To appear at ACM SACMAT 2002

A Lightweight Approach to Specification and Analysis of Role-based Access Control Extensions (2)

Andreas Schaad
Department of Computer Science
University of York
York, United Kingdom
andreas@cs.york.ac.uk

Jonathan D. Moffett
Department of Computer Science
University of York
York, United Kingdom
jdm@cs.york.ac.uk

ABSTRACT

Role-based access control is a powerful and policy-neutral concept for enforcing access control. Many extensions have been proposed, the most significant of which are the decentralised administration of role-based systems and the enforcement of constraints. However, the simultaneous integration of these extensions can cause conflicts in a later system implementation. We demonstrate how we use the Alloy language for the specification of a conflict-free role-based system. This specification provides us at the same time with a suitable basis for further analysis by the Alloy constraint analyser.

Keywords

Alloy, constraint analysis, RBAC96, ARBAC97, Separation of Duties, Delegation of Authority, Decentralisation

1. INTRODUCTION

Role-based access control is a powerful concept and can be used to enforce a variety of security policies in a system. The reference model for role-based access control is the RBAC96 model [13], [14]. Two significant extensions to this model have been proposed, one concentrating on the specification of constraints [12], the other describing a framework for the delegation of authority through administrative roles [11].

However, these two extensions create a new range of problems when integrated simultaneously within a role-based access control model. The main concern is that a model which allows for the delegation of authority through decentralised administrative actions can conflict with specified separation of duty constraints. A simple example is that two roles *r1* and *r2* are declared as mutually exclusive by a chief security officer. A valid constraint is that a user must not be assigned to the two exclusive roles at the same time. Assuming that a user *u1* already holds *r1*, the delegation of *r2* to user *u1*

would result in an unintended conflict. While there are already some mechanisms such as prerequisite conditions in place to avoid this kind of conflict, we believe that this does not solve the general problem. What is missing is a framework for the specification and analysis of role-based access control models and required constraints.

To ensure operational efficiency systems such as described in [16] require a certain degree of flexibility with respect to decentralised administration, either through designated local administrators or system users with delegation authority.

A trivial but highly realistic example is that of an employee being ill. On a short-term basis his roles might have to be temporarily assigned to another employee so that he can cover his ill colleague for a day. Decentralising administrative control would allow an assigned branch administrator to perform the required administrative actions, avoiding the bottleneck of a centralised security administration. This example also presents us with an environment where the enforcement of constraints and implementation of conflict detection mechanisms are of major importance. While there might be a corporate-wide valid constraint in the form of a separation of duty requirement, this might not necessarily be conformed to by the designated local administrator. Mechanisms must be in place to constrain his actions in accordance with a higher-level policy and resolve any conflicts.

First attempts are now being made to built systems according to these models and the established standards. However, at this stage we encounter problems when trying to integrate the functionality of several models into one product. The complexity of the models demands a structured approach to analysis and design of such systems. Access control research only recently started to address this issue [6, 1]. Using lightweight formal specification techniques such as Alloy might be a further step in that direction.

2. OUTLINE

We will demonstrate how we use the Alloy language to specify and analyse the implications of the simultaneous integration of role-based access control extensions and constraints. Having given an initial introduction (Section 1), we continue with a review of the relevant literature (Section 3). We then use Alloy to specify an initial RBAC96 model (Section 4) followed by a specification of the ARBAC97 model and its sub-models (Section 5). A set of separation of duty constraints are defined within this context (Section 6). We then discuss possible separation of duty conflicts as a result

of administrative actions (Section 7). The paper finishes with a summary and conclusion (Section 8).

3. RELATED WORK

3.1 Administrative role-based access control

The work on role-based access control systems has provided us with the RBAC96 model family of access control models [13], which since then has been subject to a NIST standard proposal [14]. In the RBAC96 model family, the central notion is that permissions are associated with roles, and users are made members of appropriate roles thereby acquiring the roles' permissions. The model family consists of four sub-models, gradually adding functionality to the basic model which consists of user, role and permission entities and the relations between them.

The administrative role-based access control model (AR-BAC97) [11] expresses the idea of using RBAC to manage RBAC through decentralisation of administrative authority. A distinction is made between regular and administrative roles and permissions (Figure 1). ARBAC97 consists of three sub-models. These describe the decentralised administration through user-role assignment (URA97), permission-role assignment(PRA97) and role-role assignment (RRA97). The two central concepts of ARBAC97 are those of the *administrative range* and *prerequisite conditions*. They regulate and impose restrictions on the way in which system objects can be administered.

The administrative range reflects the set of roles over which an administrator has authority. Depending on the context he can assign and revoke users to or from a role, alter role hierarchies, and assign or revoke permissions. In case of a user-role assignment, a prerequisite condition could be used, e.g. to express that any user to be assigned to a role *r1* must already be assigned to another role *r2*.

3.2 Separation of Duties

The separation of duties property expresses that a critical process cannot be performed through a single subject. Although the principle of separation of duties has been applied for a long time in the area of accountancy, Clark and Wilson were the first to describe it as one of the main mechanisms to control fraud and error in the context of an automated system [2]. This idea was later taken up by Nash and Poland, describing the static and dynamic separation of duty [10].

The work on role-based systems gave a new impetus to the description of static and dynamic separation of duty constraints. Kuhn addresses the mutual exclusion of roles to implement separation of duty in a role-based access control system [7]. Simon and Zurko show further variations of the separation of duty [17]. A later formalisation of separation of duty properties is presented by Gligor et al. in [3]. Ahn describes a framework for specifying separation of duty and conflict of interest policies in role-based systems. A formal language is introduced to express separation of duties properties [1].

3.3 The Alloy specification language

The Alloy language is designed for the specification of object models through graphical and textual structures [4]. It is a state-based language and invariants constrain the relationships between objects. Alloy is supported by a constraint analysis facility [5] which allows us to analyse specifications in order to detect any over- and under-constraints. As we make extensive use of the Alloy language we summarise some relevant parts of the syntax in the following.

