A Model for Attribute Based Role-Role Assignment (ARRA)

Jiwan L. Ninglekhu and Ram Krishnan
Department of Electrical and Computer Engineering
The University of Texas at San Antonio
San Antonio, Texas 78249
Email: jiwan.ninglekhu@gmail.com and ram.krishnan@utsa.edu

Abstract—Administrative Role Based Access Control (ARBAC) models specify how to manage user-role assignments (URA), permission-role assignments (PRA), and role-role assignments (RRA). Many approaches have been proposed in the literature for URA, PRA and RRA. In this paper, we propose a model for attribute-based role-role assignment (ARRA), a novel way to unify prior RRA approaches. We leverage the idea that attributes of various RBAC entities such as admin users and regular roles can be used to administer RRA in a highly-flexible manner. We demonstrate that ARRA can express and unify prior RRA models.

Index Terms—Roles, Role Hierarchy, RBAC, ARBAC, Access Control, Administration.

I. INTRODUCTION

Role-based access control (RBAC) [1], [2] model is well studied in academia [3] and well adopted in the industry [4]. In RBAC, assignment of users and permissions to roles allows users to execute permissions by activating roles.

Administrative RBAC (ARBAC), which involves in user-role assignment and revocation (URA), permission-role assignment and revocation (PRA), and role-role assignment and revocation (RRA), is a challenging task [5]. ARBAC has been well explored [5]–[9]. All of these works explore URA and PRA. However, only few cover RRA [5], [8]. Each of these models use few fixed set of properties of RBAC in making assignment decisions. For example, in RRA97 [5], an admin role is given admin authority over set of roles, called role range, where admin role can perform role assignment.

Recently, Attribute-Based Access Control (ABAC) has gained popularity because of its flexibility [10]–[14]. ABAC has also proven its ability to represent different access control models [12]. However, it has rarely been used in administration of RBAC.

Among three main aspects of ARBAC, we have explored models for attribute-based user-role assignment (AURA) and attribute-based permission-role assignment (ARPA) models [15].

In this paper we present our model for attribute-based role-role assignment (ARRA), a follow-up work on AARBAC [15]. ARRA is driven by our objective to design attribute-based model that allows us to express features from previous RRA models and more. For example, it allows us to express features as a combination of two prior models and in addition, allows us to add new features. Thus, this work is motivated largely by two critical factors: (a) an objective to build a coherent model, which can be configured to express prior models and more (b) build a unified model that can be analyzed for various desirable security features, which can provide a single codebase to express prior models and beyond.

The contributions of this paper are as follows:

- We develop an attribute-based administrative model for role-role assignment (ARRA).
- We demonstrate that ARRA is capable of expressing prior approaches to RRA such as RRA97 [5] and UARBAC’s RRA [8].

The remainder of this paper is organized as follows. In Section II we discuss related work. In Section III ARRA model is presented. Section IV presents algorithms that translate prior RRA instances into equivalent ARRA instances. Section V concludes the paper.

II. RELATED WORK

Among many prominent works done for ARBAC [5]–[9], ARBAC97 [5] and UARBAC [8] present role-role assignment (RRA) as part of their model.

In RRA97, a set of roles in the hierarchy called authority range is given to an admin role for decentralized administration. A user with admin role can perform operations like inserting an edge or creating a role, on roles in her authority range. To confirm autonomy RRA97 requires role range to be encapsulated [5].
UARBAC [8] redefines RBAC model with class objects. It treats each entity in the RBAC system such as files, roles and users as objects. UARBAC’s RRA deals with assigning/revoking role-role on the basis of admin user’s authority over individual roles specified by access modes. Class permissions in UARBAC allows an admin user to conduct an operation over every object of a class.

In both of these models, the policy for assigning a role to a role is based on a fixed set of properties of the entities such as admin role, admin user and regular roles, involved in assigning/revoking roles.

Crampton et. al [16] present models for RRA with administrative scopes. Admin scopes are plausible approach for role hierarchy operations in RBAC. However, admin scopes may not be intuitive to express as a role attribute. Many literature [10], [17], [18] present benefits of integrating attributes into an RBAC operational model. An operational model deals with making decisions on user’s access request for objects. There are many works on operational aspects of ABAC [11], [12], [19], [20]. In contrast, ARRA is an administrative model that uses attributes of RBAC entities for assigning/revoking roles.

III. ARRA MODEL

In this section, we present our approach for attribute-based role-role assignment (ARRA). Inspired by prior RRA models we have included attributes for admin users, admin roles and regular roles. Based on these attributes role assignment and revocation decisions are made.

Table I 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. We have developed admin users attribute functions (AATT) and admin roles attribute functions (ARATT). We have also introduced regular roles attribute functions (RATT) based on the need we have observed. Attributes from other RBAC entities can also be developed if needed. We will later see that we need attributes from different entities in representing properties of RRA97 and UARBAC’s RRA in ARRA.

