<u>Set and functions from URA99 to AURA:</u>

- USERS = $\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}\}$
- AU = $\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}\}$
- ROLES = $\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \mathbf{x_4}, \mathbf{x_5}, \mathbf{x_6}\}$

- RH = {<$\mathbf{x_1}$, $\mathbf{x_2}$>, <$\mathbf{x_2}$, $\mathbf{x_3}$>, <$\mathbf{x_3}$, $\mathbf{x_4}$>, <$\mathbf{x_4}$, $\mathbf{x_5}$>, <$\mathbf{x_5}$, $\mathbf{x_6}$>}
- *assigned_roles*($\mathbf{u_1}$) = {M$\mathbf{x_1}$, IM$\mathbf{x_3}$},

  assigned_roles($\mathbf{u_2}$) = {IM$\mathbf{x_1}$, IM$\mathbf{x_3}$}
- AOP = {**mob-assign, immob-assign, mob-revoke, immob-revoke**}

Define attributes and its values:

- UATT= {*exp_mob_mem, imp_mob_mem, exp_immob_mem, imp_immob_mem*}
- Scope(*exp_mob_mem*) = ROLES, attType(*exp_mob_mem*) = set,

  is_ordered(*exp_mob_mem*) = True

  $H_{exp\_mob\_mem}$ = RH
- *exp_mob_mem*($\mathbf{u_1}$) = {$\mathbf{x_1}$}, *exp_mob_mem*($\mathbf{u_2}$) = {},

  exp_mob_mem($\mathbf{u_3}$) = {}, *exp_mob_mem*($\mathbf{u_4}$) = {},
- Scope(*imp_mob_mem*)= ROLES, attType(*imp_mob_mem*) = set,

  is_ordered(*imp_mob_mem*) = True, $H_{imp\_mob\_mem}$ = RH
- *imp_mob_mem*($\mathbf{u_1}$) = {$\mathbf{x_2}$, $\mathbf{x_3}$, $\mathbf{x_4}$, $\mathbf{x_5}$, $\mathbf{x_6}$}, *imp_mob_mem*($\mathbf{u_2}$) = {},

  imp_mob_mem($\mathbf{u_3}$) = {}, *imp_mob_mem*($\mathbf{u_4}$) = {}
- Scope(*exp_immob_mem*) = ROLES,

  attType(*exp_immob_mem*) = set, is_ordered(*exp_immob_mem*) = True,

  $H_{exp\_immob\_mem}$ = RH
- *exp_immob_mem*($\mathbf{u_1}$) = {$\mathbf{x_3}$}, *exp_immob_mem*($\mathbf{u_2}$) = {$\mathbf{x_1}$, $\mathbf{x_2}$},

  exp_immob_mem($\mathbf{u_3}$) = {}, *exp_immob_mem*($\mathbf{u_4}$) = {},

- Scope(*imp_immob_mem*) = ROLES, attType(*imp_immob_mem*) = set

  is_ordered(*imp_immob_mem*) = True, $H_{imp\_immob\_mem}$ = RH

- *imp_immob_mem*($\mathbf{u_1}$) = {$\mathbf{x_4, x_5, x_6}$}, *imp_immob_mem*($\mathbf{u_2}$) = {$\mathbf{x_3, x_4, x_5, x_6}$},

  imp_immob_mem($\mathbf{u_3}$) = { }, *imp_immob_mem*($\mathbf{u_4}$) = { }

- AATT = {*aroles*}

- Scope(*aroles*) = {$\mathbf{ar_1, ar_2}$}, attType(*aroles*) = set, is_ordered(*aroles*) = True,

  H_{aroles} = {<$\mathbf{ar_1, ar_2}$>}

- *aroles*($\mathbf{u_3}$) = {$\mathbf{ar_1}$}, *aroles*($\mathbf{u_4}$) = {$\mathbf{ar_2}$}

Formulate authorization functions for mobile user-role assignment:

Authorization rules for assignment and revocation of a user as a mobile member of role can be expressed respectively, as follows:

For any user $u \in$ USERS, undertaken for assignment,

– is_authorized$U_{\mathbf{mob\text{-}assign}}$($au$: AR, u : USERS, r : ROLES) $\equiv$

$((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u) \wedge r \in \{\mathbf{x_4, x_5}\} \wedge (\mathbf{x_1} \in exp\_mob\_mem(u) \vee (\mathbf{x_1} \in imp\_mob\_mem(u) \wedge \mathbf{x_1} \notin exp\_immob\_mem(u))) \wedge (\mathbf{x_2} \in exp\_mob\_mem(u) \vee (\mathbf{x_2} \in imp\_mob\_mem(u) \wedge \mathbf{x_2} \notin exp\_immob\_mem(u)))) \vee ((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u) \wedge r \in \{\mathbf{x_5, x_6}\} \wedge (\mathbf{x_1} \notin exp\_mob\_mem(u) \wedge \mathbf{x_1} \notin imp\_mob\_mem(u) \wedge \mathbf{x_1} \notin exp\_immob\_mem(u) \wedge \mathbf{x_1} \notin imp\_immob\_mem\ (u)))$

Formulate authorization functions for revoking mobile user from role:

For any user $u \in$ USERS that needs to be revoked,

– is_authorized$U_{\mathbf{mob\text{-}revoke}}$($au$: AR, u : USERS, r : ROLES) $\equiv$

$((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u) \wedge r \in \{\mathbf{x_3, x_4, x_5}\} \wedge ((\mathbf{x_1} \in exp\_mob\_mem(u) \vee \mathbf{x_1} \in imp\_mob\_mem(u) \vee \mathbf{x_1} \in exp\_immob\_mem(u) \vee \mathbf{x_1} \in imp\_immob\_mem) \wedge (\mathbf{x_2} \in exp\_mob\_mem(u) \vee \mathbf{x_2} \in imp\_mob\_mem(u) \vee \mathbf{x_2} \in exp\_immob\_mem(u) \vee \mathbf{x_2} \in imp\_immob\_mem)) \vee ((\exists ar \geq \mathbf{ar_1}).$

$ar \in aroles(u) \wedge r \in \{\mathbf{x_5}, \mathbf{x_6}\} \wedge \mathbf{x_1} \notin exp\_mob\_mem(u) \wedge \mathbf{x_1} \notin imp\_mob\_mem(u) \wedge \mathbf{x_1} \notin exp\_immob\_mem(u) \wedge \mathbf{x_1} \notin imp\_immob\_mem(u))$

Formulate authorization functions for immobile user-role assignment:

Authorization rules for assignment and revocation of a user as an immobile member of role can be expressed respectively, as follows:

For any user $u \in$ USERS, undertaken for assignment,

– is_authorizedU$_{\textbf{immob-assign}}$(au : AR, u : USERS, r : ROLES) $\equiv$

$((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u) \wedge r \in \{\mathbf{x_5}, \mathbf{x_6}\} \wedge (\mathbf{x_1} \in exp\_mob\_mem(u) \vee (\mathbf{x_1} \in imp\_mob\_mem(u) \wedge \mathbf{x_1} \notin exp\_immob\_mem(u))) \wedge (\mathbf{x_2} \in exp\_mob\_mem(u) \vee (\mathbf{x_2} \in imp\_mob\_mem(u) \wedge \mathbf{x_2} \notin exp\_immob\_mem(u)))) \vee ((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u) \wedge r \in \{\mathbf{x_5}, \mathbf{x_6}\} \wedge (\mathbf{x_1} \notin exp\_mob\_mem(u) \wedge \mathbf{x_1} \notin imp\_mob\_mem(u) \wedge \mathbf{x_1} \notin exp\_immob\_mem(u) \wedge \mathbf{x_1} \notin imp\_immob\_mem(u)))$

Formulate authorization functions for revoking immobile user from role:

For any user $u \in$ USERS that needs to be revoked,

– is_authorizedU$_{\textbf{immob-revoke}}$(au : AR, u : USERS, r : ROLES) $\equiv$

$((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u) \wedge r \in \{\mathbf{x_5}, \mathbf{x_6}\} \wedge ((\mathbf{x_1} \in exp\_mob\_mem(u) \vee \mathbf{x_1} \in imp\_mob\_mem(u) \vee \mathbf{x_1} \in exp\_immob\_mem(u) \vee \mathbf{x_1} \in imp\_immob\_mem) \wedge (\mathbf{x_2} \in exp\_mob\_mem(u) \vee \mathbf{x_2} \in imp\_mob\_mem(u) \vee \mathbf{x_2} \in exp\_immob\_mem(u) \vee \mathbf{x_2} \in imp\_immob\_mem)) \vee$
$((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u) \wedge r \in \{\mathbf{x_5}, \mathbf{x_6}\} \wedge \mathbf{x_1} \notin exp\_mob\_mem(u) \wedge \mathbf{x_1} \notin imp\_mob\_mem(u) \wedge \mathbf{x_1} \notin exp\_immob\_mem(u) \wedge \mathbf{x_1} \notin imp\_immob\_mem(u))$

As observed, the translation process involves four basic steps. First, the primary sets from URA99 example instance is mapped to primary sets in AURA instance. URA99 can also be considered as an extension of URA97. Therefore, there exists some similarity. Users and admin users come from same set, $\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}\}$. ROLES and RH represent regular roles and their hierarchy. However, there are two types of role memberships: mobile and immobile memberships. *assigned_roles*(u) function yields mobile or immobile membership of a user on a role. There are four different types of administrative operations: **mob-assign, immob-assign, mob-revoke** and **immob-revoke** for assigning a user as a mobile member, revoking a mobile member, assigning a user as an immobile member and, revoking an immobile member of a role, respectively. Secondly, appropriate attributes that intuitively represent meaning in URA99 instance are engineered. Admin user attribute *aroles* has same meaning as URA97. In addition, there are four user attributes, *exp_mob_mem, imp_mob_mem, exp_immob_mem, imp_immob_mem* in UATT. For each attribute, appropriate attribute type, scope and hierarchies are defined. They represent roles on which a target user has explicit mobile membership, implicit mobile membership, explicit immobile membership and implicit immobile membership, respectively. For example, *exp_mob_mem*($\mathbf{u_1}$) = {$\mathbf{x_1}$} means $\mathbf{u_1}$ has explicit mobile membership on role $\mathbf{x_1}$. Similarly, *imp_mob_mem*($\mathbf{u_1}$) = {$\mathbf{x_2, x_3, x_4, x_5, x_6}$} means user $\mathbf{u_1}$ has implicit mobile membership on roles $\mathbf{x_2}$ through $\mathbf{x_6}$. In fact, a user becomes implicit member of all the roles that are junior to the roles to which that user is an explicit member. In other words, the user assigned to role *x* can activate and use all the roles that are junior to role *x*. Similarly, *exp_immob_mem*($\mathbf{u_1}$) = {$\mathbf{x_3}$} indicates that user $\mathbf{u_1}$ has explicit immobile membership on role $\mathbf{x_3}$ and *imp_immob_mem*($\mathbf{u_1}$) = {$\mathbf{x_4, x_5, x_6}$} means that user $\mathbf{u_1}$ has implicit immobile membership on roles $\mathbf{x_4, x_5}$ and $\mathbf{x_6}$. Admin user attribute *aroles*(*au*) yields admin roles assigned to admin user. In this mapping example, $\mathbf{u_3}$ has admin role $\mathbf{ar_1}$ and, $\mathbf{u_4}$ has admin role $\mathbf{ar_2}$ captured by *aroles*.

In third step and fourth steps, authorization functions that equivalently and intuitively represent mobile user-role assignment (*can-assign-M*), authorization functions for revoking mobile a user from a role (*can-revoke-M*), authorization function for assigning immobile user to role (*can-assign-IM*), and authorization function for revoking immobile user from a role (*can-revoke-IM*), using attributes are respectively established.

3.4.4 MAP$_{\text{URA99}}$

Algorithm 3.2 presents Map$_{\text{URA99}}$. It is an algorithm that maps a URA99 instance into an equivalent AURA instance. For brevity, sets and functions from URA99 model and AURA model are marked with superscripts `99` and `A`, respectively. Map$_{\text{URA99}}$ takes URA99 instance as its input and yields an equivalent instance of AURA model. In particular, input for Map$_{\text{URA99}}$ fundamentally

Algorithm 3.2 $\text{Map}_{\text{URA99}}$

```
    Input: URA99 instance
    Output: AURA instance
Step 1:    /* Map basic sets and functions in AURA */
    a. $\text{USERS}^{\mathtt{A}} \leftarrow \text{USERS}^{99}$ ; $\text{AU}^{\mathtt{A}} \leftarrow \text{USERS}^{99}$ ; $\text{ROLES}^{\mathtt{A}} \leftarrow \text{ROLES}^{99}$ ; $\text{RH}^{\mathtt{A}} \leftarrow \text{RH}^{99}$
    b. $\text{AOP}^{\mathtt{A}} \leftarrow \{$mob-assign, immob-assign, mob-revoke, immob-revoke$\}$
Step 2:    /* Map attribute functions in AURA */
    a. $\text{UATT}^{\mathtt{A}} = \{exp\_mob\_mem, imp\_mob\_mem, exp\_immob\_mem, imp\_immob\_mem\}$
    b. $\text{Range}(exp\_mob\_mem) = \text{ROLES}^{\mathtt{A}}$ ; $\text{attType}(exp\_mob\_mem) = \text{set}$
    c. $\text{is\_ordered}(exp\_mob\_mem) = \text{True}$ ; $\text{H}_{exp\_mob\_mem} = \text{RH}^{\mathtt{A}}$
    d. For each $u \in \text{U}$, $exp\_mob\_mem(u) = \phi$
    e. For each $(u, \text{M}r) \in \text{UA}^{99}$,
            $exp\_mob\_mem(u) = exp\_mob\_mem(u) \cup r$
    f. $\text{Range}(imp\_mob\_mem) = \text{ROLES}$ ; $\text{attType}(imp\_mob\_mem) = \text{set}$
    g. $\text{is\_ordered}(imp\_mob\_mem) = \text{True}$ ; $\text{H}_{imp\_mob\_mem} = \text{RH}^{\mathtt{A}}$
    h. For each $u \in \text{USERS}$, $imp\_mob\_mem(u) = \phi$
    i. For each $(u, \text{M}r) \in \text{UA}^{99}$ and for each $r > r'$,
            $imp\_mob\_mem(u) = imp\_mob\_mem(u) \cup r'$
    j. $\text{Range}(exp\_immob\_mem) = \text{ROLES}^{\mathtt{A}}$ ; $\text{attType}(exp\_immob\_mem) = \text{set}$
    k. $\text{is\_ordered}(exp\_immob\_mem) = \text{True}$ ; $\text{H}_{exp\_immob\_mem} = \text{RH}^{\mathtt{A}}$
    l. For each $u \in \text{USERS}$, $exp\_immob\_mem(u) = \phi$
    m. For each $(u, \text{IM}r) \in \text{UA}^{99}$,
            $exp\_immob\_mem(u) = exp\_immob\_mem(u) \cup r$
    n. $\text{Range}(imp\_immob\_mem) = \text{ROLES}^{\mathtt{A}}$ ; $\text{attType}(imp\_immob\_mem) = \text{set}$
    o. $\text{is\_ordered}(imp\_immob\_mem) = \text{True}$ ; $\text{H}_{imp\_immob\_mem} = \text{RH}^{\mathtt{A}}$
    p. For each $u \in \text{USERS}^{\mathtt{A}}$, $imp\_immob\_mem(u) = \phi$
    q. For each $(u, \text{IM}r) \in \text{UA}^{\mathtt{A}}$ and for each $r > r'$ ,
            $imp\_immob\_mem(u) = imp\_immob\_mem(u) \cup r'$
    r. $\text{AATT}^{\mathtt{A}} \leftarrow \{aroles\}$
    s. $\text{Range}(aroles) = \text{AR}^{99}$ ; $\text{attType}(aroles) = \text{set}$ ; $\text{is\_ordered}(aroles) = \text{True}$ ; $\text{H}_{aroles} = \text{RH}^{\mathtt{A}}$
```

Conitunuation of Algorithm 3.2 Map$_{\text{URA99}}$

```
    t. For each $u \in \mathrm{AU}^{\mathtt{A}}$, $aroles(u) = \phi$
    u. For each $(u, ar)$ in $\mathrm{AUA}^{99}$, $aroles(u) = aroles(u) \cup ar$
Step 3:    /* Construct assign rule in AURA */
    a. assign-mob-formula = $\phi$
    b. For each $(ar, cr, Z) \in can\text{-}assign\text{-}M^{99}$,
        assign-mob-formula' = assign-mob-formula $\vee$
            $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z \wedge (translate(cr, \text{assign})))$
    c. auth_mob_assign = $\text{is\_authorizedU}_{\textbf{mob-assign}}(au : \mathrm{AU}^{\mathtt{A}}, u : \mathrm{USERS}^{\mathtt{A}}, r : \mathrm{ROLES}^{\mathtt{A}}) \equiv$
            assign-mob-formula'
    d. assign-immob-formula = $\phi$
    e. For each $(ar, cr, Z) \in can\text{-}assign\text{-}IM^{99}$,
            assign-immob-formula' = assign-immob-formula $\vee$
            $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z \wedge (translate(cr, \text{assign})))$
    f. auth_immob_assign = $\text{is\_authorizedU}_{\textbf{immob-assign}}(au : \mathrm{AU}^{\mathtt{A}}, u : \mathrm{USERS}^{\mathtt{A}}, r : \mathrm{ROLES}^{\mathtt{A}}) \equiv$
            assign-immob-formula'
Step 4:    /* Construct revoke rule in AURA */
    a. revoke-mob-formula = $\phi$
    b. For each $(ar, cr, Z) \in can\text{-}revoke\text{-}M^{99}$,
        revoke-mob-formula' = revoke-mob-formula $\vee$
            $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z \wedge (translate(cr, \text{revoke})))$
    c. auth_mob_revoke = $\text{is\_authorizedU}_{\textbf{mob-revoke}}(au : \mathrm{AU}^{\mathtt{A}}, u : \mathrm{USERS}^{\mathtt{A}}, r : \mathrm{ROLES}^{\mathtt{A}}) \equiv$
            revoke-mob-formula'
    d. revoke-immob-formula = $\phi$
    e. For each $(ar, cr, Z) \in can\text{-}revoke\text{-}IM^{99}$,
        revoke-immob-formula' = revoke-immob-formula $\vee$
            $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z \wedge (translate(cr, \text{revoke})))$
    f. auth_immob_revoke = $\text{is\_authorizedU}_{\textbf{immob-revoke}}(au : \mathrm{AU}^{\mathtt{A}}, u : \mathrm{USERS}^{\mathtt{A}}, r : \mathrm{ROLES}^{\mathtt{A}}) \equiv$
            revoke-immob-formula'
    End
```

Support routine for algorithm 3.2 *translate*$_{99}$

```
    Input: A URA99 prerequisite condition ($cr$), $op \in$ {assign, revoke}
    Output: An equivalent sub-rule for AURA authorization assign rule.
    Begin:
 1: $rule\_string = \phi$
 2: For each $symbol$ in $cr$
 3:     if $op$ = assign $\wedge$ $symbol$ is a role and in the form $x$ (i.e., the user holds role $x$)
 4:         $rule\_string = rule\_string + (x \in exp\_mob\_mem(u) \vee$
                        $(x \in imp\_mob\_mem(u) \wedge x \notin exp\_immob\_mem(u))$
        else if $op$ = revoke $\wedge$ $symbol$ is a role and in the form $x$ (i.e., the user holds role $x$)
 5:         $rule\_string = rule\_string + (x \in exp\_mob\_mem(u) \vee x \in imp\_mob\_mem(u) \vee$
                        $x \in exp\_immob\_mem(u) \vee x \in imp\_immob\_mem(u))$
 6:     else if $op$ = assign $\vee$ revoke $\wedge$ $symbol$ is role and in the form $\bar{x}$
                                                        (i.e., the user doesn't hold role $x$)
 7:         $rule\_string = rule\_string + (x \notin exp\_mob\_mem(u) \wedge x \notin imp\_mob\_mem(u) \wedge$
                        $x \notin exp\_immob\_mem(u) \wedge x \notin imp\_immob\_mem(u))$
 8:     else
 9:         $rule\_string = rule\_string + symbol$   /* where a $symbol$ is a $\wedge$ or $\vee$ logical operator */
10:     end if
    End
```

includes USERS^{99}, ROLES^{99}, AR^{99}, UA^{99}, AUA^{99}, RH^{99}, ARH^{99}, *can-assign-M*^{99}, *can-assign-IM*^{99}, *can-revoke-M*^{99}, and *can-revoke-IM*^{99}.

Output from Map$_{\text{URA99}}$ algorithm is an equivalent AURA instance, with primarily consisting of following sets and functions: USERS^{A}, AU^{A}, ROLES^{A}, RH^{A}, AOP^{A}, UATT^{A}, AATT^{A}, For each attribute $att \in$ UATT^{A} $\cup$ AATT^{A}, Scope(att), attType(att), is_ordered(att) and H$_{att}$, For each user $u \in$ USERS, and for each $aatt \in$ AATT^{A}, $aatt(u)$, For each user $u \in$ USERS, and for each $uatt \in$ UATT^{A}, $uatt(u)$, Authorization rule for mobile assign (auth_mob_assign), Authorization rule for mobile revoke (auth_mob_revoke), Authorization rule for immobile assign (auth_immob_assign), and Authorization rule for immobile revoke (auth_immob_revoke).

As shown in Algorithm Map$_{\text{URA99}}$, there are four main steps required in mapping any instance of URA99 model to AURA instance. In Step 1, sets and functions from URA99 instance are mapped into AURA sets and functions. In Step 2, user attributes and administrative user attribute functions are expressed. There are four regular user attributes, *exp_mob_mem, imp_mob_mem,* *exp_immob_mem,* and *imp_immob_mem*. Each captures, a user's explicit mobile membership, implicit mobile membership, explicit immobile membership and implicit immobile membership on roles, respectively. Admin user attribute *aroles* captures admin roles assigned to admin users. Step 3 involves constructing assign-mob-formula and assign-immob-formula in AURA that is equivalent to *can-assign-M* and *can-assign-IM* in URA99, respectively. Both *can-assign-M* and *can-assign-IM* are set of triples. Each triple bears information on whether an admin role can assign a candidate user to a set of roles as a mobile member in the case of *can-assign-M* and, as an immobile member in the case of *can-assign-IM*. AURA equivalent for *can-assign-M* is given by is_authorizedU$_{\textbf{mob-assign}}$(au : AU$^{\mathtt{A}}$, u : USERS$^{\mathtt{A}}$, r : ROLES$^{\mathtt{A}}$) and an equivalent translation for *can-assignp-IM* is given by is_authorizedU$_{\textbf{immob-assign}}$(au : AU$^{\mathtt{A}}$, u : USERS$^{\mathtt{A}}$, r : ROLES$^{\mathtt{A}}$). Similarly, In Step 4, revoke-mob-formula equivalent to *can-revoke-M* and *can-revoke-IM* are presented. *translate*$_{99}$ is a support routine for Map$_{\text{URA99}}$ that translates prerequisite condition in URA99 into its AURA equivalent. A complete example instance and its corresponding equivalent AURA instances are presented in Section 3.4.2 and Section 3.4.3, respectively.

3.5 URA02 in AURA

This section first revisits a summary of URA02 model [45]. To illustrate the mapping between URA02 and AURA models, an example instance of URA02 model is presented, which is then manually converted to its equivalent AURA instance representation. A concrete mapping algorithm called Map$_{\text{URA02}}$ which gives a framework for mapping any instance of URA02 model to AURA model is then exhibited.

3.5.1 Summary of URA02 Model

– USERS is a finite set of regular users.

– AR is a finite set of administrative roles.

– CR is a finite set of prerequisite conditions.

– ROLES is a finite set of regular roles.

– ORGU is a finite set of orgnization units.

– UUA $\subseteq$ USERS $\times$ ORGU, regular user to organization unit assignment on OS-U (Organization structure represented as a user-pool).

– CR is a finite set of prerequisite conditions.

A *prerequisite condition* of URA is a boolean expression using the $\wedge$ and $\vee$ operators in terms of the form x and $\bar{x}$, where x is a *regular role or organization unit* in OS-U.

URA02 Grant Model:

User-role assignment is authorized in URA02 by the following relation:

$$can\_assign \subseteq \text{AR} \times \text{CR} \times 2^{\text{ROLES}}.$$

The meaning of *can_assign*($x, y, \{a, b, c\}$) is that a member of an administrative role x (or a member of a role that is senior to x) can assign a user whose current membership, or non-membership, in regular role or organization unit satisfies the prerequisite condition y, to regular roles, a, b or c.

Prerequisite condition is evaluated for user u by interpreting x to be true if:

Case 1:

$x \in$ ROLES: $(\exists x' \geq x)(u, x') \in$ UA

Case 2:

$x \in$ ORGU: $(\exists x' \leq x)(u, x') \in$ UUA

and $\bar{x}$ to be true if:

Case 1:

$x \in ROLES : \neg((\forall x' \geq x)(u, x') \in \text{UA})$

Case 2:

$x \in$ ORGU: $\neg((\forall x' \leq x)(u, x') \in \text{UUA})$

URA02 Revoke Model

The URA02 model controls the user-role or user-organization unit revocation by means of the following relation:

$$can\_revoke \subseteq \text{AR} \times 2^{\text{R}}.$$

The meanning 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 a membership of a user from any regular role or organization unit $y \in Y$.

3.5.2 URA02 Instance

In URA02, decision to assign/revoke user-role can be made based on two factors, a user's membership on role or a user's membership in an organization unit in organization structure of user-pool. They can be viewed as two different cases. In this example instance roles are represented with r and organization units with x for simplicity.

<u>Sets and functions:</u>

- USERS = $\{\mathbf{u_1, u_2, u_3, u_4}\}$
- ROLES = $\{\mathbf{r_1, r_2, r_3, r_4, r_5, r_6}\}$
- AR = $\{\mathbf{ar_1, ar_2}\}$
- UA = $\{(\mathbf{u_1, r_1}), (\mathbf{u_1, r_2}), (\mathbf{u_2, r_3}), (\mathbf{u_2, r_4})\}$
- AUA = $\{(\mathbf{u_3, ar_1}), (\mathbf{u_4, ar_2})\}$
- RH = $\{<\mathbf{r_1, r_2}>, <\mathbf{r_2, r_3}>, <\mathbf{r_3, r_4}>, <\mathbf{r_4, r_5}>, <\mathbf{r_5, r_6}>\}$
- ARH = $\{<\mathbf{ar_1, ar_2}>\}$
- ORGU = $\{\mathbf{x_1, x_2, x_3}\}$
- OUH = $\{<\mathbf{x_3, x_2}>, <\mathbf{x_2, x_1}>\}$
- UUA = $\{(\mathbf{u_1, x_1}), (\mathbf{u_2, x_3})\}$

<u>Case 1:</u>

- CR = $\{\mathbf{r_1} \wedge \mathbf{r_2}, \mathbf{r_1} \vee \bar{\mathbf{r}}_\mathbf{2} \wedge \mathbf{x_3}\}$

  Let $cr_1 = \mathbf{r_1} \wedge \mathbf{r_2}$ and, $cr_2 = \mathbf{r_1} \vee \bar{\mathbf{r}}_\mathbf{2} \wedge \mathbf{r_3}$

Case 2:

- CR = $\{\mathbf{x_1} \wedge \mathbf{x_2}, \mathbf{x_1} \vee \bar{\mathbf{x}}_2 \wedge \mathbf{x_3}\}$

  Let $cr_3 = \mathbf{x_1} \wedge \mathbf{x_2}$ and, $cr_4 = \mathbf{x_1} \vee \bar{\mathbf{x}}_2 \wedge \mathbf{x_3}$

Case 1:

cr_1 is evaluated as follows:

For any user $u \in$ USERS undertaken for assginment,

$(\exists r \geq \mathbf{r_1}).\ (u, r) \in \text{UA} \wedge (\exists r \geq \mathbf{r_2}).\ (u, r) \in \text{UA}$

cr_2 is evaluated as follows:

For any user $u \in$ USERS undertaken for assginment,

$(\exists r \geq \mathbf{r_1}).\ (u, r) \in \text{UA} \vee \neg((\forall r \geq \mathbf{r_2}).\ (u, r) \in \text{UA}) \wedge (\exists r \geq \mathbf{r_3}).\ (u, r) \in \text{UA}$

Case 2:

cr_3 is evaluated as follows:

For any user $u \in$ USERS undertaken for assginment,

$(\exists x \leq \mathbf{x_1}).\ (u, x) \in \text{UUA} \wedge (\exists x \leq \mathbf{x_2}).\ (u, x) \in \text{UUA}$

cr_4 is evaluated as follows:

For any user $u \in$ USERS undertaken for assginment,

$(\exists x \leq \mathbf{x_1}).\ (u, x) \in \text{UUA} \vee \neg((\forall x \leq \mathbf{x_2}).\ (u, x) \in \text{UUA}) \wedge (\exists x \leq \mathbf{x_3}).\ (u, x) \in \text{UUA}$

can_assign and *can_revoke* for respective cases are as follows:

Case 1:

$can\_assign = \{(\mathbf{ar_1}, cr_1, \{\mathbf{r_4}, \mathbf{r_5}\}), (\mathbf{ar_1}, cr_2, \{\mathbf{r_6}\})\}$

$can\_revoke = \{(\mathbf{ar_1}, \{\mathbf{r_1}, \mathbf{r_3}, \mathbf{r_4}\})\}$

Case 2:

$can\_assign = \{(\mathbf{ar_1}, cr_3, \{\mathbf{r_4}, \mathbf{r_5}\}), (\mathbf{ar_1}, cr_4, \{\mathbf{r_6}\})\}$

$can\_revoke = \{(\mathbf{ar_1}, \{\mathbf{r_1}, \mathbf{r_3}, \mathbf{r_4}\})\}$

3.5.3 Equivalent URA02 Instance in AURA

This sub-section presents equivalent translation of the URA02 instance into AURA instance.

Map sets and functions from URA02 to AURA:

- USERS = {$\mathbf{u_1, u_2, u_3, u_4}$}
- AU = {$\mathbf{u_1, u_2, u_3, u_4}$}
- AOP = {**assign, revoke**}
- ROLES = {$\mathbf{r_1, r_2, r_3, r_4, r_5, r_6}$}
- RH = {<$\mathbf{r_1, r_2}$>, <$\mathbf{r_2, r_3}$>, <$\mathbf{r_3, r_4}$>, <$\mathbf{r_4, r_5}$>, <$\mathbf{r_5, r_6}$>}
- *assigned_roles*($\mathbf{u_1}$) = {$\mathbf{r_1, r_2}$}, *assigned_roles*($\mathbf{u_2}$) = {$\mathbf{r_3, r_4}$}

Define attributes and values:

- UATT = {*org_units*}
- Scope(*org_units*) = {$\mathbf{x_1, x_2, x_3}$}, attType(*org_units*) = set

  is_ordered(*org_units*) = True, H_{org_units} = {<$\mathbf{x_3, x_2}$>, <$\mathbf{x_2, x_1}$>}
- *org_units*($\mathbf{u_1}$) = {$\mathbf{x_1}$}, *org_units*($\mathbf{u_2}$) = {$\mathbf{x_3}$}
- AATT = {*aroles*}
- Scope(*aroles*) = {$\mathbf{ar_1, ar_2}$}, attType(*aroles*) = set,

  is_ordered(*aroles*) = True, H_{aroles} = {<$\mathbf{ar_1, ar_2}$>}
- *aroles*($\mathbf{u_3}$) = {$\mathbf{ar_1}$}, *aroles*($\mathbf{u_4}$) = {$\mathbf{ar_2}$}

Construct authorization functions for user-role assignment:

For each *op* in OP, authorization rule for user to role assignment and revocation can be expressed respectively, as follows:

Case 1:

For any user $u \in$ USERS, undertaken for assignment,

– is_authorizedU$_{\textbf{assign}}$(au : AU, u : USERS, r : ROLES) $\equiv$ (($\exists ar \geq \mathbf{ar_1}$). $ar \in aroles(u) \wedge r \in \{\mathbf{r_4, r_5}\} \wedge ((\exists r \geq \mathbf{r_1}).\ r \in assigned\_roles(u) \wedge (\exists r \geq \mathbf{r_2}).\ r \in assigned\_roles(u))) \vee ((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u) \wedge r \in \{\mathbf{r_6}\} \wedge ((\exists r \geq \mathbf{r_1}).\ r \in assigned\_roles(u) \vee (\exists r \geq \mathbf{r_2}).\ r \notin assigned\_roles(u) \wedge (\exists r \geq \mathbf{r_3}).\ r \in assigned\_roles(u)))$

Case 2:

For any user $u \in$ USERS, undertaken for assignment,

– is_authorizedU$_{\textbf{assign}}$(au : AU, u : USERS, r : ROLES) $\equiv$ (($\exists ar \geq \mathbf{ar_1}$). $ar \in aroles(u) \wedge r \in \{\mathbf{r_4, r_5}\} \wedge ((\exists x \leq \mathbf{x_1}).\ x \in org\_units(u) \wedge (\exists x \leq \mathbf{x_2}).\ x \in org\_units(u))) \vee ((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u) \wedge r \in \{\mathbf{r_6}\} \wedge ((\exists x \leq \mathbf{x_1}).\ x \in org\_units(u) \vee (\exists x \leq \mathbf{x_2}).\ x \notin org\_units(u) \wedge (\exists x \leq \mathbf{x_3}).\ x \in org\_units(u)))$

Construct authorization functions for revoking user from role:

Case 1:

For any user $u \in$ USERS, undertaken for assignmet,

– is_authorizedU$_{\textbf{revoke}}$(au : AU, u : USERS, r : ROLES) $\equiv$ ($\exists ar \geq \mathbf{ar_1}$). $ar \in aroles(u) \wedge r \in \{\mathbf{r_1, r_3, r_4}\}$

Case 2:

For any user $u \in$ USERS, undertaken for assignment,

– is_authorizedU$_{\textbf{revoke}}$(au : AU, u : USERS, r : ROLES) $\equiv$ ($\exists ar \geq \mathbf{ar_1}$). $ar \in aroles(u) \wedge r \in \{\mathbf{r_1, r_3, r_4}\}$

It can be seen in this subsection that the translation process involves four general stages. First, the primary sets from URA02 example instance is mapped to primary sets in AURA equivalent instance. This step is very similar to first step in translating URA97 instance to AURA instance. Second, appropriate attributes that intuitively represent meaning in URA02 instance are engineered.

There exists an admin user attribute *aroles*(*au*). *aroles*(*au*) captures admin roles belonging to the admin user. For example, *aroles*($\mathbf{u_3}$) = {$\mathbf{ar_1}$} indicates that admin user $\mathbf{u_3}$ has admin role $\mathbf{ar_1}$. There is a regular user attribute called *org_units*(*u*). *org_units*(*u*) captures a user's association with organization units it belongs to, in an organization structure represented as user-pool (OS-U). For example, *org_units*($\mathbf{u_1}$) = {$\mathbf{x_1}$} means user $\mathbf{u_1}$ has membership on organization unit $\mathbf{x_1}$. In third step, authorization functions for user-role assignment for two different cases that equivalently and intuitively represent *can-assign* functions in URA02 are constructed. URA02 introduces a notion of organization units in addition to a notion of prerequisite roles. Finally, authorization function for revoking user from role for two different cases given in *can_revoke* relation are established. They are mapped into equivalent authorization functions in AURA equivalent instance using related entities and attributes.