Each state component is either a set, a binary relation, or an indexed relation (whose values are indexed collections of relations). The two main logical connectors are and (&&) and or (||). For sets, there are the usual set-theoretic operators, in ASCII form:

```
s + t    union of s and t
s & t    intersection of s and t
s - t    difference: s with elements of t removed
```

For relations, we have the following operators:

```
~r    the transpose of r
+r    the transitive closure of r
*r    the reflexive transitive closure of r
```

There are no set constants in Alloy. Instead, we have:

```
some s    s is non-empty
no s      s is empty
sole s    s has at most one element
one s     s has exactly one element
```

Elementary formulas in Alloy are made by comparing sets:

```
s = t      equality: s and t have the same elements
s in t     subset: every element of s is an element of t
s !in t    subset: every element of s is not an element of t
```

An important operator in Alloy is the relational image '.'. The expression `s.r` denotes the set of objects that the set `s` maps to in the relation `r`. This kind of expression is often called a "navigation expression" because we can think of it as a navigation from s along the relation `r`. Applying the image operator again yields a longer navigation. Suppose, for example, that we have the sets `Users`, `Roles` and `Permissions` denoting the components of a role-based system. The relations `ureg_assignment` and `regp_assignment` map the sets of users to roles and roles to permissions respectively. Then for a single user `u1`, `u1.ureg_assignment` will be the set of roles that the user holds and the evaluation of the expression `u1.ureg_assignemnt.regp_assignment` results in the set of permissions he holds through all his roles. If we take a specific permission `p1` from this set and write `p1.~regp_assignment.~ureg_assignment`, we will obtain the set of users who can execute permission `p1`. In case of a role hierarchy marked by a relation `reg_rr_hierarchy` between two sets of Roles, we could read `r1.+reg_rr_hierarchy` as "take a role `r1` and follow `reg_rr_hierarchy` one or more times" and `r1.*reg_rr_hierarchy` as "take a role `r1` and follow `reg_rr_hierarchy` zero or more times". In other words, we would obtain the set of roles inferior to `r1`, which excludes and includes `r1`, respectively.

For the rest of this paper we will gradually introduce further concepts of Alloy where needed.

4. SPECIFYING RBAC96 IN ALLOY

We give an example of the use of Alloy by specifying RBAC96 model properties which will be the underlying basis for the later analysis of RBAC extensions.

The initial structure of the RBAC96 model can be easily specified in Alloy. The **domain** paragraph describes the objects we make use of, with the **fixed** keyword indicating

that they are drawn from a specified pool of objects. The `state` paragraph then describes the relations between our objects. Where it is not specified the cardinality of the relation is automatically assumed to be zero or more, otherwise the `!` and `+` symbols indicate cardinalities of one and one or more respectively.

Specification 1

```
model RBAC96{
 domain {fixed User, fixed RegRole,
         fixed Session, fixed Permission}
 state {
  ureg_assignment:  User -> RegRole
  regp_assignment:  RegRole -> Permission
  us_assignment:    User! -> Session
  sr_assignment:    Session -> RegRole+
  reg_rr_hierarchy: RegRole -> RegRole
 }}
```

We can now use the the constraint analysis facilities of Alloy in order to find a model that satisfies our formulae (Figure 1). A graphical representation helps in detecting any under- or overconstraint or other undesired model properties. For better readability, we chose to represent users as boxes, roles as circles, permissions as diamonds and sessions as triangles. This graphical model is also supported by a textual output to which we only refer when necessary. We can immediately see some undesired properties in our resulting graphical model (Figure 2).
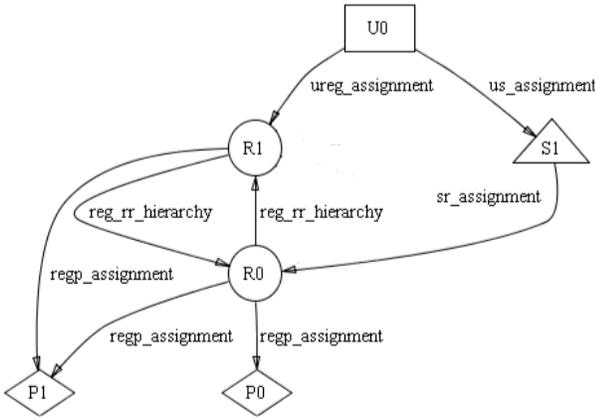

Figure 1: Unconstrained RBAC Model

To begin with we can observe that a user U0 can have access to a role R0 through a session S1, although he is not assigned to that role. Additionally, we have two roles R1 and R0 which have a cycle in their role hierarchy. To mitigate the above effects we have to specify some invariants that must always hold in our model. To begin with we introduce an invariant on the role hierarchy. Any cycles within the role hierarchy as defined by the `reg_rr_hierarchy` relation are ruled out. The star symbol (`*`) in the clause `r !in r.*reg_rr_hierarchy` represents a reflexive transitive closure operator. Here we test for the presence of role `r` in the set of roles which are inferior to `r`. The `!` symbol negates set membership.

Invariant 1:

```
all r:RegRole | r !in r.*reg_rr_hierarchy
```

The other problem is that of a user having access to a role through a session although he is not assigned to that role. We can similarly rule this out. A user can only activate his assigned roles through a currently open session (`r in u.us_assignment.sr_assignment`) if he is assigned to it (`r in u.ureg_assignment`).

