Conflict Checking of Separation of Duty Constraints in RBAC - Implementation Experiences

Mark Strembeck
Department of Information Systems, New Media Lab
Vienna University of Economics and BA, Austria
mark.strembeck@wu-wien.ac.at

ABSTRACT
Separation of duty constraints define mutual exclusion relations between two entities (e.g. two permissions). Thus, a software component that supports the definition of separation of duty constraints implicitly requires a means to control their definition and to ensure the consistency of the resulting runtime structures. In this paper, we present our experiences with the implementation of conflict-checking methods for separation of duty constraints in the XORBAC access control service.

KEY WORDS
Role-based Access Control, Separation of Duty

1 Introduction

In access control, *separation of duty constraints* enforce conflict of interest policies. Conflict of interest arises as a result of the simultaneous assignment of two mutual exclusive permissions or roles to the same subject. *Mutual exclusive* roles or permissions result from the division of powerful rights or responsibilities to prevent fraud and abuse. An example is the common practice to separate the “controller” role and the “chief buyer” role in medium-sized and large companies. A particular access control specification (an access control policy rule set and the corresponding subject-assignment relations) is said to be *safe* iff no subject can obtain an “unauthorized” right. In other words, no conflicting permission can ever be assigned to the current permission-set of a particular subject, neither directly nor via a role. However, since the verification of the safety property for general access control models, like role-based access control (RBAC), is not decidable [4], constraints are an attempt to enforce the safety property via explicit modeling-level artifacts.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of role-based access control and the XORBAC access control component. Subsequently, Section 3 describes which checks need to be performed with respect to definition of separation of duty (SOD) constraints and motivates how SOD constraints are inherited in a role hierarchy. Section 4 then introduces the methods that are needed to check SOD constraints in XORBAC, before we describe the conflict checking methods for permission-to-role assignment (Section 5), role-to-subject and permission-to-subject assignment (Section 6), and role-to-role assignment (Section 7). Section 8 concludes the paper.

2 The xoRBAC Component

Access control deals with the elicitation, specification, maintenance, and enforcement of authorization policies in software-based systems (see also [5, 11, 12]). In order to allow for an (automated) enforcement of authorization policies, the high-level control objectives of a system need to be mapped to the structures provided by an access control model. *Access control model* provide a framework for the definition of authorization policies. The three most important classes of access control models are discretionary access control (DAC), mandatory access control (MAC), and role-based access control (RBAC).

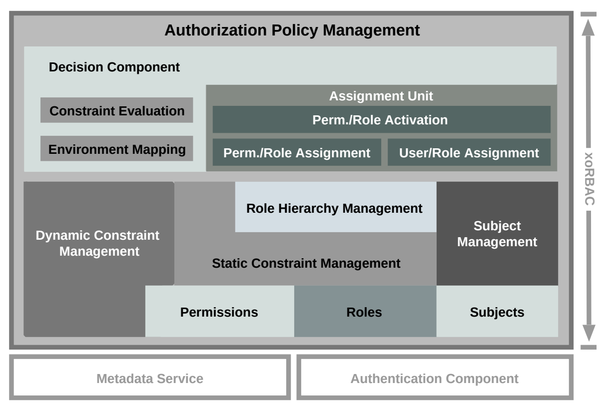

Figure 1. XORBAC: conceptual structure

DAC is sometimes criticized as conceding too many liberties to the rights-manager, while MAC commonly is regarded as being too restrictive for most applications (cf. [6, 11]). RBAC [3] offers a promising alternative. In recent years RBAC (together with various extensions) has developed into the de facto standard for access control in both research and industry. One of the advantages of RBAC is being a general access control model. This means that a sophisticated RBAC-service may be configured to enforce many different access control policies, including DAC- or MAC-based policies (see [10]). A central idea in RBAC is to support constraints on almost all