3.5.4 $\mathbf{Map_{URA02}}$

$\text{Map}_{\text{URA02}}$ is depicted in Algorighm 3.3. It is an algorithm for mapping a URA02 instance into equivalent AURA instance. For simplicity, sets and functions from URA02 and AURA are marked with superscripts `02` and `A`, respectively. An instance of $\text{Map}_{\text{URA02}}$ is given as an input to this algorithm. In particular, input for $\text{Map}_{\text{URA02}}$ fundamentally includes USERS^{02}, ROLES^{02}, AR^{02}, UA^{02}, AUA^{02}, RH^{02}, ARH^{02}, *can_assign*^{02}, *can_revoke*^{02}, ORGU^{02}, OUH^{02}, and UUA^{02}. The output from $\text{Map}_{\text{URA02}}$ algorithm is an equivalent AURA instance, with primarily consisting of following sets and functions: USERS^{A}, AU^{A}, AOP^{A}, ROLES^{A}, RH^{A}, For each $u \in \text{USERS}^{A}$, *assigned_roles*(*u*), UATT^{A}, AATT^{A}, For each attribute $att \in \text{UATT}^{A} \cup \text{AATT}^{A}$, Scope(*att*), attType(*att*), is_ordered(*att*) and H_{att}, For each user $u \in \text{USERS}^{A}$, *aroles*(*u*) and *org_units*(*u*), Authorization rule for assign (auth_assign), and Authorization rule to revoke (auth_revoke)

A shown in $\text{Map}_{\text{URA02}}$, there are four main steps required in mapping any instance of URA02 model to AURA instance. In Step 1, sets and functions from URA02 instance are mapped into AURA sets and functions. This step takes careful identification of sets that are alike between two models. In Step 2, user attributes and administrative user attribute functions are expressed. UATT

Algorithm 3.3 $\text{Map}_{\text{URA02}}$

```
Input: URA02 instance
Output: AURA instance
    Begin:
Step 1:  /* Map basic sets and functions in AURA */
    a. USERS $\leftarrow$ USERS$^{02}$ ; AU$^{\mathtt{A}}$ $\leftarrow$ AU$^{02}$ ; AOP$^{\mathtt{A}}$ $\leftarrow$ {assign, revoke}
    b. ROLES$^{\mathtt{A}}$ $\leftarrow$ ROLES$^{02}$ ; RH$^{\mathtt{A}}$ $\leftarrow$ ROLES$^{02}$
    c. For each $u \in$ USERS$^{\mathtt{A}}$, $assigned\_roles(u) = \phi$
    d. For each $(u, r) \in$ UA$^{02}$, $assigned\_roles(u) \cup r$
Step 2:  /* Map attribute functions to AURA */
    a. UATT$^{\mathtt{A}}$ $\leftarrow \{org\_units\}$
    b. Scope($org\_units$) = ORGU$^{02}$ ; attType($org\_units$) = set
    c. is_ordered($org\_units$) = True ; $\text{H}_{org\_units}$ = OUH$^{02}$
    d. For each $u \in$ USERS$^{\mathtt{A}}$, $org\_units(u) = \phi$ ;
    e. For each $(u, orgu) \in$ UUA$^{02}$, $org\_units(u) = org\_units(u) \cup orgu$
    f. AATT$^{\mathtt{A}}$ $\leftarrow \{aroles\}$
    g. Scope($aroles$) = AR$^{02}$ ; attType($aroles$) = set
    h. is_ordered($aroles$) = True ; $\text{H}_{aroles} \leftarrow$ ARH$^{02}$
    i. For each $u \in$ AU$^{\mathtt{A}}$, $aroles(u) = \phi$
    j. For each $(u, ar)$ in AUA$^{02}$, $aroles(u) = aroles(u) \cup ar$
Step 3:  /* Construct assign rule in AURA */
    a. assign_formula = $\phi$
    b. For each $(ar, cr, Z) \in can\_assign^{02}$,
        assign_formula' = assign_formula $\vee$ $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z$
        $\wedge\ (translate(cr)))$
    c. auth_assign = is_authorizedU$_{\textbf{assign}}$($au$ : AU, $u$ : USERS, $r$ : ROLES) $\equiv$ assign_formula'
Step 4:  /* Construct revoke rule in AURA */
    a. revoke_formula = $\phi$
    b. For each $(ar, cr, Z) \in can\_revoke^{02}$,
            revoke_formula' = revoke_formula $\vee$ $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z)$
    c. auth_revoke = is_authorizedU$_{\textbf{revoke}}$($au$ : AU, $u$ : USERS, $r$ : ROLES) $\equiv$ revoke_formula'
```

Support routine for algorithm 3.3 *translate*$_{02}$

```
    Input: A URA02 prerequisite condition ($cr$), Case 1, Case 2
    Output: An equivalent sub-rule for AURA authorization rule.
 1: $rule\_string = \phi$
 2: Case Of selection
 3:     ' Case 1 ' ($cr$ is based on roles) :
 4:         $translate_{97}$
 5:     ' Case 2 ' ($cr$ is based on org_units):
 6:     For each $symbol$ in $cr$
 7:         if $symbol$ is an organization unit and in the form $x$
                        (i.e., the user is a member of organization unit $x$)
 8:             $rule\_string = rule\_string + (\exists x' \leq x).\ x' \in org\_units(u)$
 9:         else if $symbol$ an organization unit and in the form $\bar{x}$
                        (i.e., the user is not a member of organization unit $x$)

10:             $rule\_string = rule\_string + (\exists x' \leq x).\ x' \notin org\_units(u)$
11:         else
12:             $rule\_string = rule\_string + symbol$
                    /* where a $symbol$ is a $\wedge$ or $\vee$ logical operator */
13:         end if
14: end Case
```

set has one user attribute called *org_units*. This attribute captures a regular user's appointment or association in an organization unit in organization structure of user-pool. There are two ways by which a user assignment decision is made in URA02, marked as Case 1 and Case 2 in the model. Case 1 checks for user's existing membership or non-membership on roles, and Case 2 checks for user's membership on organization units in organizarional structure. *org_units* captures Case 2. Case 1 is same as URA97. Admin user attribute *aroles* captures admin roles assigned to admin users. Step 3 involves constructing assign_formula in AURA that is equivalent to can_assign^{02} in URA02. can_assign^{02} is a set of triples. Each triple bears information on whether an admin role can assign a candidate user to a set of roles. Equivalent translation in AURA for URA02 is given by is_authorizedU$_{\textbf{assign}}$(au : AU$^{\mathbb{A}}$, u : USERS$^{\mathbb{A}}$, r : ROLES$^{\mathbb{A}}$). Similarly, In Step 4, revoke_formula equivalent to can_revoke^{02} is presented. Support routine for algorithm 3.5.4 presents *translate*$_{02}$. It translates prerequisite condition in URA02 into its equivalent in AURA. A complete example instance and its corresponding equivalent AURA instances were presented in Section 3.5.2 and Section 3.5.3, respectively.

3.6 Uni-ARBAC's URA in AURA

This section first re-visits Uni-ARBAC's URA model by presenting it as a short summary. To illustrate that any instance of Uni-ARBAC's URA model can be represented using AURA model, an example instance for URA in Uni-ARBAC (URA-Uni) is first considered followed by its equivalent AURA instance, which involves a manual process of translation. Then finally, an algorithm is presented to intuitively realize the mapping process.

3.6.1 Summary of Uni-ARBAC's URA Model

– USERS is a finite set of regular users.

– AU is a finite set of administrative units.

– UP is a finite set of user-pools.

– ROLES is a finite set of regular roles.

User-pools relations:

– UPH $\subseteq$ UP $\times$ UP, user-pool partial order hierarchy.

– UUPA $\subseteq$ USERS $\times$ UP, user to user-pool assignment relation.

AUand partitioned assignment:

– *roles*(au : AU) $\rightarrow 2^{\text{ROLES}}$, assignment of roles.

– *user_pools*(au : AU) $\rightarrow 2^{\text{UP}}$, assignment of user-pool.

Derived function:

– *user_pools**(au : AU) $\rightarrow 2^{\text{UP}}$ and defined as,

– *user_pools**(au) = $\{\exists up' \mid \exists\, up \in$ *user_pools*(au)) $up \succeq_{up} up'\}$

Administrative user assignment:

– UA_admin $\subseteq$ USERS $\times$ AU

– AUH $\in$ AU$\times$ AU rooted tree partial order $\succeq_{au}$

Authorization function:

– *can_manage_user_role*(u_1 : *USERS*, u_2 : *USERS*, r : *ROLES*)= $(\exists au_i, au_j)[(u_1, au_i) \in$ UA_admin $\wedge\ au_i \succeq_{au} au_j \wedge r \in roles(au_j) \wedge (\exists\, up \in \text{UP})][(u_2, up) \in \text{UUP} \wedge up \in user\_pools^*(au_j)]$

3.6.2 Uni-ARBAC's URA Instance

This segment presents an instance of URA in Uni-ARBAC model.

Sets and functions:

- USERS = $\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}\}$
- ROLES = $\{\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_3}\}$
- RH = $\{<\mathbf{r_1}, \mathbf{r_2}>, <\mathbf{r_2}, \mathbf{r_3}>\}$
- UA = $\{(\mathbf{u_3}, \mathbf{r_1}), (\mathbf{u_4}, \mathbf{r_3})\}$

user-pools sets and relations

- UPH = $\{<\mathbf{up_2}, \mathbf{up_1}>\}$
- UUPA = $\{(\mathbf{u_1}, \mathbf{up_1}), (\mathbf{u_2}, \mathbf{up_2}), (\mathbf{u_3}, \mathbf{up_1}), (\mathbf{u_4}, \mathbf{up_2})\}$

Administrative Units and Partitioned Assignments

- AU = $\{\mathbf{au_1}, \mathbf{au_2}\}$
- *roles*($\mathbf{au_1}$) = $\{\mathbf{r_1}, \mathbf{r_2}\}$, *roles*($\mathbf{au_2}$) = $\{\mathbf{r_3}\}$
- *user_pools*($\mathbf{au_1}$) = $\{\mathbf{up_1}\}$, *user_pools*($\mathbf{au_2}$) = $\{\mathbf{up_2}\}$

Derived Function

- *user_pools**($\mathbf{au_1}$) = $\{\mathbf{up_1}\}$
- *user_pools**($\mathbf{au_2}$) = $\{\mathbf{up_1}, \mathbf{up_2}\}$

Administrative User Assignments

- *UA_admin* = $\{(\mathbf{u_1}, \mathbf{au_1}), (\mathbf{u_2}, \mathbf{au_2})\}$
- AUH = $\{<\mathbf{au_1}, \mathbf{au_2}>\}$

An instance of user-role assignment condition in Uni-ARBAC:

– *can_manage_user_role*(u_1 : USERS, u_2: USERS, r: ROLES) =

$(\exists au_i, au_j)[(u_1, au_i) \in UA\_admin \wedge au_i \succeq_{au} au_j \wedge r \in roles(au_j) \wedge (\exists up \in \text{UP})[(u_2, up) \in \text{UUPA} \wedge$
$up \in user\_pools^*(au_j)]]$

3.6.3 Equivalent AURA instance of Uni-ARBAC's URA

This segment depicts an equivalent AURA instance for the example instance that was presented in section 3.6.2.

Map sets and functions from Uni-ARBAC's URA to AURA:

- USERS = {$\mathbf{u_1, u_2, u_3, u_4}$}
- AU = {$\mathbf{u_1, u_2, u_3, u_4}$}
- AOP = {**assign, revoke**}
- ROLES = {$\mathbf{r_1, r_2, r_3}$}
- RH = {<$\mathbf{r_1, r_2}$>, <$\mathbf{r_2, r_3}$>}
- *assigned_roles*($\mathbf{u_3}$) = {$\mathbf{r_1}$}, *assigned_roles*($\mathbf{u_4}$) = {$\mathbf{r_3}$}

Define attributes and values:

- UATT = {*userpools, userpool_adminunit*}
- Scope(*userpools*) = {$\mathbf{up_1, up_2}$}, attType(*userpools*) = set,

  is_ordered(*userpools*) = True, $H_{userpools}$ = {<$\mathbf{up_2, up_1}$>}
- Scope(*userpool_adminunit*) = {($\mathbf{up_1, au_1}$), ($\mathbf{up_2, au_2}$)},

  attType(*userpool_adminunit*) = set,

  is_ordered(*userpool_adminunit*) = False, $H_{userpool\_adminunit} = \phi$

- *userpools*($\mathbf{u_1}$) = {$\mathbf{up_1, up_2}$}, *userpools*($\mathbf{u_2}$) = {$\mathbf{up_2}$},

  userpools($\mathbf{u_3}$) = {$\mathbf{up_1, up_2}$}, *userpools*($\mathbf{u_4}$) = {$\mathbf{up_2}$}

- *userpool_adminunit*($\mathbf{u_1}$) = {($\mathbf{up_1, au_1}$), ($\mathbf{up_2, au_2}$)}, *userpool_adminunit*($\mathbf{u_2}$) = {($\mathbf{up_2, au_2}$)},

  userpool_adminunit($\mathbf{u_3}$) = {($\mathbf{up_1, au_1}$), ($\mathbf{up_2, au_2}$)}, *userpool_adminunit*($\mathbf{u_4}$) = {($\mathbf{up_2, au_2}$)}

- AATT = {*admin_unit, adminunit_role*}

- Scope(*admin_unit*) = {$\mathbf{au_1, au_2}$}, attType(*admin_unit*) = set,

  is_ordered(*admin_unit*) = True, H_{admin_unit} = {<$\mathbf{au_1, au_2}$>}

- Scope(*adminunit_role*) = {($\mathbf{au_1, r_1}$), ($\mathbf{au_1, r_2}$), ($\mathbf{au_2, r_3}$)},

  attType(*adminunit_role*) = set, is_ordered(*adminunit_role*) = False, $H_{adminunit\_role} = \phi$

- *admin_unit*($\mathbf{u_1}$) = {$\mathbf{au_1}$}, *admin_unit*($\mathbf{u_2}$) = {$\mathbf{au_2}$}, *admin_unit*($\mathbf{u_3}$) = { },

  admin_unit($\mathbf{u_4}$) = { }

- *adminunit_role*($\mathbf{u_1}$) = {($\mathbf{au_1, r_1}$), ($\mathbf{au_1, r_2}$), ($\mathbf{au_2, r_3}$)}, *adminunit_role*($\mathbf{u_2}$) = {($\mathbf{au_2, r_3}$)},

  adminunit_role($\mathbf{u_3}$) = { }, *adminunit_role*($\mathbf{u_4}$) = { }

Construct authorization function for user-role assignment

For each *op* in OP, authorization rule to assign a user to a role can be expressed as follows:

For any user $u_2 \in$ USERS undertaken for assignment to a role r,

– is_authorized$U_{\mathbf{assign}}$(u_1 : USERS, u_2 : USERS, r : ROLES) $\equiv$

$\exists au_1, au_2 \in$ Scope(*admin_unit*). $<au_1, au_2> \in H_{admin\_unit} \wedge (au_1 \in$ *admin_unit*(u_1) $\wedge$

$(au_2, r) \in$ *adminunit_role*$(u_1)) \wedge \exists up_1, up_2 \in$ Scope(*userpools*). $(up_2, up_1) \in H_{userpools} \wedge$

$((up_2, au_2) \in$ *userpool_adminunit*$(u_2))$

Construct authorization function for revoking user from role

To revoke any user $u_2 \in$ USERS from a role r,

– is_authorizedU$_{\textbf{revoke}}$(u_1 : USERS, u_2 : USERS, r : ROLES) $\equiv$

is_authorizedU$_{\textbf{assign}}$(u_1 : USERS, u_2 : USERS, r : ROLES)

This subsection presented a manual conversion process in which, an example instance was taken from Uni-ARBAC's URA and then mapped into an equivalent instance of AURA. Firstly, sets from Uni-ARBAC's URA are mapped into sets of AURA instance. User and admin users come from same user pool, {$\mathbf{u_1}$, $\mathbf{u_2}$, $\mathbf{u_3}$, $\mathbf{u_4}$} as per the design of Uni-ARBAC's URA. There are two administrative operations, **assign** and **revoke**. ROLES and RH represent set of roles and role hierarchy respectively. *assigned_roles*(*u*) function is assumed to exist as it captures existing roles and their relation with the user. Secondly, attribute function that capture features in Uni-ARBAC's URA are created. There are two attributes in UATT, *userpools* and *userpool_adminunit*. *userpools*(*u*) captures the number of user-pools that a user is associated with. For example, *user-pools*($\mathbf{u_1}$) = {$\mathbf{up_1}$, $\mathbf{up_2}$} means user $\mathbf{u_1}$ is mapped to user-pools $\mathbf{up_1}$ and $\mathbf{up_2}$. *userpool_adminunit* represents all the user-pools a user is associated with and admin units to which these particular user-pools are associated with. For example, *userpool_adminunit*($\mathbf{u_1}$) = {($\mathbf{up_1}$, $\mathbf{au_1}$)} indicates that $\mathbf{u_1}$ is mapped to user-pool $\mathbf{up_1}$ and, user-pool $\mathbf{up_1}$ is further mapped to admin unit $\mathbf{au_1}$. There are two admin user attribute as well. They are *admin_unit* and *adminunit_role*. *admin_unit* yields admin unit to which an admin user has admin authority for URA. For example, *admin_unit*($\mathbf{u_1}$) = {$\mathbf{au_1}$} means that admin user $\mathbf{u_1}$ has an authority for user-role assignment over admin unit $\mathbf{au_1}$. *adminunit_role* on the other hand, captures all the admin units over which that admin user has authority for user-role assignment and, in addition, captures all the roles that are associated with admin units. For example, *adminunit_role*($\mathbf{u_1}$) = {($\mathbf{au_1}$, $\mathbf{r_1}$)} indicates that $\mathbf{u_1}$ has admin authority for user-role assignment over $\mathbf{au_1}$ and role $\mathbf{r_1}$ is mapped to admin unit $\mathbf{au_1}$. For each attribute, scope, order and hierarchy are defined. Finally, authorization rule for user-role assignment and user-role revocation using attributes that were defined above are constructed.

3.6.4 Map$_{\text{URA-Uni-ARBAC}}$

Map$_{\text{URA-Uni-ARBAC}}$ algorithm represents a translation process of any URA-Uni-ARBAC instance to AURA instance. For clarity, basic sets from URA in Uni-ARBAC are marked with superscript `Uni` and basic sets from AURA are marked with superscript `A`. It takes URA-Uni instance as input. In particular it involves USERS$^{\mathtt{Uni}}$, ROLES$^{\mathtt{Uni}}$, RH$^{\mathtt{Uni}}$, UA$^{\mathtt{Uni}}$, UP$^{\mathtt{Uni}}$, UPH$^{\mathtt{Uni}}$, UUPA$^{\mathtt{Uni}}$, AU$^{\mathtt{Uni}}$, For each *au* in AU$^{\mathtt{Uni}}$, *roles*(*au*), For each *au* in AU$^{\mathtt{Uni}}$, *user_pools**(*au*), *UA_admin*$^{\mathtt{Uni}}$, AUH$^{\mathtt{Uni}}$, and *can_manage_user_role*(u_1 : USERS$^{\mathtt{Uni}}$, u_2 : USERS$^{\mathtt{Uni}}$, r : ROLES$^{\mathtt{Uni}}$).

It yields an equivalent instance of AURA as USERS$^{\mathtt{A}}$, AU$^{\mathtt{A}}$, AOP$^{\mathtt{A}}$, ROLES$^{\mathtt{A}}$, RH$^{\mathtt{A}}$, For each $u \in$ USERS$^{\mathtt{A}}$, *roles*(*u*), UATT$^{\mathtt{A}}$, AATT$^{\mathtt{A}}$, For each attribute $att \in$ UATT$^{\mathtt{A}}$ $\cup$ AATT$^{\mathtt{A}}$, Scope(*att*), attType(*att*), is_ordered(*att*) and H$_{att}$, For each user $u \in$ USERS$^{\mathtt{A}}$, and for each $att \in$ UATT$^{\mathtt{A}}$ $\cup$ AATT$^{\mathtt{A}}$, *att*(*u*), Authorization rule for assign (auth_assign), and Authorization rule for revoke (auth_revoke).

A shown in Algorithm Map$_{\text{URA-Uni-ARBAC}}$, there are four main steps required in mapping any instance of URA-Uni model to equivalent AURA instance. In Step 1, sets and functions from URA-Uni instance are mapped into AURA sets and functions. In Step 2, user attributes and administrative user attribute functions are expressed. There are two user attribute, *userpools* and *userpool_adminunit*. *userpools* captures regular user's binding with a group called user-pools. Regular user attribute *userpool_adminunit* provides regular user's association with user-pools, and for each user-pool a user is associated with, user-pool's mapping with admin unit. As a result, this attribute captures a regular user's association in an admin unit. Note that both these attributes are required. Although *userpool_adminunit* captures regular user's association with user-pools and corresponding admin units, it cannot capture user association with user-pools which do not have admin unit associated with it. It is the user-pools that are mapped to admin units. There are two admin user attributes, *admin_unit* and *adminunit_role*. *admin_unit* captures *UA_admin* relation in Uni-ARBAC, and *adminunit_role* admin user's mapping with admin unit and for each admin unit an admin user is mapped to, admin unit's associated roles.

The notion of Uni-ARBAC model is to provide an admin user with admin authority (given by

UA_admin relation) over an admin unit to assign/revoke regular user and role if both regular user and role are mapped to that same admin unit.

Step 3 involves constructing auth_assign in AURA that is equivalent to *can_manage_user_role*(u_1 : USERS$^{\texttt{Uni}}$, u_2 : USERS$^{\texttt{Uni}}$, r : ROLES$^{\texttt{Uni}}$) in Uni-ARBAC's URA. Equivalent translation of *can_manage_user_role* for user-role assignment in AURA is given by is_authorizedU$_{\textbf{assign}}$(*au* : AU, *u* : USERS, *r* : ROLES). Similarly, In Step 4, authorization rule to revoke user-role, auth_revoke, which is equivalent to *can_manage_user_role*(u_1 : USERS$^{\texttt{Uni}}$, u_2 : USERS$^{\texttt{Uni}}$, r : ROLES$^{\texttt{Uni}}$) is expressed.

3.7 UARBAC's URA in AURA

This section shows how it is possible to represent UARBAC's URA model in AURA. This is gracefully done by first taking an example instance from URABC's URA model and representing equivalently as AURA instance. Then a complete algorithm called Map$_{\text{URA-UARBAC}}$ is expressed which shows that any instance of UARBAC's URA model can be represented in AURA model. A summary of UARBAC's URA is presented as follows:

3.7.1 Summary of UARBAC's URA Model

Mao and Li [37] redefine RBAC model. They propose a notion of class of objects and *access modes* in RBAC. A summary of the UARBAC's URA model is presented below:

RBAC Model

RBAC model has the following schema:

RBAC Schema

RBAC Schema is given by following tuple.

<*C, OBJS, AM*>

- *C* is a finite set of object classes with predefined classes: user and role.

Algorithm 3.4 $\text{Map}_{\text{URA-Uni-ARBAC}}$

```
Input: Instance of URA in Uni-ARBAC
Output: AURA instance
Step 1: /* Map basic sets and functions in AURA */
    a. $\text{USERS}^{A} \leftarrow \text{USERS}^{\text{Uni}}$ ; $\text{AU}^{A} \leftarrow \text{USERS}^{\text{Uni}}$ ; $\text{AOP}^{A} \leftarrow \{\textbf{assign, revoke}\}$
    b. $\text{ROLES}^{A} \leftarrow \text{ROLES}^{\text{Uni}}$ ; $\text{RH}_{A} \leftarrow \text{RH}^{\text{Uni}}$
    c. For each $u \in \text{USERS}^{A}$, $assigned\_roles(u) = \phi$
    d. For each $(u, r) \in \text{UA}^{\text{Uni}}$, $assigned\_roles(u)' = assigned\_roles(u) \cup r$
Step 2: /* Map attribute functions in AURA */
    a. $\text{UATT}^{A} \leftarrow \{userpools, userpool\_adminunit\}$
    b. $\text{Scope}(userpools) = \text{UP}^{\text{Uni}}$ ; $\text{attType}(userpools) = \text{set}$
    c. $\text{is\_ordered}(userpools) = \text{True}$ ; $\text{H}_{userpools} = \text{UPH}^{\text{Uni}}$
    d. For each $u$ in $\text{USERS}^{A}$, $userpools(u) = \phi$
    e. For each $(u, up) \in \text{UUPA}^{\text{Uni}}$, $userpools(u)' = userpools(u) \cup up$
    f. $\text{Scope}(userpool\_adminunit) = \text{USERS}^{\text{Uni}} \times \text{AU}^{\text{Uni}}$
    g. $\text{attType}(userpool\_adminunit) = \text{set}$ ; $\text{is\_ordered}(userpool\_adminunit) = \text{False}$
    h. $\text{H}_{userpool\_adminunit}$
    i. For each $u$ in $\text{USERS}^{A}$, $userpool\_adminunit(u) = \phi$ ;
    j. For each $(u, up) \in \text{UUPA}^{\text{Uni}}$ and for each $au$ in $\text{AU}^{\text{Uni}}$, if $up \in user\_pools(au)$ then
            $userpool\_adminunit(u)' = userpool\_adminunit(u) \cup (up, au)$
    k. $\text{AATT}^{A} \leftarrow \{admin\_unit, adminunit\_role\}$
    l. $\text{Scope}(admin\_unit) = \text{AU}^{\text{Uni}}$ ; $\text{attType}(admin\_unit) = \text{set}$
    m. $\text{is\_ordered}(admin\_unit) = \text{True}$ ; $\text{H}_{admin\_unit} = \text{AUH}^{\text{Uni}}$
    n. For each $u$ in $\text{AU}^{A}$, $admin\_unit(u) = \phi$
    o. For each $(u, au) \in UA\_Admin^{\text{Uni}}$, $admin\_unit(u)' = admin\_unit(u) \cup au$
    p. $\text{Scope}(adminunit\_role) = \text{AU}^{\text{Uni}} \times \text{ROLES}^{\text{Uni}}$
    q. $\text{attType}(adminunit\_role) = \text{set}$ ; $\text{is\_ordered}(adminunit\_role) = \text{False}$ ; $\text{H}_{adminunit\_role} = \phi$
    r. For each $u$ in $\text{AU}^{A}$, $adminunit\_role(u) = \phi$
    s. For each $(u, au) \in UA\_Admin^{\text{Uni}}$ and for each $r \in roles(au)$,
            $adminunit\_role(u)' = adminunit\_role(u) \cup (au, r)$
```

Continuation of Algorithm 3.4 Map$_{\text{URA-Uni-ARBAC}}$

```
Step 3:    /* Construct assign rule in AURA */
    a. can_manage_rule = ($\exists au_1, au_2 \in$ Scope($admin\_unit$). $<au_1, au_2> \in H_{admin\_unit} \wedge$
    ($au_1 \in admin\_unit(u_1) \wedge (au_2, r) \in adminunit\_role(u_1)$) $\wedge \exists up_1, up_2 \in$ Scope($userpools$).
    $<up_2, up_1> \in H_{userpools} \wedge (up_2, au_2) \in userpool\_adminunit(u_2)$)
    b. auth_assign = is_authorizedU$_{\textbf{assign}}$($u_1$ : AU$^{\mathbb{A}}$, $u_2$ : USERS$^{\mathbb{A}}$, $r$ : ROLES$^{\mathbb{A}}$) $\equiv$
    can_manage_rule
Step 4:    /* Construct revoke rule for AURA */
    a. auth_revoke = is_authorizedU$_{\textbf{revoke}}$($u_1$ : AU$^{\mathbb{A}}$, $u_2$ : USERS$^{\mathbb{A}}$, $r$ : ROLES$^{\mathbb{A}}$) $\equiv$
    can_manage_rule
```

- $OBJS(c)$ is a function that gives all possible names for objects of the class $c \in C$. Let

  USERS = $OBJS$(user) and **ROLES** = $OBJS$(role)

- $AM(c)$ is function that maps class c to a set of access modes that can be applied on objects of class c.

Access modes for two predefined classes user and role are fixed by the model and are as follows:

AM(user) = {empower, admin}

AM(role) = {grant, empower, admin}

RBAC Permissions:

There are two kinds of permissions in this RBAC model:

- Object permissions of the form,

  $[c, o, a]$, where $c \in C$, $o \in OBJS(c)$, $a \in AM(c)$.

- Class permissions of the form,

  $[c, a]$, where, $c \in C$, and $a \in$ {create} $\cup\, AM(c)$.

RBAC State:

Given an RBAC Schema, an RBAC state is given by,

$<OB, UA, PA, RH>$

- OB is a function that maps each class in C to a finite set of object names of that class that currently exists, i.e., $OB(c) \subseteq OBJS(c)$. Let,

  $OB(\text{user}) = USERS$, and $OB(\text{role}) = ROLES$.

  Set of permissions P is given by,

  $P = \{[c, o, a] \mid c \in C \wedge o \in OBJS(c) \wedge a \in AM(c)\} \cup \{[c, a] \mid c \in C \wedge a \in \{\text{create}\} \cup AM(c)\}$

- $UA \subseteq USERS \times ROLES$, user-role assignment relation.

- $PA \subseteq P \times ROLES$, permission-role assignment relation.

- $RH \subseteq ROLES \times ROLES$, partial order in $ROLES$ denoted by $\succeq_{RH}$.

Administrative permissions in UARBAC:

All the permissions of user u who performs administrative operations can be calculated as follows:

- authorized_perms$[u] = \{p \in P \mid \exists r_1, r_2 \in ROLES\ [(u, r_1) \in UA \wedge (r_1 \succeq_{RH} r_2)$
  $\wedge\ (r_2, p) \in PA]\}$

User-Role Administration

Operations required to assign user u_1 to role r_1 and to revoke u_1 from role r_1 are respectively listed below:

- grantRoleToUser(r_1, u_1)

- revokeRoleFromUser(r_1, u_1)

A user requires one of the following two permissions to conduct grantRoleToUser(r_1, u_1) operation.

- [user, u_1, empower] and [role, r_1, grant] or,
- [user, empower] and [role, r_1, grant] or,
- [user, u_1, empower] and [role, grant] or,
- [user, empower] and [role, grant]

A user at least requires either of the following options to conduct revokeRoleFromUser(r_1, u_1) operation.

- [user, u_1, empower] and [role, r_1, empower] or,
- [user, u_1, admin] or,
- [role, r_1, admin] or,
- [user, admin] or,
- [role, admin].

3.7.2 An Instance of UARBAC's URA

<u>RBAC Schema</u>

- C = {user, role}
- $OBJS$(user) = USERS, $OBJS$(role) = ROLES
- AM(user) = {empower, admin}, AM(role) = {grant, empower, admin}

<u>RBAC State</u>

- $USERS = OBJ$(user) = {$\mathbf{u_1, u_2, u_3, u_4}$}
- $ROLES = OBJ$(role)= {$\mathbf{r_1, r_2, r_3, r_4}$}

- *P* = {[user, $\mathbf{u_1}$, empower], [user, $\mathbf{u_1}$, admin], [user, $\mathbf{u_2}$, empower], [user, $\mathbf{u_2}$, admin], [user, $\mathbf{u_3}$, empower], [user, $\mathbf{u_3}$, admin], [user, $\mathbf{u_4}$, empower], [user, $\mathbf{u_4}$, admin], [role, $\mathbf{r_1}$, grant], [role, $\mathbf{r_1}$, empower], [role, $\mathbf{r_1}$, admin], [role, $\mathbf{r_2}$, grant], [role, $\mathbf{r_2}$, empower], [role, $\mathbf{r_2}$, admin], [role, $\mathbf{r_3}$, grant], [role, $\mathbf{r_3}$, empower], [role, $\mathbf{r_3}$, admin], [role, $\mathbf{r_4}$, grant], [role, $\mathbf{r_4}$, empower], [role, $\mathbf{r_4}$, admin], [user, empower], [user, admin], [role, empower], [role, grant], [role, admin]}

- *UA* = {($\mathbf{u_1, r_1}$), ($\mathbf{u_2, r_1}$), ($\mathbf{u_2, r_2}$), ($\mathbf{u_2, r_3}$), ($\mathbf{u_3, r_3}$), ($\mathbf{u_4, r_2}$)}

- *RH* = {<$\mathbf{r_1, r_2}$>, <$\mathbf{r_2, r_3}$>, <$\mathbf{r_3, r_4}$>}

Administrative permissions for user-role assignment in UARBAC:

Following is the list of administrative permissions each user has for user-role assignment:

- authorized_perms[$\mathbf{u_1}$] = {[user, $\mathbf{u_1}$, empower], [role, $\mathbf{r_1}$, grant], [user, $\mathbf{u_2}$, empower], [role, $\mathbf{r_3}$, grant], [user, $\mathbf{u_3}$, empower], [user, $\mathbf{u_4}$, empower], [role, $\mathbf{r_2}$, grant], [user, $\mathbf{u_3}$, admin], [role, $\mathbf{r_1}$, admin], [role, $\mathbf{r_4}$, admin]}

- authorized_perms[$\mathbf{u_2}$] = {[user, $\mathbf{u_1}$, empower], [role, $\mathbf{r_1}$, grant], [user, $\mathbf{u_2}$, empower], [role, $\mathbf{r_2}$, grant]}

- authorized_perms[$\mathbf{u_3}$] = { }

- authorized_perms[$\mathbf{u_4}$] = {[role, grant], [user, empower]}

User-Role assignment condition in UARBAC's URA:

One can perform following operation to assign a user u_1 to a role r_1.

- grantRoleToUser(r_1, u_1)

To perform aforementioned operation one needs one of the following two permissions:

- [user, u_1, empower] and [role, r_1, grant] **or**,

- [user, empower] and [role, r_1, grant] **or**,

- [user, u_1, empower] and [role, grant] **or**,
- [user, empower] and [role, grant]

Condition for revoking user-role in UARBAC's URA:

One can perform following operation to revoke a user u_1 to a role r_1.

- revokeRoleFromUser(r_1, u_1)

To perform aforementioned operation one needs one of the following permissions:

- [user, u_1, empower] and [role, r_1, grant] **or**,
- [role, r_1, admin] **or**,
- [user, u_1, admin] **or**,
- [role, admin] **or**,
- [user, admin]

3.7.3 Equivalent AURA instance for UARBAC's URA

This section depicts an equivalent instance of AURA model for the example instance that was presented in previous subsection.

Map sets and functions from UARBAC's URA to AURA:

- USERS = {$\mathbf{u_1, u_2, u_3, u_4}$}
- AU = {$\mathbf{u_1, u_2, u_3, u_4}$}
- AOP = {assign, revoke}
- ROLES = {$\mathbf{r_1, r_2, r_3}$}
- RH = {<$\mathbf{r_1, r_2}$>, <$\mathbf{r_2, r_3}$>, <$\mathbf{r_3, r_4}$>}

- *assigned_roles*($\mathbf{u_1}$) = {$\mathbf{r_1}$}, *assigned_roles*($\mathbf{u_2}$) = {$\mathbf{r_1}$, $\mathbf{r_2}$, $\mathbf{r_3}$}, *assigned_roles*($\mathbf{u_3}$) = {$\mathbf{r_3}$}, *assigned_roles*($\mathbf{u_4}$) = {$\mathbf{r_2}$}

Define attributes and values:

- UATT = { }
- AATT = {*user_am, role_am, classp*}
- Scope(*user_am*) = {($\mathbf{u_1}$, empower), ($\mathbf{u_2}$, empower), ($\mathbf{u_3}$, empower), ($\mathbf{u_3}$, admin),

  ($\mathbf{u_4}$, empower)},

  attType(*user_am*) = set, is_ordered(*user_am*) = False, $H_{user\_am} = \phi$
- Scope(*role_am*) = {($\mathbf{r_1}$, **grant**), ($\mathbf{r_2}$, grant), ($\mathbf{r_3}$, grant), ($\mathbf{r_1}$, admin), ($\mathbf{r_4}$, admin)},

  attType(*role_am*) = set, is_ordered(*role_am*) = False, $H_{role\_am} = \phi$
- Scope(*classp*) = {(user, empower), (user, admin), (role, empower), (user, grant), (role, admin)},

  attType(*classp*) = set, is_ordered(*user_am*) = False, $H_{classp} = \phi$
- *user_am*($\mathbf{u_1}$) = {($\mathbf{u_1}$, empower), ($\mathbf{u_2}$, empower), ($\mathbf{u_3}$, empower), ($\mathbf{u_4}$, empower),

  ($\mathbf{u_3}$, admin)},

  user_am($\mathbf{u_2}$) = {($\mathbf{u_1}$, empower), ($\mathbf{u_2}$, empower)},

  user_am($\mathbf{u_3}$) = { }, *user_am*($\mathbf{u_4}$) = { }
- *role_am*($\mathbf{u_1}$) = {($\mathbf{r_1}$, grant), ($\mathbf{r_2}$, grant), ($\mathbf{r_3}$, grant), ($\mathbf{r_1}$, admin), ($\mathbf{r_4}$, admin)},

  role_am($\mathbf{u_2}$) = {($\mathbf{r_1}$, grant), ($\mathbf{r_2}$, grant)},

  role_am($\mathbf{u_3}$) = { }, *role_am*($\mathbf{u_4}$) = { }

- *classp*(**u₁**) = {}, *classp*(**u₂**) = {}, *classp*(**u₃**) = {},

  classp(**u₄**) = {(role, grant), (user, empower)}

Construct authorization function for user-role assignment:

For each *op* in AOP, authorization rule to assign user to role can be expressed as follows:

For any user $u_2 \in$ USERS taken for assignment to role r_1,

– is_authorized$_{\textbf{assign}}$(u_1 : USERS, u_2 : USERS, r_1 : ROLES) $\equiv$

((u_2, empower) $\in$ *user_am*(u_1) $\wedge$ (r_1, grant) $\in$ *role_am*(u_1)) $\vee$ ((u_2, empower) $\in$ *user_am*(u_1) $\wedge$ (role, grant) $\in$ *classp*(u_1)) $\vee$ ((user, empower) $\in$ *classp*(u_1) $\wedge$ (r_1, grant) $\in$ *role_am*(u_1)) $\vee$ ((user, empower) $\in$ *classp*(u_1) $\wedge$ (role, grant) $\in$ *classp*(u_1))

Construct authorization function for revoking user from role:

To revoke any user u_2 from a role r_1,

– is_authorized$_{\textbf{revoke}}$(u_1 : USERS, u_2 : USERS, r_1 : ROLES) $\equiv$

((u_2, empower) $\in$ *user_am*(u_1) $\wedge$ (r_1, grant) $\in$ *role_am*(u_1)) $\vee$ (u_2, admin) $\in$ *user_am*(u_1) $\vee$ (r_1, admin) $\in$ *role_am*(u_1) $\vee$ (user, admin) $\in$ *classp*(u_1) $\vee$ (role, admin) $\in$ *classp*(u_1)

The mapping process for UARBC's URA to AURA involves four primary stages. In first step, appropriate sets are mapped. Users and admin users come from same user pool, {**u₁, u₂, u₃, u₄**}. In UARBAC [37], administrative entity is not defined clearly. It uses the term *one*. In this dissertation, *one* is considered as admin user. Function *assigned_roles*(u) represents UA relation in UARBAC's URA. UARBAC's URA's user-role assignment model heavily depends on the administrative user's authority over the target objects, user and role. The authority is defined using *access modes*. Secondly, attributes are engineered intuitively to match the meaning defined in UARBAC's URA. Three admin user attributes, *user_am, role_am* and *classp* are defined for admin user. Regular user attribute set is empty. *user_am* defines a particular admin user's *access mode* towards a target user. For example, *user_am*(**u₂**) = {(**u₁**, empower), (**u₂**, empower)} means that admin user **u₂** has empower access mode towards user **u₁** and **u₂**. *role_am* yields admin user's *access mode* towards target role. For example, *role_am*($\mathbf{u_2}$) = {($\mathbf{r_1}$, grant), ($\mathbf{r_2}$, grant)} indicates that admin user $\mathbf{u_2}$ has access mode grant on $\mathbf{r_1}$ and $\mathbf{r_2}$. *classp* defines admin user's authority over class level objects, which gives admin user an authority over all the objects of that class with particular access mode. For example, *classp*($\mathbf{u_4}$) = {(role, grant), (user, empower)} indicates that $\mathbf{u_4}$ has access modes grant towards role class and, access mode empower over user class.

In third step, authorization functions for user-role assignment is established. In step four, an authorization function for revoking user from role is established. This is carried out using participating entities and their attributes/values.

3.7.4 Map$_{\text{URA-UARBAC}}$

Algorithm 3.5 presents Map$_{\text{URA-UARBAC}}$, an algorithm for mapping any instance of UARBAC's URA [37] to its equivalent AURA instance. For brevity, sets and function from UARBAC model are labeled with superscript U, and that of AURA with superscript A. There are four required primary steps required for this mapping. Step 1 involves translating sets and functions from UARBAC's URA to AURA equivalent sets and functions. In Step 2, user attributes and admin user attributes functions are defined. Regular user attributes set (UATT) is set to null as there is no user attributes required. An admin user's authority towards a regular user and a role, defined by unique *access modes* towards user and role, decides whether she can assign that particular user to that particular role. AURA defines three admin user attributes: *user_am, role_am* and *classp*. *user_am* attribute captures an admin user's access mode towards a regular user. Similarly, *role_am* captures an admin user's access mode towards a role. An admin user can also have a class level access mode captured by attribute *classp*. With class level access mode, an admin user gains authority over that entire class of object. For example, [grant, role] is an admin permission with class level access mode about class object role, which provides an admin user with power to grant any role to a given user.

