Detecting Conflicts in a Role-based Delegation Model

Andreas Schaad
University of York
Department of Computer Science
YO10 5DD, York, U.K.
andreas@cs.york.ac.uk

Abstract

The RBAC96 access control model has been the basis for extensive work on role-based constraint specification and role-based delegation. However, these practical extensions can also lead to conflicts at compile and run-time. We demonstrate, following a rule-based, declarative approach, how conflicts between specified Separation of Duty constraints and delegation activities can be detected. This approach also demonstrates the general suitability of Prolog as an executable specification language for the simulation and analysis of role-based systems. Using an extended definition of a role we show how at least one of the conflicts can be resolved and discuss the impacts of this extension on the specified constraints.

1 Introduction

Significant work has been done on role-based access control models, most of which has been presented in the ACM workshops on role-based access control. The RBAC96 model [14] and its extensions are the result of these discussions.

Two significant areas of extensions to the RBAC96 model have been proposed, one concentrating on the specification of constraints [3, 1], the others describing a framework for role-based delegation [2, 8]. However, these two extensions create a new range of problems within a role-based access control model such as RBAC96. The main concern is that specified Separation of Duty constraints can conflict with a model allowing for the delegation of authority through role transfer.

A simple example is that two roles *r1* and *r2* are declared as mutually exclusive. A valid Separation of Duty 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*, a delegation of *r2* to user *u1* would result in a conflict with the separation constraint.

2 Outline

We give an initial motivation for the integration of role-based constraints and delegation mechanisms into a single system (Section 3), and discuss related work (Section 4). A rule-based approach to detect possible conflicts at compile and at run-time, using Prolog as a general constraint specification and implementation language, is presented (Section 5). Using the example of a department processing cheques, we apply these rules to detect conflicts and analyse in which cases the delegation of roles causes additional violation of constraints (Sections 6 and 7). This is followed by a brief discussion on the necessity of conflict analysis and the technical suitability of Prolog for such a task (Section 8). We then discuss how a different definition of a role could help us to resolve at least one of the previously detected conflicts (Section 9 and 10) and finally provide a summary, conclusion and discussion of future work (Section 11).

3 Motivation

We discussed the role-based access control system of a European bank in [17]. This clearly showed that there are real world applications which require support for role-based delegation.

A trivial but highly realistic example is that of an employee being ill. On a short-term basis his roles might have to be delegated to another employee so that he can cover his ill colleague for a day. However, making a formal request to the system administrator, and asking for the required changes to be performed immediately, is often not feasible due to the lack of resources. Instead it would be more efficient to allow for a delegation of authority between peers without any specific administrative power.

This example also presents us with an environment in which the enforcement of Separation of Duty constraints and implementation of conflict detection mechanisms are of major importance with respect to the overall criticality of the company's operations.

4 Related Work

The seminal work on role-based access control models was the introduction of the RBAC96 model family by Sandhu et al. [14], hereafter simply referred to as the RBAC96 model. It has served as the basis for a significant part of the role-based access control research and is likely to become a NIST standard [15]. 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.

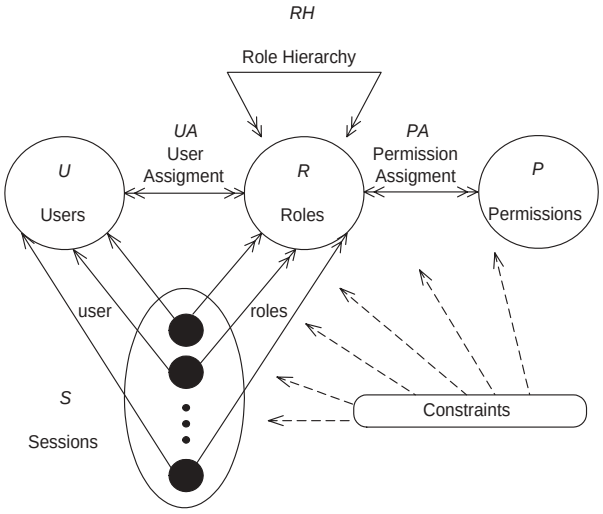

Figure 1. The RBAC96 model family

Several extensions have been proposed to the RBAC96 model, one of which is the RCL2000 language for constraint specification [1] and the other the RBDM0 delegation model [2]. The RCL2000 language provides a framework for the specification of Separation of Duty constraints within the RBAC96 model. The constraints that are described are partially based on previous work on separation of duties, here mainly [4, 12, 11, 18, 7]. Although the delegation of authority has been subject to intensive research before [10, 19, 13], the RBDM0 model was the first to describe delegation between regular roles within the RBAC96 role context. A notable extension and formalisation of the RBDM0 model is presented in the rule-based approach of Longhua et al. [8].