Invariant 2:

```
all u:User | all r:Regrole |
   r in u.us_assignment.sr_assignment ->
   r in u.ureg_assignment
```
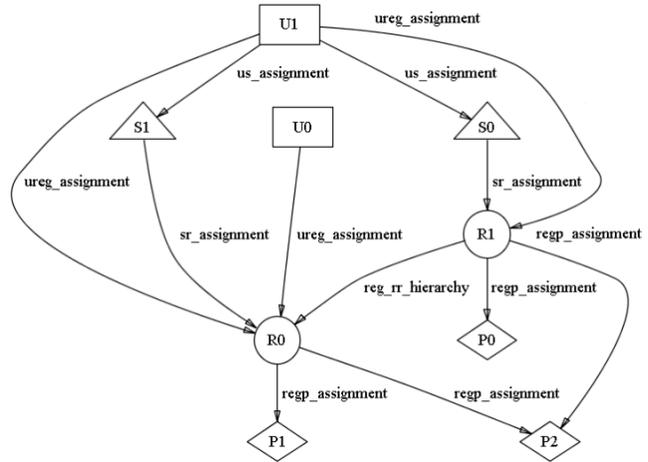

Figure 2: Constrained RBAC Model

A valid model can be seen in figure 2. However, the generation of some (graphical) models is not good enough. To obtain further assurance about the properties of our model we made use of Alloy's assertions and conditions, represented by the keywords `assert` and `cond`. With the first we can ask questions of the kind "Is it true that...?", the latter would allow us to ask "Find me a model with the following properties...!". Depending on the way in which we formulate our questions, the Alloy constraint analyser will either generate a valid example, a counterexample, or respond that no example/counterexample was found.

5. SPECIFYING ARBAC97 IN ALLOY

So far we have only made use of static invariants to constrain our model. ARBAC97 requires us to define a set of specific administrative operations. Execution of these operations must result in a valid state before and after they are carried out.

5.1 General ARBAC97 properties

The administrative access control model ARBAC97 requires us to add some new state components to our existing Alloy specification.

We now have to make a distinction between regular and administrative roles. One way we could have specified these in Alloy would have been using the Alloy concept of partitioning. However, this would have required us to change

our existing model and its invariants significantly. We decided to simply add a new object with the type `AdmRole` to the `domain` paragraph. Administrative roles have authority over a set of regular roles (`RegRole`). In ARBAC97 this is expressed using the mathematical concept of the range notation. We do not have this available in Alloy and thus we simply introduce a new binary relation. The keyword `static` specifies that the administrative range relation does not change during an operation.

```
admin_range: static AdmRole -> static RegRole
```

One ARBAC97 constraint on the authority range is that authority ranges must not overlap. The question is how far this can be seen as a policy neutral constraint. We decided to use it in our Alloy specification as it contributes to the overall legibility of the generated models.

Invariant 3:

```
inv admin_range{all adm_role1,adm_role2:AdmRole |
                all r:RegRole |
  //Authority ranges must not overlap
  r in adm_role1.admin_range ->
  r !in adm_role2.admin_range}
```

One could imagine altering this constraint, allowing the sets of roles within different administrative ranges to overlap as long as the users assigned to the respective administrative roles are different. This however, is again a policy question outside the scope of this paper.

Having specified the concept of an administrative range we can continue to specify the three sub-models into which the ARBAC97 model is split. They describe the decentralised system administration through user-role assignment (URA97), permission-role assignment (PRA97) and role-role assignment (RRA97).

5.2 User-Role Assignment (URA97)

The URA97 model describes the decentralised administration of user-role assignments. The two operations we observe within the context of our analysis are the assignment (`assign`) and revocation (`revoke`) of a user from a role. In general, ARBAC97 does not use explicit operations but defines administrative actions in terms of authorisation relations. While we try to stay as close as possible to the semantics of these relations, using Alloy operations we define what actually happens in terms of the before and after states.

As a general principle we tried to keep our specifications of operations as policy neutral as possible. This explains why certain phenomena, such as an administrator assigning a role to himself, are not explicitly ruled out.

5.2.1 User-Role Assignment Operation

The `assign` operation is simple to define in Alloy. As in other sequential specification languages, the before state is left unmarked while the after state is indicated by the primed symbol ('). The argument list of the operation `assign` defines the variables upon which the operation is carried out. As an initial definition we might write:

Operation 1a:

```
op assign (reg_role:RegRole, reg_role_member:User){
```

```
  //the receiving user must hold the role
  //after the execution of operation
  reg_role in reg_role_member.ureg_assignment'}
```

This however has some problems. We have to ensure that this operation can only be carried out by an administrator if authorised. Authorisation would mean that there exists a user who is assigned to an administrative role which in turn has the regular role to be assigned in its administrative range. We could off course add an invariant of the form `reg_role in adm_role.admin_range` to express this. However, from a security point of view we also have to allow the operation to fail in case the role to be assigned is not part of the administrative range. In this case simply nothing would happen and the before and after state are identical. In Alloy terms this would be expressed as

```
reg_role !in adm_role.admin_range ->
    reg_role_member.ureg_assignment' =
    reg_role_member.ureg_assignment
```

Thus the according `assign` operation incorporating this and some further invariants is defined as follows:

Operation 1b:

```
op assign (adm_role:AdmRole, reg_role:RegRole,
           reg_role_member:User){
 //Can adm_role assign role? - Yes!
 reg_role in adm_role.admin_range ->
    reg_role in  reg_role_member.ureg_assignment'

 //Can adm_role assign role? - No!
 reg_role !in adm_role.admin_range ->
    reg_role_member.ureg_assignment' =
    reg_role_member.ureg_assignment

 //the receiving user must not have
 //previously had the role
 reg_role !in reg_role_member.ureg_assignment

 //At least one person (not the reg_role_member)
 //must hold the adm. role
 some u : User - reg_role_member |
    adm_role in u.uadm_assignment}
```

Using Alloy's assertion mechanism, we can clearly observe the two possible effects of this operation in Figures 3 and 4. Either a new relation between a regular role and a user is created or not. So we can observe that in case of a successful assignment (Figure 3), user U1 was assigned with the regular role R0 (depicted as an ellipse) through U0 in his administrative role A0 (depicted as a circle).

In the case of an unsuccessful assignment we can clearly observe the absence of the regular role R1 in the administrative range of the administrating role A0. Thus, user U1 was not assigned with role R1.