In Step 3, assign_formula for AURA equivalent to grantRoleToUser(u_1, r_1) in UARBAC's URA is established. For example, if an admin user $\mathbf{au_1}$ wants to execute grantRoleToUser($\mathbf{u_3}$,

Algorithm 3.5 Map$_{\text{URA-UARBAC}}$

```
Input: Instance of URA in UARBAC
Output: AURA instance
Step 1:    /* Map basic sets and functions in AURA */
    a. $\text{USERS}^{\mathbb{A}} \leftarrow \textit{USERS}^{\mathbb{U}}$ ; $\text{AU}^{\mathbb{A}} \leftarrow \textit{USERS}^{\mathbb{U}}$ ; $\text{AOP}^{\mathbb{A}} \leftarrow \{\textbf{assign, revoke}\}$
    b. $\text{ROLES}^{\mathbb{A}} \leftarrow \textit{ROLES}^{\mathbb{U}}$ ; $\text{RH}^{\mathbb{A}} \leftarrow \textit{RH}^{\mathbb{U}}$
    c. For each $u_1 \in \text{USERS}^{\mathbb{A}}$, $\textit{assigned\_roles}^{\mathbb{A}}(u_1) = \phi$
    d. For each $(u_1, r_1) \in \textit{UA}^{\mathbb{U}}$, $\textit{assigned\_roles}^{\mathbb{A}}(u_1)' = \textit{assigned\_roles}^{\mathbb{A}}(u_1) \cup r_1$
Step 2:    /* Map attribute functions in AURA */
    a. $\text{UATT}^{\mathbb{A}} = \phi$
    b. $\text{AATT}^{\mathbb{A}} \leftarrow \{\textit{user\_am, role\_am, classp}\}$
    c. $\text{Scope}^{\mathbb{A}}(\textit{user\_am}) = \textit{USERS}^{\mathbb{U}} \times \textit{AM}^{\mathbb{U}}(\text{user})$
    d. $\text{attType}^{\mathbb{A}}(\textit{user\_am}) = \text{set}$ ; $\text{is\_ordered}^{\mathbb{A}}(\textit{user\_am}) = \text{False}$, $\text{H}^{\mathbb{A}}{}_{\textit{user\_am}} = \phi$
    e. For each $u$ in $\text{AU}^{\mathbb{U}}$, $\textit{user\_am}(u) = \phi$
    f. For each $u$ in $U^{\mathbb{U}}$ and for each $[c, u_1, am] \in \text{authorized\_perms}^{\mathbb{U}}[u]$,
            $\textit{user\_am}(u)' = \textit{user\_am}(u) \cup (u_1, am)$
    g. $\text{Scope}^{\mathbb{A}}(\textit{role\_am}) = \textit{ROLES}^{\mathbb{U}} \times \textit{AM}^{\mathbb{U}}(\text{role})$
    h. $\text{attType}^{\mathbb{A}}(\textit{role\_am}) = \text{set}$
    i. $\text{is\_ordered}^{\mathbb{A}}(\textit{role\_am}) = \text{False}$, $\text{H}^{\mathbb{A}}{}_{\textit{role\_am}} = \phi$
    j. For each $u$ in $\text{USERS}^{\mathbb{A}}$, $\textit{role\_am}(u) = \phi$
    k. For each $u$ in $U^{\mathbb{U}}$ and for each $[c, r_1, am] \in \text{authorized\_perms}^{\mathbb{U}}[u]$,
            $\textit{role\_am}(u)' = \textit{role\_am}(u) \cup (r_1, am)$
    l. $\text{Scope}^{\mathbb{A}}(\textit{classp}) = C^{\mathbb{U}} \times \{\textit{AM}(\text{role})^{\mathbb{U}} \cup \textit{AM}^{\mathbb{U}}(\text{user})\}$
    m. $\text{attType}^{\mathbb{A}}(\textit{classp}) = \text{set}$
    n. $\text{is\_ordered}^{\mathbb{A}}(\textit{classp}) = \text{False}$, $\text{H}^{\mathbb{A}}{}_{\textit{classp}} = \phi$
    o. For each $u$ in $\text{USERS}^{\mathbb{A}}$, $\textit{classp}(u) = \phi$
    p. For each $u$ in $U^{\mathbb{U}}$ and for each $[c, a] \in \text{authorized\_perms}^{\mathbb{U}}[u]$,
            $\textit{classp}(u)' = \textit{classp}(u) \cup (c, a)$
```

Continuation of Algorithm 3.5 Map$_{\text{URA-UARBAC}}$

```
Step 3:    /* Construct assign rule in AURA */
    a. assign_formula =
    $((u_2, \text{empower}) \in user\_am(u_1) \wedge (r_1, \text{grant}) \in role\_am(u_1)) \vee ((u_2, \text{empower}) \in user\_am(u_1)$
    $\wedge (\text{role}, \text{grant}) \in classp(u_1)) \vee ((\text{user}, \text{empower}) \in classp(u_1) \wedge (r_1, \text{grant}) \in role\_am(u_1))$
    $\vee ((\text{user}, \text{empower}) \in classp(u_1) \wedge (\text{role}, \text{grant}) \in classp(u_1))$
    b. auth_assign = is_authorizedU$_{\textbf{assign}}(u_1 : \text{USERS}^{\text{A}}, u_2 : \text{USERS}^{\text{A}}, r_1 : \text{ROLES}^{\text{A}}) \equiv$
    assign_formula
Step 4:    /* Construct revoke rule for AURA */
    a. revoke_formula =
    $((u_2, \text{empower}) \in user\_am(u_1) \wedge (r_1, \text{grant}) \in role\_am(u_1)) \vee (u_2, \text{admin}) \in user\_am(u_1) \vee$
    $(r_1, \text{admin}) \in role\_am(u_1) \vee (\text{user}, \text{admin}) \in classp(u_1) \vee (\text{role}, \text{admin}) \in classp(u_1)$
    b. auth_revoke = is_authorizedU$_{\textbf{revoke}}(u_1 : \text{USERS}^{\text{A}}, u_2 : \text{USERS}^{\text{A}}, r_1 : \text{ROLES}^{\text{A}}) \equiv$
    revoke_formula
```

$\mathbf{r_1}$), she must have one of the combination of two permissions: [user, $\mathbf{u_3}$, empower] and [role, $\mathbf{r_1}$, grant], or [user, $\mathbf{u_3}$, empower] and [role, grant], or [user, empower] and [role, $\mathbf{r_1}$, grant], or [user, empower] and [role, grant]. Each of this permission given to an admin user is expressed equivalently in AURA model using attributes of admin user. For instance, an admin user having an admin permission [user, $\mathbf{u_3}$, empower] is expressed equivalently as ($\mathbf{u_3}$, empower) $\in$ *user_am*($\mathbf{u_1}$) and so on. Equivalent assign_formula for this example is expressed as is_authorizedU$_{\textbf{assign}}$($\mathbf{au_1}$, $\mathbf{u_3}$, $\mathbf{r_1}$) $\equiv ((\mathbf{u_3}, \text{empower}) \in user\_am(\mathbf{au_1}) \wedge (\mathbf{r_1}, \text{grant}) \in role\_am(\mathbf{au_1})) \vee ((\mathbf{u_3}, \text{empower}) \in user\_am(\mathbf{au_1}) \wedge (\text{role}, \text{grant}) \in classp(\mathbf{au_1})) \vee ((\text{user}, \text{empower}) \in classp(\mathbf{au_1}) \wedge (\mathbf{r_1}, \text{grant}) \in role\_am(\mathbf{au_1})) \vee ((\text{user}, \text{empower}) \in classp(\mathbf{au_1}) \wedge (\text{role}, \text{grant}) \in classp(\mathbf{au_1}))$. Thus, assign_formula checks if admin user ($\mathbf{au_1}$) bears any combination of the aforementioned permissions, but expressed in terms of admin user attributes. Step 4 establishes revoke_formula equivalent to revokeRoleFromUser(u_1,r_1). To execute revoke function revokeRoleFromUser(u_1,r_1), an admin user must have either of these permissions: [user, u_1, empower] and [role, r_1, grant] or, [user, u_1, admin] or, [role, r_1, admin] or, [user, admin] or, [role, admin]. This requirement is expressed as revoke_formula in terms of admin user attributes. For example, equivalent authorization function to revoke a user $\mathbf{u_2}$ from role $\mathbf{r_1}$ is given as is_authorizedU$_{\text{revoke}}$($\mathbf{au_1}$, $\mathbf{u_2}$, $\mathbf{r_1}$) $\equiv$ (($\mathbf{u_2}$, empower) $\in$ *user_am*($\mathbf{au_1}$) $\wedge$ ($\mathbf{r_1}$, grant) $\in$ *role_am*($\mathbf{au_1}$)) $\vee$ ($\mathbf{u_2}$, admin) $\in$ *user_am*($\mathbf{au_1}$) $\vee$ ($\mathbf{r_1}$, admin) $\in$ *role_am*($\mathbf{au_1}$) $\vee$ (user, admin) $\in$ *classp*($\mathbf{au_1}$) $\vee$ (role, admin) $\in$ *classp*($\mathbf{au_1}$).

An article that included both attribute-based user-role assignment (AURA) model and attribute-based permission-role assignment (ARPA) models, collectively known as *'AARBAC: Attribute-Based Administration of Role-Based Access Control'* was published in IEEE 3nd International Conference on Collaboration and Internet Computing (CIC), 2017 (CIC 2017) [39]. Another version that includes detailed description of these two models is also publicly avaliable [41].

Chapter 4: ARPA: ATTRIBUTE-BASED PERMISSION-ROLE ASSIGNMENT MODEL

Portion of materials in this chapter are published in the following venue [39]:

- Jiwan Ninglekhu and Ram Krishnan. AARBAC: Attribute-Based Administration of Role-Based Access Control. In 2017 IEEE 3nd International Conference on Collaboration and Internet Computing (CIC). IEEE, 2017.

The ARPA model is very similar to the AURA model in that it deals with how to assign permissions to roles, and how to revoke permissions from roles using attributes of the involed entities. The assignment decisions are based on attributes and their values. The model's main difference with respect to AURA when it comes to permission-role assignment is that it replaces regular users (USERS) in AURA with permissions (PERMS). Similarly, it replaces user attributes (UATT) with permission attibutes (PATT). Similar to AURA, one of the main motivations behind the development of ARPA model is to capture the features that were used in assignment decisions processes in the prior PRA models such as those in PRA02, UARBAC and Uni-ARBAC. The method used to capture those features in APRA is by introducing attributes for admin users and permissions.

Likewise, ARPA also supports two ways to select permissions to which an admin user can assign a role. First way is to assign single permission to a single role. Secondly, a set of permissions can be selected using a set-builder notation whose rule is specified using permission attributes. Finally, the authorization rule is specified as a logical expression in the usual way over the attributes of the admin users and those of the permissions.

4.1 ARPA Model

Table 4.1 presents the formal ARPA model. The entities involved in ARPA include admin users (AU), admin operations (AOP), roles (ROLES) with a role hierarchy (RH), and permissions (PERMS). ARPA allows for an admin user in AU to perform an admin operation such as assign and revoke in

Table 4.1: ARPA Model

– AU is a finite set of administrative users.

– AOP is a finite set of admin operations such as assign and revoke.

– ROLES is a finite set of regular roles.

– RH $\subseteq$ ROLES $\times$ ROLES, a partial ordering on the set ROLES.

– PERMS is a finite set of permissions.

– AATT is a finite set of administrative user attribute functions.

– PATT is a finite set of permission attribute functions.

– For each *att* in AATT $\cup$ PATT, Scope(*att*) is a finite set of atomic values from which the range of the attribute function *att* is derived.

– attType : AATT $\cup$ PATT $\rightarrow$ $\{$set, atomic$\}$, which specifies whether the range of given attribute is atomic or set valued.

– Each attribute function maps elements in AU and PERMS to atomic or set values.

$$\forall aatt \in \text{AATT}.\ aatt : \text{AU} \rightarrow \begin{cases} \text{Scope}(aatt) \text{ if attType}(aatt) = \text{atomic} \\ 2^{\text{Scope}(aatt)} \text{ if attType}(aatt) = \text{set} \end{cases}$$

$$\forall patt \in \text{PATT}.\ patt : \text{PERMS} \rightarrow \begin{cases} \text{Scope}(patt) \text{ if attType}(patt) = \text{atomic} \\ 2^{\text{Scope}(patt)} \text{ if attType}(patt) = \text{set} \end{cases}$$

– is_ordered : AATT $\cup$ PATT $\rightarrow$ $\{$True, False$\}$, specifies if the scope is ordered for each of the attributes.

– For each *att* $\in$ AATT $\cup$ PATT,

if is_ordered(*att*) = True, $\text{H}_{att} \subseteq$ Scope(*att*) $\times$ Scope(*att*), a partially ordered attribute hierarchy, and $\text{H}_{att} \neq \phi$

else, if is_ordered(*att*) = False, $\text{H}_{att} = \phi$

(For some $att \in$ PATT $\cup$ AATT for which attType(*att*) = set and is_ordered(*att*) = True, if $\{a, b\}$, $\{c, d\} \in 2^{\text{Scope}(att)}$ (where $a, b, c, d \in$ Scope(att)), we infer $\{a, b\} \geq \{c, d\}$ if H^{*}_{att}.)

ARPA model allows an administrator to perform an operation on a single permission or a set of permissions at a time. The authorization rule for performing an operation on a single permission is as follows:

For each *op* in AOP, **is_authorizedP*op***(*au*: AU, *p* : PERMS, *r* : ROLES) specifies if the admin user *au* is allowed to perform the operation *op* (e.g. assign, revoke, etc.) between the permission *p* and the role *r*. This rule is written as a logical expression using attributes of the admin user *au* and attributes of the permission *p*.

The authorization rule for performing an operation on a set of permissions is as follows.

For each *op* in AOP, **is_authorizedP*op***(*au*: AU, χ : 2^{PERMS}, *r* : ROLES) specifies if the admin user *au* is allowed to perform the operation *op* (e.g. assign, revoke, etc.) between the permissions in the set of χ and the role *r*. Here χ is a set of permissions that can be specified using a set-builder notation, whose rule is written using permission attributes.

AOP between a regular user in USERS and a permission in PERMS, by using attributes of various participating entities. For that matter, sets of attribute functions for the regular users (UATT) and admin users (AATT) are defined. A motivation behind the development of ARPA is to allow ARPA to have the ability to capture the features of previously developed PRA models such as PRA97, PRA99, and PRA02. As it shall be observed, only attributes for admin users and permissions are needed to subsume the properties used by prior models considered in this dissertation. Aside from the attributes that we have defined for certain entities, one can envision attributes for ROLES or AOP. However, we limit the scope of model design based on the aforementioned motivation, and hence the attributes for selected entities.

The attributes are defined as a mapping from its domain (AU or PERMS as the case may be) to its range. The range of an attribute *att*, which can be atomic or set valued, is derived from a specified set of scope of atomic values denoted Scope(*att*). Whether an attribute is atomic or set valued is specified by a function called attType. Also, the scope of an attribute can be either ordered or unordered, which is specified by a function called is_ordered. If an attribute *att* is ordered, we require that a corresponding hierarchy, denoted H_{att}, be specified on its scope Scope(*att*). H_{att} is a partial ordering on Scope(*att*). Note that, even in the case of a set valued attribute *att*, the hierarchy H_{att} is specified on Scope(*att*) instead of $2^{\mathrm{Scope}(att)}$. We infer the ordering between two set values given an ordering on atomic values as explained in Table 4.1. Note that H^{*}_{att} denotes the reflexive transitive closure of H_{att}.

Like AURA, ARPA supports two ways to select a set of permissions for assigning them to a roles. The first one allows an admin user to identify a single permission, a role and perform an operation such as assign. The second one allows an admin user to identify a set of permissions, a role and perform an operation such as assign for all those permissions. In this case, the selection criteria for the set of permissions can be specified using a set-builder notation whose rule is stated using the regular permission attributes. For example, is_authorizedU$_{\mathbf{assign}}$(**au**, $\{p \mid p \in \mathrm{PERMS} \wedge \mathbf{t_1} \in admin\_unit(p)\}$, **r**) would specify a policy for an admin user **au** who identifies the set of all permissions that are associated with the task $\mathbf{t_1}$ in order to assign a role **r** to all those permissions.

Finally, the authorization rule is specified as a usual logical expression on the attributes of admin users and those of the set of permissions in question.

4.2 Mapping Prior PRA Models in ARPA

In this section, an idea that ARPA can intuitively simulate the features of prior PRA models is demostrated. To clarify the same, it presents example instance for each prior model that are taken into consideration, and they are manually mapped to equivalent ARPA model instance. Concrete algorithms that can convert any instance of PRA97, PRA99, PRA02, the PRA model in UARBAC, and the PRA model in Uni-ARBAC into an equivalent instance of ARPA is presented to demonstrate that any instance from each of these models can be mapped intuitively into ARPA model instance using attributes.

4.3 PRA97 in ARPA

This section presents how an instance from PRA97 model can be translated into equivalent ARPA model instance. Following that demonstration, a mapping algorithm called $\text{Map}_{\text{PRA97}}$ is exhibited to show that any instance from PRA97 can be automated to map into an equivalent instance of ARPA model. A summary of PRA97 is reviewed in the following segment:

4.3.1 Summary of PRA97 Model

Sets and functions:

– USERS is a finite set of users.

– PERMS is a finite set of permissions.

– AR is a finite set of administrative roles.

– ROLES is a finite set of regular roles.

– RH $\subseteq$ ROLES $\times$ ROLES a partial order on roles.

– CR is a finite set of prerequisite conditions.

A *prerequisite condition* is a boolean expression using the usual $\wedge$ and $\vee$ operators on terms of form x and $\bar{x}$ where x is a regular role (i.e., $x \in$ ROLES).

PRA97 Grant Model:

The PRA97 model controls the assignment by means of the relation,

$$can\_assign \subseteq \text{AR} \times \text{CR} \times 2^{\text{ROLES}}.$$

The meaning of *can_assign*(x, y, $\{a, b, c\}$) is that a member of the administrative role x (or an administrative role that is senior to x) can assign a permission whose current membership, or non-membership in a regular role satisfies the prerequisite condition y to be a member of regular roles a, b, or c.

The notion of a prerequisite condition is identical to that in URA97, except the boolean expression is now evaluated for membership and nonmembership of a permission in specified roles.

PRA97 Revoke Model:

The URA97 model controls user-role revocation by means of the following relation:

$$can\_revoke \subseteq \text{AR} \times 2^{\text{ROLES}}.$$

The meaning of *can_revoke*(x, Y) is that a member of the administrative role x (or a member of an administrative role that is senior to x) can revoke membership of a permission from any regular role $y \in Y$. Y defines the range of revocation.

4.3.2 PRA97 Instance

An example instance of aforementioned PRA97 model is instantiated with few prequisite conditions. The sets are as follows:

Sets and functions:

- USERS = $\{\mathbf{u_1, u_2, u_3, u_4}\}$
- ROLES = $\{\mathbf{x_1, x_2, x_3, x_4, x_5, x_6}\}$
- AR = $\{\mathbf{ar_1, ar_2}\}$
- PERMS = $\{\mathbf{p_1, p_2, p_3, p_4}\}$

- AUA = $\{(\mathbf{u_1}, \mathbf{ar_1}), (\mathbf{u_3}, \mathbf{ar_2})\}$

- PA = $\{(\mathbf{p_1}, \mathbf{r_1}), (\mathbf{p_2}, \mathbf{r_2}), (\mathbf{p_2}, \mathbf{r_4}), (\mathbf{p_3}, \mathbf{r_3}), (\mathbf{p_4}, \mathbf{r_3}), (\mathbf{p_4}, \mathbf{r_4})\}$

- RH = $\{<\mathbf{x_1}, \mathbf{x_2}>, <\mathbf{x_2}, \mathbf{x_3}>, <\mathbf{x_3}, \mathbf{x_4}>, <\mathbf{x_4}, \mathbf{x_5}>, <\mathbf{x_5}, \mathbf{x_6}>\}$

- ARH = $\{<\mathbf{ar_1}, \mathbf{ar_2}>\}$

- CR = $\{\mathbf{x_1} \wedge \mathbf{x_2}, \bar{\mathbf{x}}_\mathbf{1} \vee \mathbf{x_3}\}$

Let $cr_1 = \mathbf{x_1} \wedge \mathbf{x_2}$ and, $cr_2 = \bar{\mathbf{x}}_\mathbf{1} \vee \mathbf{x_3}$.

Prerequisite condition cr_1 is evaluated as follows:

For each p that is undertaken for assignment,

$(\exists x \leq \mathbf{x_1})(p, x) \in \text{PA} \wedge (\exists x \leq \mathbf{x_2})(p, x) \in \text{PA}$

cr_2 is evaluated as follows:

For each p that is undertaken for assignment,

$(\exists x \leq \mathbf{x_1})(p, x) \notin \text{PA} \vee (\exists x \leq \mathbf{x_3})(p, x) \in \text{PA}$

Let *can_assignp* and *can_revokep* be as follows:

can_assignp = $\{(\mathbf{ar_1}, cr_1, \{\mathbf{x_4}, \mathbf{x_5}\}), (\mathbf{ar_1}, cr_2, \{\mathbf{x_6}\})\}$

can_revokep = $\{(\mathbf{ar_1}, \text{ROLES})\}$

4.3.3 Equivalent Example Instance of ARPA for PRA97

This section presents an equivalent ARPA instance for the aforementioned PRA97 example instance.

Map set and functions from PRA97 to ARPA:

- AU = $\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}\}$

- AOP = {**assign, revoke**}

- ROLES = $\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}\ \ \mathbf{x_4}, \mathbf{x_5}, \mathbf{x_6}\}$

- RH = $\{<\mathbf{x_1}, \mathbf{x_2}>, <\mathbf{x_2}, \mathbf{x_3}>, <\mathbf{x_3}, \mathbf{x_4}>, <\mathbf{x_5}, \mathbf{x_6}>\}$

- PERMS = $\{\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}, \mathbf{p_4}\}$

Define attributes and values:

- AATT = $\{aroles\}$
- Scope($aroles$) = $\{\mathbf{ar_1}, \mathbf{ar_2}\}$

  attType($aroles$) = set, is_ordered($aroles$) = True, $H_{aroles} = \{\mathbf{<ar_1, ar_2>}\}$
- $aroles(\mathbf{u_1}) = \{\mathbf{ar_1}\}$, $aroles(\mathbf{u_2}) = \{\}$, $aroles(\mathbf{u_3}) = \{\mathbf{ar_2}\}$, $aroles(\mathbf{u_4}) = \{\}$
- PATT = $\{rolesp\}$
- Scope($rolesp$) = ROLES, attType($rolesp$) = set, is_ordered($rolesp$) = True, H_{rolesp} = RH
- $rolesp(\mathbf{p_1}) = \{\mathbf{r_1}\}$, $rolesp(\mathbf{p_2}) = \{\mathbf{r_2}, \mathbf{r_4}\}$, $rolesp(\mathbf{p_3}) = \{\mathbf{r_3}\}$, $rolesp(\mathbf{p_4}) = \{\mathbf{r_3}, \mathbf{r_4}\}$

Construct authorization functions for permission-role assignment:

Authorization rule for user-role assignment can be expressed as follows:

For any permission $p \in$ PERMS undertaken for assignment,

– is_authorizedP$_{\mathbf{assign}}$(au : AU, p : PERMS, r : ROLES) $\equiv$
$((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(au) \wedge r \in \{\mathbf{x_4}, \mathbf{x_5}\} \wedge (\exists x \leq \mathbf{x_1}).\ x \in rolesp(p) \wedge (\exists x \leq \mathbf{x_2}).\ x \in rolesp(p)\ \vee (\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(au) \wedge r \in \{\mathbf{x_6}\} \wedge (\exists x \leq \mathbf{x_1}).\ x \notin rolesp(p)$
$\vee (\exists x \leq \mathbf{x_3}).\ x \in rolesp(p)$

Construct authorization functions for revoking permission from role:

Authorization rule to revoke a permission from a role can be expressed as follows:

To revoke any permission $p \in$ PERMS from a role,

– is_authorizedP$_{\mathbf{revoke}}$(au : AU, p : PERMS, r : ROLES) $\equiv ((\exists ar \leq \mathbf{ar_1}).\ ar \in aroles(au) \wedge r \in$ ROLES)

The translation process follows the following process. First, appropriate sets from PRA97 model is mapped to the sets from ARPA model. In particular, sets for admin users, permissions, roles and their hierarchy, and administrative operations are mapped. Second, attributes and its values are defined. In particular, admin user attribute *aroles*(*au*) captures admin roles for each admin user. *aroles*($\mathbf{u_1}$) = {$\mathbf{ar_1}$} specifies that admin user $\mathbf{u_1}$ has admin role $\mathbf{ar_1}$. There is one permission attribute *rolesp*(*p*) that captures roles that a permission is assigned to. For example, *rolesp*($\mathbf{p_1}$) yields role $\mathbf{r_1}$, which means permission $\mathbf{p_1}$ is assigned to role $\mathbf{r_1}$. For both the attributes, corresponding range, type, order and hierarchies are defined. The attributes, values and their definition should reflect same meaning from the PRA97 model. In third step , authorization functions for assigning permission to role is constructed. In final step, authorization function for revoking permission from role is expressed.

4.3.4 Map$_{\mathbf{PRA97}}$

Algorithm Map$_{\text{PRA97}}$ is an algorithm for mapping any PRA97 instance into equivalent ARPA instance. For brevity, sets and functions from PRA97 and ARPA are marked with superscripts 97 and A, respectively. Map$_{\text{PRA97}}$ takes PRA97 instance as its input. In particular, input for Map$_{\text{PRA97}}$ fundamentally has USERS^{97}, ROLES^{97}, AR^{97}, PERMS^{97}, AUA^{97}, PA^{97}, RH^{97}, ARH^{97}, *can_assignp*^{97}, and *can_revokep*^{97}.

Output from Map$_{\text{PRA97}}$ algorithm is an equivalent ARPA instance, with primarily consisting of AU^{A}, AOP^{A}, ROLES^{A}, RH^{A}, PERMS^{A}, AATT^{A}, PATT^{A}, For each attribute $att \in$ AATT^{A} $\cup$ PATT^{A}, Scope^{A}(att), attType^{A}(att), is_ordered^{A}(att) and H$^{A}{}_{att}$, For each user $u \in$ AU^{A}, and for each $att \in$ AATT^{A}, $att(u)$, For each permission $p \in$ PERMS^{A}, and for each $att \in$ PATT^{A}, $att(p)$, Authorization rule for permission assign (auth_assign) and Authorization rule for permission revoke (auth_revoke)

As indicated in Map$_{\text{PRA97}}$, there are four main steps for mapping. In Step 1, sets and functions from PRA97 are mapped into ARPA sets and functions. In Step 2, permission attributes and administrative user attribute functions are expressed. There exists one permission attribute called *rolesp*.

Algorithm 4.1 Map$_{\text{PRA97}}$

```
Input: PRA97 instance
Output: ARPA instance
Step 1:     /* Map basic sets and functions in ARPA */
    a. $AU^{A} \leftarrow USERS^{97}$ ; $AOP^{A} \leftarrow \{$assign, revoke$\}$
    b. $ROLES^{A} \leftarrow ROLES^{97}$ ; $RH^{A} \leftarrow RH^{97}$ ; $PERMS^{A} \leftarrow PERMS^{97}$
Step 2:     /* Map attribute functions in ARPA */
    a. $AATT^{A} \leftarrow \{aroles\}$
    b. $Scope^{A}(aroles) = AR^{97}$ ; $attType^{A}(aroles) = set$
    c. $is\_ordered^{A}(aroles) = True$ ; $H^{A}_{aroles} \leftarrow ARH^{97}$
    d. For each $u \in AU^{A}$, $aroles(u) = \phi$
    e. For each $(u, ar)$ in $AUA^{97}$,
        $aroles(u)' = aroles(u) \cup ar$
    f. $PATT^{A} \leftarrow \{rolesp\}$
    g. $Scope^{A}(rolesp) = ROLES^{A}$
    h. $attType^{A}(rolesp) = set$ ; $is\_ordered^{A}(rolesp) = True$ ; $H^{A}_{rolesp} \leftarrow RH^{A}$
    i. For each $p$ in $PERMS^{A}$, $rolesp(u) = \phi$
    j. For each $(p, r)$ in $PA^{97}$, $rolesp(p)' = rolesp(p) \cup r$
Step 3:     /* Construct assign rule in ARPA */
    a. $assign\_formula = \phi$
    b. For each $(ar, cr, Z) \in can\_assign^{97}$,
        $assign\_formula' = assign\_formula \vee ((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z \wedge$
        $(translatep_{97}(cr)))$
    c. $auth\_assign = is\_authorizedP_{\mathbf{assign}}(au : AU^{A}, p : PERMS^{A}, r : ROLES^{A}) \equiv assign\_formula'$
Step 4:     /* Construct revoke rule for ARPA */
    a. $revoke\_formula = \phi$
    b. For each $(ar, cr, Z) \in can\_revokep^{97}$
        $revoke\_formula' = revoke\_formula \vee ((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z)$
    c. $auth\_revoke = is\_authorizedP_{\mathbf{assign}}(au : AU^{A}, p : PERMS^{A}, r : ROLES^{A}) \equiv assign\_formula'$
```

Support routine for algorithm 4.6 *translatep*$_{97}$

```
Input: A PRA97 prerequisite condition, cr
Output: An equivalent sub-rule for ARPA authorization assign rule.
 1: $rule\_string = \phi$
 2: For each symbol in cr
 3:     if $op$ = (mob-assignp $\vee$ immob-assignp) $\wedge$ symbol is a role and in the form $x$
                                        (i.e., the permission has membership on role $x$)
 4:         $rule\_string = rule\_string + (x \in exp\_mob\_mem(p) \vee$
                        $(x \in imp\_mob\_mem(p) \wedge x \notin exp\_immob\_mem(p))$
        else if $op$ = (mob-revokep $\vee$ immob-revokep) $\wedge$ symbol is a role and in the form $x$
                                        (i.e., the permission has membership on role $x$)
 5:         $rule\_string = rule\_string + (x \in exp\_mob\_mem(p) \vee x \in imp\_mob\_mem(p) \vee$
                        $x \in exp\_immob\_mem(p) \vee x \in imp\_immob\_mem(p))$
 6:     else if $op$ = (mob-assignp $\vee$ immob-assignp $\vee$ mob-revokep $\vee$ immob-revokep) $\wedge$
                  symbol is role and in the form $\bar{x}$
                                   (i.e., the permission doesn't have membership on role $x$)

 7:         $rule\_string = rule\_string + (x \notin exp\_mob\_mem(p) \wedge x \notin imp\_mob\_mem(p) \wedge$
                        $x \notin exp\_immob\_mem(p) \wedge x \notin imp\_immob\_mem(p))$
 8:     else
 9:         $rule\_string = rule\_string + symbol$   /* where a symbol is a $\wedge$ or $\vee$ logical operator */
10:     end if
```

It is an association between a permission and roles it has been assigned to. Admin user attribute *aroles* captures the association between admin users and admin roles in PRA97. Step 3 involves constructing assign_formula in ARPA that is equivalent to *can_assignp* in PRA97. *can_assignp* is a set of triples. Each triple bears information on whether an admin role can assign a candidate permission to a set of roles.

Equivalent translation equivalent to *can_assignp* in ARPA is given by is_authorizedP$_{\mathbf{assign}}$(au : AU$^{\mathbb{A}}$, p : PERMS$^{\mathbb{A}}$, r : ROLES$^{\mathbb{A}}$). Similarly, In Step 4, revoke_formula equivalent to *can_revokep* is presented. A support routine *translatep*$_{97}$ translates prerequisite condition.

4.4 PRA99 in ARPA

In the sections that follow, a method of mapping an instance of PRA99 model to an equivalent instance of ARPA model is presented. To realize that any instance from PRA99 model can be translated to ARPA model's equivalent instance, a concrete mapping algorithm is presented. A summary of PRA99 is re-visited as follows:

4.4.1 Summary of PRA99 Model

Sets and functions:

– USERS is a finite set of regular users.

– ROLES is a finite set of regular roles.

– PERMS is a finite set of permissions.

– AR is a finite set of administrative roles.

– CR is set of prerequisite conditions.

A *prerequisite condition* is a boolean expression using the usual $\wedge$ and $\vee$ operators on terms of form x and $\bar{x}$ where x is a regular role (i.e. $x \in$ ROLES).

PRA99 assumes two sub-roles of $x \in$ ROLES to be *Mx and IMx*. Membership of a user u in *Mx* and *IMx* are called mobile membership and immobile membership, respectively.

There are four kinds of permission-role membership in PRA99 for a given role x. They are as follows:

Explicit mobile member *EMx*

$p \in EMx \equiv (p, Mx) \in PA$

Explicit immobile member *EIMx*

$p \in EIMx \equiv (p, IMx) \in PA$

Implicit mobile member *ImMx*

$p \in ImMx \equiv (\exists x' < x)(p, Mx') \in PA$

implicit immobile member *ImIMx*

$p \in ImIMx \equiv (\exists x' < x)(p, IMx') \in PA$

PRA99 Grant Model:

User-role assignments as mobile or immobile members are authorized by the following relation:

$$\textit{can-assignp-M} \subseteq \text{AR} \times \text{CR} \times 2^{\text{ROLES}}.$$

The meaning of *can-assignp-M*(x, y, $\{a, b, c\}$) is that a member of administrative role x (or a member of administrative role senior to *x*) can assign a permission whose current membership, or non membership in roles in *R* satisfies the prerequisite *y* to a regular roles *a, b* or *c*, as a mobile member.

Similar is the definition for assigning a qualifying permission as immobile member of regular roles given by the relation:

$$\textit{can-assignp-IM} \subseteq \mathrm{AR} \times \mathrm{CR} \times 2^{\mathrm{ROLES}}.$$

A prerequisite condition is a boolean expression using the $\wedge$ and $\vee$ operators in terms of the form *r* and $\bar{x}$.

The prerequisite condition in PRA99 grant model is evaluated for a permission *p*, by interpreting *x* to be true if:

$$p \in EMx \vee (p \in ImMx \wedge p \notin EIMx)$$

and $\bar{x}$ to be true if:

$$p \notin EMx \wedge p \notin EIMx \wedge p \notin ImMx \wedge p \notin ImIMx$$

PRA99 Revoke Model:

The PRA99 model authorizes revocation of mobile membership by the relation:

$$\textit{can-revokep-M} \subseteq \mathrm{AR} \times \mathrm{CR} \times 2^{\mathrm{ROLES}}.$$

and revocation of immobile membership by the relation:

$$\textit{can-revokep-IM} \subseteq \mathrm{AR} \times \mathrm{CR} \times 2^{\mathrm{R}}.$$

The meaning of *can-revokep-M*(*x, y,* {*a, b, c*}) is that a member of administrative role *x* (or a member of a administrative role senior to *x*) can revoke mobile membership of a permission from role *a, b* or *c* subject to the prerequisite condition *y*. Similarly for *can-revokep-IM* with respect to immobile membership.

For the revoke model we do not distinguish mobile and immobile membership. Thus we have the following interpretation. A prerequisite condition in PRA99 revoke model is evaluated for a user *p* by interpreting *x* to be true if:

$$p \in EMx \vee p \in EIMx \vee p \in ImMx \vee p \in ImIMx$$

and $\bar{x}$ to be true if:

$$p \notin EMx \wedge p \notin EIMx \wedge p \notin ImMx \wedge p \notin ImIMx$$

PRA99 Instance

In this section, an example instance of the PRA99 model is presented.

Sets and functions:

- USERS = $\{\mathbf{u_1, u_2, u_3, u_4}\}$
- ROLES = $\{\mathbf{x_1, x_2, x_3, x_4, x_5, x_6}\}$
- AR = $\{\mathbf{ar_1, ar_2}\}$
- PERMS = $\{\mathbf{p_1, p_2, p_3, p_4}\}$
- AUA = $\{(\mathbf{u_3, ar_1}), (\mathbf{u_4, ar_2})\}$
- PA = $\{(\mathbf{p_1}, \mathrm{M}\mathbf{x_1}), (\mathbf{p_2}, \mathrm{IM}\mathbf{x_3}), (\mathbf{p_3}, \mathrm{IM}\mathbf{x_2}), (\mathbf{p_4}, \mathrm{M}\mathbf{x_4})\}$
- RH = $\{<\mathbf{x_1, x_2}>, <\mathbf{x_2, x_3}>, <\mathbf{x_3, x_4}>, <\mathbf{x_4, x_5}>, <\mathbf{x_5, x_6}>\}$
- ARH = $\{<\mathbf{ar_1, ar_2}>\}$
- CR = $\{\mathbf{x_2}, \bar{\mathbf{x}}_1\}$

Let $cr_1 = \mathbf{x_2}$ and, $cr_2 = \bar{\mathbf{x}}_1$. Prerequisite condition cr_1 is evaluated as follows:

For any $p \in$ PERMS that is undertaken for assignment,

$((p, \mathrm{M}\mathbf{x_2}) \in \mathrm{PA} \vee ((\exists x' \leq \mathbf{x_2}).\ (p, \mathrm{M}x') \in \mathrm{PA}) \wedge (p, \mathrm{IM}\mathbf{x_2}) \notin \mathrm{PA})$ cr_2 is evaluated as follows:

For any $p \in$ PERMS that is undertaken for assignment,

$(p, \mathrm{M}\mathbf{x_1}) \notin \mathrm{PA} \wedge ((\exists x' \leq \mathbf{x_1}).\ (p, \mathrm{M}x') \notin \mathrm{PA}) \wedge (p, \mathrm{IM}\mathbf{x_1}) \notin \mathrm{PA} \wedge ((\exists x' \leq \mathbf{x_1}).\ (p, \mathrm{IM}x') \notin \mathrm{PA})$

Let *can-assign-M* and *can-assign-IM* in PRA99 be as follows:

can-assign-M = $\{(\mathbf{ar_1}, cr_1, \{\mathbf{x_4, x_5}\})\}$

can-assign-IM= $\{(\mathbf{ar_1}, cr_2, \{\mathbf{x_3}\})\}$

For simplicity, same prerequisite conditions and target role sets are considered for grant and revoke models. Prerequisite conditions for PRA99 revoke model are evaluated as follows:

cr_1 is evaluated as follows:

$((p, \mathrm{M}\mathbf{x_2}) \in \mathrm{PA} \vee (p, \mathrm{IM}\mathbf{x_2}) \in \mathrm{PA} \vee ((\exists x' \leq \mathbf{x_2}).\ (p, \mathrm{M}x') \in \mathrm{PA}) \vee ((\exists x' \leq \mathbf{x_2}).\ (p, \mathrm{IM}x') \in \mathrm{PA}))$

cr_2 is evaluated as follows:

$(p, \mathrm{M}\mathbf{x_1}) \notin \mathrm{PA} \wedge (p, \mathrm{IM}\mathbf{x_1}) \notin \mathrm{PA} \wedge ((\exists x' \leq \mathbf{x_1}).\ (p, \mathrm{M}x') \notin \mathrm{PA}) \wedge ((\exists x' \leq \mathbf{x_1}).\ (p, \mathrm{IM}x') \notin \mathrm{PA})$

Let *can-revoke-M* and *can-revoke-IM* sets be as follows:

can-revoke-M = $\{(\mathbf{ar_1}, cr_1, \{\mathbf{x_3}, \mathbf{x_4}, \mathbf{x_5}\})\}$

can-revoke-IM= $\{(\mathbf{ar_1}, cr_2, \{\mathbf{x_5}, \mathbf{x_6}\})\}$

4.4.2 Equivalent PRA99 Instance in ARPA

This section presents an equivalent ARPA instance for the aforementioned PRA99 example instance.