The attribute functions or simply attributes are defined as mapping from its domain such as AU or AR to its range. Range of an attribute att can be atomic or set valued, which is specified by function attType. It is derived from set of scope or atomic values denoted by Scope(att). Furthermore, the scope of an attribute can be either ordered or unordered, which is given by a function called is_odered(att). If it is ordered, we must specify that attribute’s hierarchy denoted by Hatt on its scope Scope(att). Hatt is a partial ordering on Scope(att). Note that even in the case of a set valued attribute att, the hierarchy Hatt is specified on Scope(att) instead of 2^{Scope(att)}. We infer the ordering between two set values given an ordering on atomic values. H^{\text{att}}_{\text{att}} in Table II denotes the reflexive transitive closure of Hatt.

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\_assign\_au(\{r_1 | r_1 \in \text{ROLES} \land \text{Lead} \in \text{roleTitle}(r_1)\}, 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 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 insertion.

Authorization rule is specified as a logical expression on the attributes of admin users, admin roles, and that of regular roles considered for assignment.

IV. MAPPING PRIOR RRA MODELS IN ARRA

In this section, we demonstrate that ARRA can intuitively simulate the features of prior RRA models. In particular, we have developed concrete algorithms that can convert any instance of RRA97 and UARBAC’s RRA into an equivalent ARRA instance. For each of these models, we first present an example instance followed by that instance’s corresponding ARRA equivalent. Then finally, we present algorithms for each to show a programmable mapping procedure.

A. Mapping RRA97 to AARA

In this section, we present RRA97 example instance and manually convert it into its equivalent ARRA instance. We then present a mapping algorithm for the same, which demonstrates a programmable procedure for translation of any RRA97 instance to its equivalent ARRA instance.
- 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 $att \in AATT \cup ARATT \cup RATT$, $\text{Scope}(att)$ is a finite set of atomic values from which the range of the attribute function $att$ is derived.
- $\text{attType} : AATT \cup ARATT \cup RATT \rightarrow \{\text{set, atomic}\}$, 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.

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

$\forall aratt \in ARATT. aratt : AR \rightarrow \begin{cases} \text{Scope}(aratt) \text{ if } \text{attType}(aratt) = \text{atomic} \\ 2^{\text{Scope}(aratt)} \text{ if } \text{attType}(aratt) = \text{set} \end{cases}$

$\forall ratt \in RATT. ratt : ROLES \rightarrow \begin{cases} \text{Scope}(ratt) \text{ if } \text{attType}(ratt) = \text{atomic} \\ 2^{\text{Scope}(ratt)} \text{ if } \text{attType}(ratt) = \text{set} \end{cases}$

- is_ordered : $AATT \cup ARATT \cup RATT \rightarrow \{\text{True, False}\}$, 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, $H_{att} \subseteq \text{Scope}(att) \times \text{Scope}(att)$, a partially ordered attribute hierarchy, and $H_{att} \neq \emptyset$, else, if is_ordered($att$) = False, $H_{att} = \emptyset$.

(For some $att \in AATT \cup ARATT \cup RATT$ for which attType($att$) = set and is Ordered($att$) = True, if $\{a, b\}, \{c, d\} \in 2^{\text{Scope}(att)}$ (where $a, b, c, d \in \text{Scope}(att)$), 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, $\text{is\_authorizedR}_{op}(au: AU, r_1 : ROLES, r_2 : ROLES)$ 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, $\text{is\_authorizedR}_{op}(au: AU, \chi : 2^{ROLES}, r : ROLES)$ specifies if the admin user $au$ is allowed to perform the operation $op$ (e.g. assign, revoke, etc.) between the roles in the set $\chi$ and the role $r$.

Here $\chi$ is a set of roles that can be specified using a set-builder notation, whose rule is written using role attributes.
1) An RRA97 Example Instance: RRA97 [5] assumes an existing regular role hierarchy and an administrative role hierarchy as depicted in Figure [1] An example instance for RRA97 model is presented as follows:

- **Sets and Functions:**
  - **USERS** = \{u₁, u₂, u₃, u₄\}
  - **ROLES** = \{ED, E₁, PE₁, QE₁, PL₁, E₂, PE₂, QE₂, PL₂, DIR\}
  - **AR** = \{DSO, PSO₁\}
  - **AUA** = \{(u₅, DSO), (u₄, PSO₁)\}
  - **RH** = \{(ED, E₁), \{E₁, PE₁\}, \{E₁, QE₁\}, \{E₁, PL₁\}, \{ED, E₂\}, \{PE₁, PE₂\}, \{QE₁, QE₂\}, \{PL₁, PL₂\}, \{E₁, 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 u₃ is given admin role DSO and PSO₁ is given to role u₄. RH and ARH represent regular role and admin role hierarchies as depicted in Figure [1] We have taken original can-modify example from RRA97 paper [5] where each admin role is mapped with authority ranges. The can-modify set in our instance 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).