5.2.2 User-Role Revocation Operation

With respect to the `revoke` operation, ARBAC97 makes a distinction between strong and weak revocation. This is due to the possibility of a senior role R1 inheriting permissions from a junior role R2. If a user is assigned to two such roles, the weak revocation of R2 would result in only that
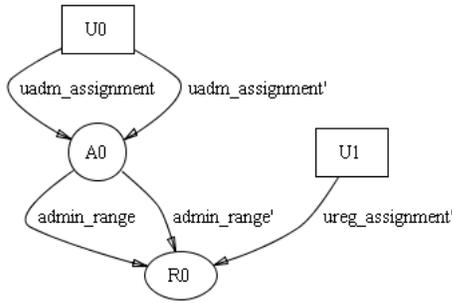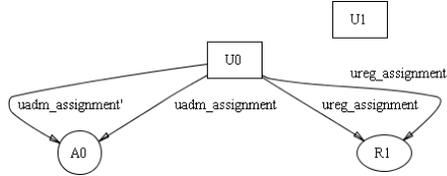
Figure 3: Successful assign

Figure 4: Failed assign

role being revoked. Strong revocation would additionally result in the user being revoked from R1 as well, since R1 is senior to R2. Weak revocation can be easily achieved and the corresponding Alloy operation is almost the same as the previous `assign` relation. Strong revocation on the other hand is not so easy to specify. We first have to make a distinction between implicit and explicit role membership. Implicit role membership occurs with a user being explicitly assigned to a role R1 which is superior to some other role R2. He then is explicit member of R1 and implicit member of R2. ARBAC97 also allows a role to be implicitly and explicitly assigned to a user at the same time. We therefore introduce two new relations for regular roles:

```
reg_explicit : User -> RegRole
reg_implicit : User -> RegRole
```

In addition we had to specify some more invariants on these two new relations such as to make sure that a relation `reg_explicit` only occurs when there is a corresponding assignment `ureg_assignment`. So the strong revocation operation is defined as follows in Alloy:

Operation 2:

```
op strong_revoke(adm_role:AdmRole, reg_role:RegRole,
                 reg_role_member:User) {
//This operation is only valid on implicit roles
reg_role !in reg_role_member.reg_implicit ->
reg_role_member.ureg_assignment' =
reg_role_member.ureg_assignment

//The role to be revoked and its seniors must be
//in the administrative range or operation fails
reg_role.~*reg_rr_hierarchy !in
adm_role.admin_range ->
reg_role_member.ureg_assignment' =
reg_role_member.ureg_assignment

//If it is in the administrative range
```

```
//then delete assignment
reg_role.~*reg_rr_hierarchy in
adm_role.admin_range ->
reg_role !in reg_role_member.reg_implicit'

//If it is in the administrative range and
//there is another direct assignment then
//delete that also
reg_role.~*reg_rr_hierarchy in
adm_role.admin_range &&
reg_role in reg_role_member.ureg_assignment ->
reg_role !in reg_role_member.ureg_assignment'

//At least one person (not reg_role_member)
//must hold the adm. role
some u : User - reg_role_member |
adm_role in u.uadm_assignment}
```

The cautious reader will wonder where we check for any role hierarchies and subsequent upward revocation of explicit roles. Indeed this is not part of the operation itself, but results out of other, not described invariants, that must hold on the `reg_explicit` relationship. In this case we specified that if a user is explicitly assigned to a role, and this role is senior to some other roles, then he must be implicitly assigned to the junior ones. Thus, a deletion of an implicit assignment will cause any explicit senior assignment to be removed as well.

Whilst a possible corresponding graphical model for the weak revocation looks just the opposite of figure 3, a model representing a strong revocation operation looks somewhat different. For a better readability we do not present before and after states in one figure. Instead, we can observe the before state in figure 5 and the after state in figure 6. Also we chose only to represent a successful operation for reasons of space.
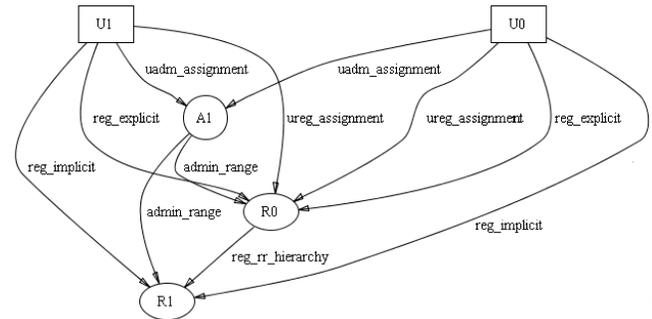

Figure 5: Strong revoke (before state)

In this case we also need to make use of the analyser's textual output to see what happened. Alloy will tell us that user U1 was declared as `reg_role_member` and role R1 as the role to be revoked (`reg_role`). Since U1 was a explicitly assigned to role R0 and R0 was superior to R1, U1 is implicitly assigned to R1. Subsequently his implicit assignment to R1 and explicit assignment to R0 were revoked.

5.3 Permission-Role Assignment (PRA97)

As pointed out in the ARBAC97 model [11], users and permissions have a similar character from the perspective of a role. Thus, PRA97 is a simple dual of URA97. We found
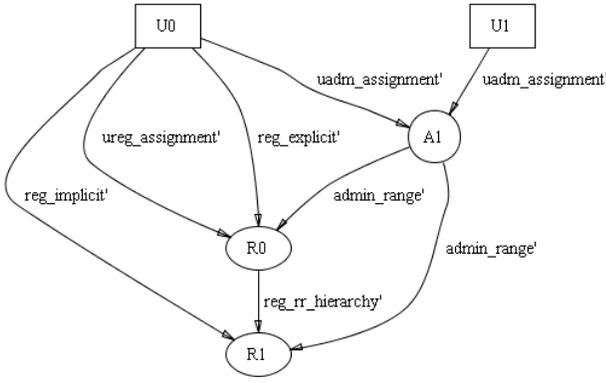
Figure 6: Strong revoke (after state)

this to be true when specifying the relevant operations and as a result we do not provide any detailed specification here. What needs to be noted however is that, unlike in a strong user-role revocation, a strong permission-role revocation cascades down a role hierarchy and not upwards.