Map sets and functions from PRA99 to ARPA:

- AU = $\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}\}$
- AOP = {**mob-assign, immob-assign, mob-revoke, immob-revoke**}
- ROLES = $\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \mathbf{x_4}, \mathbf{x_5}, \mathbf{x_6}\}$
- RH = $\{<\mathbf{x_1}, \mathbf{x_2}>, <\mathbf{x_2}, \mathbf{x_3}>, <\mathbf{x_3}, \mathbf{x_4}>, <\mathbf{x_4}, \mathbf{x_5}>, <\mathbf{x_5}, \mathbf{x_6}>\}$
- PERMS = $\{\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}, \mathbf{p_4}\}$

Define attributes and values:

- AATT = {*aroles*}
- Scope(*aroles*) = $\{\mathbf{ar_1}, \mathbf{ar_2}\}$, attType(*aroles*) = set,

  is_ordered(*aroles*) = True, $\mathrm{H}_{aroles} = \{<\mathbf{ar_1}, \mathbf{ar_2}>\}$
- *aroles*($\mathbf{u_1}$) = { }, *aroles*($\mathbf{u_2}$) = { }, *aroles*($\mathbf{u_3}$) = $\{\mathbf{ar_1}\}$, *aroles*($\mathbf{u_4}$) = $\{\mathbf{ar_2}\}$
- PATT= {*exp_mob_mem, imp_mob_mem, exp_immob_mem, imp_immob_mem*}

- Scope(*exp_mob_mem*) = ROLES, attType(*exp_mob_mem*) = set,

  is_ordered(*exp_mob_mem*) = True, $H_{exp\_mob\_mem}$ = RH
- *exp_mob_mem*($\mathbf{p_1}$) = {$\mathbf{x_1}$}, *exp_mob_mem*($\mathbf{p_2}$) = { },

  exp_mob_mem($\mathbf{p_3}$) = { }, *exp_mob_mem*($\mathbf{p_4}$) = {$\mathbf{x_4}$},
- Scope(*imp_mob_mem*)= ROLES, attType(*imp_mob_mem*) = set,

  is_ordered(*imp_mob_mem*) = True, $H_{imp\_mob\_mem}$ = RH
- *imp_mob_mem*($\mathbf{p_1}$) = { }, *imp_mob_mem*($\mathbf{p_2}$) = { },

  imp_mob_mem($\mathbf{p_3}$) = { }, *imp_mob_mem*($\mathbf{p_4}$) = {$\mathbf{x_1, x_2, x_3}$}
- Scope(*exp_immob_mem*) = ROLES, attType(*exp_immob_mem*) = set

  is_ordered(*exp_immob_mem*) = True, $H_{exp\_immob\_mem}$ = RH
- *exp_immob_mem*($\mathbf{p_1}$) = { }, *exp_immob_mem*($\mathbf{p_2}$) = {$\mathbf{x_3}$},

  exp_immob_mem($\mathbf{p_3}$) = {$\mathbf{x_2}$}, *exp_immob_mem*($\mathbf{p_4}$) = { },
- Scope(*imp_immob_mem*) = ROLES, attType(*imp_immob_mem*) = set
- is_ordered(*imp_immob_mem*) = True, $H_{imp\_immob\_mem}$ = RH
- *imp_immob_mem*($\mathbf{p_1}$) = { }, *imp_immob_mem*($\mathbf{p_2}$) = {$\mathbf{x_1, x_2}$},

  imp_immob_mem($\mathbf{p_3}$) = {$\mathbf{x_1}$}, *imp_immob_mem*($\mathbf{p_4}$) = { }

Construct authorization function for moble permission-role assignment:

Authorization rule to assign a permission as a mobile member of a role can be expressed as follows:

To assign any permission $p \in$ PERMS as a mobile member,

– is_authorizedP$_{\textbf{mob-assign}}$(au : AU, p : PERMS, r : ROLES) $\equiv$

$((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u) \wedge r \in \{\mathbf{x_4, x_5}\} \wedge (\mathbf{x_2} \in exp\_mob\_mem(p) \vee (\mathbf{x_2} \in imp\_mob\_mem(p)$

$\wedge\ \mathbf{x_2} \notin exp\_immob\_mem(p)))$

<u>Construct authorization function for revoking mobile permission from role:</u>

Authorization rule to revoke a mobile permission from a role can be expressed as follows:

To revoke any mobile permission $p \in$ PERMS from a role,

– is_authorizedP$_{\textbf{mob-revoke}}$(au : AU, p : PERMS, r : ROLES) $\equiv$

$((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u) \wedge r \in \{\mathbf{x_3, x_4, x_5}\} \wedge (\mathbf{x_2} \in exp\_mob\_mem(p) \vee \mathbf{x_2} \in imp\_mob\_mem(p)$

$\vee\ \mathbf{x_2} \in exp\_immob\_mem(p) \vee \mathbf{x_2} \in imp\_immob\_mem))$

<u>Construct authorization function for assigning immobile permission to role:</u>

Authorization functions to assign any permission $p \in$ PERMS as an immobile member of role can be expressed as follows:

To assign any permission $p \in$ PERMS as an immobile member,

– is_authorizedP$_{\textbf{immob-assign}}$(au : AU, p : PERMS, r : ROLES) $\equiv ((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(u)$

$\wedge\ r \in \{\mathbf{x_3}\} \wedge (\mathbf{x_1} \notin exp\_mob\_mem(p) \wedge \mathbf{x_1} \notin imp\_mob\_mem(p) \wedge \mathbf{x_1} \notin exp\_immob\_mem(p)$

$\wedge\ \mathbf{x_1} \notin imp\_immob\_mem(p)))$

<u>Construct authorization function for revoking immobile permission from role:</u>

Authorization rule to revoke any immobile permission from a role can be expressed as follows:

To revoke any immobile permission $p \in$ PERMS from a role,

– is_authorizedP$_{\textbf{immob-revoke}}$(au : AU, p : PERMS, r : ROLES) $\equiv ((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(p)$

$\wedge\ r \in \{\mathbf{x_5, x_6}\} \wedge \mathbf{x_1} \notin exp\_mob\_mem(p) \wedge \mathbf{x_1} \notin imp\_mob\_mem(p) \wedge \mathbf{x_1} \notin exp\_immob\_mem(p)$

$\wedge\ \mathbf{x_1} \notin imp\_immob\_mem(p))$

The manual translation involved the following process. First, the sets from PRA99 model are mapped to the sets of APRA model. Sets for admin users, perimissions, roles and their hierarchy and, administrative operations are mapped. There are four administrative operations, **mob-assign**,

immob-assign, **mob-revoke**, and **immob-revoke** for assigning permission as a mobile member, assigning a permission as an immobile member, revoking a mobile permission, and revoking an immobile permission, respectively. Second, the relations pertained in PRA99 are intuitively translated as attributes in ARPA. attribute *aroles* captures admin roles for given admin user. There are four different permission attributes *exp_mob_mem*, *imp_mob_mem*, *exp_immob_mem*, and *imp_immob_mem*. *exp_mob_mem* attribute yields all the roles to which a given permission has explicit mobile membership. For example, *exp_mob_mem*($\mathbf{p_1}$) = {$\mathbf{x_1}$} signifies permission $\mathbf{p_1}$ has explicit mobile membership on role $\mathbf{x_1}$. *imp_mob_mem* yields roles to which a permission has implicit mobile membership. For example, *imp_mob_mem*($\mathbf{p_4}$) = {$\mathbf{x_1}$, $\mathbf{x_2}$, $\mathbf{x_3}$} specifies that $\mathbf{p_4}$ has implicit mobile membership on roles $\mathbf{x_1}$ through $\mathbf{x_3}$. *exp_immob_mem* yields roles to which a permission has explicit immobile membership. For example, *exp_immob_mem*($\mathbf{p_2}$) = {$\mathbf{x_3}$} indicates that $\mathbf{p_2}$ has explicit immobile membership on role $\mathbf{x_3}$ and, *imp_immob_mem* yields roles to which a permission has implicit immobile membership. For example, *imp_immob_mem*($\mathbf{p_3}$) = {$\mathbf{x_1}$} function indicates that permission $\mathbf{p_3}$ has implicit immobile membership on role $\mathbf{x_1}$. For all the attributes, their corresponding range, type, order and hierarchies were mapped.

In step three, an equivalent authorization functions for permission-role assignment and permission-role revocation were constructed respectively for mobile users using related entities and their attributes. Finally in step four, authorization functions for assigning and revoking immobile user to/from role, respectively were constructed using related entities and their attributes.

4.4.3 Map$_{\text{PRA99}}$

Algorithm Map$_{\text{PRA99}}$ is an algorithm for mapping any PRA99 instance into equivalent ARPA instance. Sets and functions from PRA99 and ARPA are marked with superscripts 99 and A, respectively. Map$_{\text{PRA99}}$ takes PRA99 instance as its input. In particular, input for Map$_{\text{PRA99}}$ fundamentally has USERS^{99}, PERMS^{99}, ROLES^{99}, AR^{99}, PA^{99}, AUA^{99}, RH^{99}, ARH^{99}, *can-assignp-M*^{99}, *can-assignp-IM*^{99}, *can-revokep-M*^{99}, and *can-revokep-IM*^{99}.

Output from Map$_{\text{PRA99}}$ algorithm is an equivalent ARPA instance, with primarily consisting of $\mathrm{AU}^{\mathtt{A}}$, $\mathrm{AOP}^{\mathtt{A}}$, $\mathrm{ROLES}^{\mathtt{A}}$, $\mathrm{RH}^{\mathtt{A}}$, $\mathrm{PERMS}^{\mathtt{A}}$, $\mathrm{AATT}^{\mathtt{A}}$, $\mathrm{PATT}^{\mathtt{A}}$, For each attribute $att \in \mathrm{AATT}^{\mathtt{A}} \cup \mathrm{PATT}^{\mathtt{A}}$, $\mathrm{Range}^{\mathtt{A}}(att)$, $\mathrm{attType}^{\mathtt{A}}(att)$, $\mathrm{is\_ordered}^{\mathtt{A}}(att)$ and ${\mathrm{H}^{\mathtt{A}}}_{att}$, For each user $u \in \mathrm{AU}^{\mathtt{A}}$, and for each $att \in \mathrm{AATT}^{\mathtt{A}}$, $att(u)$, For each permission $p \in \mathrm{PERMS}^{\mathtt{A}}$, and for each $att \in \mathrm{PATT}^{\mathtt{A}}$, $att(p)$, Authorization rule for mobile assign (auth_mob_assign), Authorization rule for mobile revoke (auth_mob_revoke), Authorization rule for immobile assign (auth_immob_assign), and Authorization rule for immobile revoke (auth_immob_revoke)

As shown in Algorithm $\mathrm{Map}_{\mathrm{PRA99}}$, there are four main steps required in mapping any instance of PRA99 model to ARPA instance. In Step 1, sets and functions from PRA99 instance are mapped into ARPA sets and functions. In Step 2, permission attributes and administrative user attribute functions are expressed. There are four permission attributes: *exp_mob_mem, imp_mob_mem, exp_immob_mem,* and *imp_immob_mem*. Each captures, a permission's explicit mobile membership, implicit mobile membership, explicit immobile membership and implicit immobile membership, respectively, on roles. Admin user attribute *aroles* captures admin roles assigned to admin users. Step 3 involves constructing assign-mob-formula and assign-immob-formula in ARPA that is equivalent to *can-assignp-M* and *can-assignp-IM* in PRA99, respectively, in PRA99. Both *can-assignp-M* and *can-assignp-IM* are set of triples. Each triple bears information on whether an admin role can assign a candidate permission to a set of roles as a mobile member in the case of *can-assignp-M* and, as an immobile member in the case of *can-assignp-IM*. AURA equivalent for *can-assignp-M* is given by $\mathrm{is\_authorizedP}_{\textbf{mob-assign}}(au : \mathrm{AU}^{\mathtt{A}}, u : \mathrm{USERS}^{\mathtt{A}}, r : \mathrm{ROLES}^{\mathtt{A}})$ and an equivalent translation for *can-assignp-IM* is given by $\mathrm{is\_authorizedP}_{\textbf{immob-assign}}(au : \mathrm{AU}^{\mathtt{A}}, u : \mathrm{USERS}^{\mathtt{A}}, r : \mathrm{ROLES}^{\mathtt{A}})$. Similarly, In Step 4, revoke-mob-formula equivalent to *can-revokep-M* and *can-revokep-IM* are presented. A support routine $\mathit{translatep}_{99}$ translates prerequisite condition in PRA99 into its ARPA equivalent. A complete example instance and its corresponding equivalent APRA instances were presented in Section 4.4.1 and Section 4.4.2, respectively.

Algorithm 4.7 $\text{Map}_{\text{PRA99}}$

```
Input: PRA99 instance
Output: ARPA instance
Step 1:  /* Map basic sets and functions in ARPA */
    a. $AU^{A} \leftarrow USERS^{99}$ ; $AOP^{A} \leftarrow \{$mob-assign, mob-revoke, immob-assign, immob-revoke$\}$
    b. $ROLES^{A} \leftarrow ROLES^{99}$ ; $RH^{A} \leftarrow RH^{99}$ ; $PERMS^{A} \leftarrow PERMS^{99}$
Step 2:  /* Map attribute functions in ARPA */
    a. $AATT^{A} \leftarrow \{aroles\}$
    b. $Scope^{A}(aroles) = AR^{99}$ ; $attType^{A}(aroles)$ = set
    c. $is\_ordered^{A}(aroles)$ = True ; $H^{A}_{aroles} \leftarrow ARH^{99}$
    d. For each $u \in AU^{A}$, $aroles(u) = \phi$
    e. For each $(u, ar)$ in $AUA^{99}$, $aroles(u) = aroles(u) \cup ar$
    f. $PATT^{A} \leftarrow \{exp\_mob\_mem, imp\_mob\_mem, exp\_immob\_mem, imp\_immob\_mem\}$
    g. $Scope^{A}(exp\_mob\_mem) = ROLES^{A}$ ; $attType^{A}(exp\_mob\_mem)$ = set
    h. $is\_ordered^{A}(exp\_mob\_mem)$ = True ; $H^{A}_{exp\_mob\_mem} \leftarrow RH^{A}$
    i. For each $p$ in $PERMS^{A}$, $exp\_mob\_mem(p) = \phi$
    j. For each $(p, Mr)$ in $PA^{99}$, $exp\_mob\_mem(p)' = exp\_mob\_mem(p) \cup r$
    k. $Scope^{A}(imp\_mob\_mem) = ROLES^{A}$ ; $attType^{A}(imp\_mob\_mem)$ = set ;
    l. $is\_ordered^{A}(imp\_mob\_mem)$ = True ; $H^{A}_{imp\_mob\_mem} \leftarrow RH^{A}$
    m. For each $p$ in $PERMS^{A}$, $imp\_mob\_mem(p) = \phi$
    n. For each $(p, Mr)$ in $PA^{99}$ and for each role $r' > r$,
            $imp\_mob\_mem(p)' = imp\_mob\_mem(p) \cup r'$
    o. $Scope^{A}(exp\_immob\_mem) = ROLES^{A}$ ; $attType^{A}(exp\_immob\_mem)$ = set
    p. $is\_ordered^{A}(exp\_immob\_mem)$ = True ; $H^{A}_{exp\_immob\_mem} \leftarrow RH^{A}$
    q. For each $p$ in $PERMS^{A}$, $exp\_immob\_mem(p) = \phi$ ;
    r. For each $(p, IMr)$ in $PA^{99}$, $exp\_immob\_mem(p)' = exp\_immob\_mem(p) \cup r$
    s. $Scope^{A}(imp\_immob\_mem) = ROLES^{A}$ ; $attType^{A}(imp\_immob\_mem)$ = set
    t. $is\_ordered^{A}(imp\_immob\_mem)$ = True ; $H^{A}_{imp\_immob\_mem} \leftarrow RH$
```

Algorithm 4.7 $\text{Map}_{\text{PRA99}}$

```
u. $\text{attType}^{A}(imp\_immob\_mem)$ = set ; $\text{is\_ordered}^{A}(imp\_immob\_mem)$ = True
v. $H^{A}_{imp\_immob\_mem} \leftarrow RH^{A}$ ; For each $p$ in $\text{PERMS}^{A}$, $imp\_immob\_mem(p) = \phi$
w. For each ($p$, IM$r$) in $PA^{99}$ and for each role $r' > r$,
        $imp\_immob\_mem(p)' = imp\_immob\_mem(p) \cup r'$
Step 3:    /* Construct assign rule in ARPA */
  a. assign-mob-formula = $\phi$
  b. For each $(ar, cr, Z) \in can\text{-}assign\text{-}M^{99}$,
       assign-mob-formula' = assign-mob-formula $\vee$ $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z \wedge$
          $(translatep_{99}(cr$, mob-assign$)))$
  c. auth_mob_assign = $\text{is\_authorizedP}_{\textbf{mob-assign}}(au : \text{AU}, p : \text{PERMS}, r : \text{ROLES}) \equiv$
  assign-mob-formula'
  d. assign-immob-formula = $\phi$
  e. For each $(ar, cr, Z) \in can\text{-}assign\text{-}IM^{99}$,
       assign-immob-formula' = assign-immob-formula $\vee$ $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z$
       $\wedge\ (translatep_{99}(cr$, immob-assign$)))$
  f. auth_immob_assign = $\text{is\_authorizedP}_{\textbf{immob-assign}}(au : \text{AU}, p : \text{PERMS}, r : \text{ROLES}) \equiv$
  assign-immob-formula'
Step 4:    /* Construct revoke rule in ARPA */
  a. revoke-mob-formula = $\phi$
  b. For each $(ar, cr, Z) \in can\text{-}revoke\text{-}M^{99}$,
       revoke-mob-formula' = revoke-mob-formula $\vee$ $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z \wedge$
          $(translatep_{99}(cr$, mob-revoke$)))$
  c. auth_mob_revoke = $\text{is\_authorizedP}_{\textbf{mob-revoke}}(au : \text{AU}, p : \text{PERMS}, r : \text{ROLES}) \equiv$
  revoke-mob-formula'
  d. revoke-immob-formula = $\phi$
  e. For each $(ar, cr, Z) \in can\text{-}revoke\text{-}IM^{99}$,
       revoke-immob-formula' = revoke-immob-formula $\vee$ $((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge$
          $r \in Z \wedge (translatep_{99}(cr$, immob-revoke$)))$
  f. auth_immob_revoke = $\text{is\_authorizedP}_{\textbf{immob-revoke}}(au : \text{AU}, p : \text{PERMS}, r : \text{ROLES}) \equiv$
  revoke-immob-formula'
```

Support routine for algorithm 4.7 *translatep*$_{99}$

Input: A PRA99 prerequisite condition (*cr*),

$op \in \{$**mob-assign, immob-assign, mob-revoke, immob-revoke**$\}$

Output: An equivalent sub-rule for AURA authorization assign rule.

```
1:  $rule\_string = \phi$
2:  For each $symbol$ in $cr$
3:      if $op$ = (mob-assignp $\vee$ immob-assignp) $\wedge$ $symbol$ is a role and in the form $x$
                                        (i.e., the permission has membership on role $x$)
4:          $rule\_string = rule\_string + (x \in exp\_mob\_mem(p) \vee$
                        $(x \in imp\_mob\_mem(p) \wedge x \notin exp\_immob\_mem(p))$
        else if $op$ = (mob-revokep $\vee$ immob-revokep) $\wedge$ $symbol$ is a role and in the form $x$
                                        (i.e., the permission has membership on role $x$)
5:          $rule\_string = rule\_string + (x \in exp\_mob\_mem(p) \vee x \in imp\_mob\_mem(p) \vee$
                        $x \in exp\_immob\_mem(p) \vee x \in imp\_immob\_mem(p))$
6:      else if $op$ = (mob-assignp $\vee$ immob-assignp $\vee$ mob-revokep $\vee$ immob-revokep) $\wedge$
                    $symbol$ is role and in the form $\bar{x}$
                                    (i.e., the permission doesn't have membership on role $x$)

7:          $rule\_string = rule\_string + (x \notin exp\_mob\_mem(p) \wedge x \notin imp\_mob\_mem(p) \wedge$
                        $x \notin exp\_immob\_mem(p) \wedge x \notin imp\_immob\_mem(p))$
8:      else
9:          $rule\_string = rule\_string + symbol$    /* where a $symbol$ is a $\wedge$ or $\vee$ logical operator */
10:     end if
```

4.5 PRA02 in ARPA

In this section, a method by which any instance of PRA02 model is translated into an equivalent instance of APRA model is demonstrated. For that matter, firstly, a summary of PRA02 model is presented. An example, instance for PRA02 model is manually converted into its equivalent representation in ARPA model. This shows an intuitive approach by which attributes can represent the properties presented in PRA02 model. Finally, a concrete algorithm that can take any instance of PRA02 and map it into ARPA model is shown. The summary of PRA02 model is presented below:

Sets and functions:

4.5.1 Summary of PRA02 Model

– USERS is a finite set of regular users.

– PERMS is a finite set of permissions.

– AR is a finite set of administrative roles.

– CR is a finite set of prerequisite conditions.

– ROLES is a finite set of regular roles.

– ORGU is a finite set of orgnization units.

– PPA $\subseteq$ PERMS $\times$ ORGU, regular user to organization unit assignment on OS-P (Organization structure represented as a permission-pool).

– CR is a finite set of prerequisite conditions.

A *prerequisite condition* of PRA is a boolean expression using the $\wedge$ and $\vee$ operators on terms of the form *x* and $\bar{x}$, where *x* is a *regular role or organization unit* in OS-P.

PRA02 Grant Model:

User-role assignment is authorized in PRA02 by the following relation:

$$can\_assignp \subseteq \text{AR} \times \text{CR} \times 2^{\text{ROLES}}.$$

The meaning of *can_assign*(*x, y,* {*a, b, c*}) is that a member of an administrative role *x* (or a member of a role that is senior to *x*) can assign a pemrission whose current membership, or non-membership, in regular role or organization unit satisfies the prerequisite condition *y*, to regular roles, *a, b* or *c*.

PRA02 has same grant model as PRA97 but the prerequisite condition is different. A prerequisite condition is evaluated for user *u* by interpreting *x* to be true if:

Case 1:

$x \in$ ROLES : $(\exists x' \leq x)(u, x') \in$ PA

Case 2:

$x \in$ ORGU : $(\exists x' \geq x)(u, x') \in$ PPA

and $\bar{x}$ to be true if:

Case 1:

$x \in$ ROLES : $\neg((\forall x' \leq x)(u, x') \in$ PA)

Case 2:

$x \in$ ORGU : $\neg((\forall x' \geq x)(u, x') \in$ PPA)

<u>PRA02 Revoke Model</u>

The PRA02 model controls the permission-role or permission-organization unit revocation by means of the following relation:

$$can\_revokep \subseteq \text{AR} \times 2^{\text{R}}.$$

The meaning of *can_revokep*(*x, Y*) is that a member of the administrative role *x* (or a member of an administrative role senior to *x*) can revoke a membership of a permission from any regular role or organization unit $y \in Y$.

4.5.2 PRA02 Instance

In PRA02, decision about permission-role assignment and revocation is made on the basis of two factors: a permission's membership on role(s) or a permission's membership in organization unit(s). They can be viewed as two different cases. In this example instance we represent roles with *r* and organization units with *x*, for simplicity and clarity.

Sets and functions:

- USERS = {$\mathbf{u_1, u_2, u_3, u_4}$}
- ROLES = {$\mathbf{r_1, r_2, r_3, r_4, r_5, r_6}$}
- AR = {$\mathbf{ar_1, ar_2}$}
- PERMS = {$\mathbf{p_1, p_2, p_3, p_4}$}
- AUA = {($\mathbf{u_3, ar_1}$), ($\mathbf{u_4, ar_2}$)}
- PA = {($\mathbf{p_1, r_1}$), ($\mathbf{p_1, r_2}$), ($\mathbf{p_2, r_3}$), ($\mathbf{p_2, r_4}$)}
- RH = {<$\mathbf{r_1, r_2}$>, <$\mathbf{r_2, r_3}$>, <$\mathbf{r_3, r_4}$>, <$\mathbf{r_4, r_5}$>, <$\mathbf{r_5, r_6}$>}
- ARH = {<$\mathbf{ar_1, ar_2}$>}
- ORGU = {$\mathbf{x_1, x_2, x_3}$}
- OUH = {<$\mathbf{x_3, x_2}$>, <$\mathbf{x_2, x_1}$>}
- PPA = {($\mathbf{p_1, x_1}$), ($\mathbf{p_2, x_2}$), ($\mathbf{p_3, x_3}$), ($\mathbf{p_1, x_3}$)}

  Case 1:

- CR = {$\mathbf{r_1} \wedge \mathbf{r_2}, \mathbf{r_1} \vee \mathbf{\bar{r}_2} \wedge \mathbf{x_3}$}

  Let $cr_1 = \mathbf{r_1} \wedge \mathbf{r_2}$ and, $cr_2 = \mathbf{r_1} \vee \mathbf{\bar{r}_2} \wedge \mathbf{r_3}$

  Case 2:

- CR = {$\mathbf{x_1} \wedge \mathbf{x_2}, \mathbf{x_1} \vee \mathbf{\bar{x}_2} \wedge \mathbf{x_3}$}

  Let $cr_3 = \mathbf{x_1} \wedge \mathbf{x_2}$ and, $cr_4 = \mathbf{x_1} \vee \mathbf{\bar{x}_2} \wedge \mathbf{x_3}$

Prerequisite conditions are evaluated as follows:

Case 1:

cr₁ is evaluated as follows:

For each *p* that is undertaken for assignment,

$(\exists r \leq \mathbf{r_1}).\ (p, r) \in \text{PA} \wedge (\exists r \leq \mathbf{r_2}).\ (p, r) \in \text{PA}$

cr₂ is evaluated as follows: For each *p* that is undertaken for assignment,

$(\exists r \leq \mathbf{r_1}).\ (p, r) \in \text{PA} \vee \neg((\forall r \leq \mathbf{r_2}).\ (p, r) \in \text{PA}) \wedge (\exists r \leq \mathbf{r_3}).\ (p, r) \in \text{PA}$

<u>Case 2:</u>

cr_3 is evaluated as follows:

For each *p* that is undertaken for assignment,

$(\exists x \geq \mathbf{x_1}).\ (p, x) \in \text{PPA} \wedge (\exists x \geq \mathbf{x_2}).\ (p, x) \in \text{PPA}$

cr_4 is evaluated as follows:

For each *p* that is undertaken for assignment,

$(\exists x \geq \mathbf{x_1}).\ (p, x) \in \text{PPA} \vee \neg((\forall x \geq \mathbf{x_2}).\ (p, x) \in \text{PPA}) \wedge (\exists x \leq \mathbf{x_3}).\ (p, x) \in \text{PPA}$

Let *can_assign* and *can_revoke* be as follows:

<u>Case 1:</u>

can_assign = $\{(\mathbf{ar_1}, cr_1, \{\mathbf{r_4}, \mathbf{r_5}\}), (\mathbf{ar_1}, cr_2, \{\mathbf{r_6}\})\}$

can_revoke = $\{(\mathbf{ar_1}, \{\mathbf{r_1}, \mathbf{r_3}, \mathbf{r_4}\})\}$

<u>Case 2:</u>

can_assign = $\{(\mathbf{ar_1}, cr_3, \{\mathbf{r_4}, \mathbf{r_5}\}), (\mathbf{ar_1}, cr_4, \{\mathbf{r_6}\})\}$

can_revoke = $\{(\mathbf{ar_1}, \{\mathbf{r_1}, \mathbf{r_3}, \mathbf{r_4}\})\}$

4.5.3 Equivalent PRA02 Instance in ARPA

<u>Map sets and functions from PRA02 to ARPA:</u>

- AU = $\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, \mathbf{u_4}\}$
- AOP = {assign, revoke}
- ROLES = $\{\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_3}, \mathbf{r_4}, \mathbf{r_5}, \mathbf{r_6}\}$
- RH = $\{<\mathbf{r_1}, \mathbf{r_2}>, <\mathbf{r_2}, \mathbf{r_3}>, <\mathbf{r_3}, \mathbf{r_4}>, <\mathbf{r_4}, \mathbf{r_5}>, <\mathbf{r_5}, \mathbf{r_6}>\}$
- PERMS = $\{\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}, \mathbf{p_4}\}$

Define attributes and values:

- AATT = {*aroles*}
- Scope(*aroles*) = {$\mathbf{ar_1}$, $\mathbf{ar_2}$}, attType(*aroles*) = set

  is_ordered(*aroles*) = True, H_{aroles} = {<$\mathbf{ar_1}$, $\mathbf{ar_2}$>}
- *aroles*($\mathbf{u_3}$) = {$\mathbf{ar_1}$}, *aroles*($\mathbf{u_4}$) = {$\mathbf{ar_2}$}
- PATT = {*rolesp, org_units*}
- Scope(*rolesp*) = ROLES, attType(*rolesp*) = set

  is_ordered(*rolesp*) = True, H_{rolesp} = RH,
- *rolesp*($\mathbf{p_1}$) = {$\mathbf{r_1}$, $\mathbf{r_2}$}, *rolesp*($\mathbf{p_2}$) = {$\mathbf{r_3}$, $\mathbf{r_4}$}, *rolesp*($\mathbf{p_3}$) = { }, *rolesp*($\mathbf{p_4}$) = { }
- Scope(*org_units*) = {$\mathbf{x_1}$, $\mathbf{x_2}$, $\mathbf{x_3}$} ; attType(*org_units*) = set

  is_ordered(*org_units*) = True ; H_{org_units} = {($\mathbf{x_3}$, $\mathbf{x_2}$), ($\mathbf{x_2}$, $\mathbf{x_1}$)}
- *org_units*($\mathbf{p_1}$) = {$\mathbf{x_1}$, $\mathbf{x_3}$}, *org_units*($\mathbf{p_2}$) = {$\mathbf{x_2}$}, *org_units*($\mathbf{p_3}$) = {$\mathbf{x_3}$}

For each *op* in OP, authorization rule for permission to role assignment and revocation can be expressed respectively, as follows:

Construct authorization function for permission-role assignment:

Case 1:

For any permission $p \in$ PERMS undertaken for assignment,

– is_authorized$P_{\mathbf{assign}}$(au : AU, p : PERMS, r : ROLES) $\equiv$

$((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(au) \wedge r \in \{\mathbf{r_4}, \mathbf{r_5}\} \wedge ((\exists r \leq \mathbf{r_1}).\ r \in rolesp(p) \wedge (\exists r \leq \mathbf{r_2}).\ r \in rolesp(p))) \vee ((\exists ar \geq \mathbf{ar_1}).\ ar \in aroles(au) \wedge r \in \{\mathbf{r_6}\} \wedge ((\exists r \leq \mathbf{r_1}).\ r \in rolesp(p)$

$\vee (\exists r \leq \mathbf{r_2}).\ r \notin rolesp(p) \wedge (\exists r \leq \mathbf{r_3}).\ r \in rolesp(p)))$

Case 2:

or any permission $p \in$ PERMS undertaken for assignment,

– is_authorized$\text{P}_{\textbf{assign}}(au : \text{AU}, p : \text{PERMS}, r : \text{ROLES}) \equiv$
$((\exists ar \geq \textbf{ar}_1).\ ar \in aroles(au) \land r \in \{\textbf{r}_4, \textbf{r}_5\} \land ((\exists x \geq \textbf{x}_1).\ x \in org\_units(p) \land (\exists x \geq \textbf{x}_2).$
$x \in org\_units(p))) \lor ((\exists ar \geq \textbf{ar}_1).\ ar \in aroles(au) \land r \in \{\textbf{r}_6\} \land ((\exists x \geq \textbf{x}_1).\ x \in org\_units(p)$
$\lor (\exists x \geq \textbf{x}_2).\ x \notin org\_units(p) \land (\exists x \geq \textbf{x}_3).\ x \in org\_units(p)))$

Construct authorization function for revoking permission from role:

Case 1:

For any pemrission $p \in$ PERMS undertaken for revocation,

– is_authorized$\text{P}_{\textbf{revoke}}(au : \text{AU}, p : \text{PERMS}, r : \text{ROLES}) \equiv (\exists ar \geq \textbf{ar}_1).\ ar \in aroles(u) \land$
$r \in \{\textbf{r}_1, \textbf{r}_3, \textbf{r}_4\}$

Case 2:

For any pemrission $p \in$ PERMS undertaken for revocation,

– is_authorized$\text{P}_{\textbf{revoke}}(au : \text{AU}, p : \text{PERMS}, r : \text{ROLES}) \equiv (\exists ar \geq \textbf{ar}_1).\ ar \in aroles(p) \land$
$r \in \{\textbf{r}_1, \textbf{r}_3, \textbf{r}_4\}$

The manual translation shows that it is possible to convert an instance of PRA02 model into an instance of ARPA model. It primarily takes few steps for translation. First, it maps all the sets from PRA02 model into sets of ARPA model. In particular, sets for admin users, administrative operations, roles and their hierarchy and, permissions are mapped. There are two administrative operations **assign** and **revoke**. Second, the relations and functions are intuitively engineered using attributes. The only admin role attribute *aroles* yields admin roles assigned to admin user. For example, $aroles(\textbf{u}_3) = \{\textbf{ar}_1\}$ specifies that admin user $\textbf{u}_3$ has admin role $\textbf{ar}_1$. There are two permission attributes *org_units* and *rolesp*. Permission attribute *org_units* gives back all the organization units that a permission is associated with. For example, $org\_units(\textbf{p}_1) = \{\textbf{x}_1, \textbf{x}_3\}$ specifies that $\textbf{p}_1$ is mapped to organization units, $\textbf{x}_1$ and $\textbf{x}_3$. Attribute *rolesp* yields roles that a permission is assigned

Algorithm 4.8 $\text{Map}_{\text{PRA02}}$

```
Input: PRA02 instance
Output: AURA instance
Step 1:    /* Map basic sets and functions in ARPA */
    a. $AU^{A} \leftarrow AU^{02}$ ; $AOP^{A} \leftarrow \{\textbf{assign, revoke}\}$ ; $ROLES^{A} \leftarrow ROLES^{02}$
    b. $RH^{A} \leftarrow RH^{02}$ ; $PERMS^{A} \leftarrow PERMS^{02}$
Step 2:    /* Map attribute functions to ARPA */
    a. $AATT^{A} \leftarrow \{aroles\}$
    $Scope^{A}(aroles) = AR^{02}$ ; $attType^{A}(aroles) = set$
    b. $is\_ordered^{A}(aroles) = True$ ; $H^{A}_{aroles} \leftarrow ARH^{02}$
    c. For each $u \in AU^{A}$, $aroles(u) = \phi$
    d. For each $(u, ar)$ in $AUA^{02}$, $aroles(u)' = aroles(u) \cup ar$
    e. $PATT^{A} \leftarrow \{org\_units, rolesp\}$
    f. $Scope^{A}(org\_units) = ORGU^{02}$ ; $attType^{A}(org\_units) = set$
    g. $is\_ordered^{A}(org\_units) = True$ ; $H^{A}_{org\_units} = OUH^{02}$
    h. For each $p \in PERMS^{A}$, $org\_units(p) = \phi$
    i. For each $(p, orgu) \in PPA^{02}$, $org\_units(p)' = org\_units(p) \cup orgu$
    j. $Scope^{A}(rolesp) = ROLES^{A}$ ; $attType^{A}(rolesp) = set$
    k. $is\_ordered^{A}(rolesp) = True$ ; $H^{A}_{rolesp} = RH^{A}$
    l. For each $p \in PERMS^{A}$, $rolesp(p) = \phi$
    m. For each $(p, r) \in PPA^{02}$, $rolesp(p)' = rolesp(p) \cup r$
Step 3:    /* Construct assign rule in ARPA */
    a. $assign\_formula = \phi$
    b. For each $(ar, cr, Z) \in can\_assign^{02}$,
        $assign\_formula' = assign\_formula \vee ((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z \wedge$
            $(translatep_{02}(cr)))$
    c. $auth\_assign = is\_authorized_{\textbf{assign}}(au : AU, p : PERMS, r : ROLES) \equiv assign\_formula'$
Step 4:    /* Construct revoke rule in ARPA */
    a. $revoke\_formula = \phi$
    b. For each $(ar, cr, Z) \in can\_revoke^{02}$,
        $revoke\_formula' = revoke\_formula \vee ((\exists ar' \geq ar).\ ar' \in aroles(au) \wedge r \in Z)$
    c. $auth\_revoke = is\_authorized_{\textbf{revoke}}(au : AU, p : PERMS, r : ROLES) \equiv revoke\_formula'$
```

Support routine for algorithm 4.8 *translatep*$_{02}$

Input: A PRA02 prerequisite condition (*cr*), Case 1, Case 2

Output: An equivalent sub-rule for ARPA authorization rule. STATE *rule_string* = ϕ

```
1:  Case Of selection
2:      ' Case 1 ' (cr is based on roles) :
3:          translatep97
4:      ' Case 2 ' (cr is based on org_units):
5:      For each symbol in cr
6:          if symbol is an organization unit and in the form x
                        (i.e., the permission has a membership on organization unit x)
7:              rule_string = rule_string + $(\exists x' \geq x).\ x' \in org\_units(p)$
8:          else if symbol an organization unit and in the form $\bar{x}$
                        (i.e., the permission doesn't have a membership on organization unit x)

9:              rule_string = rule_string + $(\exists x' \geq x).\ x' \notin org\_units(p)$
10:         else
11:             rule_string = rule_string + symbol  /* where a symbol is a $\wedge$ or $\vee$ logical operator
    */
12:         end if
13: end Case
```

to. For example, *rolesp*($\mathbf{p_1}$) = {$\mathbf{r_1, r_2}$} indicates that $\mathbf{p_1}$ is assigned to roles, $\mathbf{r_1}$ and $\mathbf{r_2}$. Range, type, order and hierarchy for values of each attribute are mapped. In step three, authorization functions that intuitively translate the meaning of *can_assignp* for two different cases, as per PRA02 are established. In the final step, authorization functions equivalent to *can_revokep* in PRA02 for two different cases are constructed using attributes.