As far as we are aware of, no work on conflict detection exists within the scope of the RBAC96 model. Other role-based frameworks [9] and description languages [6] address this topic, but the proposed detection and resolution mechanisms heavily depend on their specific definition of roles and policies and cannot be directly applied within RBAC96-type models.

5 Role-based Conflict Analysis

5.1 Using Prolog as a declarative language

Prolog is a declarative language based on facts, rules and questions with built-in support for backtracking. Strings beginning with an upper case letter indicate the use of a variable. Strings beginning with a lower case are constants. For a more thorough introduction to Prolog and its backtracking mechanism we refer to [5]. Facts are represented as n-ary relations of the form `fact(x1,x2,...,xn)`, always followed by a fullstop. So a valid set of facts to express that supervisor is a role, andreas is a user and andreas holds or is assigned to the role of a supervisor would be:

```
role(supervisor).
user(andreas).
holds(andreas, supervisor).
```

Rules take the form of the form `Head:-Body`, where the body is a conjunction of facts or other rules, all of which need to be satisfied such that the head of the rule can succeed. So the following rule would express that a role `R1` can be delegated by a user `U1` to a user `U2` if `role(R1)`, `user(U1)`, `user(U2)` can all be inferred from the facts given to the system, and the rule `holds(U1,R1)` evaluates to true:

```
can_delegate(R1, U1, U2):-
 role(R1),
 user(U1),
 user(U2),
 holds(U1, R1).
```

Prolog also allows us to ask questions. So in order to determine all the users with their assigned roles in our system, we ask:

```
| ?- holds(User, Role).
```

and would receive an answer such as:

```
User = andreas, Role = supervisor;
User = jonathan, Role = accountant;
User = james, Role = clerk;
```

This indicates that andreas holds the role of a supervisor, jonathan that of an accountant and james that of a clerk in the current system configuration. Asking the system a question such as "is 'clerk' a valid role in our system?" could be asked by typing

```
| ?- role(clerk).
```

and would result in `yes` as an answer, assuming that the fact `role(clerk)` is part of the fact base.

5.2 Specifying Role-based access control in Prolog

The basis for our simulation and later analysis is the RBAC96 access control model and the RBDM0 delegation model. A summary of the formal RBAC96 elements, the RBDM0 delegation extensions and our corresponding Prolog code can be found in Table 1.

Translating the RBAC96 model into Prolog clauses is straightforward. The basic many-to-many user-role and role-permission relations are expressed in the clauses `holds(User, Role)` and `cando(Role, Permission)`. The concept of sessions is captured using the simplified relation `plays(User, Role)` to express activation of a role. Role hierarchies are presented as a binary relation between roles using the predicate `superior(Role1, Role2)`. RBAC96 functions such as *Users:*$R \rightarrow U^2$, which delivers the set of users assigned to a role, had to be expressed in more complex rules and are not explained in more detail here. However, the full Prolog translation and the facts we used for describing our later scenario are also described in the Appendix.

The RBDM0 model defines the ability of a user to authorize another user to become a member of a delegated role. So unlike the decentralised administration and delegation of authority through defined administrative roles [13], a user can now make delegation decisions by himself and delegate roles he was originally assigned with to other users. Thus, a role can be assigned with original members and delegated members, a property defined by the RBDM0 functions *Users_O(r)* and *Users_D(r)* respectively. This is modeled in Prolog extending the user-role assignment relation, using the predicates `holds_o(User, Role)` and `holds_d(User, Role)`. So in order to determine which roles a user is generally assigned to we specified the following rules:

```
holds(User, Role):-
 holds_o(User, Role).

holds(User, Role):-
 holds_d(User, Role).
```

Delegation in the RBDM0 model is further based on the following assumptions:

- No delegation between members in the same role.
- Delegation is only allowed between the original holder of a role and a delegate who does not possess that role so far (One-step property).
- Delegation is total and all permissions associated with a role are delegated.

In Prolog, these assumptions are either implemented as specific rules or can be inferred from the given facts. Although the RBDM0 model specifies role revocation and timing properties we do not discuss these in this context and they are not part of the implementation.

5.3 Adding constraints to our specification

We implemented Separation of Duty constraints as a set of Prolog rules and used them for asking questions to our system. However, the rules could also be triggered by specified events such as administrative actions. In this case any administrative action would be checked against the specified constraints. Ideally conflicts should be detected even before the operation is fully carried out.