5.4 Role-Role Assignment (RRA97)

The specification of the RRA97 model and its operations is not as straightforward as the earlier specifications of the URA97 and PRA97 sub-models. We see two main reasons for this. On the one hand the administration of role-role assignments almost always requires policy decisions. On the other hand, the ARBAC97 concept of a range of roles cannot be modeled satisfactorily in terms of Alloy. As already described in section 5.1 we do not have this at hand in Alloy but chose to use the simple binary relation `admin_range` to show which regular roles can be administered. Accordingly, many of the constraints on the administrative range as defined in ARBAC97 cannot be directly expressed (e.g. no partially overlapping ranges).

Four administrative actions are specified in the role-role assignment model: Role Creation; Role Deletion; Edge Insertion; and Edge Deletion. ARBAC97 requires some specific constraints to be enforced for each of these operations. Creation of a role requires the specification of its immediate parent and child in the existing hierarchy. Deletion of a role will cause any existing edges to be deleted. Edge insertion is only meaningful between incomparable roles, whilst edge deletion is only meaningful if that edge is not transitively implied by other edges.

5.4.1 Create Role Operation

ARBAC97 requires that with the creation of a role we specify its immediate parent and child in the existing hierarchy. Only the chief security officer can create roles outside the authority range or without a parent or child. We only specify a role creation operation with respect to a decentralised administration, however removing some constraints would show the general creation of roles.

Operation 3:

```
op create_role (new_role:RegRole'!, sup:RegRole!,
                inf:RegRole!, adm_role:AdmRole!)
{
 //There is only one new_role created
```

```
no (new_role & RegRole)
RegRole' = RegRole + new_role

//Place new_role into hierarchy
new_role in sup.reg_rr_hierarchy'
inf in new_role.reg_rr_hierarchy'

//Make sure roles are in the admin_range
(inf + sup) in adm_role.admin_range
(inf + sup + new_role) in adm_role.admin_range'

//Remove transitive edges
inf in sup.reg_rr_hierarchy ->
inf !in sup.reg_rr_hierarchy'

//The new role must have no other
//relations apart from sup and inf
no (new_role.reg_rr_hierarchy - inf)
no (new_role.~reg_rr_hierarchy - sup)

//inf must be inferior to sup in unprimed
inf in sup.reg_rr_hierarchy}
```

While most parts of the operation are very similar to what we have specified before we can observe that a new role is part of the operation signature `new_role:RegRole'`. This role is now placed between a superior (`sup`) and inferior role (`inf`), at the same time ensuring that any transitive edges are removed.

A valid model for the role creation operation can be seen in figure 7. Here we can observe that role R2 is the newly created role which is placed in between the hierarchy of R0 and R1. The previous edge R0→R1 is replaced by the two primed edges R0→R2 and R2→R1.
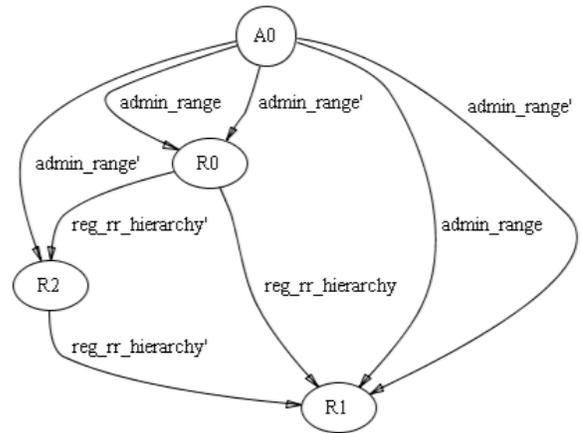

Figure 7: Creating a role

5.4.2 Delete Role Operation

Deleting a role is the natural counterpart to role creation but needs to consider several issues. Permissions associated with a role are inherited upwards in a role hierarchy. Deleting a role with associated permissions can thus have serious effects on a role hierarchy. ARBAC97 suggest three ways to mitigate this. By deactivating a role the semantics of the hierarchy remain intact, at the same time prohibiting any assignments to the role. Complete deletion of a role can only

be performed if there are either no permissions and users associated with that role, or if all its permissions are assigned to the next senior role and all users are assigned to the next junior role. For reasons of space we only show the relevant excerpts to achieve the later two options. Apart from these, the entire operation is almost identical to the `create_role` operation. Thus, an empty role would be specified as:

```
no del_role.ureg_assignemnt
no del_role.regp_assignment
```

In case any users or permissions are assigned to the role to be deleted, their re-allocation is specified as follows:

```
sup.regp_assignment' =
sup.regp_assignment + del_role.regp_assignment

inf.~ureg_assignment' =
inf.~ureg_assignment + del_role.~ureg_assignment
```

In this case `sup` and `inf` are the respective superior and inferior roles, while `del_role` is the role to be deleted. The effects of such a shift can be observed in figure 8.
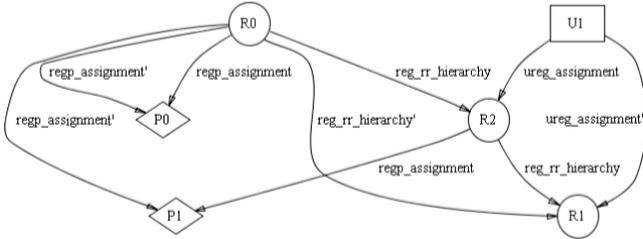

Figure 8: Deleting a role

Here we see that role R2 is the role that will be deleted from the hierarchy. Thus, the existing edges between R0→R2 and R2→R1 cease to exist and a new edge R0→R1 is introduced. The permission P1 assigned to R2 is re-assigned to R0, and the user U1 is re-assigned to R1.

5.4.3 Insert Edge Operation

The insertion and deletion of a single edge between two roles is again very simple to define as an operation. In terms of the ARBAC97 model further constraints have to be enforced. We chose to only specify that the roles between which an edge is created must lie within the same authority range. Additionally, the insertion of transitive edges is avoided by demanding that the roles between which an edge is created must be incomparable.