4.5.4 Map$_{\text{PRA02}}$

Algorithm 4.8 presents Map$_{\text{PRA02}}$. It maps a PRA02 instance into an equivalent ARPA instance. For brevity, sets and functions from PRA02 and ARPA are marked with superscripts `02` and `A`, respectively. Map$_{\text{PRA02}}$ takes PRA02 instance as its input. In particular, input for Map$_{\text{PRA02}}$ fundamentally has USERS^{02}, ROLES^{02}, AR^{02}, PERMS^{02}, AUA^{02}, PA^{02}, RH^{02}, ARH^{02}, *can_assignp*^{02}, *can_revokep*^{02}, ORGU^{02}, OUH^{02}, and PPA^{02}

Output from Map$_{\text{PRA02}}$ algorithm is an equivalent ARPA instance, with primarily consisting of

$\mathrm{AU}^{\mathbb{A}}$, $\mathrm{AOP}^{\mathbb{A}}$, $\mathrm{ROLES}^{\mathbb{A}}$, $\mathrm{RH}^{\mathbb{A}}$, $\mathrm{PERMS}^{\mathbb{A}}$, $\mathrm{AATT}^{\mathbb{A}}$, $\mathrm{PATT}^{\mathbb{A}}$, For each attribute $att \in \mathrm{AATT}^{\mathbb{A}} \cup \mathrm{PATT}^{\mathbb{A}}$, $\mathrm{Range}^{\mathbb{A}}(att)$, $\mathrm{attType}^{\mathbb{A}}(att)$, $\mathrm{is\_ordered}^{\mathbb{A}}(att)$ and $\mathrm{H}^{\mathbb{A}}{}_{att}$, For each user $p \in \mathrm{PERMS}^{\mathbb{A}}$ and for each $att \in \mathrm{PATT}^{\mathbb{A}}$, $att(p)$, For each user $u \in \mathrm{AU}^{\mathbb{A}}$ and for each $att \in \mathrm{AATT}^{\mathbb{A}}$, $att(u)$, Authorization rule to assign (auth_assignp), and Authorization rule to revoke (auth_revokep).

As shown in Algorithm 4.8, there are four main steps required in mapping any instance of PRA02 model to ARPA instance. In Step 1, sets and functions from PRA02 instance are mapped into ARPA sets and functions. In Step 2, permission attributes and administrative user attribute functions are expressed. PATT set has two permission attributes, *org_units* and *rolesp*. *org_units* attribute captures a permission's association in an organization unit and, *rolesp* captures roles to which permission have been assigned to. There are two options for user-role assignment in PRA02. They are marked as Case 1 and Case 2 in the model. Case 1 checks for permission's existing membership on roles while Case 2 checks for user's membership on organization units. *org_units* is captured in Case 2. Case 1 is same as PRA97 in Section 4.4. Admin user attribute *aroles* captures admin roles assigned to admin users. Step 3 involves constructing assignp_formula in ARPA that is equivalent to $can\_assignp^{02}$ in PRA02. $can\_assignp^{02}$ is a set of triples. Each triple bears information on whether an admin role can assign a candidate permission to a set of roles. Equivalent translation in ARPA for PRA02 is given by $\mathrm{is\_authorizedU}_{\mathbf{assign}}(au : \mathrm{AU}^{\mathbb{A}}, u : \mathrm{USERS}^{\mathbb{A}}, r : \mathrm{ROLES}^{\mathbb{A}})$. Similarly, In Step 4, revoke_formula equivalent to $can\_revokep^{02}$ given by $\mathrm{is\_authorizedU}_{\mathbf{revoke}}(au : \mathrm{AU}^{\mathbb{A}}, u : \mathrm{USERS}^{\mathbb{A}}, r : \mathrm{ROLES}^{\mathbb{A}})$ is presented. A support routine $translate_{02}$ translates prerequisite condition in PRA02 into its equivalent in ARPA. A complete example instance and its corresponding equivalent ARPA instances were presented in Section 4.5.2 and Section 4.5.3, respectively.

4.6 Uni-ARBAC's PRA in ARPA

This seciton shows that it is possible to represent Uni-ARBAC's URA in ARPA. Firstly, it is shown by a manual translation process of an example instance of Uni-ARBAC's PRA into its equivalent instance in ARPA model. Then to prove that any instance of Uni-ARBAC's PRA can be represented in ARPA, a formal algorithm for mapping Uni-ARBAC's PRA to ARPA equivalent instance is presented. A summary of Uni-ARBAC's PRA model is given as follows:

4.6.1 Summary of Uni-ARBAC's PRA Model

Traditional RBAC Sets & Relations

– USERS is a finite set of regular users.

– ROLES is a finite set of regular roles.

– RH $\subseteq$ ROLES $\times$ ROLES, partial order hierarchy on roles.

– PERMS is a finite set of permissions.

Additional RBAC Sets & Relations

– T is a finite set of tasks.

– TH $\subseteq$ T $\times$ T is a partial order hierarchy on tasks.

– PA $\subseteq$ PERMS $\times$ T permission task assignment relation.

– TA $\subseteq$ T $\times$ ROLES, task-role assignment relation.

Derived functions

authorized_perms(r : ROLES) $\rightarrow 2^{\text{PERMS}}$

Administrative Units and Partitioned Assignments

– AU is a finite set of administrative units.

– *roles*(au : AU) $\rightarrow 2^{\text{ROLES}}$, assignment of roles.

- *tasks*(au : AU) $\rightarrow 2^{\text{T}}$, assignment of tasks

Derived functions

– *tasks**(au : AU) $\rightarrow 2^{\text{T}}$, defined as *tasks**(au) = $\{t' \mid (\exists t \in tasks(au)) \wedge \succeq_{\text{t}} t'\}$

Administrative user assignment:

– *TA_admin* $\subseteq$ USERS $\times$ AU

– AUH $\in$ AU$\times$ AU rooted tree partial order $\succeq_{au}$

Authorization function:

– *can_manage_task_role*(u_1 : USERS, t : T, r : ROLES)= $(\exists au_i, au_j)[(u_1, au_i) \in TA\_admin \wedge au_i$

$\succeq_{au}$ $au_j \wedge r \in roles(au_j) \wedge t \in tasks^*(au_j))]$

4.6.2 Instance of PRA in Uni-ARBAC

In this section an example instance for PRA in UARBAC (PRA-U) model is presented.

Traditional RBAC Sets & Relations:

- USERS = {$\mathbf{u_1, u_2, u_3, u_4}$}
- ROLES = {$\mathbf{r_1, r_2, r_3, r_4}$}
- PERMS = {$\mathbf{p_1, p_2, p_3, p_4}$}
- RH = {<$\mathbf{r_1, r_2}$>, <$\mathbf{r_2, r_3}$>}

Additional RBAC Sets & Relations:

- T = {$\mathbf{t_1, t_2, t_3, t_4}$}
- TH = {<$\mathbf{t_1, t_2}$>, <$\mathbf{t_2, t_3}$>, <$\mathbf{t_2, t_4}$>}
- PA = {($\mathbf{p_1, t_1}$), ($\mathbf{p_1, t_4}$), ($\mathbf{p_2, t_4}$), ($\mathbf{p_4, t_3}$), ($\mathbf{p_3, t_2}$)}
- TA = {($\mathbf{t_1, r_2}$), ($\mathbf{t_2, r_1}$), ($\mathbf{t_3, r_4}$), ($\mathbf{t_4, r_3}$)}

Derived functions

- authorized_perms($\mathbf{r_1}$) = {$\mathbf{p_3}$}
- authorized_perms($\mathbf{r_2}$) = {$\mathbf{p_1}$}
- authorized_perms($\mathbf{r_3}$) = {$\mathbf{p_1, p_2}$}
- authorized_perms($\mathbf{r_4}$) = {$\mathbf{p_4}$}

Administrative Units and Partitioned Assignments

- AU = {$\mathbf{au_1, au_2}$}

- *roles*($\mathbf{au_1}$) = {$\mathbf{r_1, r_2}$}, *roles*($\mathbf{au_2}$) = {$\mathbf{r_3}$}

- *tasks*($\mathbf{au_1}$) = {$\mathbf{t_1, t_2}$}, *tasks*($\mathbf{au_2}$) = {$\mathbf{t_3, t_4}$}

Derived Function

- *tasks**($\mathbf{au_1}$) = {$\mathbf{t_1, t_2, t_3, t_4}$}

- *tasks**($\mathbf{au_2}$) = {$\mathbf{t_3, t_4}$}

Administrative User Assignments

- *TA_admin* = {($\mathbf{u_1, au_1}$), ($\mathbf{u_2, au_2}$)}

- AUH = {<$\mathbf{au_1, au_2}$>}

Task-role assignment condition in uni-ARBAC:

– *can_manage_task_role*(*u* : USERS, *t*: T, *r*: ROLES) =

$(\exists au_i, au_j)[(u, au_i) \in TA\_admin \wedge au_i \succeq_{au} au_j \wedge r \in roles(au_j) \wedge t \in tasks^*(au_j)]$

4.6.3 Equivalent ARPA instance of PRA in Uni-ARBAC

This section presents an equivalent instance of ARPA for the example instance presented above in section 4.6.2.

Map set and functions:

- AU = {$\mathbf{u_1, u_2, u_3, u_4}$}

- AOP = {**assign, revoke**}

- ROLES = {$\mathbf{r_1, r_2, r_3, r_4}$}

- RH = {<$\mathbf{r_1, r_2}$>, <$\mathbf{r_2, r_3}$>, <$\mathbf{r_3, r_4}$>}

- PERMS = {$\mathbf{p_1, p_2, p_3, p_4}$}

Define attributes and values:

- AATT = {*admin_unit, adminunit_role*}
- Scope(*admin_unit*) = {$\mathbf{au_1}$, $\mathbf{au_2}$}, attType(*admin_unit*) = set,

  is_ordered(*admin_unit*) = True, H_{admin_unit} = {<$\mathbf{au_1}$, $\mathbf{au_2}$>}
- *admin_unit*($\mathbf{u_1}$) = {$\mathbf{au_1}$}, *admin_unit*($\mathbf{u_2}$) = {$\mathbf{au_2}$},

  admin_unit($\mathbf{u_3}$) = { }, *admin_unit*($\mathbf{u_4}$) = { }
- Scope(*adminunit_role*) = {($\mathbf{au_1}$, $\mathbf{r_1}$), ($\mathbf{au_1}$, $\mathbf{r_2}$), ($\mathbf{au_2}$, $\mathbf{r_3}$)}, attType(*adminunit_role*) = set,

  is_ordered(*adminunit_role*) = False, $H_{adminunit\_role} = \phi$
- *adminunit_role*($\mathbf{u_1}$) = {($\mathbf{au_1}$, $\mathbf{r_1}$), ($\mathbf{au_1}$, $\mathbf{r_2}$), ($\mathbf{au_2}$, $\mathbf{r_3}$)}, *adminunit_role*($\mathbf{u_2}$) = {($\mathbf{au_2}$, $\mathbf{r_3}$)},

  adminunit_role($\mathbf{u_3}$) = { }, *adminunit_role*($\mathbf{u_4}$) = { }
- PATT = {*tasks, task_adminu*}
- Scope(*tasks*) = {$\mathbf{t_1}$, $\mathbf{t_2}$, $\mathbf{t_3}$, $\mathbf{t_4}$}, attType(*tasks*) = set, is_ordered(*tasks*) = True, H_{tasks} = TH
- *tasks*($\mathbf{p_1}$) = {$\mathbf{t_1}$, $\mathbf{t_2}$, $\mathbf{t_4}$}, *tasks*($\mathbf{p_2}$) = {$\mathbf{t_1}$, $\mathbf{t_2}$, $\mathbf{t_4}$}, *tasks*($\mathbf{p_3}$) = {$\mathbf{t_1}$, $\mathbf{t_2}$}, *tasks*($\mathbf{p_4}$) = {$\mathbf{t_1}$, $\mathbf{t_2}$, $\mathbf{t_3}$}
- Scope(*task_adminu*) = {($\mathbf{t_1}$, $\mathbf{au_1}$), ($\mathbf{t_2}$, $\mathbf{au_2}$)}, attType(*task_adminu*) = set,

  is_ordered(*task_adminu*) = False, $H_{task\_adminu} = \phi$
- *task_adminu*($\mathbf{p_1}$) = {($\mathbf{t_1}$, $\mathbf{au_1}$), ($\mathbf{t_4}$, $\mathbf{au_2}$)}, it*task_adminu*($\mathbf{p_2}$) = {($\mathbf{t_4}$, $\mathbf{au_2}$)},

  task_adminu($\mathbf{p_3}$) = {($\mathbf{t_2}$, $\mathbf{au_1}$)}, *task_adminu*($\mathbf{p_4}$) = {($\mathbf{t_3}$, $\mathbf{au_2}$)}

Set of permissions that are mapped to each task in T can be expressed as follows:

Let each set be represented with χ_i as shown below.

- $\chi_1 = \{p \mid \mathbf{t_1} \in tasks(p)\}$
- $\chi_2 = \{p \mid \mathbf{t_2} \in tasks(p)\}$

- $\chi_3 = \{p \mid \mathbf{t_3} \in tasks(p)\}$
- $\chi_4 = \{p \mid \mathbf{t_4} \in tasks(p)\}$

<u>Construct authorization function for permission-role assignment:</u>

For each χ_i in $\{\chi_1, \chi_2, \chi_3, \chi_4\}$, authorization rule for whether an admin user in AU is authorized to assign χ_i to a roles r in ROLES is given below:

– is_authorizedP$_{\mathbf{assign}}$(u : USERS, χ_i : 2^{PERMS}, r : ROLES) $\equiv$

$\exists au_1, au_2 \in \text{Scope}(admin\_unit). <au_1, au_2> \in \text{H}_{admin\_unit} \wedge (au_1 \in admin\_unit(u) \wedge$
$(au_2, r) \in adminunit\_role(u)) \wedge \exists t_1, t_2 \in \text{Scope}(tasks). [<t_1, t_2> \in \text{TH} \wedge \forall q \in \chi.\ t_2 \in tasks(q)$
$\wedge\ \exists q' \in (\text{PERMS} - \chi).\ t_2 \notin tasks(q') \wedge (t_2, au_2) \in tasks\_adminu(q)]$

<u>Construct authorization function for revoking permission from role:</u>

For each χ_i in $\{\chi_1, \chi_2, \chi_3, \chi_4\}$, authorization function for whether an admin user in AU is authorized to revoke χ from a roles $r \in$ ROLES is given below:

– is_authorizedP$_{\mathbf{revoke}}$(u : USERS, χ_i : 2^{PERMS}, r : ROLES) $\equiv$
is_authorizedP$_{\mathbf{assign}}$(u : USERS, χ_i : 2^{PERMS}, r : ROLES)

This manual translation shows that there are four steps involved in mapping Uni-ARBAC's PRA instance to ARPA model instance. Firstly, the sets from Uni-ARBAC's PRA instance is mapped to ARPA instance. In particular, sets for admin users, administrative operations, roles and their hierarchy and, permissions are mapped in ARPA. Then the relations and appropriate functions that capture the features from Uni-ARBAC's PRA are intuitively engineered as attributes with their corresponding values in the range. There are two admin user attributes *admin_unit* and *adminunit_role*. *admin_unit* captures an admin user's task assignment authority over admin units. For example, *admin_unit*($\mathbf{u_1}$) = $\{\mathbf{au_1}\}$ specifies that admin user $\mathbf{u_1}$ has task assignment authority in admin unit $\mathbf{au_1}$. *adminunit_role* yields admin unit to which admin user has task-role assignment authority and all the roles that have been associated with those admin units. For example, *adminunit_role*($\mathbf{u_2}$) = $\{(\mathbf{au_2}, \mathbf{r_3})\}$ specifies $\mathbf{u_2}$ has admin authority over admin unit $\mathbf{au_2}$ and $\mathbf{r_3}$ is mapped to admin unit, $\mathbf{au_2}$. There are two permission attributes defined, *tasks* and *task_adminu*. Attribute *tasks* yields all the tasks that a permission is grouped to. For example, *tasks*($\mathbf{p_1}$) = $\{\mathbf{t_1, t_2, t_4}\}$ means permission $\mathbf{p_1}$ is in tasks $\mathbf{t_1, t_2, t_4}$. Attribute *task_adminu*(p) yields all the tasks that a permission p belongs to and all those admin units to which those tasks (with permission p) is associated with. For example, *task_adminu*($\mathbf{p_3}$) = $\{(\mathbf{t_2, au_1})\}$ indicates that permission $\mathbf{p_3}$ belongs to task $\mathbf{t_2}$ and task $\mathbf{t_2}$ is mapped to admin unit $\mathbf{au_1}$. For all the attributes, their range, type, order and hierarchies are mapped. Since, Uni-ARBAC's PRA involves task assignment, which are group of permissions, it needs to be translated into ARPA equivalent. Fortunately, ARPA has a mechanism for group of permission to a role assignment function. Therefore, permissions are put together as a set builder notion based on permission attribute represented by χ_i.

In third step, authorization functions that translate task-role assignment is established using defined attributes and values. In step four, attributes are used to construct task-role revocation authorization functions.

Algorithm 4.9 Map$_{\text{PRA-Uni-ARBAC}}$

```
Input: Instance of PRA in Uni-ARBAC
Output: AURA instance
Step 1:   /* Map basic sets and functions in ARPA */
    a. $AU^{A} \leftarrow USERS^{Uni}$ ; $AOP^{A} \leftarrow \{\textbf{assign, revoke}\}$ ; $ROLES^{A} \leftarrow ROLES^{Uni}$
    b. $RH^{A} \leftarrow RH^{Uni}$ ; $PERMS^{A} \leftarrow PERMS^{Uni}$
Step 2:   /* Map attribute functions in ARPA */
    a. $AATT^{A} \leftarrow \{admin\_unit, adminunit\_role\}$
    b. $Scope^{A}(admin\_unit) = AU^{Uni}$ ; $attType^{A}(admin\_unit) = set$
    c. $is\_ordered^{A}(admin\_unit) = True, H^{A}_{admin\_unit} = AUH^{Uni}$
    d. For each $u$ in $AU^{A}$, $admin\_unit(u) = \phi$
    e. For each $(u, au) \in TA\_admin^{Uni}$, $admin\_unit(u)' = admin\_unit(u) \cup au$
    f. $Scope^{A}(adminunit\_role) = AU^{Uni} \times ROLES^{Uni}$ ; $attType^{A}(adminunit\_role) = set$
    g. $is\_ordered^{A}(adminunit\_role) = False, H_{adminunit\_role} = \phi$
    h. For each $u$ in $AU^{A}$, $adminunit\_role(u) = \phi$
    i. For each $(u, au) \in TA\_admin^{Uni}$ and for each $r \in assigned\_roles^{Uni}(au)$,
            $adminunit\_role(u)' = adminunit\_role(u) \cup (au, r)$
    j. $PATT^{A} \leftarrow \{tasks, task\_adminu\}$
    k. $Scope^{A}(tasks) = T^{Uni}$ ; $attType^{A}(tasks) = set$
    l. $is\_ordered^{A}(tasks) = True$ ; $H^{A}_{tasks} = TH^{Uni}$
    m. For each $p$ in $PERMS^{A}$, $tasks(p) = \phi$ ;
    n. For each $(p, t) \in PA^{Uni}$, $tasks(p)' = tasks(p) \cup t$
    o. $Scope^{A}(task\_adminu) = T^{Uni} \times AU^{Uni}$ ; $attType^{A}(task\_adminu) = set$ ;
    p. $is\_ordered^{A}(task\_adminu) = False$ ; $H^{A}_{task\_adminu} = \phi$
    q. For each $p$ in $PERMS^{A}$, $task\_adminu(p) = \phi$
    r. For each $(p, t) \in PA^{Uni}$ and for each $t \in tasks*^{Uni}(au)$,
        $task\_adminu(p)' = task\_adminu(p) \cup (t, au)$
Step 3:   /* Construct assign rule in ARPA */
    a. can_manage_rule = $\exists au_1, au_2 \in Range(admin\_unit).\ (au_1, au_2) \in H_{admin\_unit} \wedge$
                         $(au_1 \in admin\_unit(u) \wedge (au_2, r) \in adminunit\_role(u)) \wedge$
                         $\exists t_1, t_2 \in Scope(tasks).\ [(t_1, t_2) \in TH \wedge \forall q \in \chi.\ t_2 \in tasks(q) \wedge$
                         $\exists q' \in (PERMS - \chi).\ t_2 \notin tasks(q') \wedge (t_2, au_2) \in tasks\_adminu(q)]$
    b. auth_assign = $is\_authorizedP_{\textbf{assign}}(u : AU^{A}, \chi : 2^{PERMS^{A}}, r : ROLES^{A}) \equiv$ can_manage_rule
Step 4:   /* Construct revoke rule for ARPA */
    a. auth_revoke = $is\_authorizedP_{\textbf{revoke}}(u : AU^{A}, \chi : 2^{PERMS^{A}}, r : ROLES^{A}) \equiv$ can_manage_rule
```

4.6.4 Map$_{\textbf{PRA-Uni-ARBAC}}$

Algorithm 4.9 presents Map$_{\text{PRA-Uni}}$. It translates any instance of Uni-ARBAC's PRA to an equivalent instance of ARPA. For brevity, sets and functions from Uni-ARBAC's PRA and ARPA are labeled as `Uni` and superscript `A`, respectively.

In Step 1, sets and functions from PRA-Uni instance are mapped to ARPA instance. In Uni-ARBAC, both admin users and regular users belong to same set, USERS$^{\text{Uni}}$. Thus, USERS$^{\text{Uni}}$ is mapped to AU$^{\text{A}}$. In Step 2, admin user attributes and permission attributes are defined. There are two admin user attributes: *admin_unit* and *adminunit_role*. *admin_unit* captures the *TA_admin*$^{\text{Uni}}$ relation in Uni-ARBAC's URA, and *adminunit_role* captures admin user's mapping with admin unit, and the roles mapped to that admin unit. There are two permission attributes defined: *tasks* and *task_adminu*. Attribute *tasks* gives a mapping between permission and tasks. That is for each permission *p*, *tasks*(*p*) yields set of tasks it is mapped to. For a given permission, attribute *task_adminu* gives its mapping with tasks, and admin units that each task is mapped to. Step 3 constructs an assignment rule equivalent to *can_manage_task_role*(u : USERS$^{\texttt{Uni}}$, t: T$^{\texttt{Uni}}$, r: ROLES$^{\texttt{Uni}}$) in Uni-ARBAC. In PRA-Uni, it evaluates if an admin user *u* can assign/revoke a task *t* to/from a role *r* if admin user *u* has *Task_Admin* relation with some admin unit *au* to which task *t* and role *r* are mapped. An equivalent assignment rule in ARPA is expressed in Step 3 as is_authorizedP$_{\textbf{assign}}$(u, χ, r). Finally, auth_revoke, which represents an equivalent revoke function to revoke a task (or set of permissions) from a role for *can_manage_task_role*(u : USERS$^{\texttt{Uni}}$, t: T$^{\texttt{Uni}}$, r: ROLES$^{\texttt{Uni}}$) in Uni-ARBAC is expressed in Step 4 as is_authorizedP$_{\textbf{revoke}}$(u, χ, r). Authorization criteria for **assign** and **revoke** is identical in Uni-ARBAC's PRA.

4.7 UARBAC's PRA in ARPA

This section shows that UARBAC's PRA model can be mapped to ARPA model. To illustrate this in detail, first, a manual translation of UARBAC's PRA example instance into its equivalent ARPA instance is presented. Next, a formal algorithm for mapping any instance of UARBAC's PRA into equivalent ARPA instance is exhibited. A brief summary of UARBAC's PRA model is re-visited as follows:

4.7.1 Summary of UARBAC's PRA Model

RBAC Model

UARBAC model is designed with a notion of class objects. Thus, in addition to object level permissions, it also includes class level administrative permissions. RBAC schema is for this RBAC model is as follows:

RBAC Schema:

RBAC Schemas is given by following tuple.

<C, OBJS, AM>

- *C* is a finite set of object classes with predefined classes: user and role.
- *OBJS*(c) is a function that gives all possible names for objects of the class $c \in C$.

  Let **USERS** = *OBJS*(user) and **ROLES** = *OBJS*(role)
- *AM*(c) is function that maps class c to a set of access modes that can be applied on objects of class c.

Access modes for two predefined classes user and role are fixed. By observation we find it relevant to consider files as resource objects. We take file as example resource object to which we will define access.

AM(user) = {empower, admin}

AM(role) = {grant, empower, admin}

AM(file) = {read, write, append, execute, admin}

RBAC Permissions:

There are two kinds of permissions in this RBAC model:

- Object permissions of the form,

$[c, o, a]$, where $c \in C$, $o \in OBJS(c)$, $a \in AM(c)$.

- Class permissions of the form,

  $[c, a]$, where, $c \in C$, and $a \in \{\textsf{create}\} \cup AM(c)$.

<u>RBAC State:</u>

Given an RBAC Schema, an RBAC state is given by,

$<OB, UA, PA, RH>$

- OB is a function that maps each class in C to a finite set of object names of that class that currently exists, i.e., $OB(c) \subseteq OBJS(c)$.

  Let $OB(\textsf{user}) = USERS$, $OB(\textsf{role}) = ROLES$ and, let $OB(\textsf{file}) = FILES$

  Set of permissions, P is given by,

  $P = \{[c, o, a] \mid c \in C \wedge o \in OBJS(c) \wedge a \in AM(c)\} \cup \{[c, a] \mid c \in C \wedge a \in \{\textsf{create}\} \cup AM(c)\}$

- $UA \subseteq USERS \times ROLES$, user-role assignment relation.

- $PA \subseteq P \times ROLES$, permission-role assignment relation.

- $RH \subseteq ROLES \times ROLES$, partial order in $ROLES$ denoted by $\succeq_{RH}$.

<u>Administrative permissions in UARBAC:</u>

All the permissions of user u who performs administrative operations can be calculated as follows:

- authorized_perms$[u] = \{p \in P \mid \exists r_1, r_2 \in \mathrm{R}\ [(u, r_1) \in UA \wedge (r_1 \succeq_{RH} r_2) \wedge (r_2, p) \in PA]\}$

<u>Permission-Role Administration</u>

Operations required to assign object permission $[c, o_1, a_1]$ to role r_1 and to revoke object permission $[c, o_1, a_1]$ from role r_1 are respectively listed below:

- grantObjPermToRole($[c, o_1, a_1]$, r_1)

- revokeObjPermFromRole([c, o_1, a_1], r_1)

A user at requires one of the following two permissions to conduct grantObjPermToRole([c, o_1, a_1], r_1) operation.

- [c, o_1, admin] and [role, r_1, empower] or,
- [c, o_1, admin] and [role, empower] or,
- [c, admin] and [role, r_1, empower] or,
- [c, admin] and [role, empower]

A user at least requires one of the following (two) options to conduct revokeObjPermFromRole([c, o_1, a_1], r_1) operation.

- [c, o_1, admin] and [role, r_1, empower] or,
- [c, o_1, admin] or,
- [role, r_1, admin] or,
- [c, admin] or,
- [role, admin]

4.7.2 Instance of UARBAC's PRA

RBAC Schema

Let us consider objects to which users need access via roles to be of file object class.

- C = {user, role, file}
- $OBJS$(user) = USERS,
- $OBJS$(role) = ROLES
- $OBJS$(file) = FILES

Access modes for role class and file class are as follows:

- *AM*(role) = {grant, empower, admin}
- *AM*(file) = {read, write, append, execute, admin}

RBAC State

- *USERS* = *OBJ*(user)= {$\mathbf{u_1, u_2, u_3, u_4}$}
- *ROLES* = *OBJ*(role)= {$\mathbf{r_1, r_2, r_3, r_4}$}
- *O* = *OBJ*(file)= {$\mathbf{o_1, o_2, o_3}$}
- *P* = {[role, $\mathbf{r_1}$, grant], [role, $\mathbf{r_1}$, empower], [role, $\mathbf{r_1}$, admin], [role, $\mathbf{r_2}$, grant], [role, $\mathbf{r_2}$, empower], [role, $\mathbf{r_2}$, admin], [role, $\mathbf{r_3}$, grant], [role, $\mathbf{r_3}$, empower], [role, $\mathbf{r_3}$, admin], [role, $\mathbf{r_4}$, grant], [role, $\mathbf{r_4}$, empower], [role, $\mathbf{r_4}$, admin], [file, $\mathbf{o_1}$, read], [file, $\mathbf{o_1}$, write], [file, $\mathbf{o_1}$, append], [file, $\mathbf{o_1}$, execute], [file, $\mathbf{o_1}$, admin], [file, $\mathbf{o_2}$, read], [file, $\mathbf{o_2}$, write], [file, $\mathbf{o_2}$, append], [file, $\mathbf{o_2}$, execute], [file, $\mathbf{o_2}$, admin], [file, $\mathbf{o_3}$, read], [file, $\mathbf{o_3}$, write], [file, $\mathbf{o_3}$, append], [file, $\mathbf{o_3}$, execute], [file, $\mathbf{o_3}$, admin], [file, read], [file, write], [file, append], [file, execute], [file, admin]}
- *PA* = {([file, $\mathbf{o_1}$, read], $\mathbf{r_1}$), ([file, $\mathbf{o_2}$, execute], $\mathbf{r_1}$), ([file, $\mathbf{o_2}$, execute], $\mathbf{r_2}$), ([file, $\mathbf{o_3}$, admin], $\mathbf{r_3}$), ([file, $\mathbf{o_1}$, read], $\mathbf{r_3}$), ([file, $\mathbf{o_3}$, write], $\mathbf{r_2}$)}
- *RH* = {<$\mathbf{r_1, r_2}$>, <$\mathbf{r_2, r_3}$>, <$\mathbf{r_3, r_4}$>}

Authorized permissions in UARBAC

Following is the list of authorized permissions each user has that includes administrative permissions for permission-role assignment:

- authorized_perms[$\mathbf{u_1}$] = {[file, $\mathbf{o_1}$, read], [role, $\mathbf{r_1}$, grant], [file, $\mathbf{o_1}$, write], [role, $\mathbf{r_3}$, grant], [file, $\mathbf{o_2}$, admin], [file, $\mathbf{o_3}$, append],[role, $\mathbf{r_2}$, grant], [file, $\mathbf{o_3}$, admin], [role, $\mathbf{r_1}$, admin], [role, $\mathbf{r_4}$, admin]}

- authorized_perms[$\mathbf{u_2}$] = {[file, $\mathbf{o_1}$, append], [role, $\mathbf{r_1}$, grant], [file, $\mathbf{o_2}$, admin], [role, $\mathbf{r_2}$, grant]}
- authorized_perms[$\mathbf{u_3}$] = {[file, admin]}
- authorized_perms[$\mathbf{u_4}$] = {}

Permission-Role assignment condition in PRA-UARBAC:

One can perform following operation to assign a permission [file, o_1, a] to a role r_1.

- grantObjPermToRole([file, o_1, a], r_1)

To perform aforementioned operation one needs the following two permissions:

- [file, o_1, admin] and [role, r_1, empower]

Condition for revoking permission-role in PRA-UARBAC:

One can perform following operation to revoke a user [file, o_1, a] to a role r_1.

- revokeObjPermFromRole([file, o_1, a], r_1)

To perform aforementioned operation one needs one of the following permissions:

- [file, o_1, admin] **or**,
- [role, r_1, admin]

4.7.3 Equivalent ARPA instance of UARBAC's PRA

Map sets and functions:

- AU = {$\mathbf{u_1, u_2, u_3, u_4}$}
- AOP = {assignp, revokep}
- ROLES = {$\mathbf{r_1, r_2, r_3, r_4}$}
- RH = {<$\mathbf{r_1, r_2}$>, <$\mathbf{r_2, r_3}$>, <$\mathbf{r_3, r_4}$>}

- PERMS = {[role, $\mathbf{r_1}$, grant], [role, $\mathbf{r_1}$, empower], [role, $\mathbf{r_1}$, admin], [role, $\mathbf{r_2}$, grant], [role, $\mathbf{r_2}$, empower], [role, $\mathbf{r_2}$, admin], [role, $\mathbf{r_3}$, grant], [role, $\mathbf{r_3}$, empower], [role, $\mathbf{r_3}$, admin], [role, $\mathbf{r_4}$, grant], [role, $\mathbf{r_4}$, empower], [role, $\mathbf{r_4}$, admin], [file, $\mathbf{o_1}$, read], [file, $\mathbf{o_1}$, write], [file, $\mathbf{o_1}$, append], [file, $\mathbf{o_1}$, execute], [file, $\mathbf{o_1}$, admin]

  [file, $\mathbf{o_2}$, read], [file, $\mathbf{o_2}$, write], [file, $\mathbf{o_2}$, append], [file, $\mathbf{o_2}$, execute], [file, $\mathbf{o_2}$, admin]

  [file, $\mathbf{o_3}$, read], [file, $\mathbf{o_3}$, write], [file, $\mathbf{o_3}$, append], [file, $\mathbf{o_3}$, execute], [file, $\mathbf{o_3}$, admin]

  [file, read], [file, write], [file, append], [file, execute], [file, admin]}

Define attributes and values:

- AATT = {*object_am, role_am, classp*}

- Scope(*object_am*) = {($\mathbf{o_1}$, read), ($\mathbf{o_1}$, write), ($\mathbf{o_1}$, execute), ($\mathbf{o_1}$, append), ($\mathbf{o_1}$, admin), ($\mathbf{o_2}$, read), ($\mathbf{o_2}$, write), ($\mathbf{o_2}$, append), ($\mathbf{o_2}$, execute), ($\mathbf{o_2}$, admin), ($\mathbf{o_3}$, read), ($\mathbf{o_3}$, write), ($\mathbf{o_3}$, append), ($\mathbf{o_3}$, execute), ($\mathbf{o_3}$, admin)},

  attType(*object_am*) = set, is_ordered(*object_am*) = False, $H_{object\_am} = \phi$

- Scope(*role_am*) = {($\mathbf{r_1}$, grant), ($\mathbf{r_1}$,empower), ($\mathbf{r_1}$, admin), ($\mathbf{r_2}$, grant), ($\mathbf{r_2}$, empower), ($\mathbf{r_2}$, admin), ($\mathbf{r_3}$, grant), ($\mathbf{r_3}$, empower), ($\mathbf{r_3}$, admin), ($\mathbf{r_4}$, grant), ($\mathbf{r_4}$, empower), ($\mathbf{r_4}$, admin)},

  attType(*role_am*) = set, is_ordered(*role_am*) = False, $H_{role\_am} = \phi$

- Scope(*classp*) = {(file, read), (file, write), (file, append), (file, execute), (file, admin), (role, grant), (role, empower), (role, admin)},

  attType(*classp*) = set, is_ordered(*classp*) = False, $H_{classp} = \phi$

- *object_am*($\mathbf{u_1}$) = {($\mathbf{o_1}$, read), ($\mathbf{o_1}$, write), ($\mathbf{o_2}$, admin), ($\mathbf{o_3}$, append), ($\mathbf{o_3}$, admin)}

- *object_am*($\mathbf{u_2}$) = {($\mathbf{o_1}$, append), ($\mathbf{o_2}$, admin)}, *object_am*($\mathbf{u_3}$) = { }, *object_am*($\mathbf{u_4}$) = { }

- *role_am*($\mathbf{u_1}$) = {($\mathbf{r_1}$, grant), ($\mathbf{r_2}$, grant), ($\mathbf{r_3}$, grant), ($\mathbf{r_1}$, admin), ($\mathbf{r_4}$, admin)} *role_am*($\mathbf{u_2}$) = {($\mathbf{r_1}$, grant), ($\mathbf{r_2}$, grant)}, *role_am*($\mathbf{u_3}$) = { }, *role_am*($\mathbf{u_4}$) = { }

- *classp*($\mathbf{u_1}$) = { }, *classp*($\mathbf{u_2}$) = { }, *classp*($\mathbf{u_3}$) = {(file, admin)}, *classp*($\mathbf{u_4}$) = { }
- PATT = {*object_id*}
- Scope(*object_id*) = {$\mathbf{r_1, r_2, r_3, r_4, o_1, o_2, o_3}$}, attType(*object_id*) = atomic,

  is_ordered(*object_id*) = False, $H_{object\_am} = \phi$

Let object permissions in PERMS be represented by $\mathbf{p_1, p_2, p_3 ... p_n}$. Therefore *object_id* for each permission *p* is:

- *object_id*(*p*) = *o*.
- *object_id*($\mathbf{p_1}$) = $\mathbf{r_1}$, *object_id*($\mathbf{p_2}$) = $\mathbf{r_2}$ and so on, and *object_id*(p_1) = ϕ, if *p* is a class level permission.

Construct authorization functions for assigning permission tp role:

For each *op* in AOP, authorization function for assignment and revocation of permission of the form *p* = [file, *o, a*] to role *r* can be expressed as follows:

For any permission $p \in$ PERMS undertaken for assignment,

– is_authorized$P_{\mathbf{assign}}$(u : AU, p : PERMS, r : ROLES) $\equiv$

((*object_id*(*p*), admin) $\in$ *object_am*(*u*) $\wedge$ (*r*, empower) $\in$ *role_am*(*u*)) $\vee$ ((file, admin) $\in$ *classp*(*u*) $\wedge$ (*r*, empower) $\in$ *role_am*(*u*)) $\vee$ ((*object_id*(*p*), admin) $\in$ *object_am*(*u*) $\wedge$ (role, empower) $\in$ *classp*(*u*)) $\vee$ (file, admin) $\in$ *classp*(*u*) $\wedge$ (role, empower) $\in$ *classp*(*u*))

Construct authorization functions for revoking permission from role:

For any permission $p \in$ PERMS undertaken for revocation,

– is_authorized$P_{\mathbf{revoke}}$(u : AU, p : PERMS, r : ROLES) $\equiv$

(*object_id*(*p*), admin) $\in$ *object_am*(*u*) $\vee$ (*r*, admin) $\in$ *role_am*(*u*) $\vee$ (file, admin) $\in$ *classp*(*u*) $\vee$ (role, admin) $\in$ *classp*(*u*)

First step in the manual translation process above involves mapping the sets and functions from UARBAC's PRA to sets and functions from ARPA instance. In particular, sets of admin users (AU), roles (ROLES) and their hierarchy (RH) and, permissions (PERMS) are mapped into ARPA model. There are two administrative operations **assign** and **revoke** that map with the idea of grant and revoke operations in grantObjPermToRole() and revokeObjPermFromRole() functions, respectively. Secondly, the attributes that intuitively define the relations and functions described in UARBAC's PRA are defined in ARPA. Since a permission in UARBAC's PRA are tracked using the object id involved in each permission, object ids are mapped with respect to admin user attributes. There are three admin user attributes, namely, *object_am*, *role_am*, and *classp*. Admin user attribute *object_am*(*au*) yields object id/name and an access mode pair, which indicates the type of access mode that an admin user has over that particular object. For example, *object_am*($\mathbf{u_1}$) = {($\mathbf{o_1}$, read)} means admin user $\mathbf{u_1}$ has a read access mode towards object with id $\mathbf{o_1}$. *role_am*(*au*) gives a tuple of role id and an access mode for a given admin user. It indicates the type of access mode that admin user (*au*) has over that particular role. For instance, *role_am*($\mathbf{u_1}$) = {($\mathbf{r_1}$, grant)} indicates that admin user $\mathbf{u_1}$ has grant access mode towards role $\mathbf{r_1}$. *classp*(*au*) yields a tuple with object class name and an access mode indicating the type of access mode an admin user has over that class object. For example, *classp*($\mathbf{u_3}$) = {(file, admin)} means admin user, $\mathbf{u_3}$ has admin access mode towards file class object. This allows admin user to have admin access mode towards each object of class type file.