In RRA97, an authority range, \((x, y)\) is given by \(r \in \text{ROLES} | x < r < y\), 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,

\[
\forall r_1 \in (x, y) \land \forall r_2 \notin (x, y), \quad [r_2 > r_1 \leftrightarrow r_2 > y \land r_2 < r_1 \leftrightarrow r_2 < x]
\]

A new edge AB can be inserted between two incomparable roles A and B under following two conditions:

- \(AR_{\text{immediate}}(A) = AR_{\text{immediate}}(B)\). An immediate authority range of a role \(r\), \(AR_{\text{immediate}}(r)\) 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 \((A = y \land B > x) \lor (B = x \land A < y)\) then insertion of AB must preserve encapsulation of \((x, y)\).

2) ARRA Instance Equivalent to RRA97 Instance: In this section, we show a manual translation of RRA97 example instance presented in previous segment into an equivalent ARRA instance. Equivalent ARRA instance is as follows:

Map RRA97 sets to ARRA sets

- **AU** = \{u₁, u₂, u₃, u₄\}
- **AOP** = \{insertEdge, deleteEdge\}
- **ROLES** = \{ED, DIR, E₁, PE₁, QE₁, PL₁, E₂, PE₂, QE₂, PL₂, DIR\}
- **AUA** = \{(u₅, DSO), (u₄, PSO₁)\}
- **RH** = \{(ED, E₁), \{E₁, PE₁\}, \{E₁, QE₁\}, \{E₁, PL₁\}, \{QE₁, QE₂\}, \{PL₁, PL₂\}, \{E₁, DIR\}, \{PL₂, DIR\}\}
- **ARH** = \{\{SSO, DSO\}, \{DSO, PSO₁\}, \{DSO, PSO₂\}\}
- **can-modify** = \{(DSO, (ED, DIR)), (PSO₁, (E₁, PL₁)), (PSO₁, (E₂, PL₂))\}

Define attributes

- **ATT** = \{\}, **ATT** = \{\}
- **ARATT** = \{authRange\}
- **Scope(authRange)** = RH⁺, attType(authRange) = set, is_authorized(authRange) = False, H_{authRange} = \emptyset
- **authRange(DSO)** = \{(ED, DIR), authRange(PSO₁) = \{(E₁, DIR), (E₂, PL₂)\}

Construct authorization rule for **insertEdge**

To assign role \(r₁\) to role \(r₂\) (or to insert an edge \(r₁r₂\) and hence add \(<r₂, r₁\>\) to RH),

- is_authorizedR_{insertEdge}(au : AU, r₁ : ROLES, r₂ : ROLES) \(\equiv\)

\[
\exists (au, ar₁) \in \text{AUA, } \exists (s, t) \in \text{authRange(ar₁), } r₁, r₂ \in [s, t] \land (\exists (m, n), (m’, n’) \in \bigcup_{ar₂ \in \text{AR}} \text{authRange(ar₂)}.
\]

\[
r₁, r₂ \in [m, n] \land ([m, n] \rightarrow r₁, r₂ \notin (m’, n’))
\]
∀ (∃(x, y) ∈ ∪ authRange(ar_2)). ((r_1 = y ∧ r_2 > x) \\
∀ (r_2 = x ∧ r_1 < y) ∧ (∃p ∈ [x, y] ∧ ∀q ∉ [x, y] . (∈ {q, p} ∈ (RH ∪ <r_2, r_1>)* ⇔ <q, y> ∈ (RH ∪ <r_2, r_1>)*) ∧ (∃p, q ∉ (RH ∪ <r_2, r_1>)* ⇔ <x, q> ∈ (RH ∪ <r_2, r_1>)*)))

Construct authorization rule for deleteEdge

= is_authorizedR deleteEdge (au : AU, r_1 : ROLES, r_2 : ROLES) ⇔ ∃(au, ar) ∈ AU. ∃(x, y) ∈ authRange(ar), r_1, r_2 ∈ [x, y] ∧ ∀(u, v) ∈ ∪ authRange(ar_2), ∃<r_2, r_1> ∈ RH. (r_1 ≠ u ∧ r_2 ≠ v)

As we can observe, 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. We have designed an admin role attribute authRange that maps authority ranges given to admin roles. Attribute authRange is set valued and unordered. Scope of authRange is a transitive closure of regular role hierarchy, RH*. That is, if <a, b>, <b, c>, <c, d> ∈ RH*, then <a, d> is also true.

We know that authority range represents a set of roles over which an admin role has authority to conduct operations such as assign and revoke. In ARRA, we use symbolic representation of authority range with end points a and b as (a, b). However, whenever we need to work with roles present in the set represented by authority range (a, b), we denote the set of roles with [a, b]. That is, [a, b] = {r | r ∈ ROLES ∧ a < r < b}.