We can avoid transitive edges by specifying that the inferior role must not already lie within the transitive closure of the superior role (`inf !in sup.+reg_rr_hierarchy`) and that in the after state the inferior role is directly related to the superior role (`inf in sup.reg_rr_hierarchy'`).

We can see in figure 9 how an edge was inserted between the roles R4 and R3 according to these invariants.

5.4.4 Delete Edge Operation

As described in ARBAC97, deletion of transitive edges is meaningless. If a non-transitive edge is deleted we will have to create new edges to maintain the semantics of the role hierarchy. In this case if the edge between a senior role `sup`
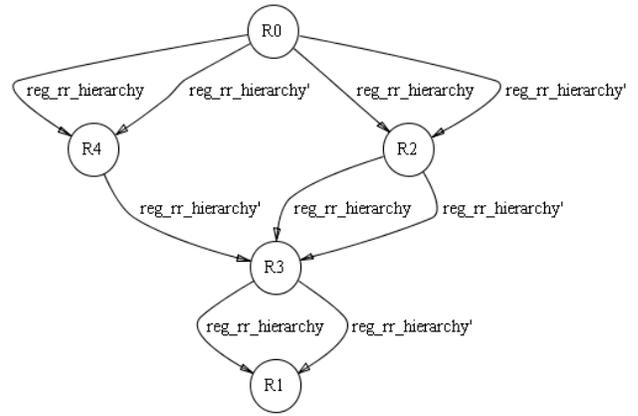

Figure 9: Inserting an edge

and a junior role `inf` is deleted, we have to create an edge between `inf` and the role senior to `sup` and vice versa. Within our specification we specify the deletion of an edge as

```
inf in sup.reg_rr_hierarchy
inf !in sup.reg_rr_hierarchy'
```

while the creation of new edges would be done for the roles `sup` and `inf` as described in the previous section 5.4.

The result of this can be observed in figure 10. Here the edge between role R4(`sup`) and R3(`inf`) is deleted. As a consequence the new edges R4→R0 and R2→R3 are created.
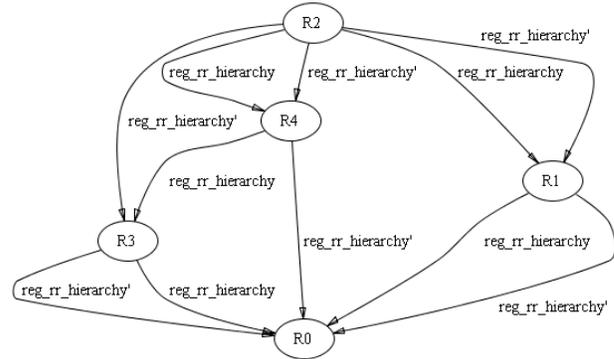

Figure 10: Deleting an edge

6. SPECIFYING SEPARATION OF DUTY PROPERTIES IN ALLOY

The separation of duties is one of the best understood constraints in role-based models so far. However, many different ways of specifying and enforcing separation of duties constraints exist. We chose to use the taxonomy proposed by Simon and Zurko [17], enforced through the concept of mutually exclusive roles [7]. For reasons of space we only present a subset of the constraints we specified.

The Static Separation of Duty (SSoD) constraint defines that if roles are strongly exclusive, no person is ever allowed to hold both of them at the same time. Two exclusive roles have thus no common assigned user. The dynamic Separation of Duty constraints we implemented are the Simple

Dynamic Separation of Duty (SDSoD) and the Operational Separation of Duty (OpSoD). SDSoD requires that any two exclusive roles must not be activated at the same time by the same user. Preserving OpSoD means that all permissions a user has through his exclusive roles should not allow him to perform all the actions required for the completion of a critical workflow.

To enforce these constraints we need to expand our specification by adding the following two new relations:

```
regr_exclusive: Regrole -> Regrole
wf_requires: Workflow -> Permission+
```

The first defines the relation in which regular roles are exclusive to each other, the second the set of permissions required by a workflow (We also have to add `Workflow` as `fixed` to our domain paragraph). This is all the information we need to specify our separation of duty constraints. As suggested in [7], we ask for the set intersection (`&`) of users assigned to pairs of mutually exclusive roles to be empty (`no`). Accordingly the SSoD and DSoD invariants express that for all pairs of mutually exclusive roles, these pairs must not have a common assigned user or a common session respectively.

SSoD Invariant:

```
inv static_sod {all r1, r2:Regrole |
 r1 in r2.regr_exclusive ->
 no(r1.~ureg_assignment & r2.~ureg_assignment)}
```

SDSoD Invariant:

```
inv dynamic_sod {all r1, r2:Regrole |
 r1 in r2.regr_exclusive ->
 no(r1.~sr_assignment & r2.~sr_assignment)}
```

For preserving operational separation of duties we only have to check that the set of permissions that every user holds through his roles does not contain all permissions needed for a critical operation. We express this by requiring the difference set (`-`) of the required permissions for a workflow and the permissions a user user holds through his roles to be non-empty (`some`). In this case we only specified this constraint for a static operational separation of duties, however, this could be easily extended to a more relaxed dynamic invariant.

OpSoD Invariant:

```
inv operational_sod {all u:User | all wf:Workflow |
 some(wf.wf_requires -
 u.ureg_assignment.regp_assignment) }
```

When checking the specified invariants we can see an immediate problem. The role hierarchies might enable a user to be implicitly assigned to an exclusive role and subsequently break our intended separation of duty properties [9]. An example for breaking a static separation of duties property can be seen in figure 11, however all other separation constraints could be broken in a similar way. In this case, the roles R2 and R0 are exclusive to each other. Although not being directly assigned to R0, user U0 nevertheless breaks a static separation of duty constraint as his role R1 is senior to R0.
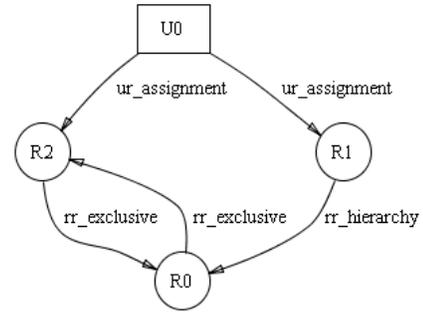

Figure 11: SoD violation through role hierarchy

To mitigate this situation we have to change our specification to check for mutually exclusive roles within role hierarchies. Below we give an example of how to do this for the simple static separation of duties, in a similar manner this would have to be done for the other separation properties. We use the reflexive transitive closure operator `*` to traverse eventual role hierarchies upwards (`~`).