There is only one permission attribute called *object_id*. *object_id*(*p*) yields the object id towards which that particular permission is dedicated. Note that permission can be dedicated towards any class of object. For example, permissions $\mathbf{p_{15}}$ = [role, $\mathbf{r_4}$, admin] and $\mathbf{p_{12}}$ = [file, $\mathbf{o_1}$, read] are dedicated for role class type and file class type, respectively. *object_id*($\mathbf{p_{12}}$) yields object id $\mathbf{o_1}$ and, *object_id*($\mathbf{p_{15}}$) yields object id $\mathbf{r_4}$. If the permission is of class type, then *object_id*(*p*) is equal to null because it covers all the objects of particular class type. For instance, [file, admin] is a class level permission and doesn't bear an object id.

Thirdly, authorization function for permission-role assignment is constructed using related entities and their attributes. Finally in step four, authorization functions for revoking permission from role is established using attributes.

4.7.4 Map$_{\text{PRA-UARBAC}}$

Map$_{\text{PRA-UARBAC}}$ is an algorithm for mapping any instance of UARBAC's PRA [37] to its equivalent ARPA instance. For clarity, sets and function from UARBAC model are labeled with U, and that of ARPA with A.

Map$_{\text{PRA-UARBAC}}$ algorithm takes following sets and functions as input from UARBAC's PRA model. C^{U}, $USERS^{U}$, $ROLES^{U}$, $PERMS^{U}$, PA^{U}, RH^{U}, AM^{U}(role), AM^{U}(file), For each $u \in USERS^{U}$, authorized_perms[u], For each [file, o_1, a] $\in PERMS^{U}$ and for each $r_1 \in ROLES^{U}$, grantObjPermToRole([file, o_1, a], r_1) is true if the granter has one of the following combination of permissions:

- [file, o_1, admin] and [role, r_1, empower], or
- [file, o_1, admin] and [role, empower], or
- [file, admin] and [role, r_1, empower], or
- [file, admin] and [role, empower]

For each [file, o_1, a] $\in PERMS^{U}$ and for each $r_1 \in ROLES^{U}$, revokeObjPermFromRole([file, o_1, a], r_1) is true if the granter has either of the following permissions:

- [file, o_1, admin] or,
- [role, r_1, admin] or,
- [file, admin] or,
- [role, admin]

Output from Map$_{\text{PRA-UARBAC}}$ algorithm is an equivalent ARPA instance, with primarily consisting of AU^{A}, AOP^{A}, $ROLES^{A}$, RH^{A}, $PERMS^{A}$, $AATT^{A}$, $PATT^{A}$, For each attribute $att \in AATT^{A} \cup$

Algorithm 4.10 $\text{Map}_{\text{PRA-UARBAC}}$

```
    Input: Instance of PRA in UARBAC
    Output: ARPA instance
Step 1:        /* Map basic sets and functions in ARPA */
    a. $AU^A \leftarrow USERS^U$ ; $AOP^A \leftarrow \{\textbf{assign, revoke}\}$ ; $ROLES^A \leftarrow ROLES^U$
    b. $RH^A \leftarrow RH^U$ ; $PERMS^A \leftarrow PERMS^U$
Step 2:     /* Map attribute functions in ARPA */
    a. $AATT^A \leftarrow \{object\_am, role\_am, classp\}$
    b. Scope($object\_am$) = $O^U \times AM^U(\text{file})$ ; attType($object\_am$) = set
    c. is_ordered($object\_am$) = False, $H_{object\_am} = \phi$
    d. For each $u$ in $AU^U$, $object\_am(u) = \phi$
    e. For each $u$ in $U^U$ and for each $[c, o_1, a] \in$ authorized_perms[$u$],
            $object\_am(u)' = object\_am(u) \cup (o, a)$
    f. Scope($role\_am$) = $ROLES^U \times AM^U(\text{role})$ ; attType($role\_am$) = set
    g. is_ordered($role\_am$) = False, $H_{role\_am} = \phi$
    h. For each $u$ in $AU^A$, $role\_am(u) = \phi$
    i. For each $u$ in $U^U$ for each $[c, r_1, a] \in$ authorized_perms[$u$],
            $role\_am(u)' = role\_am(u) \cup (r, a)$
    j. Scope($classp$) = $C^U \times \{AM^U(\text{file}) \cup AM^U(\text{role})\}$
    k. attType($classp$) = set ; is_ordered($classp$) = False, $H_{classp} = \phi$
    l. For each $u$ in $AU^A$, $object\_am(u) = \phi$
    m. For each $u$ in $U^U$ for each $[c, a] \in$ authorized_perms[$u$],
            $classp(u)' = object\_am(u) \cup (c, a)$
    n. $PATT^A = \{object\_id\}$
    o. Scope($object\_id$) = $ROLES^U \cup PERMS^U$ ; attType($object\_id$) = atomic
    p. is_ordered($object\_id$) = False, $H_{object\_id} = \phi$
    q. For each $u$ in $U^U$ and for each $p \in$ authorized_perms[$u$],
            where $p$ is of the form $[c, o_i, a]$ or $[c, a]$, $object\_id(p) = o_i$ or $\phi$
```

Continuation of Algorithm 4.10 Map$_{\text{PRA-UARBAC}}$

```
Step 3:  /* Construct assign rule in ARPA */
    a. assign_formula =
    $((object\_id(p), \text{admin}) \in object\_am(au) \wedge (r, \text{empower}) \in role\_am(au)) \vee ((\text{file}, \text{admin}) \in classp(au) \wedge (r, \text{empower}) \in role\_am(au)) \vee ((object\_id(p), \text{admin}) \in object\_am(au) \wedge (\text{role}, \text{empower}) \in classp(au)) \vee (\text{file}, \text{admin}) \in classp(au) \wedge (\text{role}, \text{empower}) \in classp(au))$
    b. auth_assign = $\text{is\_authorizedP}_{\textbf{assign}}(au : \text{AU}, p : \text{PERMS}, r : \text{ROLES}) \equiv$ assign_formula
Step 4:  /* Construct revoke rule for ARPA */
    a. revoke_formula =
    $(object\_id(p), \text{admin}) \in object\_am(au) \vee (r, \text{admin}) \in role\_am(au)) \vee (\text{file}, \text{admin}) \in classp(au) \vee (\text{role}, \text{admin}) \in classp(au))$
    b. auth_revoke = $\text{is\_authorizedP}_{\textbf{revoke}}(au : \text{AU}, p : \text{PERMS}, r : \text{ROLES}) \equiv$ revoke_formula
```

PATT$^{\text{A}}$, Scope(att), attType(att), is_ordered(att) and H$_{att}$, For each user $u \in$ AU$^{\text{A}}$, and for each $att \in$ AATT$^{\text{A}} \cup$ PATT$^{\text{A}}$, $att(u)$, Authorization rule for assign (auth_assign), Authorization rule for revoke (auth_revoke)

Step 1 in Map$_{\text{PRA-UARBAC}}$ involves translating sets and functions from UARBAC's PRA to ARPA equivalent sets and functions. In Step 2, permission attributes and admin user attributes functions are defined. There exists one permission attributes *object_id*, which captures id of an object for given permission. Note that a permission defines class type, object id and access mode. ARPA defines three admin user attributes: *object_am, role_am* and *classp*. *object_am* attribute captures an admin user's access mode towards an object. Similarly, *role_am* captures an admin user's access mode towards a role. An admin user can also have a class level access mode captured by attribute *classp*. With class level access mode, an admin user gains authority over an entire class of objects. For example [grant, role] admin permission provides an admin user with power to grant any role to a given permission (or a user).

In Step 3, assign_formula for ARPA that is equivalent to grantObjPermToRole([file, o_1, a], r_1) in UARBAC's PRA is established. Equivalent assign_formula is expressed as $\text{is\_authorizedP}_{\textbf{assign}}(au_1 : \text{AU}^{\text{A}}, p_1 : \text{PERMS}^{\text{A}}, r_1 : \text{ROLES}^{\text{A}})$ using attributes of permissions and admin user. Note that [file,

o_1, a] is equal to p_1. Step 4 establishes revoke_formula equivalent to revokeRoleFromUser(u_1, r_1). It is expressed as is_authorized$\text{P}_{\textbf{revoke}}$($au_1$: $\text{AU}^{\mathtt{A}}$, p_1 : $\text{PERMS}^{\mathtt{A}}$, r_1 : $\text{ROLES}^{\mathtt{A}}$) using attributes of permissions and admin user.

An article that included both attribute-based user-role assignment (AURA) model and, attribute-based permission-role assignment (ARPA) model, collectively known as *'AARBAC: Attribute-Based Administration of Role-Based Access Control'* was published in IEEE 3nd International Conference on Collaboration and Internet Computing (CIC), 2017 (CIC2017) [39]. Another version that includes detailed description of these two models is also available [41].

Chapter 5: ARRA: ATTRIBUTE-BASED ROLE-ROLE ASSIGNMENT MODEL

Portion of materials in this chapter are published in the following venue [40]:

- Jiwan Ninglekhu and Ram Krishnan. ARRA: Attribute-Based Role-Role Assignment. International Workshop on Secure Knowledge Management 2017 (SKM2017), 2017.

This chapter introduces an approach for attribute-based role-role assignment (ARRA). Like the previous two models, ARRA's objective is to unify prior RRA approaches such that those models can be expressed using ARRA. In addition, new features essential for making access control decisions can be introduced for role-role assignment. Inspired by prior RRA models, attributes for admin users, admin roles and regular roles are introduced. Based on these attributes role assignment and revocation decisions are made.

5.1 ARRA Model

Table 5.1 presents formal ARRA model. The entities involved in ARRA comprise of admin users (AU), regular roles (ROLES) and their hierarchy (RH), admin roles (AR) and their hierarchy (ARH), admin user to admin role relation (AUA) and admin operations (AOP).

In ARRA, admin user in AU wants to perform admin operation such as assign or revoke from AOP using attributes of entities in the model. Admin users attribute functions (AATT) and admin roles attribute functions (ARATT) are introduced. In addition, based on the need observed, regular roles attribute function called (RATT) has also been introduced. Attributes from other RBAC entities can also be developed if needed. It shall be later observed that there exist a need for attributes from different entities in representing properties of RRA97 and UARBAC's RRA in ARRA.

The ARRA model is very similar in its core with previously described two models. The challenge is in the representation of prior RRA models, as the scenarios under which role-role assignments are performed may vary and, may be different from scenarios under which user-role or permission-role assignments are done. Like previous two models, this version of ARRA is also limited to assignment and revocation of roles.

In ARRA, there are two ways by which an admin user can select a set of regular roles for assignment to a target regular role. The first way allows an admin user to select a single role and a target role, and perform an admin operation like assign or revoke. The second way allows an admin user to select a set of regular roles, the target role and perform similar operation on those roles. In the latter case, the selection criteria for the set of regular roles can be expressed using a set-builder notation whose rule is based on the regular role attributes. For example, is_authorized$\text{R}_{\textbf{assign}}$(**au**, $\{r_1 \mid r_1 \in \text{ROLES} \land \textbf{Lead} \in \mathit{roleTitle}(r_1)\}$, $\mathbf{r_2}$) would specify a policy for an admin user **au** that selects the set of all the roles with role title **Lead** in order to assign those roles to a role $\mathbf{r_2}$. Assigning a role r_1 to role r_2 makes role r_1 junior to r_2. In other words, it adds an entry $<r_2, r_1>$ in RH. It is also referred to as edge insetion.

Authorization rule is specified as a logical expression on the attributes of admin users, admin roles, and that of regular roles considered for assignment.

5.2 Mapping Prior RRA Models in ARRA

In this section, ARRA model's ability to intuitively simulate the features of prior RRA models is demostrated. For each of the prior models considered, an example instance is expressed. For each example instance of RRA corresponding equivalent instance in ARRA is manually mapped . Then eventually, concrete algorithms that can convert any instance of prior RRA models into their equivalent ARRA instances are depicted.

5.3 RRA97 in AARA

This segment presents RRA97 example instance and its manual convertion into its equivalent ARRA instance. It is followed by a mapping algorithm, which demonstrates a programmable procedure for translation of any RRA97 instance to its equivalent ARRA instance.

Table 5.1: ARRA Model

– AU, AOP, ROLES, AR are finite sets of administrative users, administrative operations such as assign and revoke, regular roles and administrative roles, respectively.

– AUA $\subseteq$ AU $\times$ AR, administrative user to administrative role assignment relation.

– RH $\subseteq$ ROLES $\times$ ROLES, a partial ordering on the set ROLES.

– ARH $\subseteq$ AR $\times$ AR, a partial ordering on the set AR.

– AATT, ARATT, and RATT are finite sets of administrative user attribute functions, administrative role attribute functions, and regular role attribute functions, respectively.

– For each *att* in AATT $\cup$ ARATT $\cup$ RATT, Scope(*att*) is a finite set of atomic values from which the range of the attribute function *att* is derived.

– attType : AATT $\cup$ ARATT $\cup$ RATT $\rightarrow$ {set, atomic}, which specifies whether the range of a given attribute is atomic or set valued.

– Each attribute function maps elements in AU, AR and ROLES to atomic or set values.

$$\forall aatt \in \text{AATT}.\ aatt : \text{AU} \rightarrow \begin{cases} \text{Scope}(aatt) \text{ if attType}(aatt) = \text{atomic} \\ 2^{\text{Scope}(aatt)} \text{ if attType}(aatt) = \text{set} \end{cases}$$

$$\forall aratt \in \text{ARATT}.\ aratt : \text{AR} \rightarrow \begin{cases} \text{Scope}(aratt) \text{ if attType}(aratt) = \text{atomic} \\ 2^{\text{Scope}(aratt)} \text{ if attType}(aratt) = \text{set} \end{cases}$$

$$\forall ratt \in \text{RATT}.\ ratt : \text{ROLES} \rightarrow \begin{cases} \text{Scope}(ratt) \text{ if attType}(ratt) = \text{atomic} \\ 2^{\text{Scope}(ratt)} \text{ if attType}(ratt) = \text{set} \end{cases}$$

– is_ordered : AATT $\cup$ ARATT $\cup$ RATT $\rightarrow$ {True, False}, specifies if the scope is ordered for each of the attributes.

– For each *att* $\in$ AATT $\cup$ ARATT $\cup$ RATT,
if is_ordered(*att*) = True, $H_{att} \subseteq$ Scope(*att*) $\times$ Scope(*att*), a partially ordered attribute hierarchy, and $H_{att} \neq \phi$, else, if is_ordered(*att*) = False, $H_{att} = \phi$
(For some *att* $\in$ AATT $\cup$ ARATT $\cup$ RATT for which attType(*att*) = set and isord(*att*) = True, if $\{a, b\}$, $\{c, d\} \in 2^{\text{Scope}(att)}$ (where $a, b, c, d \in$ Scope(att)), we infer $\{a, b\} \geq \{c, d\}$ if (a, c), (a, d), (b, c), $(b, d) \in H^{*}_{att}$.)

ARRA model allows an administrator to perform an operation on a single role or a set of roles at a time. The authorization rule for performing an operation on a single role is as follows:

For each *op* in AOP, **is_authorizedR$_{op}$**(*au*: AU, r_1 : ROLES, r_2 : ROLES) specifies if the admin user *au* is allowed to perform the operation *op* (e.g. assign, revoke, etc.) between the regular roles r_1 and the role r_2. Assigning a role r_1 to r_2 makes r_1 junior to r_2. This rule is written as a logical expression using attributes of admin user *au*, admin role, *ar*, and regular role, *r*.

The authorization rule for performing an operation on a set of users is as follows:

For each *op* in AOP, **is_authorizedR$_{op}$**(*au*: AU, χ : 2^{ROLES}, *r* : ROLES) specifies if the admin user *au* is allowed to perform the operation *op* (e.g. assign, revoke, etc.) between the roles in the set χ and the role *r*.
Here χ is a set of roles that can be specified using a set-builder notation, whose rule is written using role attributes.

5.3.1 An RRA97 Example Instance

RRA97 [55] assumes an existing regular role hierarchy and an administrative role hierarchy as depicted in Figure 5.1. An example instance for RRA97 model is presented as follows:

Sets and Functions:

- USERS = {$\mathbf{u_1}$, $\mathbf{u_2}$, $\mathbf{u_3}$, $\mathbf{u_4}$}
- ROLES = {**ED, E1, PE1, QE1, PL1, E2, PE2, QE2, PL2, DIR**}
- AR = {**DSO, PSO1**}
- AUA = {($\mathbf{u_3}$, **DSO**), ($\mathbf{u_4}$, **PSO1**)}
- RH = {**<ED, E1>**, **<E1, QE1>**, **<E1, PE1>**, **<PE1, PL1>**, **<QE1, PL1>**, **<ED, E2>**,

  <E2, PE2>, **<E2, QE2>**, **<PE2, PL2>**, **<QE2, PL2>**, **<PL1, DIR>**, **<PL2, DIR>**}
- ARH = {**<SSO, DSO>**, **<DSO, PSO1>**, **<DSO, PSO2>**}
- *can-modify* = {(**DSO, (ED, DIR)**), (**PSO1, (E1, PL1)**), (**PSO1, (E2, PL2)**)}

  Where, (**ED, DIR**) = {**E1, E2, PE1, PE2, QE1, QE2, PL1, PL2**},

  (**E1, PL1**) = {**PE1, QE1**} and, (**E2, PL2**) = {**PE2, QE2**}

USERS, ROLES, and AR are sets of users, regular roles and admin roles, respectively. User $\mathbf{u_3}$ is given admin role **DSO** and **PSO1** is given to role $\mathbf{u_4}$. RH and ARH represent regular role and admin role hierarchies as depicted in Figure 5.1. Note that original *can-modify* example from RRA97 paper [55] is considered where each admin role is mapped with authority ranges. The *can-modify* set in the example instance above depicts the same scenario with three elements: (**DSO**, (**ED, DIR**)), (**PSO1**, (**E1, PL1**)) and (**PSO1**, (**E2, PL2**)). It shows that role **DSO** is given an authority range (**ED, DIR**) while role **PSO1** is given two authority ranges (**E1, PL1**) and (**E2, PL2**).

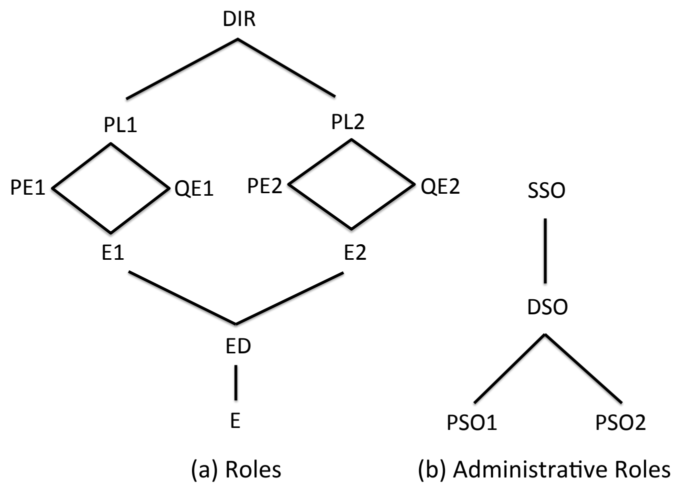

Figure 5.1: Role and Administrative Role Hierarchies in RRA97 [55]

In RRA97, an authority range, (x, y) = $\{r : \text{ROLES} \mid x < r < y\}$, where x and y are end points of the range.

For every (*ar*, (x, y)) in *can-modify*, the range denoted by (x, y) must be an encapsulated authority range. A range (x, y) is said to be encapsulated if,

$\forall r_1 \in (x, y) \wedge \forall r_2 \notin (x, y). [r_2 > r_1 \leftrightarrow r_2 > y \wedge r_2 < r_1 \leftrightarrow r_2 < x]$

A new edge AB can be inserted between two incomparable roles A and B under following two conditions:

- $AR_{immediate}(A) = AR_{immediate}(B)$. An immediate authority range of a role *r*, $AR_{immediate}(r)$ is the authority range (x,y) such that $r \in (x, y)$ and for all authority ranges (x', y') junior to (x, y) we have $r \notin (x', y')$. **or,**
- if (x, y) is an authority range such that $(A = y \wedge B > x) \vee (B = x \wedge A < y)$ then insertion of AB must preserve encapsulation of (x, y).

5.3.2 ARRA Instance Equivalent to RRA97 Instance

In this section, a manual translation of RRA97 example instance presented in previous segment into an equivalent ARRA instance is described. Equivalent ARRA instance is as follows:

Map sets and functions:

- AU = {$\mathbf{u_1, u_2, u_3, u_4}$}
- AOP = {**insertEdge, deleteEdge**}
- ROLES = {**ED, DIR, E1, PE1, QE1, PL1, E2, PE2, QE2, PL2, DIR**}
- AUA = {($\mathbf{u_3}$, **DSO**), ($\mathbf{u_4}$, **PSO1**)}
- RH = {**<ED, E1>**, **<E1, QE1>**, **<E1, PE1>**, **<PE1, PL1>**, **<QE1, PL1>**, **<ED, E2>**,

  <E2, PE2>, **<E2, QE2>**, **<PE2, PL2>**, **<QE2, PL2>**, **<PL1, DIR>**, **<PL2, DIR>**}

Define attributes

- AATT = { }, RATT = { }
- ARATT= {*authRange*}
- Scope(*authRange*) = RH^+, attType(*authRange*) = set, is_ordered(*authRange*) = False,

  $H_{authRange} = \phi$
- *authRange*(**DSO**) = {**ED, DIR**},

  authRange(**PSO1**) = {(**E1, PL1**), (**E2, PL2**)}

Construct authorization rule for **insertEdge**

To assign role r_1 to role r_2 (or to insert an edge r_1r_2 and hence add $<r_2, r_1>$ to RH),

– is_authorizedR$_{\textbf{insertEdge}}$(au : AU, r_1 : ROLES, r_2 : ROLES) ≡

$(\exists(au, ar_1) \in \text{AUA}, \exists(\text{s, t}) \in authRange(ar_1).\ r_1, r_2 \in \lceil \text{s, t}\rceil) \wedge ((\exists(\text{m, n}), (\text{m', n'}) \in \bigcup_{ar_2 \in \text{AR}} authRange(ar_2).\ r_1, r_2 \in \lceil \text{m, n}\rceil \wedge (\lceil \text{m', n'}\rceil \subset \lceil \text{m, n}\rceil \rightarrow r_1, r_2 \notin (\text{m', n'}))) \vee (\exists(\text{x,y}) \in \bigcup_{ar_3 \in \text{AR}} authRange(ar_3).\ ((r_1 = \text{y} \wedge r_2 > \text{x}) \vee (r_2 = \text{x} \wedge r_1 < \text{y})) \wedge (\forall \text{p} \in \lceil \text{x, y}\rceil \wedge \forall \text{q} \notin \lceil \text{x, y}\rceil.$
$(<\text{q, p}> \in (\text{RH} \cup <r_2, r_1>)^* \leftrightarrow <\text{q, y}> \in (\text{RH} \cup <r_2, r_1>)^*) \wedge (<\text{p, q}> \in (\text{RH} \cup <r_2, r_1>)^* \leftrightarrow$
$<\text{x, q}> \in (\text{RH} \cup <r_2, r_1>)^*))))$

Construct authorization rule for **deleteEdge**

– is_authorized$\mathrm{R}_{\mathbf{deleteEdge}}(au : \mathrm{AU}, r_1 : \mathrm{ROLES}, r_2 : \mathrm{ROLES}) \equiv$

$$\exists(au, ar) \in \mathrm{AUA}, \exists(\mathrm{x}, \mathrm{y}) \in authRange(ar).\ r_1, r_2 \in \lceil \mathrm{x}, \mathrm{y} \rceil \wedge \forall(r_1, r_2) \notin \bigcup_{ar \in \mathrm{AR}} authRange(ar)$$

As it can be observed, the process of translation involves four main steps. First, we map given sets from RRA97 to equivalent sets in ARRA. Second, required attribute functions are defined. Admin role attribute *authRange* maps authority ranges to admin roles. Attribute *authRange* is set valued and unordered. Scope of *authRange* is a transitive clousure of regular role hierarchy, RH^+. That is, if <a, b>, <b, c>, <c, d> $\in \mathrm{RH}^+$, then <a, d> is also true.

Note that authority range represents a set of roles over which an admin role have authority to conduct operations such as assign and revoke. In ARRA, a symbolic representation of authority range with end points a and b as (a, b) is used. However, whenever there is a need to work with roles present in the set represented by authority range (a, b), the set of roles is denoted with $\lceil \mathrm{a}, \mathrm{b} \rceil$. That is, $\lceil \mathrm{a}, \mathrm{b} \rceil = \{r \mid r \in \mathrm{ROLES} \wedge \mathrm{a} < r < \mathrm{b}\}$.

Next, authorization rule for the admin operation **insertEdge** is constructed. There are fundamentally three requirements that an admin user must consider to assign role r_1 to role r_2. Firstly, both the roles r_1 and r_2 must be within the authority range of an admin role assigned to a given admin user. Secondly, both the roles taken for edge insertion must have same immediate authority range, or (thirdly) if the edge is to be inserted between a role on the top end point of an authority range and a role senior to the lower end point of authority range, or if the edge is to be inserted between a role on the bottom end of an authority range and a role junior to the top end of the authority range, then that edge insertion must preserve encapsulation of an authority range with those (top and bottom) end points. Finally, an authorization formula for operation **deleteEdge** is constructed. An admin user with an admin role can delete an edge between any two roles in an authority range except that any edge between the end points of an authority range can not be deleted.

Algorithm 5.1 Map$_{\text{RRA97}}$

```
Input: RRA97 instance
Output: ARRA instance
Step 1: /* Map basic sets and functions in ARRA */
    a. $\text{AU} \leftarrow \text{USERS}^{97}$ ; $\text{AOP}^{\mathbb{A}} \leftarrow \{\textbf{insertEdge, deleteEdge}\}$
    b. $\text{ROLES}^{\mathbb{A}} \leftarrow \text{ROLES}^{97}$ ; $\text{AUA}^{\mathbb{A}} \leftarrow \text{AUA}^{\mathbb{U}}$ ; $\text{RH}^{\mathbb{A}} \leftarrow \text{RH}^{97}$
Step 2: /* Map attribute functions in ARRA */
    a. $\text{AATT}^{\mathbb{A}} \leftarrow \{\}$ ; $\text{RATT}^{\mathbb{A}} \leftarrow \{\}$
    b. $\text{ARATT}^{\mathbb{A}} \leftarrow \{\mathit{authRange}\}$
    c. $\text{Scope}^{\mathbb{A}}(\mathit{authRange}) = \text{RH}^{+\mathbb{A}}$ ; $\text{attType}^{\mathbb{A}}(\mathit{authRange}) = \text{set}$
    d. $\text{is\_ordered}^{\mathbb{A}}(\mathit{authRange}) = \text{False}$ ; $\text{H}^{\mathbb{A}}_{\mathit{authRange}} = \phi$
    e. For each $ar \in \text{AR}^{\mathbb{A}}$, $\mathit{authRange}(ar) = \phi$
    f. For each $(ar, (r_i, r_j)) \in \mathit{can\text{-}modify}^{97}$, $\mathit{authRange}(ar) = \mathit{authRange}(ar) \cup (r_i, r_j)$
Step 3: /* Construct assign rule in ARRA */
    a. assign_formula = $\phi$
    b. For each $(ar, (r_i, r_j)) \in \mathit{can\text{-}modify}^{97}$,
        $\text{assign\_formula'} = \text{assign\_formula} \vee (\exists(au, ar_1) \in \text{AUA}, \exists(\text{s, t}) \in \mathit{authRange}(ar_1).$
        $r_1, r_2 \in \lceil \text{s, t} \rceil) \wedge ((\exists(\text{m, n}), (\text{m', n'}) \in \bigcup_{ar_2 \in \text{AR}} \mathit{authRange}(ar_2).\ r_1, r_2 \in \lceil \text{m, n} \rceil \wedge$
        $(\lceil \text{m', n'} \rceil \subset \lceil \text{m, n} \rceil \rightarrow r_1, r_2 \notin (\text{m', n'}))) \vee (\exists(\text{x,y}) \in \bigcup_{ar_3 \in \text{AR}} \mathit{authRange}(ar_3).\ ((r_1 = \text{y} \wedge$
        $r_2 > \text{x}) \vee (r_2 = \text{x} \wedge r_1 < \text{y})) \wedge (\forall \text{p} \in \lceil \text{x, y} \rceil \wedge \forall \text{q} \notin \lceil \text{x, y} \rceil.\ (<\text{q, p}> \in (\text{RH} \cup <r_2, r_1>)^* \leftrightarrow$
        $<\text{q, y}> \in (\text{RH} \cup <r_2, r_1>)^*) \wedge (<\text{p, q}> \in (\text{RH} \cup <r_2, r_1>)^* \leftrightarrow <\text{x, q}> \in (\text{RH} \cup <r_2, r_1>)^*))))$
    c. $\text{auth\_assign} = \text{is\_authorizedR}_{\textbf{insertEdge}}(au : \text{AU}^{\mathbb{A}}, r_1 : \text{ROLES}^{\mathbb{A}}, r_2 : \text{ROLES}^{\mathbb{A}}) \equiv$
        assign_formula'
Step 4: /* Construct revoke rule for ARRA */
    a. revoke_formula = $\phi$
    b. For each $(ar_1, (r_1, r_2)) \in \mathit{can\text{-}modify}^{97}$,
        $\text{revoke\_formula'} = \text{revoke\_formula} \vee \exists(au, ar) \in \text{AUA}, \exists(\text{x, y}) \in \mathit{authRange}(ar).$
        $r_1, r_2 \in \lceil \text{x, y} \rceil \wedge \forall(r_1, r_2) \notin \bigcup_{ar \in \text{AR}} \mathit{authRange}(ar)$
    c. $\text{auth\_revoke} = \text{is\_authorizedR}_{\textbf{deleteEdge}}(au : \text{AU}^{\mathbb{A}}, r_1 : \text{ROLES}^{\mathbb{A}}, r_2 : \text{ROLES}^{\mathbb{A}}) \equiv$
        revoke_formula'
```

5.3.3 Map$_{\textbf{RRA97}}$

Algorithm 5.1 presents Map$_{\text{RRA97}}$. It is a translation algorithm that maps any RRA97 instance into an equivalent ARRA instance. Superscript labels 97 and A in the algorithm represent sets and functions from RRA97 and ARRA, respectively. Map$_{\text{RRA97}}$ takes an instance of RRA97 as input. In particular, input consists of USERS^{97}, ROLES^{97}, AR^{97}, AUA^{97}, RH^{97}, ARH^{97}, and *can-modify*^{97}. The *can-modify* instruction covers operations for inserting an edge, deleting an edge, creating a role and deleting a role. ARRA can simulate inserting edge and deleting an edge. However, creating and deleting roles are beyond the scope of current ARRA model.

Output from Map$_{\text{RRA97}}$ algorithm is an equivalent ARRA instance, with following sets and functions. AU$^{\mathbb{A}}$, AOP$^{\mathbb{A}}$, ROLES$^{\mathbb{A}}$, AUA$^{\mathbb{A}}$, RH$^{\mathbb{A}}$, ARH$^{\mathbb{A}}$, AATT$^{\mathbb{A}}$, ARATT$^{\mathbb{A}}$, For each attribute *att* $\in$ ARATT$^{\mathbb{A}}$, Scope$^{\mathbb{A}}$(*att*), attType$^{\mathbb{A}}$(*att*), is_ordered$^{\mathbb{A}}$(*att*) and H$^{\mathbb{A}}{}_{att}$, For each admin role *ar* $\in$ AR$^{\mathbb{A}}$ and for each *att* $\in$ ARATT$^{\mathbb{A}}$, *att*(*ar*), Authorization rule for assign (auth_assign), and Authorization rule for revoke (auth_revoke).

Steps indicated in Map$_{\text{RRA97}}$ correspond to the steps described in equivalent ARRA instance for RRA97 in section 5.3.2. It here briefly explained here. Step 1 maps sets from RRA97 to ARRA sets and functions. In Step 2, admin user attributes and admin role attributes are expressed. AATT and RATT are left empty as there is no use case for these attributes in translating RRA97. Admin role attribute *authRange* captures the mapping between an authority range as defined in RRA97, and an admin role. For each (*ar*, (r_i, r_j)) in set *can-modify*^{97}, *authRange*(*ar*) populates set of authority ranges for *ar*.

In Step 3, assign_formula rule for ARRA that is equivalent to inserting an edge in *can_modify*^{97} is constructed. For each (*ar*, (r_i, r_j)) in set *can-modify*^{97}, logical expression that checks the assignment conditions is constructed. Equivalent translation in ARRA is given by is_authorizedR$_{\textbf{insertEdge}}$(*au* : AU$^{\mathbb{A}}$, r_1 : ROLES$^{\mathbb{A}}$, r_2 : ROLES$^{\mathbb{A}}$). This formula must checks if roles r_1 and r_2 belong in an authority range of an admin role, which is mapped to admin user *au*. It further checks if r_1 and r_2 have same immediate authority range, or checks if any other assignment action doesn't violate encapsulation of admin role's authority range. Similarly, In Step 4, revoke_formula equivalent to deleting an edge from *can_modify*[97] is expressed. It checks if admin role mapped to admin role *au* has authority range where r_1 and r_2 belong and, restricts from deleting an edge between end points of any authority range.

5.3.4 UARBAC's RRA in ARRA

In this section, an example instance for UARBAC's RRA and its equivalent ARRA instance are presented. Followed by the manual translation process, a formal algorithm that outlines the translation procedure is presented.

5.3.5 An Example Instance of UARBAC's RRA

RBAC schema

- *C* = {role}
- *OBJS*(user) = **USERS**, *OBJS*(role) = **ROLES**
- *AM*(role) = {grant, empower, admin}

RBAC state

- *USERS* = *OBJ*(user) = {$\mathbf{u_1}$, $\mathbf{u_2}$, $\mathbf{u_3}$, $\mathbf{u_4}$}
- *ROLES* = *OBJ*(role)= {$\mathbf{r_1}$, $\mathbf{r_2}$, $\mathbf{r_3}$}
- *P* = [role, $\mathbf{r_1}$, grant], [role, $\mathbf{r_1}$, empower], [role, $\mathbf{r_1}$, admin], [role, $\mathbf{r_2}$, grant], [role, $\mathbf{r_2}$, empower], [role, $\mathbf{r_2}$, admin], [role, $\mathbf{r_3}$, grant], [role, $\mathbf{r_3}$, empower], [role, $\mathbf{r_3}$, admin], [role, grant], [role, empower], [role, admin]}
- *RH* = {<$\mathbf{r_2}$, $\mathbf{r_3}$>}

Administrative permissions of UARBAC's RRA

Following is the list of administrative permissions each user has for role-role assignment:

- authorized_perms[$\mathbf{u_1}$] = {[role, $\mathbf{r_1}$, grant], [role, $\mathbf{r_2}$, admin], [role, $\mathbf{r_2}$, empower], [role, $\mathbf{r_3}$, admin]}
- authorized_perms[$\mathbf{u_2}$] = {[role, $\mathbf{r_1}$, admin], [role, $\mathbf{r_1}$, empower], [role, $\mathbf{r_2}$, empower], [role, $\mathbf{r_3}$, grant]}
- authorized_perms[$\mathbf{u_3}$] = {[role, admin]}
- authorized_perms[$\mathbf{u_4}$] = {[role, grant], [role, empower], [role, admin]}

<u>Role-role assignment condition</u>

One can perform following operation to assign a role $\mathbf{r_1}$ to another role $\mathbf{r_2}$.

- grantRoleToRole($\mathbf{r_1, r_2}$)

To perform this operation one needs one of the following two permissions:

- [role, $\mathbf{r_1}$, grant] and [role, $\mathbf{r_2}$, empower] or,
- [role, grant] and [role, $\mathbf{r_2}$, empower] or,
- [role, $\mathbf{r_1}$, grant] and [role, empower] or,
- [role, grant] and [role, empower]

<u>Condition for revoking a role from another role</u>

To revoke a role $\mathbf{r_2}$ from a role $\mathbf{r_3}$, admin user performs following operation.

- revokeRoleFromUser($\mathbf{r_2, r_3}$)

To conduct this operation one needs one of the following options:

- [role, $\mathbf{r_2}$, grant] and [role, $\mathbf{r_3}$, empower]
- [role, $\mathbf{r_2}$, admin]
- [role, $\mathbf{r_3}$, admin]
- [role, admin]

5.3.6 ARRA Instance Equivalent to UARBAC's RRA Instance

This segment presents equivalent instance for UARBAC's RRA example instance presented in previous segment.