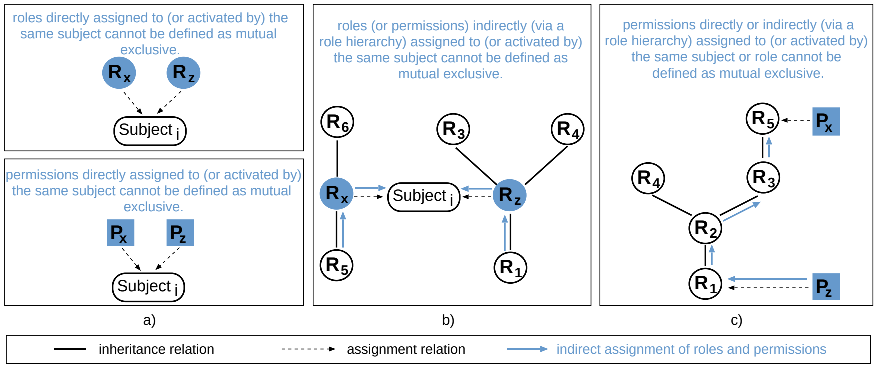

Figure 2. Required checks when defining SOD constraints

parts of an RBAC model (e.g. permissions, roles, or assignment relations) to achieve high flexibility. Static and dynamic separation of duty (see [2]) are two of the most common types of RBAC constraints (see e.g. [1]).

The XORBAC component [7, 8] provides an RBAC service that can be used on Unix and Windows systems with applications providing C or Tcl linkage. XORBAC is implemented with XOTcl [9]. While originally developed as an RBAC service, XORBAC was extended to provide a multi-policy access control system which can enforce RBAC, as well as DAC or MAC based policies including *conditional permissions* (see [8]). Figure 1 depicts the conceptual structure of XORBAC. The *Static Constraint Management* of XORBAC enables the definition of static separation of duty (SSD) constraints and cardinalities. The *Dynamic Constraint Management* allows for the definition of dynamic separation of duty constraints as well as the definition of context conditions and context constraints (see [8]).

3 Separation of Duty Constraints

Mutual exclusion relations are defined via separation of duty (SOD) constraints. SOD constraints can be subdivided in static separation of duty (SSD) constraints and dynamic separation of duty (DSD) constraints:

- *Static separation of duty* constraints specify that two mutual exclusive roles (or permissions) must *never* be *assigned* to the same subject *simultaneously*.

- *Dynamic separation of duty* constraints define that two mutual exclusive roles (or permissions) must *never* be *activated* by the same subject *simultaneously*. This means that two dynamically mutual exclusive roles may be assigned to the same subject. The corresponding subject, however, is only allowed to activate at most one of its dynamically mutual exclusive roles (permissions) at the same time.

Figure 3 depicts the assignment relations between subject, roles, and permissions as they are defined in XORBAC. Each of these relations influences, and is itself influenced by, the definition of SOD constraints. The XORBAC API offers methods to define separation of duty constraints for permissions and roles. Subsequently, we provide a detailed discussion how and when the permission-to-role assignment (PRA), the permission-to-subject assignment (PSA), the role-to-subject assignment (RSA), and the definition of a role hierarchy (role-to-role assignment, RRA) influence the definition of SOD constraints or are themselves influenced by existing (already defined) SOD constraints.

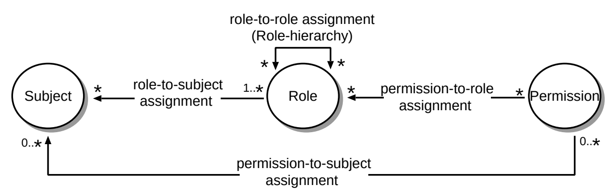

Figure 3. xoRBAC Assignment relations

Figure 2 indicates which checks need to be performed prior to establishing a new SOD constraint. Two roles (or permissions) that are assigned to the same subject cannot be defined as statically mutual exclusive, while two roles (permissions) that are concurrently activated by the same subject cannot be defined as dynamically mutual exclusive. This equally applies to roles and permissions that are directly or indirectly (via a role-hierarchy) assigned to a subject.