Ext. SSoD Invariant:

```
inv static_sod_hierarchy {all r1, r2:RegRole |
 r1 in r2.regr_exclusive ->
  no(r1.~*reg_rr_hierarchy.~ureg_assignment &
  r2.~*reg_reg_rr_hierarchy.~ureg_assignment)}
```

7. CONFLICT ANALYSIS

To summarise, we have specified an RBAC96 model in section 4, described the delegation of authority through a decentralised user-role, role-permission and role-role administration in section 5 and defined a set of separation of duty constraints in section 6.

Our initial motivation was to demonstrate that role-based extensions such as decentralised administrative actions can conflict with separation of duty constraints. We have specified a conflict free system, and to illustrate the conflicts we encountered during that process we have to 'switch off' the separation of duty constraints. We will further use Alloy assertions to prompt the Alloy constraint analyser to generate separation of duty conflicts as we expect them to happen when performing an administrative action [15]. We are going to analyse this with respect to the three ARBAC97 models in more detail.

7.1 URA97 vs. SoD

In case of a simple user-role assignment operation and an existing pair of exclusive roles we might expect a possible conflict. We now ask Alloy to verify some intended properties of the assign operation. This property is identical to the static separation of duty invariant as specified in section 6.

```
assert SSoD {all adm_role : AdmRole |
           all reg_role, excl : RegRole |
           all reg_role_member : User |
 assign(adm_role, reg_role, reg_role_member) &&
 no(reg_role.~ureg_assignment'&
    excl.~ureg_assignment')}
```

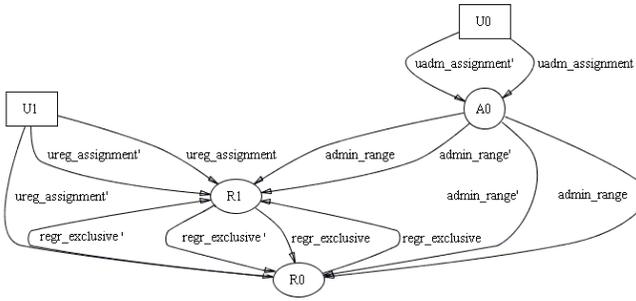However, as expected, a counterexample is found which can be seen in figure 12. Here user U1 was already assigned

Figure 12: SoD violation/URA97 Assign

to one exclusive role R1 before the operation and is now also assigned to the second exclusive role R0, thus breaking the static separation of duty property.

If we now 'switch' our static separation of duty invariant on we can observe that no counterexample will be found within our specification.

We could now specify an assertion to check for violation of our dynamic separation of duty properties. However, the way we specified our administrative operations is that they are atomic. Thus the only change of state that occurs is that of a user-role assignment. So even if two roles are exclusive and one of them was already held and activated by some regular user, the possible assignment of the second exclusive role would not lead to a conflict with respect to the after state of the operation.

The third possibility for a conflict due to a user-role assignment is that of violating operational separation of duty properties. However, we do not have the space to follow this through here.

Obviously, considering the revocation of user-role assignments we do not have to check for any conflicting situations.

7.2 PRA97 vs. SoD

Considering the decentralised assignment of permissions to roles we would expect a conflict with operational separation of duty constraints as they are the only ones which take permissions into consideration. Using assertions we can again instruct the constraint analyser to look for a possible breach of an operational separation of duty invariant. As we can see in figure 13, workflow W0 requires two permissions P1, P2. While the regular role R1 was only assigned with one of these before the execution of the operation, it is assigned with both of them in the after state, giving rise to a possible conflict.

What can also be considered is the degree to which permissions are shared by exclusive and non-exclusive roles. This was suggested as a separation property in [7]. We have specified these properties below but do not further analyse them in this paper.

1. Disjoint/Disjoint (DD): If two roles are exclusive, then each permission is assigned to only one of them.

    ```
    inv DD {all r1, r2:Regrole| all p:Permission |
     r1 in r2.regr_exclusive ->
      no(r1.regp_assignment&r2.regp_assignment)&&
      (no(p.~regp_assignment - r1) ||
       no(p.~regp_assignment - r2))}
    ```

2. Disjoint/Shared (DS): As DD, but a permission can also be assigned to non-exclusive roles.

    ```
    inv DS {all r1, r2 : Regrole|
     r1 in r2.regr_exclusive ->
      no (r1.regp_assignment & r2.regp_assignment)}
    ```

3. Shared/Disjoint (SD): Exclusive roles may share permissions, but each role must have at least one permission not available to the other.

    ```
    inv SD {all r1, r2 : Regrole|
     r1 in r2.regr_exclusive ->
      some(r1.regp_assignment-r2.regp_assignment)&&
      some(r2.regp_assignment-r1.regp_assignment)}
    ```

4. Shared/Shared (SS): As SD, but again a permission can also be assigned to non-exclusive roles.

    ```
    inv SS {all r1, r2 : Regrole|
     r1 in r2.regr_exclusive ->
      no(r1.regp_assignment & r2.regp_assignment)}
    ```

7.3 RRA97 vs. SoD

So far we have only shown what possible conflicts could arise out of administrative actions with respect to user/role and role/permission relationships. For the sake of simplicity we have only shown direct conflicts. However, indirect conflicts as they might occur due to possible role hierarchies can equally well occur in all three administrative models. We already showed that separation of duty invariants can be easily modified to handle hierarchies (Section 6).