Map UARBAC's RRA sets to ARRA sets

- AU = {$\mathbf{u_1}$, $\mathbf{u_2}$, $\mathbf{u_3}$, $\mathbf{u_4}$}
- AOP = {**assign, revoke**}
- ROLES = {$\mathbf{r_1}$, $\mathbf{r_2}$, $\mathbf{r_3}$},
- AUA = { }
- RH = {<$\mathbf{r_2}$, $\mathbf{r_3}$>}

Define Attributes and values

- AATT = {*grantAuth, empowerAuth, adminAuth, roleClassAuth*},

  ARATT = { }, RATT = { }
- Scope(*grantAuth*) = {$\mathbf{r_1}$, $\mathbf{r_2}$, $\mathbf{r_3}$}, attType(*grantAuth*) = set,

  is_ordered(*grantAuth*) = ROLES, $H_{grantAuth}$ = RH
- *grantAuth*($\mathbf{u_1}$) = {$\mathbf{r_1}$}, *grantAuth*($\mathbf{u_2}$) = {$\mathbf{r_3}$},

  grantAuth($\mathbf{u_3}$) = { }, *grantAuth*($\mathbf{u_4}$) = { }
- Scope(*empowerAuth*) = {$\mathbf{r_1}$, $\mathbf{r_2}$, $\mathbf{r_3}$}, attType(*empowerAuth*) = set,

  is_ordered(*empowerAuth*) = ROLES, $H_{empowerAuth}$ = RH
- *empowerAuth*($\mathbf{u_1}$) = {$\mathbf{r_2}$}, *empowerAuth*($\mathbf{u_2}$) = {$\mathbf{r_1}$, $\mathbf{r_2}$},

  empowerAuth($\mathbf{u_3}$) = { }, *empowerAuth*($\mathbf{u_4}$) = { }

- Scope(*adminAuth*) = {$\mathbf{r_1, r_2, r_3}$}, attType(*adminAuth*) = set,

  is_ordered(*adminAuth*) = ROLES, $H_{adminAuth}$ = RH

- *adminAuth*($\mathbf{u_1}$) = {$\mathbf{r_2, r_3}$}, *adminAuth*($\mathbf{u_2}$) = {$\mathbf{r_1}$},

  adminAuth($\mathbf{u_3}$) = { }, *adminAuth*($\mathbf{u_4}$) = { }

- Scope(*roleClassAuth*) = *AM*(role), attType(*roleClassAuth*) = set,

  is_ordered(*roleClassAuth*) = False, $H_{roleClassAuth} = \phi$

- *roleClassAuth*($\mathbf{u_1}$) = { }, *roleClassAuth*($\mathbf{u_2}$) = { },

  roleClassAuth($\mathbf{u_3}$) = {admin},*roleClassAuth*($\mathbf{u_4}$) = {grant, empower, admin}

For each *op* in AOP, authorization rule to assign/revoke role-role can be expressed as follows:

Construct authorization rule for role-role assignment

To assign any regular role $r_1 \in$ ROLES to regular role $r_2 \in$ ROLES,

– is_authorizedU$_{\mathbf{assign}}$(au : AU, r_1 : ROLES, r_2 : ROLES) $\equiv$

$(r_1 \in grantAuth(au) \wedge r_2 \in empowerAuth(au)) \vee (r_1 \in grantAuth(au) \wedge$

$\text{empower} \in roleClassAuth(au)) \vee (\text{grant} \in roleClassAuth(au) \wedge r_2 \in empowerAuth(au)) \vee$

$(\text{grant} \in roleClassAuth(au) \wedge \text{empower} \in roleClassAuth(au))$

Construct authorization rule for revoking role-role

To revoke any regular role $r_1 \in$ ROLES from another regular role $r_2 \in$ ROLES,

– is_authorizedU$_{\mathbf{revoke}}$(au_1 : AU, r_1 : ROLES, r_2 : ROLES) $\equiv$

$(r_1 \in grantAuth(au) \wedge r_2 \in empowerAuth(au)) \vee r_1 \in adminAuth(au) \vee r_2 \in adminAuth(au) \vee$

$\text{admin} \in roleClassAuth(au)$

There are four main stages in this translation process. First, main sets from UARBAC's RRA to ARRA model are mapped. Set of admin users consists of set of users from UARBAC's RRA, namely $\mathbf{u_1, u_2, u_3}$, and $\mathbf{u_4}$. There are two admin operations, **assign** and **revoke**. Sets ROLES and RH map to set of roles and role hierarchy from UARBAC's RRA, respectively. AUA is left empty as there is no notion of admin roles in UARBAC.

Next, required attribute functions that capture the properties introduced in UARBAC's RRA are defined. Admin user attributes are used to express functions and relations in ARRA for UARBAC's RRA. Among four admin user attributes, *grantAuth, empowerAuth* and *adminAuth* are object level attributes. Each of these attributes express the nature of authority an admin user has over different roles. For example, *empowerAuth* over a role r_x, allows admin user to assign any object such as a user and/or another role to role r_x. Class level attribtue *roleClassAuth* captures the nature of *access mode* an admin user has towards role class. The scope of this attribute is $AM(\mathsf{role})$, which has three different types of access modes, namely grant, empower and admin. For example, an admin user with *roleClassAuth* of grant allows that admin user to grant (assign) any role to other object such as a user and/or role. Translation of UARBAC's RRA doesn't require admin role and regular role attributes. Hence, they are left empty.

After attributes have been defined, the assignment formula (assign_formula) is constructed using usual logical expression. As mentioned in the UARBAC's RRA instance, there are four different combination of permissions an admin user can use to make role-role assignment. An equivalent logical expression is captured as is_authorized$\text{U}_{\textbf{assign}}(au : \text{AU}, r_1 : \text{ROLES}, r_2 : \text{ROLES})$. Finally, a logical expression, which captures four different permission options that an admin user can use for revoking a role from another role are constructed. The logical expression is equivalent to is_authorized$\text{U}_{\textbf{revoke}}(au_1 : \text{AU}, r_1 : \text{ROLES}, r_2 : \text{ROLES})$.

5.3.7 Map$_{\textbf{RRA-UARBAC}}$

Algorithm 5.2 presents Map$_{\text{RRA-UARBAC}}$. It is a procedure to map any instance of UARBAC's RRA [37] to its equivalent ARRA instance. It take instance of UARBAC's RRA as its input and yields an equivalent AARA instance as its output. Sets and functions from UARBAC and ARRA are labeled with superscripts U and A, respectively. Input consists of C^{U}, $USERS^{\text{U}}$, $ROLES^{\text{U}}$, P^{U}, RH^{U}, $AM^{\text{U}}(\mathsf{role})$, For each $u \in USERS^{\text{U}}$, authorized_perms$^{\text{U}}[u]$, and For every r_1, $r_2 \in ROLES^{\text{U}}$. Grant operation grantRoleToRole(r_1, r_2) will be true if the granter has either [role, r_2, empower] and [role, r_1, grant] **or**, [role, r_2, empower] and [role, grant] **or**, [role, empower] and [role, r_1,

Algorithm 5.2 $\text{Map}_{\text{RRA-UARBAC}}$

```
Input: Instance of RRA in UARBAC
Output: ARRA instance
Step 1: /* Map basic sets and functions in ARRA */
    a. $AU^A \leftarrow USERS^U$ ; $AOP^A \leftarrow \{\textbf{assign, revoke}\}$ ; $ROLES^A \leftarrow ROLES^U$ ; $AUA^A = \phi$
    b. $RH^A \leftarrow RH^U$
Step 2: /* Map attribute functions in ARRA */
    a. $AATT^A \leftarrow \{grantAuth, empowerAuth, adminAuth, roleClassAuth\}$
    b. $ARATT^A = \{\}$, $RATT^A = \{\}$
    c. $Scope^A(grantAuth) = ROLES^A$ ; $attType^A(grantAuth) = set$
    d. $is\_ordered^A(grantAuth) = True$, $H^A_{grantAuth} = RH^A$
    e. For each $u$ in $AU^U$, $grantAuth(u) = \phi$
    f. For each $u$ in $USERS^U$ and for each $[\mathsf{role}, r, \mathsf{grant}] \in authorized\_perms^U[u]$,
            $grantAuth(u)' = grantAuth(u) \cup (r)$
    g. $Scope^A(empowerAuth) = ROLES^A$ ; $attType^A(empowerAuth) = set$
    h. $is\_ordered^A(empowerAuth) = True$ ; $H^A_{empowerAuth} = RH^A$
    i. For each $u$ in $AU^U$, $empowerAuth(u) = \phi$
    j. For each $u$ in $USERS^U$ and for each $[\mathsf{role}, r, \mathsf{empower}] \in authorized\_perms^U[u]$,
            $empowerAuth(u)' = empowerAuth(u) \cup r$
    k. $Scope^A(adminAuth) = ROLES^A$ ; $attType^A(adminAuth) = set$
    l. $is\_ordered^A(adminAuth) = True$ ; $H^A_{adminAuth} = RH^A$
    m. For each $u$ in $USERS^A$, $adminAuth(u) = \phi$
    n. For each $u$ in $USERS^U$ and for each $[\mathsf{role}, r, \mathsf{admin}] \in authorized\_perms^U[u]$,
            $adminAuth(u)' = adminAuth(u) \cup r$
    o. $Scope^A(roleClassAuth) = AM^U(\mathsf{role})$ ; $attType^A(roleClassAuth) = set$
    p. $is\_ordered^A(roleClassAuth) = False$ ; $H^A_{roleClassAuth} = \phi$
    q. For each $u$ in $USERS^A$, $roleClassAuth(u) = \phi$
    r. For each $u$ in $USERS^U$ and for each $[c, am] \in authorized\_perms^U[u]$,
            $roleClassAuth(u)' = roleClassAuth(u) \cup am$
Step 3: /* Construct assign rule in ARRA */
    a. assign_formula = $(r_1 \in grantAuth(au) \land r_2 \in empowerAuth(au)) \lor (r_1 \in grantAuth(au) \land$
      $\mathsf{empower} \in roleClassAuth(au)) \lor (\mathsf{grant} \in roleClassAuth(au) \land r_2 \in empowerAuth(au)) \lor$
      $(\mathsf{grant} \in roleClassAuth(au) \land \mathsf{empower} \in roleClassAuth(au))$
    b. auth_assign = $is\_authorizedU_{\textbf{assign}}(au : AU, r_1 : ROLES, r_2 : ROLES) \equiv$ assign_formula
Step 4: /* Construct revoke rule for ARRA */
    a. revoke_formula = $r_1 \in grantAuth(au) \land$
      $r_2 \in empowerAuth(au)) \lor r_1 \in adminAuth(au) \lor r_2 \in adminAuth(au) \lor$
      $\mathsf{admin} \in roleClassAuth(au)$
    b. auth_revoke = $is\_authorizedU_{\textbf{revoke}}(au_1 : AU, r_1 : ROLES, r_2 : ROLES) \equiv$ revoke_formula
```

grant] **or**, [user, empower] and [role, grant] permissions towards roles.

For each r_1, $r_2 \in ROLES^U$, revokeRoleFromUser(r_1, r_2) is true if the granter has either [role, r_2, empower] and [role, r_1, grant] **or**, [role, r_1, admin] **or**, [role, r_2, admin] **or**, [role, admin] permission on roles.

$\text{Map}_{\text{RRA-UARBAC}}$ yields an ARRA instance consisting of AU^A, AOP^A, $ROLES^A$, AR^A, AUA^A, RH^A, ARH^A, $AATT^A$, For each attribute $att \in AATT^A$, $Scope^A(att)$, $attType^A(att)$, is_ordered$^A(att)$ and $H^A{}_{att}$, For each user $au \in AU^A$, and for each $att \in AATT^A$, $att(au)$, Authorization rule for assign (auth_assign), and Authorization rule for revoke (auth_revoke).

In Step 1, primary sets from UARBAC's RRA are mapped to AURA equivalent sets. In Step 2, admin user attributes are defined. In UARBAC model, role-role assignment decisions are based on the admin user's *access modes* such as grant, empower and admin towards regular roles. For example, for each permission [role, *r*, grant] in authorized_perms$^U[u]$, *grantAuth*(*au*) attribute extracts set of roles towards which an admin user *au* has grant authority. In other words, admin user can grant those roles to any other object such as user and role. Similarly, *empowerAuth* yields set of roles to which an admin user has empower *access mode* and *adminAuth* yields sets of roles about which admin user has *admin access mode*. The meaning of each *access mode* is preserved by UARBAC model. For each class level attribute [*c*, *am*] in authorized_perms$^U[u]$, where *c* is a type of class and *am* is type of *access mode*, *roleClassAuth*(*au*) specifies the nature of class level *access mode* such as grant or admin admin user *au* has on role class.

In Step 3, an authorization formula equivalent to grantRoleToRole(r_1, r_2) in UARBAC is expressed. Equivalent expression is_authorized$\text{R}_{\textbf{assign}}$($au : AU^A$, $r_1 : ROLES^A$, $r_2 : ROLES^A$) checks if an admin user *au* has grant authority over role r_1 and empower authority over role r_2. Similarly, in Step 4 revoke_formula equivalent to revokeRoleFromRole(r_1, r_2) is constructed using revoke permissions given to admin user. It checks whether an admin user *au* has an authority to revoke roles r_1 and r_2.

5.3.8 An Example Instance For ARRA With Role Attributes

Previous ARRA model simulation examples did not include regular role attribute. In this section, a simple yet plausible example that demonstrates a use case for role attributes in assigning role to role is presented.

There are two admin users, **Sam** and **Tom**, and regular roles **IT Director, Development Manager, Quality Manager, Marketing Mganager, Finance Manager, Support Engineer** and **System Analyst**. There can be many other roles in an organization. Only few roles enough to illustrate a case have been considered. Admin user attribute *dept* maps each admin user to set of departments they have admin authority over. Although, it is likely that departments have hierarchy in practice, we have not included hierarchy for departments for simplicity. There are three departments in the organization, namely **Operations, Account** and **IT** departments. **Sam** has admin authority over all the departments while **Tom** has authority over **IT** department only. There is a regular role attribute *dept* for mapping regular roles to their departments.

Authorization condition is that if an admin user *au* with admin role *ar* was given authority over a department *d* and if roles r_1 and r_2 belonged to the same department *d* then r_1 can be assigned to r_2.

Basic Sets

- AU = {**Sam, Tom**}
- AOP = {**assign, revoke**}
- ROLES = {**IT Director, Devlopment Manager, Quality Manager, Marketing Manager, Finance Manager, Support Engineer, System Analyst**}
- AR = { }
- AUA = { }
- RH = { },

- ARH = { }

<u>Attributes definition</u>

- AATT = {*dept*},
- Scope(*dept*) = {**Operations, Account, IT**}, attType(*dept*) = set, is_ordered(*dept*) = False, $\mathrm{H}_{dept} = \phi$
- *dept*(**Sam**) = {**Operations, Account, IT**}, *dept*(**Tom**) = {**IT**}
- ARATT = { }, RATT = {*dept, level*}
- Scope(*dept*) = {**Operations, Account, IT**}, attType(*dept*) = atomic, is_ordered(*dept*) = False, $\mathrm{H}_{dept} = \phi$
- *dept*(**IT Director**) = **IT**, *dept*(**Dev. Manager**) = **IT**, *dept*(**Marketing Manager.**) = **Operations**, *dept*(**QA Manager**) = **IT**

<u>Authorization function for role-role assignment</u>

– is_authorized$\mathrm{R}_{\mathbf{assign}}(au : \mathrm{AU}, r_1 : \mathrm{ROLES}, r_2 : \mathrm{ROLES}) \equiv$

$$\exists d \in \mathrm{Scope}(dept).\ dept(au) = dept(r_1) = dept(r_2)$$

<u>Authorization function for role-role assignment</u>

– is_authorized$\mathrm{R}_{\mathbf{revoke}}(au : \mathrm{AU}, r_1 : \mathrm{ROLES}, r_2 : \mathrm{ROLES}) \equiv$

$$\text{is\_authorized}\mathrm{R}_{\mathbf{assign}}(au : \mathrm{AU}, r_1 : \mathrm{ROLES}, r_2 : \mathrm{ROLES})$$

An article on attribute-based role-role assignment was published in International Conference on Secure Knowledge Management 2017 (SKM2017) with following title, *ARRA: Attribute-Based Role-Role Assignment* [40]. A separate full description paper is also available in [42].

Chapter 6: IMPLEMENTATION OF AURA MODEL IN OPENSTACK CLOUD IAAS

Indentity service in OpenStack Cloud IaaS (or simply OpenStack) is known as Keystone [46]. In this chapter, the details involved in the implementation process of the attribute-based user-role assignment (AURA) model in Keystone is described. It includes the motivation behind the implementation, AURA design description as compared to OpenStack's original administrative RBAC design for user-role assignment, the outcome of implementation as an improved administrative access control architecture, the evaluation of any overhead introduced by the newly introduced code, and the actual step-by-step process used in modifying Keystone's code. AURA is introduced in implementation layer of OpenStack's stable version of Ocata release [50] [21].

6.1 Background

Cloud computing has been a *game-changing* technology from individual to business needs. Infrastructure as a Service (IaaS) technology has let the cloud service providers (CSPs) to supply with a package of computing resources with storage and software networking capabilities. Cloud computing as a service has become a new normal for doing business in recent years. Cloud service providers like Google (Alphabet), Microsoft and Amazon are current leaders to name a few. It is relatively hard to find any innovative company that do not use cloud platform to meet their needs. It is driven by economy, flexibility and scalability.

Since, AURA implementation is solely in the OpenStack cloud, there are few important terminologies specific to OpenStack cloud that will be used in rest of this article. A *domain* in OpenStack can be considered as an *area of authority*, which reflects an *organization* in a real world scenario. A project represents a unit of ownership [46] which means that all the resources are owned by a *project*. A project is contained in a *domain*. Technically, a *domain* is a higher-level container that contains projects, users and groups. All projects are unique to their domain. By default all the projects are added to a default domain known as 'Default'. A domain name must be globally unique, while role names, user names, group names and project names must be unique within its owning domain.

Keystone is responsible for user authentication and high-level authorization. It involves token based authentication and user-service (services for user) authorization.

This chapter presents a proof-of-concept implementation of the theoretical AURA model that was presented in Chapter 3. As it is the implementation regarding administrative operations, namely, user-role assignment and user-role revocation, the authorization policies and operations is scoped within Keystone.

The prime objectives of using access control features in administrative RBAC has been discussed in previous chapters. Chapter 3 depicted an attribute based approach and its capability of expressing combination of features in controlling access in assigning and revoking users to/from roles. This chapter takes into consideration the approach, and emulates the attribute-based policies discussed, as close as possible to the proposed theory in the context of OpenStack identity service.

6.2 Need for Attribute-Based Administrative Approach in IaaS Cloud

This section summarizes importance of having access control in administrative RBAC, its corresponding implications presented by the attribute-based approach and its implementation in the OpenStack's identity service for fine-grained access control so as to motivate its usage. The approach taken in the design of the AURA model virtually aligns with every design principle followed in developing UARBAC [37]. In addition, as shown in Chapter 3, AURA is capable of representing many other prior URA models. Here, a summary of the requirement of access control in administrative access control in IaaS cloud is presented.

6.2.1 Flexibility and Scalability

As motivated in Chapter 1, one of the key limiting factors of previous models in representing various features for access control were their limited flexibility, and hence the scalability. However, two core features offered by cloud computing are flexibility and scalability. Therefore, the access control engine in the cloud should be able to provide a wide variety of access control features for users [27]. The limitations of flexibility regarding access control posed by prior administrative models have already been observed. As access control needs may vary according to organization's (*project*) needs, the cloud administrative architecture must facilitate a framework that is flexible and scalable. For example, with large number of users, roles and permissions, the administrative models for RBAC must be flexible enough to scale with increasing size. Many times this requires decentralization of administrative domains and administrative-operational duties. Many different projects in the cloud may implement different approaches for user-role assignment. For example, in one project an admin may assign a user to a role based on the organization unit the user belongs to. In another project, a user may only be assigned to a role if the user has certain security clearance level and if the assignment is done within a certain time frame. These varying approaches demand a flexible architecture for user-role assignment. With AURA's approach, in one hand it is possible to define features for decentralization. On the other hand, it is also possible to customize the user-role assignment approach in different projects in the same domain or different projects in different domains running the same user-role administration architecture. In the following sections, it shall be observed how attributes are able to enforce decentralization based on administrative duties. Hence, it is fair to state at this end that attribute-based administrative approach can bring much more flexibility and scalability in RBAC administration.

6.2.2 Least Privilege

Principle of least privilege is one of the defining factors for access control system. It is a practice of limiting access to the minimum level so that it will allow only required action. For instance, an employee can have the lowest level of user rights and still can do their job. Administrative rights are relatively more critical to the system as administrative users bear more power than the regular users. Usually, a dedicated administrative user with an administrative role has an authority to assign a regular user to a regular role with an assumption that regular users and regular roles are within administrative user's access authority. In many cases, the cases may not be as simple.

There is a need for accurate user-role assignment, that is, assigning users to roles that allows only the intended roles (or collection of permissions) to intended users, need for avoiding mistakes from unknown side-effects during/by assignment and, only authorized users and given roles are allowed to conduct such assignments. For instance, if a user and roles are in the same domain but if a role can be granted only if the user must have certain level of *clearance*, the principle of least privilege can control critical permissions to be assigned to accurate users. This way, even if the user and roles are in same organization an additional requirement can control for accurate user-role assignment. OpenStack has adopted RBAC in its system. With AURA, it is possible to further control least privilege and elevate the level of assurance in providing accurate access.

6.2.3 Fine Grained Access Control

OpenStack offers coarse grained access control in operation and administrative levels. As the organization scales or even when an organization needs micro-level security needs, the number of administrators can increase, and hence their duties can vary. Flexibility, scalability as well as having confined and accurate least privileges demands fine-grained access control. AWS [3] has adopted limited fine-grained access control by considering time, location, address etc. Attribute-based approach can capture all that AWS can offer as well as many other features that were discussed in the previous chapters. Although this chapter discusses attribute-based approach in which only attributes of users and administrative users are taken into account, it is possible to combine attribute of other entities such as roles and environmental conditions to grant roles to users. Properties such as context-driven requirements, conflict of interest, separation of duties are some other powerful aspects important in access control design in the cloud environment. Attribute-based approach for RBAC has made these objectives viable to achieve.

6.2.4 Principle of Minimalism

Economy of mechanism is building a block for simplicity. Li et. al [37] assert that the idea of economy of mechanism is to keep the design as simple and small as possible. However, simplic-

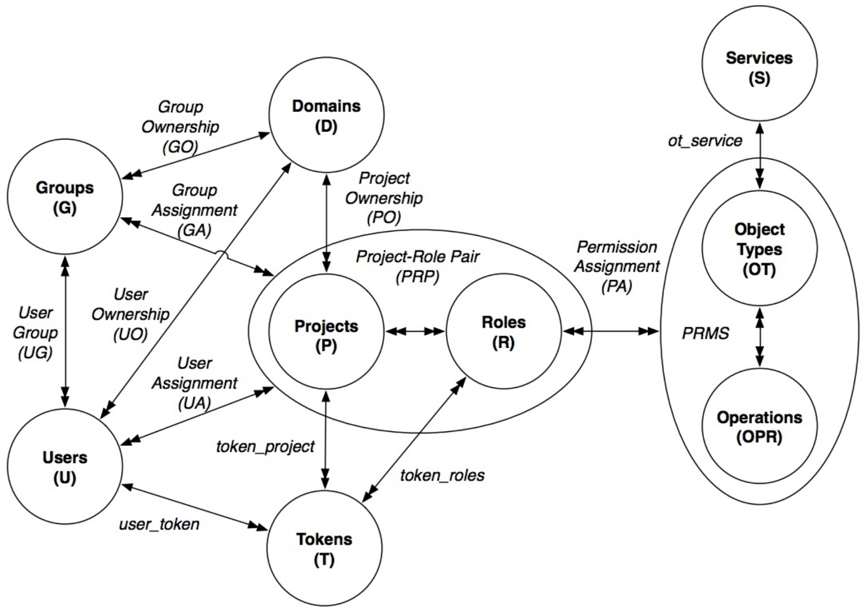

Figure 6.1: OSAC Model [61]

ity is again a relative term. With the emergence of the internet, and rapid growth of computing power the world has witnessed unprecedented ways of computing. Cloud is just one of them. This emerging architectures demand simple yet robust, minimal yet flexible and, scalable software architecture. Attribute-based RBAC approach presented in this dissertation has met that philosophy. Each model that have been presented in previous chapters are able to represent all the properties from corresponding prior models and have laid a framework where new features can be introduced. For instance, in AURA model attributes from only two entities, namely, users and admin users are taken into account and yet, it is able to specify all the properties used in user-role assignment decisions in past five models. Therefore, AARBAC approach brings in novel models that have minimum specification criteria and have an ability to address the scalability and flexibility needed in administration of IaaS cloud. OpenStack uses its unique version of RBAC as its basis for operation as well as administration. Attribute-based approach introduced by AURA model facilitates extended level of access control of URA for OpenStack with minimum extension.

6.3 OpenStack Access Control Overview

This section provides an overview of access control model in OpenStack cloud. A formal OpenStack access control (OSAC) model was introduced based on OpenStack Identity API v3 [47] as shown in Figure 6.1 by Tang et. al [61]. The components introduced in OSAC are core components. They include Users(U), Roles(R), Groups(G), Projects(P), Domains(D), Services(S), Operations(O), and Tokens(T). Each of these components are described briefly.

Users and Groups

User represents individual or user's name and its corresponding unique identifier. Each user can access cloud resources once authenticated and authorized. A group simply represents a collection of users. Users and Groups are unique to a domain.

Roles

Roles are global names within a domain used to associate users with projects. Unlike user-role assignment in general RBAC model, users are assigned to role and project at a time. In other words, a role-project pair is granted to a user. This means that a user must belong to a role in a project. For an administrative usage, a user can be assigned to role-domain pair as well. Roles are means by which users are given permissions. This is the only way in OpenStack cloud that users are given access authority.

Domains and Projects

A project represents a repository of resources in the cloud. A domain is a container of a project and other resources. In OpenStack, a project is unique to a domain. In other words, a project cannot belong to more than one domain. A user can belong to one domain but can be associated with multiple project-role pairs. To have access to the resources in a domain, a user must be granted with role-project pair. By default, a domain admin is an admin of projects in that domain.

Table 6.1: OSAC Model [61]

– U, G, P, D, R, S, OT, OP and T are finite sets of users, groups, projects, domains, roles, services, object types, operations and tokens respectively.
– user_owner : U $\rightarrow$ D, a function mapping a user to its owning domain. Equivalently viewed as a many-to-one relation UO $\subseteq$ U $\times$ D.
– group_owner : G $\rightarrow$ D, a function mapping a group to its owning domain. Equivalently viewed as a many-to-one relation GO $\subseteq$ G $\times$ D.
– project_owner : P $\rightarrow$ D, a function mapping a project to its owning domain. Equivalently viewed as a many-to-one relation P O $\subseteq$ P $\times$ D.
UG $\subseteq$ U $\times$ G, a many-to-many relation assigning users to groups where the user and group must be owned by the same domain.
– PRP = P $\times$ R, the set of project-role pairs.
– PERMS = OT $\times$ OP, the set of permissions.
– ot_service : OT $\rightarrow$ S, a function mapping an object type to its associated service.
– PA $\subseteq$ PERMS $\times$ R, a many-to-many permission to role assignment relation.
– UA $\subseteq$ U $\times$ PRP, a many-to-many user to project-role assignment relation.
– GA $\subseteq$ G $\times$ PRP, a many-to-many group to project-role assignment relation.
– user_tokens : U $\rightarrow 2^{T}$, a function mapping a user to a set of tokens; correspondingly, token_user : T $\rightarrow$ U, mapping of a token to its owning user.
– token_project : T $\rightarrow$ P, a function mapping a token to its target project.
– token_roles : T $\rightarrow 2^{R}$, a function mapping token to its set of roles.
Formally, token_roles(t) = $\{r \in R \mid (\text{token\_user}(t),(\text{token\_project}(t), r)) \in UA\} \cup (\bigcup_{g \in \text{user\_groups}(\text{token\_user}(t))} \{r \in R \mid (g, (\text{token\_project}(t), r)) \in GA\})$.
– avail_token_perms : T $\rightarrow 2^{\text{PERMS}}$, the permissions available to a user through a token,
Formally, avail_token_perms(t) = $\bigcup_{r \in \text{token\_roles}(t)} \{\text{perm} \in \text{PERMS} \mid (\text{perms}, r) \in PA\}$.

Tokens

After a user is authenticated, the identity service generates a token. A token is a representation of an authenticated identity of a user, which helps to grant authorization to user in specific domain and project. There are two types of tokens: scoped and unscoped tokens. Unscoped tokens are used for initial authentication to a particular domain. A scoped token that is specific to a project is obtained using the unscoped token from Keystone. The content of the token is encrypted using public key infrastructure (PKI) so that it remains intact during token transport process.

Services, Object Types and Operations

OpenStack and other cloud architecture is based on the service-oriented architecture. Therefore, any resource such as virtual machine or an application is delivered to the consumer as a service. The main service types in OpenStack include compute, identity, network, volume and image. In OpenStack cloud, compute service is known as Nova, identity service is known as Keystone, network service is called Neutron, block storage service is known as Cinder, and image service is known as Glance.

Objects are cloud resources. Object types can be different cloud resources such as virtual machines and storage volumes. An operation is an access technique for objects. Create, read, update and delete (CRUD) are most general operation in OpenStack as well. As shown in the diagram above, a permission is a combination of an operation and object. For example, “Create role” has a combination of an operation and object type.

Figure 6.1 depicts structure of OSAC and Table 6.1 presents corresponding formal OSAC model. OSAC model shows that OpenStack follows RBAC model in which there are specific role assignments: user assignment (UA), permission assignment (PA) and group assignment (GA). Assignment processes for user and group are identical. In both the cases role-project pair is granted to users. This way permission granted to a user is valid within the project. For instance, **Tim** can be assigned to a role **Engineer** in a project **Engineering Division**. In this case **Tim** acts as an **Engineer** only in **Engineering Division**.

6.3.1 Administrative OSAC model

As discussed, assignment information of user, roles, groups and projects are maintained by identity service in OpenStack called Keystone.

This dissertation deals with administration of RBAC. Therefore, administrative operations such as **assign** and **revoke** will be taken into account. Users, roles and projects will be target objects among which assignment or revocation operation is performed. Permission-role assignment policies in OpenStack are maintained separately in a policy file. This policy file for identity service

Table 6.2: UA Relation in OSAC

– PRP = P $\times$ R, the set of project-role pairs.
– UA $\subseteq$ U $\times$ PRP, a many-to-many user to project-role assignment relation.

specifies the permission to manage identities as well as assignments for any admin or regular roles. Permission-role assignment (PA) process in OpenStack is still a manual process which is done by editing the policy file in Keystone.

Administrative OSAC provides an abstract way to introduce administrative roles as desired. By default, there is one domain administrator for every domain in OpenStack, who is also a project administrator. By deafult, the admin role name is set to **admin**. This administrator can be regarded as a super-administrator. She can create a role, assign it with desired administrative permissions, and grant that role to a user, making that user an administrative user.

Table 6.1 presents formal model for OSAC. It also includes administrative relation for the user to project-role assignment given by set UA, which is a many-to-many user to project-role assignment relation. That means, a user can be associated with multiple role-project pair and vice-versa. Project-role pair is a cross product of project and role sets given by PRP. The two relations are separately re-expressed in Table 6.2. The OSAC model also presents administrative relation for permission to role assignment as PA. It is a many-to-many permission to role assignment relation. That is, a permission can be assigned to many roles and that a role can have many permissions. Group to role-project assignment relation is similar to UA, in that group behaves just like a user in assignment process. Group is nothing but a collection of users with a name (group name). Administrative relations present the outcome for administrative operations. Administrative models, however present procedures and control needed to be taken into consideration as presented in previous chapters.

6.4 Proof of Concept AURA Implementation in OpenStack

The objective of this chapter is to implement pre-presented AURA model from Chapter 3 as close as possible in the OpenStack's identity service. OSAC model presented an overview of Open-

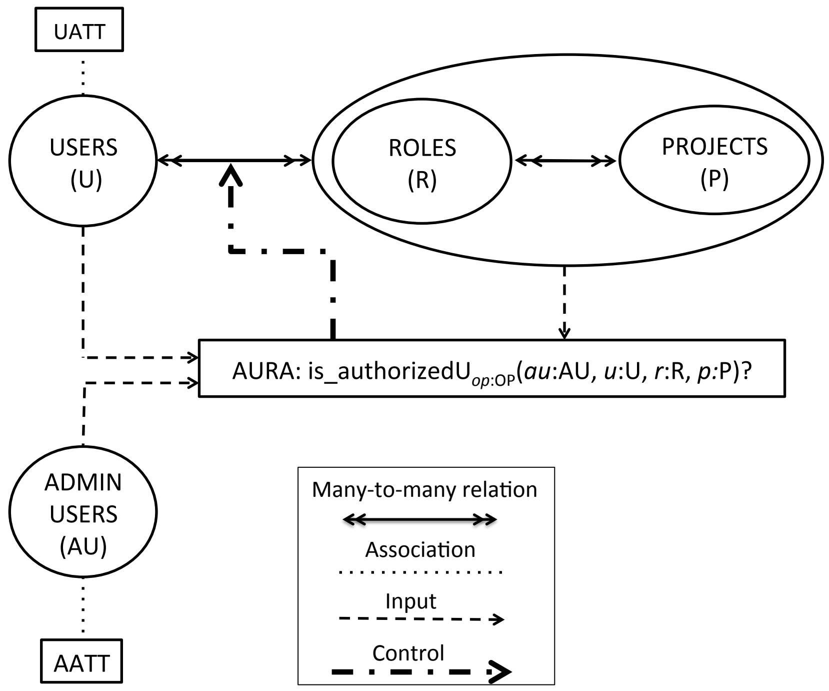

Figure 6.2: Attribute-Based User-Project-Role Assignment in OpenStack

Stack's access control model which included operational as well as administrative access control. However, administrative aspect of AURA model is scoped to granting user to role and, revoking user from a role. Therefore, this section scopes description and implementation towards the same.

Figure 6.2 shows a conceptual model for attribute-based user-role-project assignment in OpenStack. It shows a set of users (U) with user attributes (UATT), a set of admin users (AU) and its attributes (AATT), and a set of roles (R) linked with set of projects (P). The UA assignment decision is based on access control policy made up of attributes of user and admin user. The user to role-project assignment function takes admin user, target user, target role and target project as its input.

6.4.1 The Notion of Administrative Users in OpenStack

In AURA model administrative authority is given to administrative user unlike some prior models such as URA97 and URA02 where admin authority is given to admin roles. However, the notion of administrative user (admin user) doesn't exist in OpenStack. In other words, a user becomes an administrative user only by acquiring administrative permissions via some role (preferably admin role).

Therefore, to pivot AURA model according to the demand of implementing AURA in OpenStack, administrative users are differentiated from the regular users from same set of users by assigning users to an administrative role called **aura_admin**. **aura_admin** bears two key permissions: a permission to grant a user to role-project pair and, a permission to remove role-project pair from a user. It also bears other permissions that are necessary as a prerequisite for these two permissions such as as permission to list users, a permission to list roles, a permission to list projects and permissions to view their corresponding associations, if any. Table 6.5 shows permission assigned to **aura_admin**. For instance, during implementation, users **Sam** and **Ken** were assigned to **aura_admin** role that officially made them admin users.

6.4.2 Access Control Policy Management in OpenStack

For each service in OpenStack, authorization is enforced by Policy Enforcement Point (PEP). For instance, Nova, Neutron and Keystone have their own policy enforcement points. Because AURA Implementation is limited to administrative operations, the service is scoped to Keystone. Keystone stores identity information for all the entities including users, roles, projects and user assignments. Keystone provides user information in the form of a token. A token is a signed by Keystone using private key. In addition, public key information is shared among trusted components only. Trusted components authorize user once the user information is verified by decoding the token. Each component has its own policy enforcement point (PEP), policy information point (PIP), policy administration point (PAP) and, policy decision point (PDP) [27]. The PEP is responsible for receiving access requests and enforcing access control decisions; policy decision point (PDP) accepts requests from PEP, and evaluates the request based on proper access control policy that it gets from the PAP and; policy information point (PIP) is responsible for looking up additional information involving the subjects, objects and the system environment [38]. In many cases, some of these policy points are cached together, or even all PEP, PDP and PIP can be cached in a single

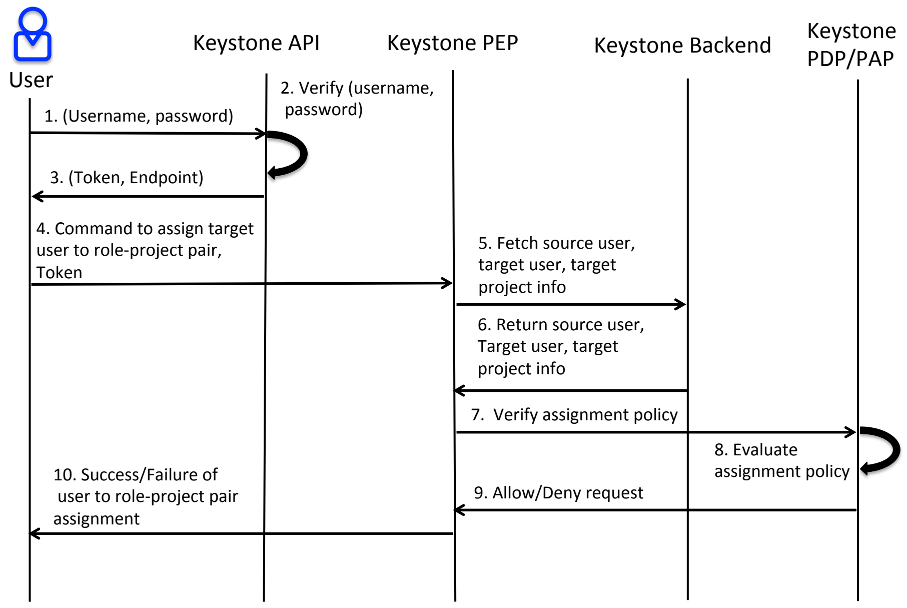
User
Keystone API
Keystone PEP
Keystone Backend
Keystone PDP/PAP
1. (Username, password)
2. Verify (username, password)
3. (Token, Endpoint)
4. Command to assign target user to role-project pair, Token
5. Fetch source user, target user, target project info
6. Return source user, Target user, target project info
7. Verify assignment policy
8. Evaluate assignment policy
9. Allow/Deny request
10. Success/Failure of user to role-project pair assignment

Figure 6.3: Keystone Authorization

point, to provide with faster service.

Figure 6.3 shows keystone authorization process with a user to role-project pair assignment (UA) use case. A user is authenticated with the Keystone via by sending request to Keystone API. After verifying requesting user with her Username and Password, user is granted a token to communicate with specific endpoint (Keystone for this example). Using token, the source user sends a command to assign a target role and target project pair to a target user to PEP. Upon receiving the request, PEP gathers the requested information from Keystone backend and returns them back to PEP. Based on the information PEP gets, it requests PDP/PAP to verify the assignment policy in place. PDP/PAP evaluates the assignment policy against the source user, target user and target project. It then sends back decision about authorization to PEP. Based on the success or failure of the command, the PEP enforces the policy in place and send back the message about the same to the source user.