At any time, an XORBAC subject possesses a set of permissions that results from the set union of the permissions which are directly assigned to this subject, the permissions it receives from its directly assigned role(s), and the permissions it receives via the role hierarchy, i.e. the permissions that are assigned to the junior-roles of its directly assigned role(s). For example, the roles R_x and R_z in Figure 2b) cannot be defined as mutual exclusive since both are assigned to $Subject_i$. Likewise, the junior-roles R_1 and R_5 may neither be defined as mutual exclusive since they are indirectly assigned to $Subject_i$ via the role-hierarchy. Moreover, it is not sensible to define a role

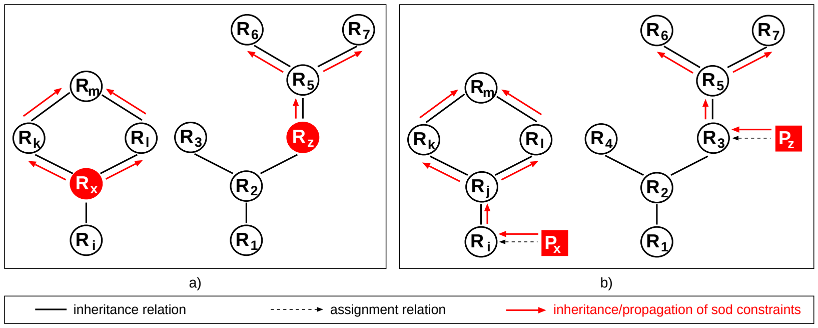

Figure 4. Inheritance/propagation of SOD constraints in role hierarchies

as mutual exclusive to one of its junior- or senior-roles (for example the roles R_x and R_5 in Figure 2). Similarly, defining P_x and P_z in Figure 2c) as mutual exclusive would cause a conflict, since R_5 simultaneously possesses P_x and P_z. Thus, in the presence of a role hierarchy SOD constraints are also subject to inheritance (see e.g. [3]). XORBAC allows to define SOD constraints on roles and on permissions. Both types of SOD constraints are inherited in XORBAC role-hierarchies. To avoid inconsistencies or unreasonable configurations, XORBAC performs the tests discussed above prior to setting a SOD-constraint.

Figure 4 shows examples for the inheritance/propagation of SOD constraints in role-hierarchies. Throughout the remainder of this paper we use red (filled) circles (for roles) and red (filled) rectangles (for permissions) to indicate that two roles or permissions are defined as mutual exclusive. In Figure 4a), the roles R_x and R_z are mutual exclusive. This mutual exclusion constraint is inherited by the corresponding senior-roles. Therefore, the SOD constraint between R_x and R_z also prevents that a senior-role of R_x (for example R_m) and a senior-role of R_z (for example R_6) are simultaneously assigned to (or activated by) the same subject. In other words, if a given role R_x is defined as mutual exclusive to a role R_z, then R_x is also mutual exclusive to all senior-roles of R_z (and vice versa). This is true since a subject that is assigned to a senior-role of R_z (e.g. to role R_6) also transitively possesses R_z (and all other junior-roles of role R_6). Thus, two roles may be mutual exclusive because two of their junior-roles are mutual exclusive, like R_m and R_6 in Figure 4a) for example.

Similar to SOD constraints on roles, SOD constraints on permissions are inherited within a role-hierarchy. In Figure 4 b) the permissions P_x and P_z are defined as mutual exclusive. Thus, R_i and R_3 (and all respective senior-roles) inherit this mutual exclusion constraint. In other words, R_i and R_3 are mutual exclusive because they own mutual exclusive permissions. Therefore, neither R_i and R_3 nor two of the corresponding senior-roles can be simultaneously assigned to the same subject. If, however, P_x or P_z are revoked from R_i resp. R_3, the mutual exclusion constraint between these roles is automatically erased at the same time.