The constraints we implemented are a subset (Table 2) of the static and dynamic Separation of Duty constraints proposed in the taxonomy of Simon and Zurko [18] . Mutually exclusive roles as described in [7] are added as facts to the Prolog fact base. Thus, declaring the two roles of accountant and clerk to be exclusive would be specified as the symmetric relation:

```
mutex(accountant, clerk).
```

The Static Separation of Duty (SSoD) constraint defines that two roles are strongly exclusive, if 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), the Object-based Separation of Duty (ObjSoD) 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. ObjSoD allows for simultaneous activation, but a user must not use any of his exclusive roles to act upon an object he has acted upon before in another of his exclusive roles. 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 process.

When specifying these Prolog conflict detection rules, we tried to keep our specification as simple as possible using pure Prolog. However, at certain points we had to make use of standard built-in predicates such as `setof/3`, as they would allow us to perform more complex operations such as determining the set of assigned roles for a user. Static conflict detection is solely based on the facts given to our system at compile-time. Dynamic conflict detection required the simulation of user behaviour at run-time. We had to use the `assert/1` and `retract/1` database manipulation features of Prolog to insert and delete new or obsolete facts to or from the database.

An example for such a dynamic manipulation would be the insertion of the fact that user andreas activated his supervisor role, which would be expressed as:

```
assert(plays(andreas, supervisor)).
```

This adds the fact `plays(andreas, supervisor)` to our fact base and is from the time of insertion used for any search by the Prolog inference engine.

6 A cheque processing scenario

We use the standard simplified example of an accountancy department processing cheques as a basis for the simulation of Separation of Duty conflicts and their further analysis. Issuing a cheque is a sensitive process. Separation of Duty constraints are used in order to prevent a single user from preparing, signing and dispatching a cheque all by himself. In our scenario preparing a cheque means that the details are filled in by an accountant. The supervisor then signs the cheque, as he alone has the legally required signature authority over the account from which the money is drawn. A clerk finally dispatches the cheque to the recipient. We assume that in this case the role of the supervisor will be assigned with the permission to print a signature onto the cheque, thus giving any member of the supervisor role the ability to sign the cheque, although he might not necessarily be legally entitled to do so. The initial configuration of our system is defined as follows (See Appendix and Figure 2): User andreas is assigned to the role of the supervisor, user jonathan is assigned to the role of the accountant and clerk, and the users jeremy and james are assigned to the role of a clerk. The accountant can prepare cheques, the supervisor can sign cheques, and the clerk can only dispatch cheques. The roles of the supervisor and accountant and the roles of the accountant and clerk are mutually exclusive. Mutual exclusiveness is non-transitive.

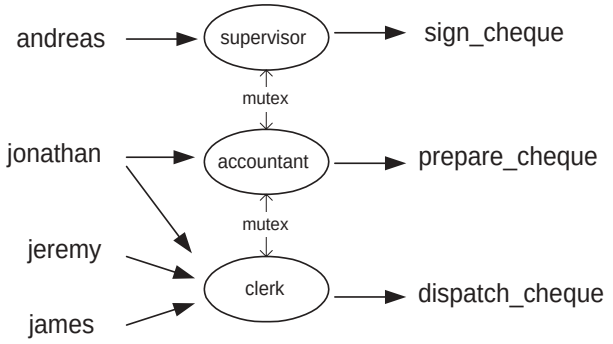

Figure 2. Original assignments

In this initial configuration the system will report a Static Separation of Duties conflict for user jonathan in his role as an accountant and clerk when posed with the query:

```
| ?- staticsod(User, Role1, Role2).

User = jonathan ,
Role1 = accountant ,
Role2 = clerk ;
```

We can imagine this constraint to be relaxed in order to allow the accountant to prepare and dispatch cheques in times of limited processing capacities due to staff shortage. Now the users activate all their roles, simulated by asserting facts such as `assert(plays(jonathan, accountant))` into the database. As expected, the simple dynamic Separation of Duties constraint is broken for user jonathan:

```
| ?- dynamicsod(User, Role1, Role2).

User = jonathan ,
Role1 = accountant ,
Role2 = clerk ;
```

Again, we relax this constraint and continue to simulate the execution of permissions. The fact that the prepare cheque permission was executed on the supplier cheque object by user jonathan in his role as an accountant is again asserted to the system through the following clause:

```
assert(was_executed_on(
       prepare_cheque, supplier_cheque,
       jonathan, accountant)).
```

In a similar way we let andreas sign the supplier cheque and james dispatch the cheque. So far no further dynamic constraints are broken. Things are different for the customer cheque. This refund to a priority customer requires fast processing and so jonathan prepares the cheque in his role as an accountant and later dispatches the same cheque in his role as a clerk. If we now check for any constraints to be broken, the system will report:

```
| ?- objectsod(User,Object,Role1,Role2).

User = jonathan,
Object = customer_cheque,
Role1 = accountant,
Role2 = clerk ;
```

Jonathan accessed the customer cheque object twice, once in his role as an accountant and once in his role as a clerk, which are mutually exclusive. The object-based Separation of Duty constraint is broken.