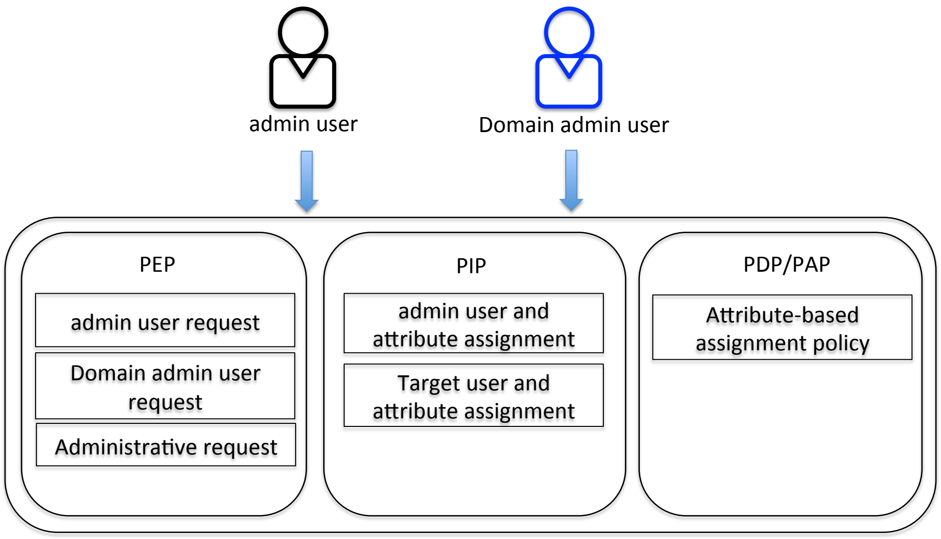
admin user
Domain admin user
PEP
admin user request
Domain admin user request
Administrative request
PIP
admin user and attribute assignment
Target user and attribute assignment
PDP/PAP
Attribute-based assignment policy

Figure 6.4: AURA Authorization Enforcement

6.4.3 AURA Enforcement Model

The organization of AURA authorization enforcement in Keystone is depicted in Figure 6.4. It shows that domain admin user and admin user with user-role assignment authority can conduct UA. Keystone stores regular user and administrative user identities, user and admin user attribute definitions, user and admin user attribute assignments. An admin user is authenticated though Keystone who initiates a request to grant role-project pair to a regular user. Based on the user and admin user ids received in the policy enforcement point (PEP), Keystone fetches target user and admin user information from the database in the Keystone backend. Additional information regarding defined attributes and assigned values for target user and admin user is fetched from attribute assignment file/database. This location is referred to as policy information point (PIP). Keystone then references the administrative user to user-role-project assignment policy and checks against the information gathered from the database at the policy decision point (PDP). As per the policy decision, the result is sent back to PEP where it is enforced.

6.5 A Practical Scenario

Consider a commercial technology oraganization called **DigiEvo**. DigiEvo has four main departments, namely Innovation, Legal, Accounts, Human Resources. Each department has their own sub-divisions and teams within them. For simplicity, lets call each department, a *project* in OpenStack. **DigiEvo** is a *domain*. There is one super administrator (super admin) role called **admin**, and an admin user called **admin** by defalut in OpenStack. This administrator (admin) role can be called as Domain admin.

A user with Domain admin role is responsible for adding new users in the domain, unless that task is dedicated to some other administrator role. Any user assigned to the Domain admin role gets the super administrator privilege. In one hand, all other administrative roles can be created by the Domain admin, and administrative permissions assigned to administrator roles, accordingly. On the other hand, Domain admin can also create regular roles with operational permissions. In a company simple as this, there may exist hundreds of thousands of users. There could exist hundreds of roles and permissions. Administrative operations such as user-role-project assignment can be non-trivial. Therefore, there is a need for administrative tasks delegation. Hence the need to create administrators junior to Domain admin with distributed privileges. In other words, decentralization of administration. For instance, one way of decentralization can be based on the department. A department itself is defined based on the tasks it involves with. A user in Legal department in most cases has different permissions (hence, different roles) than a user in the Innovation department. Therefore, the access control design should be able to address at least the following objectives:
1) Make sure that user-role-project assignment is accurate. 2) Admin users junior to Domain admin have necessary permissions only 3) An admin user can perform administrative operations towards proper project, role and users.

DigiEvo, has a Domain admin named, Ken. Ken is the super admin of the organization. Since, it is a huge task to handle administrative duties for different types of departments, for four different projects, he creates an administrative role for user-role-project assignment only. He calls this admin role as **aura_admin**. There exists another admin for permission-role assignment. **aura_admin**

role is given or assumed to have a permission to grant user to role-project pair; a permission to remove user from existing user-role-project relation; and other related permissions such as listing users and projects.

Any user assigned to aura_admin role becomes an admin user for UA. However, OpenStack consists of a single pool of users. As mentioned in section 6.4.1, the way by which admin users are virtually decoupled from user pool is by assigning these users to **aura_admin** role. Lets assume there are four different admin users, Sam, Gina, Will and Kat. Each of these admins are responsible administering users-role assignment in each of the projects. The fact that admin users are associated with particular project(s) is defined using attributes. This is done using the process of attribute assignment.

According to AURA model, there are only two entities that may have attributes. They are regular users (or simply users) and admin users. Therefore, attributes must be defined around these two entities. Lets assume there are three different attributes for each entities, which need to satisfy according to assignment policy defined. Assignment policies will be explained shortly. Let attributes of users be : *admin_unit*, *clearance*, *location* and attributes of admin users be *admin_unit*, *location*, *admin_roles*.

Domain admin user defines the values for admin users and users. Let the values for attributes be defined as follows for admin users,

admin_unit(*admin_user*) ={innov, legal, hr, accounts}

location(*admin_ user*) = {austin, san_antonio, dallas}

admin_roles(*admin_ user*) = {sr_admin, sec_admin}

Similarly, for regular users,

admin_unit(*user*) ={innov, legal, hr, accounts}

location(*user*) = {austin, san_antonio, dallas}

clearance(*user*) = {confidential, unclassified, classified}

Domain admin also makes attribute assignment for admin users and users. An example for attribute assignment based on json format is presented in Table 6.3: This example simply means admin user

Table 6.3: An example for JSON format attribute assignment

```
[
    {
        "entity": "admin",
        "id": "sam",
        "admin_unit": ["accounting"],
        "location": ["san_antonio"],
        "admin_roles": ["sec_officer"]

    },          { "entity": "user",
        "id": "john",
        "admin_unit": ["accounting"],
        "location": ["dallas"],
        "clearance": ["classified"]
    }
...
]
```

Sam has the following admin user attributes and corresponding attribute (range of) values: admin_unit is accounting, location is san_antonio, and admin_roles is sec_officer. Similarly, regular user John has admin_unit equal to accounting, location is dallas, and clearance is classified.

For instance, if Sam requests to assign target role chief accountant to regular user john in *admin_unit* accounting, the assignment process has to check for assigned attributes and its values, fetch the attributes and its values, and check against the attribute-based assignment policies. An example of attribute-based assignment policy is given in Table 6.4. The aforementioned access control policy is placed as a json. It is a json array of json objects. Each json object in json array is one policy for UA. For intance, first json object indicates a policy which means: if any admin user who has admin_unit of accounting, location is san_antonio and has admin roles sr_sec_officer and/or security_officer; and if the target user has same admin unit, accounting, same location san_antonio and has a user clearance top_secret, then that user can be assigned to roles sr_accountant or auditor or both in an existing project in a domain.

Table 6.4: An example of attribute-based assignment policy

```
[
    {
        "admin": {
        "admin_unit": ["accounting"],
        "location": ["san_antonio"],
        "admin_roles": ["sr_sec_officer"]
    },
        "user": {
        "admin_unit": ["accounting"],
        "location": ["san_antonio"],
        "clearance": ["confidential"]
    },
        "role": [ "chief_accountant",
                  "sr_accountant",
                  "auditor" ]
    },
    {
        "admin": {
        "admin_unit": ["accounting"],
        "location": ["san_antonio"],
        "admin_roles": ["sr_sec_officer",
                  "security_officer" ]
    },
        "user": {
        "admin_unit": ["accounting"],
        "location": ["san_antonio"],
        "clearance": ["top_secret"]
    },
        "role": [ "sr_accountant",
                  "auditor" ]
    },
...
...
]
```

User-role-project assignment command in OpenStack must provide the target user, target role and target project. The user-role-project assignment command is sent to the keystone. Assigned attributes for admin user and target user is extracted and sent for evaluation. The values for corresponding attributes for admin user and target regular user along with the target project and target role information is checked against the attribute-based assignment policy and then authorization decision is made to true or false.

6.6 Experimental Setup and Application

This section presents a closer look on how the actual experiment was set up for proof of concept implementation. A developer environment DevStack was installed that emulated OpenStack cloud software. In particular, an all-in-one DevStack running OpenStack Ocata with 8 GB of RAM and 4 vCPU's with the basic services was installed. The AURA model involves entities like users and roles, and doesn't involve any resources (objects) like files. It can be observed from OpenStack's latest architechture [50] that there is an independent block for authorization called Keystone. These blocks are also referred to as nodes. Other nodes serve their respective objectives. For example Nova block serves in starting or ending virtual machines, Neutron provides networking services, Cinder serves to provide with virtual storage volumes and so on. Other services except for Keystone can be treated as objects or resources. Keystone keeps track of actors involved in accessing resources in OpenStack. User-role-project assignment in OpenStack, hence involves only Keystone service block.

OpenStack's recent stable version called Ocata was selected to carryout experiment. Complete code for OpenStack's Ocata release is available in GitHub [50] and Keystone code is available at [21]. After an admin user authenticates with keystone, she gets a token (a token granting token) to conduct operation. This token is sent with a request to Keystone. Based on the token and the kind of request, keystone grants another token with respect to the service node. In the case of URA, the only node involved is Keystone. Therefore, the token to passed back and forth between keystone and keystone.

The general methodology by which any user-role-project assignment or revocation happens in Keystone (OpenStack) follows a process. In this process, once requesting entities are authenticated, the administrative operation such as grant for UA is sent to keystone. A user-role-project assignment command is as follows:

```
$openstack role add –user <USER_NAME> –project <TENANT_ID> <ROLE_NAME>
```

This command issuing entity is some admin user. In a most early phase, it is the Domain admin user. The identity information of admin user is separate from the target user, role and project. The information on admin user ID, domain it belongs to, the project ID for admin user, admin role the admin user has, etc., are packaged together as a python dictionary block, sometimes very casually referred to as token. The request command is received at Policy Enforcement Point (PEP). Evidently, these information on admin user and user are extracted from back-end database system. In the case of admin user, her information can be traced as admin user information token at the following file in Keystone (Ocata Release):

```
</opt/stack/keystone/keystone/common/authorization.py>
```

The token with admin user information is referred to as ***credentials***. On the other hand, the target user and target role are received following file of the PEP:

```
</opt/stack/keystone/keystone/assignment/controllers.py>
```

Similar to ***credentials*** token, this controllers.py file contains a block called ***target***. This token contains information on target user (regular user), such as user id, domain that user belongs to, any role that user has, etc. In addition, it also contains the target role name to which this user is supposed to be assigned to.

The information on admin user and, target user and role from aforementioned two files in PEP is sent to Policy Decision Point (PDP) where the user, admin user and role credentials are received and checked against the existing OpenStack's user-role-project assignment policy. The PDP file in the case of Ocata release is as follows:

```
</opt/stack/keystone/keystone/policy/backends/rules.py>
```

After identifying the necessary file in PEP, attributes and their values defined for admin user and regular user were put into place by Domain admin user. Attributes assigned for both admin user and user were placed in a JSON file called:

</opt/stack/keystone/keystone/common/assigned_attributes.json>

An example for attribute assignment in JSON format was presented in previous section. This file emulates a database which contains a relation among entities, attributes and their assigned values.

As mentioned previously, in authorization.py file, admin user information is populated in a token container called *credentials*. Some lines in this authorization.py file were modified so that it is able to extract the attribute values for admin user from assigned_attributes.json file. A short program which extracted proper assigned attributes and attribute values were saved as :

<opt/stack/keystone/keystone/common/aura_attributes.py>

This program extracts attribute names and their corresponding values from assigned_attributes.json file. These attributes and their values are then appended into the *credentials* and sent towards PDP. The PDP involves the evaluation of the access control policies that are in place. In addition to existing access control policies for granting or revoking user-role-project, codes that check the attribute-based policies for granting or revoking user-role-project are introduced in the file below:

</opt/stack/keystone/keystone/policy/backends/rules.py>

As the interpreter goes through the code that verifies the policies, the attribute-based policies are read from the policy file below:

</opt/stack/keystone/keystone/policy/backends/attribute_policy.json>

An example for attribute-based policy for granting or revoking user-role-project was presented above. It should be noted that same policies exist for both granting and revoking user to role-project pair. The request is either granted or rejected based on the attributes and their assigned values as well as previously defined policies. Depending upon the decision made at rules.py, the PEP enforces the decision made at the PDP.

Codes after the implementation of AURA in OpenStack is available publicly at GitHub [12].

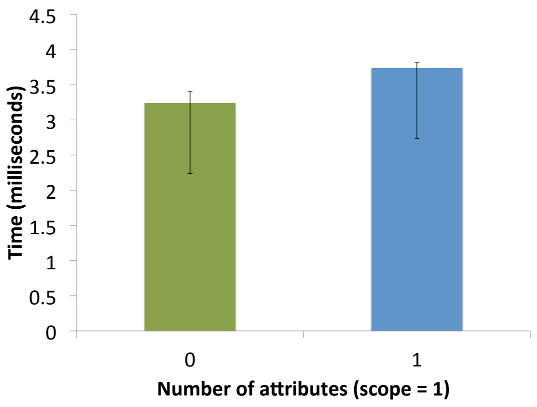

Figure 6.5: Avg. URA Time Comparison

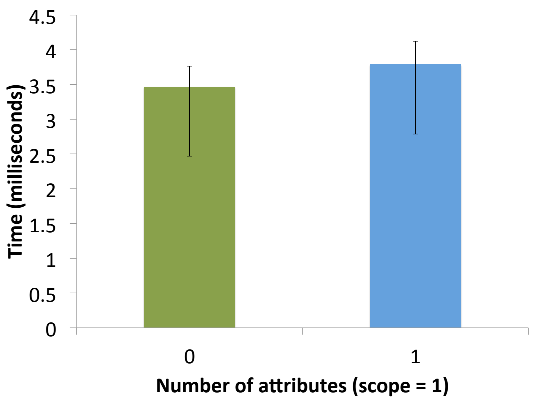

Figure 6.6: Avg. URR Time Comparison

6.7 Performance Analysis

This subsection discusses time performance evaluation after attributes and attribute-based policy were enforced. This implementation and experiments were carried out in OpenStack development environment called DevStack [16]. All the core OpenStack services along with Keystone were running in the same virtual machine. All the administrative operations involved in the experiment as presented by AURA model that were implemented in OpenStack involved modification of Keystone codes only.

Admin user makes request towards Keystone to grant target role and target project pair to the target regular user. As mentioned earlier, OpenStack doesn't have a notion of admin users. However, it is possible to create an admin user based on the model requirement. One way by which admin user could be created was to assign regular users with the project name to an administrative role (**aura_admin**).

Desired attributes were engineered and attribute assignments were done. The code in Keystone was modified in such a manner that any request for granting user to role-project pair or a request for removing user from role-project pair would consult Keystone backend. The request then received admin user's and target regular user's assigned attributes and their values, populated as a token and sent towards policy decision code along with the target role and project names. At the receiving end, the attributes and attribute values for admin user and target regular user, and role name along with project information were checked against the Keystone's existing policy. It also checked for attribute-based policies introduced and, either authorized or denied authorization for user-role-project assignment and user-role-project revocation.

6.7.1 Evaluation

Evaluation for time performance was carried out for policy decision and enforcement segment for three different cases. For simplicity, user to role-project assignment is called user-role assignment (URA) and revoking user from role-project pair is called user-role revocation (URR). First off, original OpenStack command was run for the process of granting role-project pair to user, i.e., without introducing any new code, and average time with at least 10 runs, each for granting and revoking role, along with standard deviation were recorded. Secondly, code that would enforce the policy for one attribute with one value (scope = 1) was introduced and time periods were recorded. These steps were repeated for URR. Average decision-making time taken and corresponding standard deviation values were recorded for this step as well. Figure 6.2 shows the graph for attribute-based user-role assignment in OpenStack with and without a single attribute with single value in its scope. Similarly, Figure 6.5 and 6.6 show a corresponding graphs for comparing time taken in OpenStack with and without a single attribute with single value in its scope for URA and URR, respectively. In both the cases, It shows that negligible time (about 1 milliseconds) was added in policy decision-making.

In the second part of experiment, average time was measured by increasing the number of attributes from 3 attributes to 30 attributes. Figures 6.7 and 6.8 show such times for attribute-based URA and attribute-based URR. It can be observed from these figures that even with increasing attributes the attribute-based policy decision and enforcement time stayed almost the same. At most, it increased with upto 2 milliseconds with minimum error. In addition, the error appeared to be less on the higher end of the average time, and time in most of the runs fell below the average value of time in both URA and URR.

Finally, in the third part of the experiment, both the dimensions of attributes were varied.

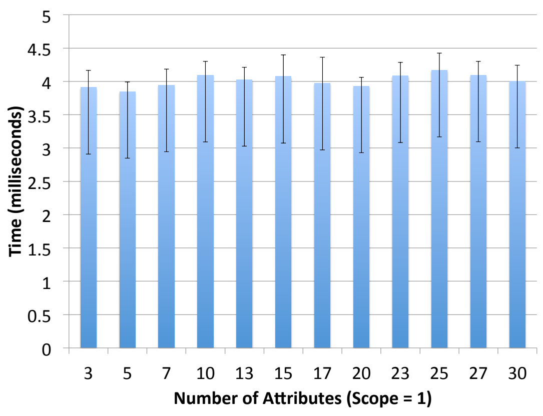

Figure 6.7: Avg. URA Time with Varying No. of Attributes

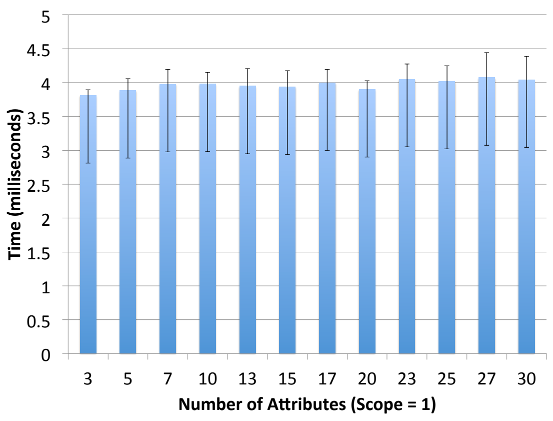

Figure 6.8: Avg. URR Time with Varying No. of Attributes

That is, the number of attributes and range of scope for each attribute was varied and times were recorded. Figures 6.9 and 6.10 depicts graphs for attribute-based URA and attribute-based URR. The number of attributes was varied from 1 through 30 with some interval. At the same time, range of scope was varied from 1 to 20 with an interval. A slow increase in policy decision and enforcement times can be observed in both the graphs. Even in an extreme case with 30 attributes and each attribute with a scope of 20 shows only a slight increase in time. For attribute-based user to role-project assignment, the highest average time for extreme case was 5.501461029 ms. For attribute-based user to role-project revocation, it came to 5.742168427 ms. The lowest times being 3.733801842 ms and 3.787136078 ms for URA and URR respectively, with an increase of up to 1.955 ms.

There existed some outliers in each of the previous experiments. They are ignored with an assumption that those high numbers were impacted by cold start or overall organization of computer.

It should be noted that all the experiments were conducted for positive policies only. For example, one of the qualifying policies for attribute-based URA was if the admin user had administrative unit and location given by attributes admin_unit and location respectively, same as regular user (admin_unit = finance and location = austin) and, admin user also had certain administrative role represented by attribute admin_role (for example, admin_role = sr_security_officer) and that regular user had met certain clearance represented by attribute clearance (e.g. clearance = clas-

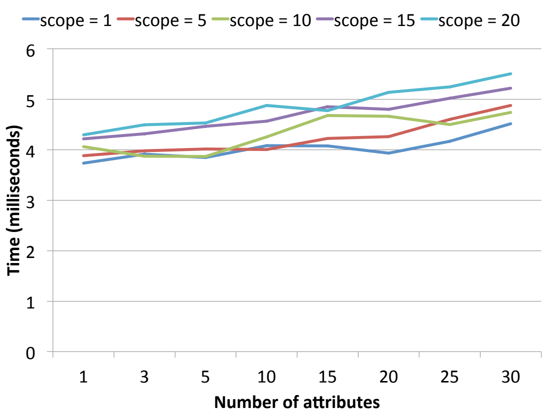

Figure 6.9: URA Time with Varying Attributes and Scope

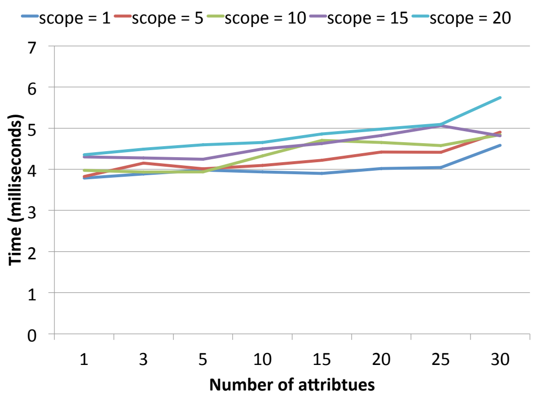

Figure 6.10: URR Time with Varying Attributes and Scope

sified), then the policy returned true and authority was granted. Experiments were not conducted for negative policies. It would be interesting to explore combination of positive and negative rules.

6.7.2 PRA and RRA in OpenStack

This dissertation explores three different attribute-based administrative models: AURA, ARPA and ARRA. Previous sections in this chapter presented a successful implementation of AURA model in OpenStack. OpenStack also involves permission-role assignment (PRA). PRA process in OpenStack is slightly different than URA process. Permission to role assignment is done manually by a Domain admin user. Permissions are predefined in Keystone and the relation among roles and permissions are manually linked in a proper format in a policy file known as policy.json. In OpenStack Ocata, it is positioned as:

</etc/keystone/policy.json>

Like URA assignment process, one can expect an automatic process of permission role assignment which OpenStack currently doesn't support. A new method of attribute-based approach in manual permission-role assignment process could also be incorporated. However, Keystone (OpenStack) may need to be redesigned in many aspects. For example, the way attributes and its values could be introduced or designing optimal set of commands. These needs appear far from scope and objectives. Therefore, its implementation is scoped out. However, a scenario where

Table 6.5: Permissions assigned to **aura_admin** role

```
"identity:list_domains": "rule:admin_required or role:aura_admin",
"identity:get_project": "rule:admin_required or project_id:%(target.project.id)s or role:aura_admin",
"identity:list_projects": "rule:admin_required or role:aura_admin",
"identity:get_user": "rule:admin_or_owner or role:aura_admin",
"identity:list_users": "rule:admin_required or role:aura_admin",
"identity:get_credential": "rule:admin_required or role:aura_admin",
"identity:list_credentials": "rule:admin_required or role:aura_admin",
"identity:get_role": "rule:admin_required or role:aura_admin",
"identity:list_roles": "rule:admin_required or role:aura_admin",
"identity:create_grant": "rule:admin_required or role:aura_admin",
"identity:revoke_grant": "rule:admin_required or role:aura_admin",
"identity:list_role_assignments": "rule:admin_required or role:aura_admin"
```

permission-role assignment was appropriate and, did appear as a part of AURA implementation. As explained in 6.4.1, there was a need to separate admin users from regular users in user pool. For that matter, users were assigned to an admin role called **aura_admin**. All the required permissions for user-role-project assignment were manually assigned to this **aura_admin** role. This illustrates that users with admin authority could be separated from regular users based on the administrative permissions. Actual permissions that were assigned to **aura_admin** role that made all the users assigned to it as admin users (with admin authority only for URA) is presented in Table 6.5. The list of permissions show that there are many permissions apart from these two:
"identity:create_grant": "rule:admin_required or role:aura_admin" and
"identity:revoke_grant":"rule:admin_required or role:aura_admin",
which are primarily for granting and revoking UA. However, other permissions such as "identity:get_user": "rule:admin_or_owner or role:aura_admin" are required for automatic query for Keystone when for instance, role name to role id translation is required. Some of them are for convenience purposes.

Role-role assignment (RRA) is an essential feature offered by RBAC model. Assigning a role to another role creates a hierarchy. Essentially, when a role is assigned to another role, the general assumption is that the role that was assigned becomes a junior role and the role to which another role is assigned become senior to the role assigned. Figure 5.1 shows an example of role hierarchy in RRA97 model. But, eventually the permissions from junior role is inherited by the senior role. That is, any user assigned to senior role becomes an implicit member of junior roles. In other words, the user indirectly owns junior roles. Therefore, creating a hierarchy greatly reduces the time and effort for URA and/or PRA processes.

Although, OpenStack's identity management system is based on RBAC model, it has not defined a framework for role-role assignment. One of the objectives of this dissertation was to administer existing RBAC model in more flexible, scalable and secure manner by leveraging the power of attributes. Hence, RRA in OpenStack remains out of scope for implementation of ARRA model.

Chapter 7: CONCLUSION

Attribute-based access control policies has been studied quite thoroughly in operational aspect of access control. However, investigation of attributes in the administrative access control domain has been subtle. This research work was focused on exploring, and leveraging the power of attributes to bring in flexibility and scalability in the administrative role-based access control.

In this dissertation, novel family of models for attribute-based administration of role-based access control (AARBAC) was presented. In particular, it consisted of a model for Attribute-Based User-Role Assignment (AURA), a model for Attribute-Based Permission-Role Assignment (ARPA) and, a model for Attribute-Bases Role-Role Assignment (ARRA).

One of the motivations behind approach utilized for AURA, ARPA and ARRA model designs was to make each of these models sufficient enough to represent their respective prior ARBAC models. For instance, AURA can represent any instance of URA97, URA02, URA99, Uni-ARBAC's URA and UARBAC's URA models, or virtually any combination of these five models. In the process, a relevant example instance from each prior model was taken, and corresponding equivalent instance in attribute-based approach for respective model was demonstrated. To further underscore the idea, translation algorithms that would take any instance from respective prior model and translate into their corresponding attribute-based assignment model were exhibited. These models are not only capable of expressing prior ARBAC models but carry potential to express new features.

Finally, the AURA model was implemented as a proof of concept in a development environment (DevStack) for OpenStack IaaS Cloud's identity service called Keystone. The procedure consisted of implementing policy based on user and administrative user attributes and making assignment decisions. On top of an existing policy enforcement point, new policies that are based on attributes were integrated; related codes for populating relevant attribute information into dedicated tokens were introduced and; time performances for varying number of attributes and, number of attribute values were examined. With an intention to stretch the limits due to addition attributes, up to 30 attributes with up to 20 values for each attribute were placed and measured for time analysis. The results were satisfactory in that even with considerably large number of attributes with large number of attribute values, very negligible time overhead of up to (about) 2 milliseconds was added. This signifies that even with introduction of attributes and additional complexity in the code, the overall performance is almost intact.

7.1 Future Work

This dissertation addresses a powerful approach on how to do user-role assignment, how to do permission-role assignment and, how to do role-role assignment in a highly flexible manner. In addition, addresses and subsumes plausible prior ARBAC models with concise yet extensible attribute-based approach. However, it does not cover many credible ARBAC approaches like SARBAC [13]. Some of the models are cited in Chapter 2. It would be interesting to explore attribute-based approach to cover these models.

OpenStack software explored in this research does not have an automatic approach for permission-role assignment. Designing an attribute-based automatic approach for permission-role assignment can an interesting implementation task.

Many entities were engineered as attributes of other entities. There may be a potential in exploring attributes of attributes in developing models and designing access control policies.

BIBLIOGRAPHY

[1] Mohammad A Al-Kahtani and Ravi Sandhu. A model for attribute-based user-role assignment. In *Proc. of the 18th Annual Computer Security Applications Conference*, pages 353–362. IEEE, 2002.

[2] Asma Alshehri and Ravi Sandhu. Access control models for cloud-enabled internet of things: A proposed architecture and research agenda. In *The 2nd IEEE International Conference on Collaboration and Internet Computing*, pages 530–538. IEEE, 2016.

[3] Amazon AWS. https://aws.amazon.com/. Accessed: 2017-11-23.

[4] Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. An access control framework for cloud-enabled wearable internet of things. 2016.

[5] Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. An attribute-based access control extension for openstack and its enforcement utilizing the policy machine. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 37–45, San Jose, USA, 2016. IEEE.

[6] Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. ABAC with group attributes and attribute hierarchies utilizing the policy machine. In *Proc. of the 2nd ACM Workshop on Attribute-Based Access Control*, pages 17–28. ACM, 2017.

[7] Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. An access control framework for cloud-enabled wearable internet of things. In *Collaboration and Internet Computing (CIC), 2017 3rd International Conference on*. IEEE, 2017.

[8] Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. Access control model for AWS internet of things. In *International Conference on Network and System Security*, pages 721–736. Springer, 2017.

[9] Prosunjit Biswas, Farhan Patwa, and Ravi Sandhu. Content level access control for openstack swift storage. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 123–126. ACM, 2015.

[10] Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan. Uni-ARBAC: A unified administrative model for role-based access control. In *International Conference on Information Security*, pages 218–230. Springer, 2016.

[11] Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan. Attribute transformation for attribute-based access control. In *Proc. of the 2nd ACM Workshop on Attribute-Based Access Control*, pages 1–8. ACM, 2017.

[12] Modified Keystone code. https://github.com/jiwanlimbu/aura. Accessed: 2017-11-12.

[13] Jason Crampton. Administrative scope and role hierarchy operations. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 145–154. ACM, 2002.

[14] Jason Crampton and George Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Transactions on Information and System Security (TISSEC)*, 6(2):201–231, 2003.

[15] Frédéric Cuppens and Alexandre Miège. Administration model for Or-BAC. In *OTM Workshops*, pages 754–768. Springer, 2003.

[16] DevStack. http://www.devstack.org/. Accessed: 2017-10-30.

[17] David Ferraiolo, Janet Cugini, and D Richard Kuhn. Role-based access control (rbac): Features and motivations. In *Proceedings of 11th annual computer security application conference*, pages 241–48, 1995.

[18] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions Information System Security*, 4:224–274, August 2001.

[19] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[20] L. Fuchs, G. Pernul, and R. Sandhu. Roles in information security - A survey and classification of the research area. *Computers & Security*, 30:748 – 769, 2011.

[21] Keystone Code (GitHub). https://github.com/openstack/keystone. Accessed: 2017-11-1.

[22] G Scott Graham and Peter J Denning. Protection: principles and practice. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, pages 417–429. ACM, 1972.

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

[24] Vincent C Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (ABAC) definition and considerations. *NIST special publication*, 800(162), 2014.

[25] Vincent C Hu, D Richard Kuhn, and David F Ferraiolo. Attribute-based access control. *Computer*, 48(2):85–88, 2015.

[26] Vincent C Hu, D Richard Kuhn, and David F Ferraiolo. Attribute-based access control. *Computer*, 48:85–88, 2015.

[27] Xin Jin. *Attribute-based access control models and implementation in cloud infrastructure as a service*. PhD thesis, The University of Texas at San Antonio, 2014.

[28] Xin Jin, Ram Krishnan, and Ravi Sandhu. A role-based administration model for attributes. In *Proceedings of the First International Workshop on Secure and Resilient Architectures and Systems*, pages 7–12. ACM, 2012.

[29] Xin Jin, Ram Krishnan, and Ravi Sandhu. A unified attribute-based access control model covering DAC, MAC and RBAC. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 41–55. Springer, 2012.

[30] Xin Jin, Ram Krishnan, and Ravi Sandhu. Role and attribute based collaborative administration of intra-tenant cloud IaaS. In *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 261–274. IEEE, 2014.

[31] Xin Jin, Ram Krishnan, and Ravi Sandhu. Role and attribute based collaborative administration of intra-tenant cloud iaas. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2014 International Conference on*, pages 261–274. IEEE, 2014.

[32] Xin Jin, Ravi Sandhu, and Ram Krishnan. RABAC: role-centric attribute-based access control. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 84–96. Springer, 2012.

[33] James BD Joshi, Elisa Bertino, and Arif Ghafoor. Hybrid role hierarchy for generalized temporal role based access control model. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 951–956. IEEE, 2002.

[34] Axel Kern, Andreas Schaad, and Jonathan Moffett. An administration concept for the enterprise role-based access control model. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 3–11. ACM, 2003.

[35] D Richard Kuhn, Edward J Coyne, and Timothy R Weil. Adding attributes to role-based access control. *Computer*, 43:79–81, 2010.

[36] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[37] Ninghui Li and Ziqing Mao. Administration in role-based access control. In *Proc. of the 2nd ACM symposium on Information, computer and communications security*, pages 127–138, 2007.

[38] Dang Nguyen. *PROVENANCE-BASED ACCESS CONTROL MODELS*. PhD thesis, The University of Texas at San Antonio, 2014.

[39] Jiwan Ninglekhu and Ram Krishnan. AARBAC: Attribute-based administration of role-based access control. In *2017 IEEE 3nd International Conference on Collaboration and Internet Computing (CIC)*. IEEE, 2017.

[40] Jiwan Ninglekhu and Ram Krishnan. ARRA: Attribute-based role-role assignment. *International Workshop on Secure Knowledge Management 2017 (SKM 2017)*, 2017.

[41] Jiwan Ninglekhu and Ram Krishnan. Attribute based administration of role based access control: A detailed description. *arXiv preprint arXiv:1706.03171*, 2017.

[42] Jiwan Ninglekhu and Ram Krishnan. A model for attribute based role-role assignment (ARRA). *arXiv preprint arXiv:1706.10274*, 2017.

[43] Alan C OConnor and Ross J Loomis. 2010 economic analysis of role-based access control. *NIST, Gaithersburg, MD*, 2010.

[44] Sejong Oh and Ravi Sandhu. A model for role administration using organization structure. In *Proc. of the seventh ACM symposium on Access control models and technologies*, pages 155–162. ACM, 2002.

[45] Sejong Oh, Ravi Sandhu, and Xinwen Zhang. An effective role administration model using organization structure. *ACM Transactions on Information and System Security (TISSEC)*, 9(2):113–137, 2006.

[46] OpenStack. https://www.openstack.org/. Accessed: 2017-10-29.

[47] OpenStack. Identity api v3 (CURRENT). https://developer.openstack.org/api-ref/identity/v3/index.html#what-s-new-in-version-3-9. Accessed: 2017-10-29.

[48] Qasim Mahmood Rajpoot, Christian Damsgaard Jensen, and Ram Krishnan. Attributes enhanced role-based access control model. In *International Conference on Trust and Privacy in Digital Business*, pages 3–17. Springer, 2015.

[49] Qasim Mahmood Rajpoot, Christian Damsgaard Jensen, and Ram Krishnan. Integrating attributes into role-based access control. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 242–249. Springer, 2015.

[50] Ocata Release. https://releases.openstack.org/ocata/. Accessed: 2017-11-1.

[51] OpenStack Conceptual Architecture (Ocata Release). https://docs.openstack.org/ocata/install-guide-rdo/common/get-started-conceptual-architecture.html. Accessed: 2017-11-12.

[52] Sushmita Ruj. Attribute based access control in clouds: A survey. In *Signal Processing and Communications (SPCOM), 2014 International Conference on*, pages 1–6. IEEE, 2014.

[53] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29:38–47, Feb 1996.

[54] Ravi Sandhu, Venkata Bhamidipati, Edward Coyne, Srinivas Ganta, and Charles Youman. The ARBAC97 model for role-based administration of roles: preliminary description and outline. In *Proceedings of the second ACM workshop on Role-based access control*, pages 41–50. ACM, 1997.

[55] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security (TISSEC)*, 2:105–135, 1999.

[56] Ravi Sandhu and Qamar Munawer. The ARBAC99 model for administration of roles. In *Proc. of the 15th Annual Computer Security Applications Conference (ACSAC'99)*, pages 229–238. IEEE, 1999.

[57] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[58] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[59] Daniel Servos and Sylvia L Osborn. HGABAC: Towards a formal model of hierarchical attribute-based access control. In *International Symposium on Foundations and Practice of Security (FPS 2014)*, pages 187–204. Springer, 2014.

[60] Bo Tang, Qi Li, and Ravi Sandhu. A multi-tenant rbac model for collaborative cloud services. In *Privacy, Security and Trust (PST), 2013 Eleventh Annual International Conference on*, pages 229–238. IEEE, 2013.

[61] Bo Tang and Ravi Sandhu. Extending openstack access control with domain trust. In *International Conference on Network and System Security*, pages 54–69. Springer, 2014.

[62] Kan Yang and Xiaohua Jia. ABAC: Attribute-based access control. In *Security for cloud storage systems*, pages 39–58. Springer, 2014.

[63] Eric Yuan and Jin Tong. Attributed based access control (ABAC) for web services. In *Proc. of the IEEE International Conference on Web Services (ICWS'05)*. IEEE, 2005.

[64] Yue Zhang and James BD Joshi. ARBAC07: a role-based administration model for RBAC with hybrid hierarchy. In *2007 IEEE International Conference on Information Reuse & Integration (2007 IRI)*, pages 196–202. IEEE, 2007.

VITA

Jiwan Ninglekhu was born in Gairi Gaun, Tehrathum, Nepal. He went to St. Joseph's School in Biratnagar, Nepal. He received his Bachelor of Engineering degree in Electronics and Communication from Kathmandu Engineering College (KEC), Tribhuvan University (TU) in Nepal. He received his Master of Science in Electrical Engineering with Computer Networking concentration from Wichita State University (WSU), Wichita, KS, USA in 2009, and Master of Science in Computer Engineering with Cyber Security concentration from The University of Texas at San Antonio (UTSA), San Antonio, TX, USA in 2013. He subsequently entered the doctoral program in Electrical Engineering with Cyber Security concentration at the department of Electrical and Computer Engineering in Fall 2013. He research is affiliated with the Institute for Cyber Security (I.C.S) at UTSA.

His research interests include but not limited to authorization and authentication, privacy and human behavior in cyber security. He is also fascinated by other creative works in the Internet space.