Next, we construct authorization rule for the admin operation insertEdge. 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, we construct an authorization formula for operation deleteEdge. 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 1: MapRRA97

Input: RRA97 instance
Output: ARRA instance

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

a. AU ← USERS^97
b. AOP^A ← {insertEdge, deleteEdge}
c. ROLES^A ← ROLES^97 ; AUA^A ← AUA^U
r. RH^A ← RH^97

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

a. AATT^A ← {} ; RATT^A ← {} 
b. ARATT^A ← {attribute}
c. Scope^A(authRange) = RH^A^A 
d. attType^A(authRange) = set
 e. is_ordered^A(authRange) = False ; H^A^authRange = φ 
f. For each ar ∈ AR^A, authRange(ar) = φ 
g. For each (ar, (r_1, r_2)) ∈ can-modify^97.

authRange(ar) = authRange(ar) ∪ (r_1, r_2)

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

a. assign_formula = φ
b. For each (ar, (r_1, r_2)) ∈ can-modify^97.

assign_formula' = assign_formula ∨
(∃(au, ar_1) ∈ AU, ∃(s, t) ∈ authRange(ar_1), r_1, r_2 ∈ [s, t] ∧ (∃(m, n), (m', n') ∈ ∪ authRange(ar_2). r_1, r_2 ∈ [m, n] ∧ ∃(m', n') ∈ (m, n) → r_1, r_2 ≠ (m', n')) ∨ (∃(x, y) ∈ ∪ authRange(ar_3). (r_1 = y ∧ r_2 > x) ∨ (r_2 = x ∧ r_1 < y)) ∧ (∃p ∈ [x, y] ∧ ∀q ∉ [x, y] . (∈ {q, p} ∈ (RH ∪ <r_2, r_1>)* ⇔ <q, y> ∈ (RH ∪ <r_2, r_1>)*) ∧ (∃p, q ∉ (RH ∪ <r_2, r_1>)* ⇔ <x, q> ∈ (RH ∪ <r_2, r_1>)*))

c. auth_assign = is_authorizedR insertEdge (au : AU^A, 

r_1 : ROLES^A, r_2 : ROLES^A) ⇔ assign_formula'

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

a. revoke_formula = φ
b. For each (ar_1, (r_1, r_2)) ∈ can-modify^97.

revoke_formula' = revoke_formula ∨
∃(au, ar) ∈ AU, ∃(x, y) ∈ authRange(ar), r_1, r_2 ∈ [x, y] ∧ ∀(u, v) ∈ ∪ authRange(ar_4),
∃<r_2, r_1> ∈ RH. (r_1 ≠ u ∧ r_2 ≠ v)
c. auth_revoke = is_authorizedR deleteEdge (au : AU^A, 

r_1 : ROLES^A, r_2 : ROLES^A) ⇔ revoke_formula'
Algorithm \[1\] presents Map\(_{RRA97}\), a translation algorithm for 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\(_{RRA97}\) 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\(_{RRA97}\) algorithm is an equivalent ARRA instance, with following sets and functions. AU\(_{A}\), AOP\(_{A}\), ROLES\(_{A}\), AUA\(_{A}\), RH\(_{A}\), ARH\(_{A}\), AATT\(_{A}\), ARATT\(_{A}\). For each attribute att \(\in\) ARATT\(_{A}\), Scope\(_{A}(att)\), attType\(_{A}(att)\), is_ordered\(_{A}(att)\) and H\(_{A}\). For each admin role ar \(\in\) AR and for each att \(\in\) ARATT\(_{A}\), att(ar), Authorization rule for assign (auth_assign), and Authorization rule for revoke (auth_revoke).

Steps indicated in Map\(_{RRA97}\) correspond to the steps described in equivalent ARRA instance for RRA97 in section [V-A2]. We explain it here briefly. 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-modifySS\(_{97}\), authRange(ar) populates set of authority ranges for ar.

In Step 3, we construct assign_formula rule for ARRA that is equivalent to inserting an edge in can-modify\(_{97}\). For each (ar, \(r_i\), \(r_j\)) in set can-modifySS\(_{97}\), logical expression that checks the assignment conditions is constructed. Equivalent translation in ARRA is given by is_authorized\(_R\)insertEdge\((au : AU^A, r_b1 : ROLES^A, r_b2 : ROLES^A)\). This formula must checks if roles \(r_b1\) and \(r_b2\) belong in an authority range of an admin role, which is mapped to admin user au. It further checks if \(r_b1\) and \(r_b2\) have same immediate authority range, or checks if any other assignment 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 \(r_b1\) and \(r_b2\) belong and restricts from deleting an edge between end points of any authority range.

C. Mapping UARBAC’s RRA to ARRA

In this section, we present an example instance for UARBAC’s RRA and its equivalent ARRA instance. We then present an algorithm that outlines the translation procedure.