RBAC96 and RBDM0 model components	Prolog implementation counterparts
1. P, R, U and S are sets of permissions, roles, users and sessions respectively. 2. *UA* $\subseteq U \times R$ is a many to many user to role assignment relation. 3. *PA* $\subseteq P \times R$ is a many to many permission to role assignment relation 4. *RH* $\subseteq R \times R$ is a partial order on R, expressing the role hierarchy. 5. *UAO* $\subseteq U \times R$ is a many to many original user to role assignment relation. 6. *UAD* $\subseteq U \times R$ is a many to many delegated user to role assignment relation. 7. *UA* = *UAO* $\cup$ *UAD* 8. *Users_O(r)*=$\{u \mid (\exists r' \geq r')(u, r') \in UAO\}$ 9. *Users_D(r)*=$\{u \mid (\exists r' \geq r')(u, r') \in UAD\}$ 10. *Users(r)= Users_O(r)* $\cup$ *Users_D(r)* 11. *Users*: $R \rightarrow 2^U$ is a function mapping each role to a set of users. 12. *User*: $S \rightarrow U$ is a function mapping each session to a single user. 13. *Roles*: $S \rightarrow 2^R$ is a function mapping each session to a set of roles. 14. *Permissions*: $S \rightarrow 2^P$ is a function derived from *PA*, mapping each session to a set of permissions	1. RBAC96 definition 1. represented by the facts `permission(P), role(R), user(U).` Sessions are not explicitly modeled. 2. RBAC96 definition 2. represented by the fact `holds(User, Role).` 3. RBAC96 definition 3. represented by the fact `cando(Role, Permission).` 4. RBAC96 definition 4. represented by the fact `superior(Role1, Role2).` Partial order not checked, could be implemented as a spanning tree. 5. RBAC96 definition 5. represented by the fact `holds_o(User, Role).` 6. RBAC96 definition 6. represented by the fact `holds_d(User, Role).` 7. RBAC96 definition 7. represented by the rule `holds(User, Role):-holds_o(User, Role), holds_d(User, Role).` 8. RBAC96 definitions 8.-10. implied by Prolog 5.-7. implementations. 9. RBAC96 definitions 11. represented by clause `setof(User, holds(User, Role), Set_of_assigned_users).` 10. RBAC96 definitions 12. and 13. indirectly represented by clause `setof(User, plays(User, Role), Set_of_assigned_users).` 11. RBAC96 definition 14. requires the more complex operation clause `collect_all_permissions(List_of_Roles_for_User, List_of_permissions).`

Table 1. RBAC96/RBDM0 components and Prolog counterparts

Static SoD Rules	Description

`staticsod(U, R1, R2):-`
`  holds(U, R1),`
`  mutex(R1, R2),`
`  holds(U, R2).`
	There is a conflict if a user U assigned to two exclusive roles R1, R2.

Dynamic SoD Rules	Description

`dynamicsod(U, R1, R2):-`
`  staticsod(U, R1, R2),`
`  plays(U, R1),`
`  plays(U, R2).`
	There is a conflict if a user U activates two exclusive roles R1, R2 simultaneously.

`objectsod(U, Ob, R1, R2):-` `  ...(See Appendix)`	There is a conflict if a user U has accessed an object Ob twice through different exclusive roles R1, R2.
`operationalsod(U, Op, P1, P2):-` `  ...(See Appendix)`	There is a conflict if a user U has all permissions needed in an operation Op through the union of his exclusive roles. P1 represents the set of his permissions, P2 the permissions required in the operation.

Table 2. Separation of Duty Rules

Still the operational Separation of Duty constraint has not been broken. Assuming standard RBAC96 mechanisms and the given initial configuration, this constraint will never be broken by any regular user activities. However, simulating a user to user role delegation, we can cause a conflict. We now assume that andreas delegates his role as a supervisor to jonathan. This is inserted as the fact `holds_d(jonathan,supervisor)` to the fact base using the `delegates(andreas, supervisor, jonathan)` rule. The newly delegated assignment is now being represented by the dashed arrow in Figure 3.