When creating a new role or generating a new edge within an existing role hierarchy permissions are inherited upwards the hierarchy. Again, it is not difficult to imagine how an operational separation of duty constraint can be broken in such a way. Interestingly, according to our specification, any static or dynamic separation of duty constraints on a pair of exclusive roles could not be broken. This is due to the invariants placing constraints on roles only, neglecting the associated permissions. We have observed this phenomenon in the previous section with respect to permission-role assignments as well as in one of our earlier papers [15]. But clearly, if we imagine two exclusive roles $r1$ and $r2$, assigned with permission $p1$ and $p2$ respectively, a direct or indirect assignment of $p1$ to $r2$ should be prohibited. We require that for each pair of exclusive roles neither set of assigned permissions is a subset of the other. This can be easily included into our Alloy specification:

```
all r1,r2 : RegRole | r1 in r2.regr_exclusive ->
r1.regp_assignment !in r2.regp_assignment &&
r2.regp_assignment !in r1.regp_assignment
```

To deal with any hierarchies each navigation will have to be extended using the reflexive transitive closure operator. For example, `r1.regp_assignment` would become `r1.*reg_rr_hierarchy.regp_assignment`.

8. SUMMARY AND CONCLUSION

Using the Alloy specification language and its analysis facilities, we have shown how to specify a RBAC96-style model, ARBAC97-style extensions and a set of separation
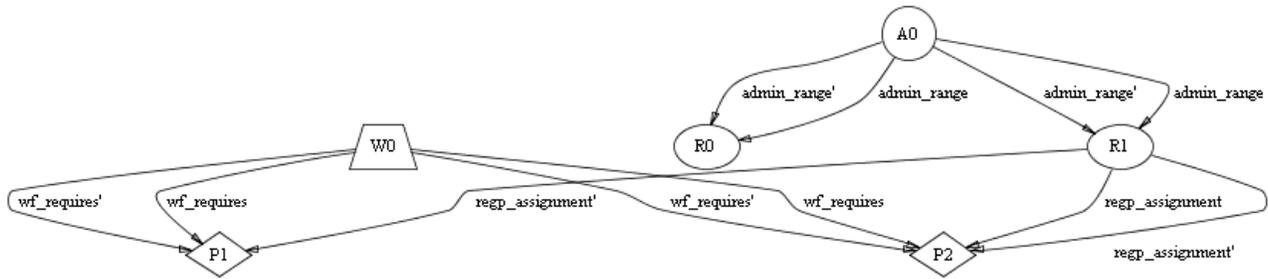
Figure 13: OPSoD violation/PRA97 Assign

of duty properties. We analysed and discussed possible conflicts that could arise out of decentralised administrative actions with respect to the separation constraints. This approach also showed that it is difficult to keep the specification policy neutral.

We believe that we have demonstrated the general suitability of the Alloy language for specification and analysis of role-based systems. The simultaneous integration of decentralised administration mechanisms and separation of duty controls is highly desirable with respect to real-world applications. However, to find the optimum balance between operational efficiency and operational control is nearly impossible. Yet, we hope that our approach can help system developers to gain a deeper understanding of this problem. We also believe that the delegation of authority through decentralised administration and the specification of separation of duty constraints are part of a much wider set of organisational controls [8]. The specification and analysis of these controls will be part of our future work. We have experimented with the Prolog language as an executable specification language to enforce control [15], and will continue to investigate the relationships between Alloy and Prolog.

While an exhaustive analysis of conflicts is not possible within such limited scope, we nevertheless hope that our approach has conveyed the general idea and problem of integrating role-based extensions.

9.   ACKNOWLEDGEMENTS

10.   REFERENCES

[1] G. Ahn. *RCL 2000*. Phd dissertation, George Mason University, 2000. Good literature review with respect to separation of duties. Lots of quotes and pointers to separation of duties.

[2] D. Clark and D. Wilson. A comparison of commercial and military security policies. In I. C. S. Press, editor, *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, California, 1987.

[3] V. Gligor, S. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In I. C. S. Press, editor, *IEEE Symposium on Security and Privacy*, pages 172–185, Oakland, CA, 1998.

[4] D. Jackson. Alloy: A leightweight object modelling notation. Technical Report 797, MIT Laboratory for Computer Science, 2000. In Folder.

[5] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proc. International Conference on Software Engineering*, Limerick, Ireland, 2000.

[6] T. Jaeger and J. Tidswell. Practical safety in flexible access control models. *ACM Transactions on Information and System Security (TISSEC)*, 4(2), 2001.

[7] R. Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *Proceedings of the second ACM workshop on Role-based access control*, pages 23–30, 1997.

[8] J. Moffett. Control principles and role hierarchies. In *3rd ACM Workshop on Role Based Access Control (RBAC)*, pages 63–72, George Mason University, Fairfax, VA, 1998.

[9] J. Moffett and E. Lupu. The uses of role hierarchies in access control. In *4th ACM Workshop on Role-Based Access Control*, pages 153–160, Fairfax, Virginia, 1999.

[10] M. Nash and K. Poland. Some conundrums concerning separation of duty. In I. C. S. Press, editor, *IEEE Symposium on Security and Privacy*, pages 201–209, Oakland, CA, 1990.

[11] R. Sandhu, V. Bhamidipati, and Q. Munawer. The arbac97 model for role-based administration of roles. *ACM Transactions. Inf. Syst. Security*, 2(1):105 – 135, 1999.

[12] R. Sandhu and F. Chen. Constraints for role-based access control. In *Proceedings of the first ACM Workshop on Role-based access control*, 1996.

[13] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[14] R. Sandhu, D. Ferraiolo, and R. Kuhn. The nist model for role-based access control: Towards a unified standard. In *5th ACM RBAC*, Berlin, Germany, 2000.

[15] A. Schaad. Conflict detection in a role-based delegation model. In *Annual Computer Security Applications Conference*, New Orleans, 2001.

[16] A. Schaad, J. Moffett, and J. Jacob. The access control system of a european bank - a case study. In *6th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Chantilly, VA, USA, 2001.

[17] R. Simon and M. Zurko. Separation of duty in role-based environments. In *Computer Security Foundations Workshop X*, Rockport, MA, 1997.