1) An Example UARBAC’s RRA Instance:

**RBAC schema**

- \(C = \{\text{role}\}\)
- \(\text{OBJ}(user) = \text{USERS}, \text{OBJ}(role) = \text{ROLES}\)
- \(\text{AM}(\text{role}) = \{\text{grant, empower, admin}\}\)

**RBAC state**

- \(\text{USERS} = \text{OBJ(user)} = \{u_1, u_2, u_3, u_4\}\)
- \(\text{ROLES} = \text{OBJ(role)} = \{r_1, r_2, r_3\}\)
- \(P = \{\text{role, r_1, grant}, \text{role, r_1, empower}, \text{role, r_2, admin}, \text{role, r_2, grant}, \text{role, r_2, empower}, \text{role, r_3, admin}, \text{role, r_3, grant}, \text{role, r_3, empower}, \text{role, r_3, admin}\}\)
- \(\text{RH} = \langle r_2, r_3 \rangle\)

**Administrative permissions of UARBAC’s RRA**

Following is the list of administrative permissions each user has for role-role assignment:
- \(\text{authorized_perms}[u_1] = \{[\text{role, r_1, grant}], [\text{role, r_2, admin}], [\text{role, r_2, grant}]\}\)
- \(\text{authorized_perms}[u_2] = \{[\text{role, r_1, admin}], [\text{role, r_2, empower}], [\text{role, r_3, grant}]\}\)
- \(\text{authorized_perms}[u_3] = \{[\text{role, admin}]\}\)
- \(\text{authorized_perms}[u_4] = \{[\text{role, grant}], [\text{role, empower}], [\text{role, admin}]\}\)

**Role-role assignment condition**

One can perform following operation to assign a role \(r_1\) to another role \(r_2\):
- \(\text{grantRoleToRole}(r_1, r_2)\)

To perform this operation one needs one of the following two permissions:
- \(\text{[role, r_1, grant] and [role, r_2, empower]}\) or,
- \(\text{[role, grant] and [role, r_2, empower]}\) or,
- \(\text{[role, r_1, grant] and [role, empower]}\) or,
- \(\text{[role, grant] and [role, empower]}\)

**Condition for revoking a role from another role**

To revoke a role \(r_2\) from a role \(r_3\), admin user performs following operation:
- \(\text{revokeRoleFromUser(r_2, r_3)}\)

To conduct this operation one needs one of the following options:
- \(\text{[role, r_2, grant] and [role, r_3, empower]}\)
D. 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 = \{u_1, u_2, u_3, u_4\}
- AOP = (assign, revoke)
- ROLES = \{r_1, r_2, r_3\}
- AUA = {}
- RH = \{\langle r_2, r_3 \rangle\}

Define Attributes

- AATT = \{grantAuth, empowerAuth, adminAuth, roleClassAuth\}, ARATT = {}, RATTT = {}  
- Scope(grantAuth) = \{r_1, r_2, r_3\}
  - attType(grantAuth) = set, is_authorizedU(grantAuth) = ROLES, H_{grantAuth} = RH
  - grantAuth(u_1) = \{r_1\}, grantAuth(u_2) = \{r_2\},
  - grantAuth(u_3) = \{\}, grantAuth(u_4) = \{\}
- Scope(empowerAuth) = \{r_1, r_2, r_3\}
  - attType(empowerAuth) = set, is_authorizedU(empowerAuth) = ROLES
  - empowerAuth(u_1) = \{r_2\},
  - empowerAuth(u_2) = \{r_1, r_2\},
  - empowerAuth(u_3) = \{\}, empowerAuth(u_4) = \{\}
- Scope(adminAuth) = \{r_1, r_2, r_3\}
  - attType(adminAuth) = set, is_authorizedU(adminAuth) = ROLES
  - adminAuth(u_1) = \{r_2, r_3\},
  - adminAuth(u_2) = \{r_1\},
  - adminAuth(u_3) = \{\}, adminAuth(u_4) = \{\}
- Scope(roleClassAuth) = AM(role)
  - attType(roleClassAuth) = set, is_authorizedU(roleClassAuth) = False
  - roleClassAuth(u_1) = \{\}, roleClassAuth(u_2) = \{\},
  - roleClassAuth(u_3) = \{admin\}, roleClassAuth(u_4) = \{grant, empower, admin\}

Construct authorization rule for role-role assignment

For each op in AOP, authorization rule to assign/revoke role-role can be expressed as follows:

To assign any regular role \(r_1\) \(\in\) ROLES to regular role \(r_2\) \(\in\) ROLES,

\[
is\text{\_}\text{authorized}_{\text{assign}}(au : AU, r_1 : ROLES, r_2 : ROLES) \equiv \\
(r_1 \in \text{grantAuth}(au) \land r_2 \in \text{empowerAuth}(au)) \lor \\
(r_1 \in \text{grantAuth}(au) \land \text{empower} \in \text{roleClassAuth}(au)) \lor \\
(grant \in \text{roleClassAuth}(au) \land r_2 \in \text{empowerAuth}(au)) \lor \\
(grant \in \text{roleClassAuth}(au) \land \text{empower} \in \text{roleClassAuth}(au))
\]