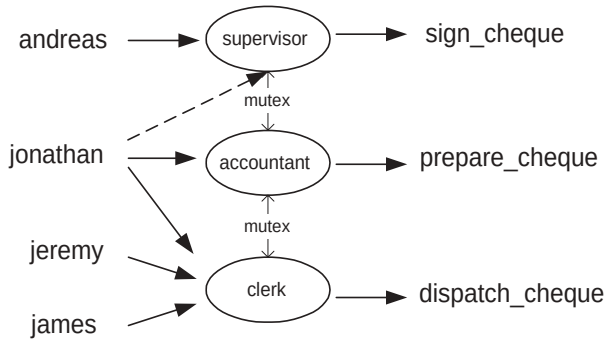

Figure 3. Original and delegated assignments

This will now cause an operational separation of duty constraint to be broken as jonathan holds the original roles of an accountant and clerk and the delegated role of a supervisor, where accountant and supervisor and accountant and clerk are declared as exclusive. Through these roles, he now holds all the permissions (P1) required for processing a cheque (P2):

```
| ?- operationalsod(User,Object,P1,P2).

User = jonathan ,
Object = process_cheque ,
P1 = [prepare_cheque, sign_cheque,
      dispatch_cheque],
P2 = [sign_cheque, dispatch_cheque,
      prepare_cheque];
```

Although we only demonstrated how delegation activities between users can cause an operational Separation of Duty constraint to be broken, it is clear that also any other dynamic constraint can be proven to conflict with delegation activities. We can no longer rely on checking for conflicts at compile time only. Conflict detection checks must be made with any delegation activity, since it is possible to create new user/role assignments at run-time.

7 Analysis of Conflicts

Considering the simulation of our conflict detection rules, we can make the following observations.

7.1 Non-hierarchical RBAC96

In a standard RBAC96 model without role hierarchies but with sessions we were able to simulate static, dynamic and object-based Separation of Duty constraints to be broken. This was on the basis of a given initial configuration and a set of user actions.

The interesting question in this case is where the origin of these conflicts lies. We see two main possibilities. On the one hand it may be the case that mutual exclusion properties are too strict and do not reflect operational needs. On the other hand the system is too complex and the side-effects of administrative actions are difficult to determine.

7.2 Hierarchical RBAC96

Role inheritance in the RBAC96 context means that permissions associated with a role are inherited upwards. Adding hierarchies to the standard RBAC96 model increases model complexity to a great extent. Now the administrator not only has to be careful about the definition of mutually exclusive roles and user to role assignment, but also about the specification of an inheritance hierarchy. In our implementation, the conflict detection rules are not greatly affected by the introduction of role hierarchies and only another rule allowing us to traverse the role hierarchy using the Prolog backtracking mechanisms is needed.

The rule for the traversal of role hierarchies is a standard backtracking rule as described in [5].

```
inherits_from(Super_Role,Sub_Role):-
 superior(Super_Role,Sub_Role).

inherits_from(Super_Role,Sub_Role):-
 superior(Super_Role,Sub_Sub_Role),
 inherits_from(Sub_Sub_Role,Sub_Role).
```

Thus in case of the static separation of duty rule we would now have to ask: 1) Which role(s) does a user hold either directly or be means of inheritance 2) Are these role(s) part of a mutual exclusion relationship? The efficiency of this was already demonstrated by us in a prototype tool presented in an invited talk at [16].

Again, we can observe that Separation of Duty constraints can be broken as described in Section 7.1. Additionally, we can obtain conflicts through a) manipulation of role hierarchies and b) assignment of users to roles part of a hierarchy. Private roles as introduced in [14] might be used as a mitigating mechanism to suppress unwanted inheritance.

7.3 Non-hierarchical RBAC96 and RBDM0

However, the focus of this paper is on the extension of a hierarchy free RBAC96 model with delegation mechanisms. The simulation we made showed that now also administrative actions by a user of the system can lead to conflicts. All the static, dynamic, object and operational constraints we specified can be broken by a) ordinary user activities (e.g. role activation, object access) and b) simple user to user delegation activities.

7.4 Hierarchical RBAC96 and RBDM0

Again, introducing role hierarchies into an extended RBAC96 model results in an increasing complexity. Now the delegation of a role which is part of a role hierarchy can also lead to explicit or implicit conflicts with static and operational separation of duty conflicts. In combination with certain user activities all four constraints can be broken.

8 Conflict Analysis - Why and How?

The scenarios and conflict examples we have presented in this paper may appear trivial. Yet, real role-based systems are far more complex. The declaration of mutually exclusive roles and role hierarchies might be done centrally for the whole organisation, whilst user-role assignment activities might occur within an application specific context. Thus, it is often difficult to support an administrator in his work, enabling him to observe what the consequences of his actions would be prior to final commitment.

We have experimented coupling Prolog with a relational database and a graphical interface. The database would be used to store the facts, while Prolog would be used to express rules. We found that the management mechanisms of a database are very useful for maintaining the integrity of our data whilst Prolog is far more efficient for processing recursive queries as they result out of role hierarchies. Considering the performance of such an approach depends on the search strategy, depth and width of eventual hierarchies, design of rules and facts, and the compiler strategy and query engine. We have not yet experienced any technical problems with this approach and would rather see the real difficulty in mapping information about organisational structures and workflows to the restricted form of database tables.