Construct authorization rule for revoking role-role

To revoke any regular role \(r_1\) \(\in\) ROLES from another regular role \(r_2\) \(\in\) ROLES,

\[
is\text{\_}\text{authorized}_{\text{revoke}}(au : AU, r_1 : ROLES, r_2 : ROLES) \equiv \\
(r_1 \in \text{grantAuth}(au) \land r_2 \in \text{empowerAuth}(au)) \lor \\
(r_1 \in \text{adminAuth}(au) \land r_2 \in \text{adminAuth}(au)) \lor \\
\text{admin} \in \text{roleClassAuth}(au)
\]

There are four main stages in this translation process. First, we map main sets from UARBAC’s RRA to ARRA model. Set of admin users consists of set of users from UARBAC’s RRA, namely \(u_1, u_2, u_3, u_4\). There are two admin operations, assign and revoke. Sets ROLES and RH map 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, we define required attribute functions. We used admin user attributes to express 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 attribute roleClassAuth captures the nature of access mode an admin user has towards role class. The scope of this attribute is \(AM(\text{role})\), 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 left empty.

After attributes have been defined we construct the assignment formula (assign_formula), 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\_authorizedUassign(au : AU, r_1 : ROLES, r_2 : ROLES). Finally, we construct a logical expression which captures four different permission options that an admin user can
use for revoking a role from another role. The logical expression is equivalent to \( \text{is\_authorized}\text{\_revoke}(\text{au}t : \text{AU}, r_1 : \text{ROLES}, r_2 : \text{ROLES}) \).

E. \( \text{Map}_{\text{RRA}}^{\text{UARBAC}} \)

We presented example instances for UARBAC’s RRA and its equivalent AARA instance in previous segments. Algorithm 2 presents \( \text{Map}_{\text{RRA}}^{\text{UARBAC}} \), a procedure to map any instance of UARBAC’s RRA to its equivalent AARA instance. It takes instance of UARBAC’s RRA as its input and yields an equivalent AARA instance as its output. Sets and functions from UARBAC and AARA are labeled with superscripts \( U \) and \( A \), respectively. Input consists of \( C^U, \text{USERS}^U, \text{ROLES}^U, P^U, \text{RH}^U, \text{AM}^U(\text{role}) \). For each \( u \in \text{USERS}^U \), \text{authorized\_perms}^U[u] \), and for every \( r_1, r_2 \in \text{ROLES}^U \). Grant operation \( \text{grantRole}\text{\_To}\text{\_Role}(r_1, r_2) \) will be true if the granter has either \( \text{[role, r_2, empower]} \) and \( \text{[role, r_1, grant]} \) or, \( \text{[role, r_2, empower]} \) and \( \text{[role, grant]} \) or, \( \text{[role, empower]} \) and \( \text{[role, grant]} \) or, \( \text{[user, empower]} \) and \( \text{[role, grant]} \) permissions towards roles.

For each \( r_1, r_2 \in \text{ROLES}^U \), \( \text{revoke\_RoleFrom}\text{\_User}(u, r_2) \) is true if the granter has either \( \text{[role, r_2, empower]} \) and \( \text{[role, r_1, grant]} \) or, \( \text{[role, r_1, admin]} \) or, \( \text{[role, r_2, admin]} \) or, \( \text{[role, admin]} \) permission on roles.

\( \text{Map}_{\text{RRA}}^{\text{UARBAC}} \) yields an AARA instance consisting of \( \text{AU}^A, \text{AOP}^A, \text{ROLES}^A, \text{AR}^A, \text{AUA}^A, \text{RH}^A, \text{ARH}^A, \text{AATT}^A \). For each attribute \( \text{att} \in \text{AATT}^A \), \text{Scope}^A(\text{att}) \), \text{attType}^A(\text{att}) \), \text{is\_ordered}^A(\text{att}) \) and \( \text{H}^A(\text{att}) \). For each user \( u \in \text{AU}^A \), and for each \( \text{att} \in \text{AATT}^A, \text{att}(\text{au}) \), Authorization rule for assign (\text{auth\_assign}), and Authorization rule for revoke (\text{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 \( \text{[role, r, grant]} \) in \text{authorized\_perms}^U[u], \( \text{grantAuth}(\text{au}) \) attribute extracts set of roles towards which an admin user \( au \) has \text{grant} \) authority. In other words, admin user can grant those roles to any other object such as user and role. Similarly, \text{empowerAuth} \) yields set of roles to which an admin user has \text{empower} \text{access mode} \) and \text{adminAuth} \) yields sets of roles about which admin user has \text{admin access mode} \). The meaning of each access mode is preserved by UARBAC model. For each class level attribute \( \text{[c, am]} \) in \text{authorized\_perms}^U[u], where \( c \) is a type of class and \( am \) is type of access mode, \text{roleClassAuth}(au) \) specifies the nature of class level

---

**Algorithm 2. \( \text{Map}_{\text{RRA}}^{\text{UARBAC}} \)**

**Input:** Instance of RRA in UARBAC  
**Output:** AARA instance

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

a. \( \text{AU}^A \leftarrow \text{USERS}^U \); \( \text{AOP}^A \leftarrow \{\text{grant, revoke}\} \)

b. \( \text{ROLES}^A \leftarrow \text{ROLES}^U \); \( \text{AU}^A = \phi \)

c. \( \text{RH}^A \leftarrow \text{RH}^U \)

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

a. \( \text{AATT}^A \leftarrow \{\text{grantAuth, empowerAuth, adminAuth, roleClassAuth}\} \)

b. \( \text{ARATT} = \{\}, \text{RATT} = \{\} \)

c. \( \text{attType}^A(\text{grantAuth}) = \text{ROLES}^A \)

d. \( \text{is\_ordered}^A(\text{grantAuth}) = \text{True}, \text{RH}^A \text{grantAuth} = \text{RH}^A \)

e. For each \( u \in \text{AU}^U \), \( \text{grantAuth}(u) = \phi \)

f. For each \( u \in \text{USERS}^U \)

for each \( \text{[role, r, grant]} \in \text{authorized\_perms}^U[u], \text{grantAuth}(u) = \text{grantAuth}(u) \cup (r) \)

g. \( \text{Scope}^A(\text{empowerAuth}) = \text{ROLES}^A \)

h. \( \text{attType}^A(\text{empowerAuth}) = \text{set} \)

i. \( \text{is\_ordered}^A(\text{empowerAuth}) = \text{True} \)

j. \( \text{H}^A(\text{empowerAuth}) = \text{RH}^A \)

k. For each \( u \in \text{AU}^U \), \( \text{empowerAuth}(u) = \phi \)

l. For each \( u \in \text{USERS}^U \)

for each \( \text{[role, r, empower]} \in \text{authorized\_perms}^U[u], \text{empowerAuth}(u) = \text{empowerAuth}(u) \cup r \)

m. \( \text{Scope}^A(\text{adminAuth}) = \text{ROLES}^A \)

n. \( \text{attType}^A(\text{adminAuth}) = \text{set} \)

o. \( \text{is\_ordered}^A(\text{adminAuth}) = \text{True} \)

p. \( \text{H}^A(\text{adminAuth}) = \text{RH}^A \)

q. For each \( u \in \text{USERS}^A \), \( \text{adminAuth}(u) = \phi \)

r. For each \( u \in \text{USERS}^U \)

for each \( \text{[role, r, admin]} \in \text{authorized\_perms}^U[u], \text{adminAuth}(u) = \text{adminAuth}(u) \cup r \)

s. \( \text{Scope}^A(\text{roleClassAuth}) = \text{AM}^U(\text{role}) \)

t. \( \text{attType}^A(\text{roleClassAuth}) = \text{set} \)

u. \( \text{is\_ordered}^A(\text{roleClassAuth}) = \text{False} \)

v. \( \text{H}^A(\text{roleClassAuth}) = \phi \)

w. For each \( u \in \text{USERS}^A \), \( \text{roleClassAuth}(u) = \phi \)

x. For each \( u \in \text{USERS}^U \)

for each \( \text{[c, am]} \in \text{authorized\_perms}^U[u], \text{roleClassAuth}(au) = \text{roleClassAuth}(au) \cup \text{am} \)

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

a. \( \text{assign\_formula} = (\text{r}_1 \in \text{grantAuth}(au) \land \text{r}_2 \in \text{empowerAuth}(au) \lor (\text{r}_1 \in \text{grantAuth}(au) \land \text{empower} \in \text{roleClassAuth}(au)) \lor (\text{grant} \in \text{roleClassAuth}(au) \land \text{r}_2 \in \text{empowerAuth}(au)) \lor (\text{grant} \in \text{roleClassAuth}(au) \land \text{empower} \in \text{roleClassAuth}(au)) \lor \end{align*} \)
Step 4: /* Construct revoke rule for AURA */

a. revoke_formula = r1 ∈ grantRoleToRole(au) ∧
r2 ∈ adminAuth(au) ∨ r1 ∈ adminAuth( au) ∨

b. auth_revoke =

is_authorizedU(revoke(au : AU, r1 : ROLES, r2 : ROLES)) ⇒ revoke_formula

Sets
- AU = {Sam, Tom}, AOP = {assign, revoke}
- ROLES = {IT Director, Development Mgr., Quality Mgr., Marketing Mgr., Finance Mgr., Support Engineer, System Analyst}
- AR = {}
- AUA = {}
- RH = {}, ARH = {}

Attributes definition
- AATT = {dept},
- Scope(dept) = {Operations, Account, IT},
attType(dept) = set, is_ordered(dept) = False, Hdept = φ
- dept(Sam) = {Operations, Account, IT},
depTTom) = {IT}
- ARATT = {}, RATT = {dept, level}
- Scope(dept) = {Operations, Account, IT},
attType(dept) = atomic,
is_ordered(dept) = False, Hdept = φ
- dept(IT Director) = IT, dept(Dev. Manager) = IT,
dep(Marketing Mgr.) = Operations,
dep(QA Mgr.) = IT.