9 Resolving Conflicts

We argue that both, constraint specification and role delegation, are valid and useful extensions to a role-based system. However, we have seen that conflicts are possible. Too many conflicts indicate an inefficient system configuration, either because separation rules are too strict or the delegation activities are not sufficiently restricted. Simply stating that separation rules always have precedence would be one possible way, but other ways of resolving these conflicts must be discussed, especially when considering the implementation of role hierarchies, more elaborate delegation models such as [8], and the decentralisation of administrative activities through administrative roles [13]. Within our simple scenario and the limitations of the RBAC96 and RBDM0 models we see the following possibilities:

- Constraining the delegation by introducing sets of roles that cannot be delegated, sets of users that cannot delegate and sets of certain users that cannot delegate certain roles.
- Constraining delegation by evaluating the current context, e.g. which user already holds and plays which original and delegated roles.
- Constraining delegation with respect to the history of a user in his role, e.g. executed permissions or accessed objects.
- Immediate revocation of delegated roles according to the principle of least privilege.
- Temporary revocation or deactivation of his original roles, e.g. a user has to cover his colleague, thus he will be delegated the needed roles but his original roles are revoked for the time of coverage.

A further possibility for conflict resolution would be the application of a different definition of a role as we suggested and described in [17]. This however, would require the RBAC96 and RBDM0 model to be changed.

10 An extended definition of a role

In the RBAC96 model ”Role” is an atomic concept, defined as ”...a named job function within the organization”. We provide an extended definition of a role, distinguishing between official positions within the organisational hierarchy and descriptions of the job function of employees. From now on we will refer to a role using the construct *function/Official Position*. We will use lower case letters for functions and title case letters for positions. An example of this would be the *supervisor/Group Manager* role, indicating that somebody has the function of being a supervisor and holds the official position of a Group Manager. If we decided to delegate not the entire role but only functions and compose new delegated roles, we would be able to solve some of the previously detected conflicts. However, this notion of an extended role would also require a different definition of delegation rules.

10.1 Delegating functions

Let us assume the following scenario. We have four extended roles defined by the tuples (Figure 4):

- *clerk/Employee*
- *clerk/Team Manager*
- *accountant/Team Manager*
- *supervisor/Group Manager*

These are identical to our previously used examples in the cheque issuing process. However, the assignment of users to those roles is different. We remember that we had assigned the user jonathan to the exclusive roles of accountant and clerk such that we could deal with certain types of cheques more effectively. In our new definition of a role we would only assign jonathan to a different function, his position will remain unchanged.

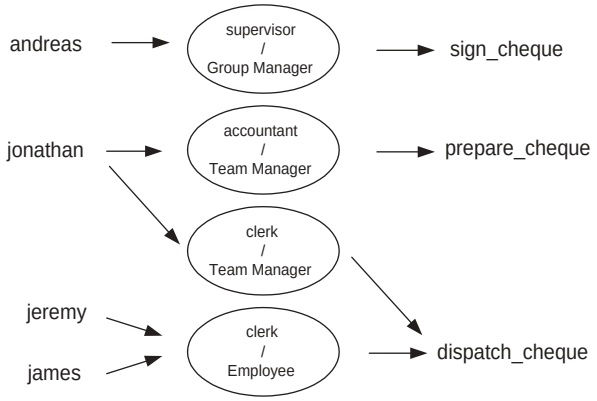

Figure 4. Extended role assignment

Of course, we could have also chosen to let the *accountant/Team Manager* role inherit the functionality of a clerk to achieve the same effect, but we assume that no inheritance mechanisms are present. Let us now see how this extended definition could be used to resolve some of the conflicts identified earlier on. We first have to provide a new definition for role delegation. A possible extended functional delegation rule could be:

Def. 1 *A user u1 can only delegate the function f1 of a role (f1/P1) to another user u2, if:*

- *u1 is assigned to an original role (f1/P1) and*
- *u2 also holds Position P1*

So what effect does this rule have with respect to the problem of conflicts between Separation of Duty properties and role delegation? According to our definition of simple static and dynamic Separation of Duty properties (Table 2), we might still have conflicts for the above initial assignment. This depends on whether we declare two roles to be mutually exclusive on basis of their functions or not. If the functions of accountant and clerk are still exclusive and simple static and dynamic constraints are based on this property we will still obtain conflicts as in our earlier scenario. However, the difference to our scenario is that user andreas will not be able to delegate the supervisor function to jonathan anymore, as their positions are different. Thus, an operational Separation of Duty constraint will be difficult to break assuming that sensitive permissions such as signing a cheque are only combined with senior positions and are thus less likely to be delegated.

11 Summary and Conclusion

We have demonstrated how to implement and enforce a set of static and dynamic Separation of Duty constraints in a role-based access control model, using a rule-based, declarative approach. Depending on the type of RBAC model, the initial configuration, administrative actions and user behaviour, these can be broken. Extending the model with simple delegation mechanisms is an additional source of conflict.

Simulating a role-based model with integrated constraints and delegation mechanisms is only a first step. We intend to pursue further work on how to detect and resolve these conflicts in a more formal model. According to our definition of an extended role we might want to make a distinction between function and position hierarchies. This extended definition of a role and distinction between the function and position of an employee seems to be a first step into the right direction. However, apart from redefining delegation rules, well-known Separation of Duty constraints might also have to be changed in order to cater for this extension.

We have recently finished work describing how we use the specification language Alloy and its model checking facilities to analyse the implications of the simultaneous integration of administrative role-based access control (ARBAC) extensions and constraints. What we need to investigate now is the relationship between such a formal specification and a set of executable rules as presented here.

12 Acknowledgements

The author is sponsored by the Engineering and Physics Research Council (EPSRC) under award no. 99311141. The comments from the reviewers and from Dr. J. D. Moffett helped to clarify this paper. Further support was given from the HISE research group under Prof. J. McDermid.

References

[1] G. Ahn. *RCL 2000*. Phd dissertation, George Mason University, 2000.

[2] E. Barka and R. Sandhu. Framework for Role-Based Delegation Models. In *16th Annual Computer Security Applications Conference*, New Orleans, Louisiana, 2000.

[3] F. Chen and R. Sandhu. Constraints for RBAC. In *1st ACM Workshop on Role-Based Access Control*, pages 39–46, Gaithersburg, MD, 1995.

[4] D. Clark and D. Wilson. A Comparison of Commercial and Military Security Policies. In IEEE Computer Society Press, editor, *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, California, 1987.

[5] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer, 4th edition, 1996.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Policies for Distributed Systems and Networks*, volume 1995, pages 18–38, Bristol, 2001. Springer Lecture Notes in Computer Science.

[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] Z. Longhua, G. Ahn, and Chu. B. A Rule-based Framework for Role-Based Delegation. In *ACM SACMAT*, Chantilly, VA, USA, 2001.

[9] E. Lupu, D. Marriott, M. Sloman, and N. Yialelis. A policy based role framework for access control. *Proceedings of the first ACM Workshop on Role-based access control*, pages 215–224, 1996.

[10] J. Moffett and M. Sloman. The Source of Authority for Commercial Access Control. *IEEE Computer*, pages 59–69, 1988.

[11] M. Nash and K. Poland. Some Conundrums Concerning Separation of Duty. In IEEE Computer Society Press, editor, *IEEE Symposium on Security and Privacy*, pages 201–209, Oakland, CA, 1990.

[12] R. Sandhu. Transaction Control Expressions for Separation of Duties. In *4th Aerospace Computer Security Conference*, pages 282–286, Arizona, 1988.

[13] 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.

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

[15] 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.

[16] A. Schaad and J. Moffett. The Incorporation of Control Principles into Access Control Policies (Extended Abstract). In *Hewlett Packard Policy Workshop*, Bristol, 2001.

[17] A. Schaad, J. Moffett, and J. Jacob. The access control system of a European bank - a case study. In *ACM Symposium on access control models and technologies (SACMAT)*, Chantilly, VA, USA, 2001.

[18] R. Simon and M. Zurko. Separation of Duty in Role-Based Environments. In *Computer Security Foundations Workshop X*, Rockport, Massachusetts, 1997.

[19] M. Sloman and J. Moffett. Delegation of Authority. In *Integrated Network Management II*, pages 595–606. North Holland, 1991.