Authorization functions
- is_authorizedAassign(au : AU, r1 : ROLES, r2 : ROLES) = ∃d ∈ Scope(dept). dept(au) = dept(r1) =
dep(r2)
- is_authorizedAr revoke(au : AU, r1 : ROLES, r2 : ROLES) = is_authorizedAassign(au : AU, r1 : ROLES, r2 : ROLES)

V. Conclusion

In this paper, we presented ARRA, a model for attribute based role-role assignment. A design motivation behind ARRA model was to make it enough to express prior RRA models. In particular, we took RRA97 and UARBAC’s RRA model as foundation. For these models, we have presented MapRA97 and MapRA-UARBAC algorithms, which map any instances of these models to ARRA model, respectively.

ARRA model can not only express existing RRA models but also has a capability to do more. For instance, prior models included in our research did not fit the notion of regular role attributes. However, we acknowledge that it is an essential attribute in making role assignment decisions to say the least. Motivated by this notion, we presented a simple example scenario where we made use of regular role attributes for role-role assignment.
ACKNOWLEDGMENT

This work is partially supported by NSF grants CNS-1423481 and CNS-1553696.

REFERENCES

[1] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, “Proposed NIST standard for role-based access control,” ACM Transactions Inf. Syst. Secur., vol. 4, pp. 224–274, Aug. 2001.
[2] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, “Role-based access control models,” Computer, vol. 29, pp. 38–47, Feb 1996.
[3] L. Fuchs, G. Pernul, and R. Sandhu, “Roles in information security - A survey and classification of the research area,” Computers & Security, vol. 30, pp. 748 – 769, 2011.
[4] A. C. O'Connor and R. J. Loomis, “2010 economic analysis of role-based access control,” NIST, Gaithersburg, MD, 2010.
[5] R. Sandhu, V. Bhambidipati, and Q. Munawer, “The ARBAC97 model for role-based administration of roles,” ACM Transactions on Information and System Security (TISSEC), vol. 2, pp. 105–135, 1999.
[6] R. Sandhu and Q. Munawer, “The ARBAC99 model for administration of roles,” in Proc. of the 15th Annual Computer Security Applications Conference (ACSAC’99). IEEE, 1999, pp. 229–238.
[7] S. Oh and R. Sandhu, “A model for role administration using organization structure,” in Proc. of the seventh ACM symposium on Access control models and technologies. ACM, 2002, pp. 155–162.
[8] N. Li and Z. Mao, “Administration in role-based access control,” in Proc. of the 2nd ACM symposium on Information, computer and communications security, 2007, pp. 127–138.
[9] P. Biswas, R. Sandhu, and R. Krishnan, “Uni-ARBAC: A unified administrative model for role-based access control,” in International Conference on Information Security. Springer, 2016, pp. 218–230.
[10] D. R. Kuhn, E. J. Coyne, and T. R. Weil, “Adding attributes to role-based access control,” Computer, vol. 43, pp. 79–81, 2010.
[11] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone et al., “Guide to attribute based access control (ABAC) definition and considerations,” NIST special publication, vol. 800, no. 162, 2014.
[12] X. Jin, R. Krishnan, and R. Sandhu, “A unified attribute-based access control model covering DAC, MAC and RBAC,” in IFIP Annual Conference on Data and Applications Security and Privacy. Springer, 2012, pp. 41–55.
[13] S. Bhattacharya, F. Patwa, and R. Sandhu, “ABAC with group attributes and attribute hierarchies utilizing the policy machine,” in Proc. of the 2nd ACM Workshop on Attribute-Based Access Control. ACM, 2017, pp. 1–8.
[14] P. Biswas, R. Sandhu, and R. Krishnan, “Attribute transformation for attribute-based access control,” in Proc. of the 2nd ACM Workshop on Attribute-Based Access Control. ACM, 2017, pp. 1–8.
[15] J. Ninglekhu and R. Krishnan, “Attribute based administration of role based access control: A detail description,” arXiv preprint arXiv:1706.03171, 2017.
[16] J. Crampton, “Administrative scope and role hierarchy operations,” in Proceedings of the seventh ACM symposium on Access control models and technologies. ACM, 2002, pp. 145–154.
[17] X. Jin, R. Sandhu, and R. Krishnan, “RABAC: role-centric attribute-based access control,” in International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security. Springer, 2012, pp. 84–96.
[18] Q. M. Rajpoot, C. D. Jensen, and R. Krishnan, “Attributes enhanced role-based access control model,” in International Conference on Trust and Privacy in Digital Business. Springer, 2015, pp. 3–17.
[19] E. Yuan and J. Tong, “Attributed based access control (ABAC) for web services,” in Proc. of the IEEE International Conference on Web Services (ICWS’05). IEEE, 2005.
[20] D. Servos and S. L. Osborn, “HGABAC: Towards a formal model of hierarchical attribute-based access control,” in International Symposium on Foundations and Practice of Security (FPS 2014). Springer, 2014, pp. 187–204.