A Prolog Source Code

```
%------------------------------------------------------------------------
%       Conflict detection in a role-based delegation model
%------------------------------------------------------------------------
%Author:    Andreas Schaad
%Date:      01/06/2001

%Simple Simulation of the RBAC96 model and RBDM0 delegation extensions
%Integration of static and dynamic Separation of Duty properties
%based on mutually exclusive roles.

%Facts are represented as standard scenario of cheque processing
%------------------------------------------------------------------------

%------------------------------------------------------------------------
%               Pre-processor
%------------------------------------------------------------------------
:- dynamic [plays/2].               %Simulate role activation
:- dynamic [was_executed_on/4].     %Simulate permission execution
:- dynamic [holds_d/2].             %Simulate delegation

unknown_predicate_handler(_,fail).
%------------------------------------------------------------------------

%------------------------------------------------------------------------
%                  Facts
%------------------------------------------------------------------------
user(andreas).                      %System Users
user(jonathan).
user(jeremy).
user(james).

role(supervisor).                   %System Roles
role(accountant).
role(clerk).

permission(sign_cheque).            %System Permissions
permission(issue_cheque).
permission(prepare_cheque).

cheque(customer_cheque).            %System Objects
cheque(supplier_cheque).

holds_o(andreas, supervisor).       %Original User - Role assignemnt
holds_o(jonathan, accountant).
holds_o(jonathan, clerk).
holds_o(jeremy, clerk).
holds_o(james, clerk).

cando(supervisor, sign_cheque).     %Role - Permission assignment
cando(accountant, prepare_cheque).
cando(clerk, dispatch_cheque).

%superior(Role1, Role2)             %Role Hierarchy

mutexclusive(supervisor, accountant).   %Mutualy exclusive roles
mutexclusive(accountant, clerk).

mutex(R1, R2):-                     %Symmetry rule
    mutexclusive(R1, R2);
    mutexclusive(R2, R1).

%required permissions for processing a cheque
operation(process_cheque, [prepare_cheque, sign_cheque, dispatch_cheque]).

%------------------------------------------------------------------------
%           Separation of Duty constraints
%------------------------------------------------------------------------
%Simple static SoD: A user must not be assigned to any two
%mutually exclusive roles r1,r2.

staticsod(User, Role1, Role2):-
    holds(User, Role1),
    mutex(Role1, Role2),
    holds(User, Role2).

%Simple dynamic SoD: A user can be assigned to any two
%mutually exclusive role r1,r2, but must not activate them at the same time.

dynamicsod(User, Role1, Role2):-
    staticsod(User, Role1, Role2),
    plays(User, Role1),
    plays(User, Role2).

%Object-based Separation of Duties: User can hold and play
%mutually exclusive roles.
%He just may not act upon the same object through any of his mutex roles.
%Fact that object was accessed is recorded in
%was_executed_on(sign_cheque, cheque(customer_cheque), jonathan, supervisor).

objectsod(User, Object, Role1, Role2):-
    was_executed_on(Permission1, Object, User, Role1),
    was_executed_on(Permission2, Object, User, Role2),
    Role1\=Role2,
    mutex(Role1, Role2).

%Operational Separation of Duties: must not be in possession of all permissions
%required in a sensitive operation.
operationalsod(User, Operation, Processlist, Permissionlist):-
    user(User),
    operation(Operation, Processlist),
    collect_mutex_roles(User, Mutexlist),
    collect_all_permissions(Mutexlist, Permissionlist),
    subset(Processlist, Permissionlist).

%------------------------------------------------------------------------
%           Delegation extensions
%------------------------------------------------------------------------

holds(User, Role):-                 %General rule: UA = UAO union UAD
    holds_o(User, Role).

holds(User, Role):-
    holds_d(User, Role).

delegates(User1, Role1, User2):-    %inserts role delegation as fact
    holds_o(User1, Role1),
    not holds_o(User2, Role1),
    asserta(holds_d(User2, Role1)).

%------------------------------------------------------------------------
%       Additional functions - Not model specific
%------------------------------------------------------------------------

%Collects all mutually exclusive roles for a user and
%make sure that the resulting list contains no doubles.
collect_mutex_roles(User, Nodoubleslist):-
    call(holds(User, Role)),
    call(mutex(Role, R)),
    assertz(queue(Role)),
    fail;
    assertz(queue(end)),
    collect(Total),
    set(Total, Nodoubleslist).

collect(Total):-
    retract(queue(List)),
    !,
    (List==end,!,Total=[];
    append([List], Rest, Total), collect(Rest)).

%Collects for a given set of roles, the union of all permissions
collect_all_permissions([H|T], List_of_permissions):-
    T=[], !,
    assert(rolestack(H)),
    assert(rolestack(end)),
    retrieve(List_of_permissions);
    assert(rolestack(H)),
    collect_all_permissions(T, List_of_permissions).

retrieve(Total):-
    retract(rolestack(Role)),
    !,
    (Role==end,!,Total=[];
    setof(Perms, cando(Role, Perms), Permlist),
    append(Permlist, Rest, Total), retrieve(Rest)).

%Union of two sets
union([], Ys, Ys).
union([X|Xs], Ys, Zs):-
   member(X, Ys), !, union(Xs, Ys, Zs).
union([X|Xs], Ys,[X|Zs]):-union(Xs, Ys, Zs).

%Removes double entries in a set X and gives cleared set Y.
set(Xs, Ys):-
    set_1(Xs, [], Ys).
set_1([], As, As). set_1([X|Xs], As, Ys):-
    member(X, As), !, set_1(Xs, As, Ys).
set_1([X|Xs], As, Ys):-set_1(Xs, [X|As], Ys).

%Is X a subset of Y? Careful! subset(Y, X).
subset([], _). subset([X|Xs], Ys):-
    member(X, Ys),
    subset(Xs, Ys).
```