4 Checking of SOD Constraints

Figure 5 shows four methods that are applied to check SOD constraints for roles. The `isMutualExclusive` method of the `Role` class expects a mandatory parameter `role` and checks if the `Role` object which is calling this method is mutual exclusive to the `Role` object identified through `role`. In particular, the `isMutualExclusive` method calls two other methods to check the current definitions of SSD and DSD constraints. In the following, we especially show and describe source code examples for SSD constraints. This is sufficient since the source code for SSD and DSD constraints is nearly identical. The only difference is that, due to their static nature, SSD constraints must hold for all sessions simultaneously, while DSD constraints only apply to the roles and permissions activated in one particular session.

```
Role instproc isMutualExclusive {role} {
  if {[[self] isStaticallyMutualExclusive $role]} {return 1}
  if {[[self] isDynamicallyMutualExclusive $role]} {return 1}
  return 0
}
Role instproc isStaticallyMutualExclusive{role} {
  if {[[self] hasSSDRoleConstraintTo $role]} {return 1}
  if {[[self] hasSSDPermConstraintTo $role]} {return 1}
  return 0
}
Role instproc hasSSDPermConstraintTo{role} {
  foreach p [my getAllPerms] {
    foreach op [$role getAllPerms] {
      if {[$p isStaticallyMutualExclusive $op]} {return 1}
    }
  }
  return 0
}
Role instproc hasSSDRoleConstraintTo{role} {
  set mutlExclRoles [my getSSDRoleConstraints]
  if {$mutlExclRoles != ""} {
    if {[lsearch -exact $mutlExclRoles $role] != -1} {return 1}
  }
  return 0
}
```

Figure 5. Checking of SOD constraints in xoRBAC

The `isStaticallyMutualExclusive` method receives a parameter `role` and checks if the `Role` object calling the method has an SSD role constraint or an SSD permission constraint to the `Role` object identified through the `role` parameter (see Figure 5). In particular, the `hasSSDPermConstraintTo` method first fetches a list of all permissions assigned to the calling `Role` ob-

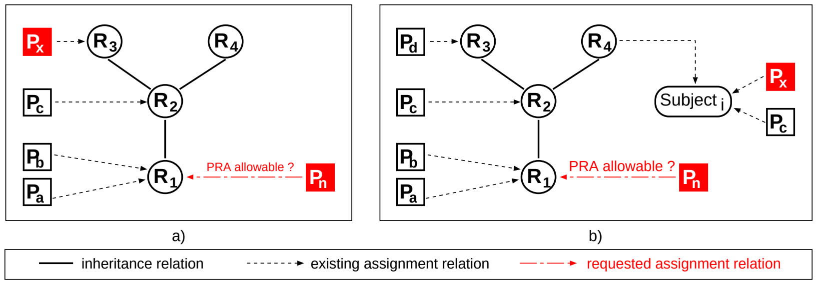

Figure 6. Example conflicts in PRA

ject (call of `[my getAllPerms]`) and to the `Role` object identified through the `role` parameter (call of `[$role getAllPerms]`). Subsequently, it checks if a permission in the first list is statically mutual exclusive to a permission in the second list. The `hasSSDRoleConstraintTo` method performs a similar check for mutual exclusive roles (cf. Figure 5).

```
Role instproc getSSDRoleConstraints{} {
  set all [concat [my getDirectSSDRoleConstraints]
                  [my getTransitiveSSDRoleConstraints]
                  [my getInheritedSSDRoleConstraints]]
  return [lsort -unique $all]
}
Role instproc getDirectSSDRoleConstraints{} {
  if {[my exists ssdconstraitns]} {
     return [my set ssdconstraints]
  }
  return ""
}
# Each owner of a "senior-role" can activate the corresponding
# junior-roles. Thus all senior-roles of a directly mutual
# exclusive role are also (transitively) mutual exclusive.
Role instproc getTransitiveSSDRoleConstraints{} {
  set transitive ""
  foreach role [my getDirectSSDRoleConstraints] {
    set transitive [concat $transitive [$role getAllSeniorRoles]]
  }
  return [lsort -unique $transitive]
}
Role instproc getInheritedSSDRoleConstraints{} {
  set inherited ""
  foreach role [my getAllJuniorRoles] {
    set inherited [concat $inherited [$role getSSDRoleConstraints]]
  }
  return [lsort -unique $inherited]
}
```

Figure 7. The `getSSDRoleConstraints` method

The `getSSDRoleConstraints` method returns a list of all `Role` objects that are statically mutual exclusive to the calling `Role` object (see Figure 7). This list consists of all direct, inherited, and transitive SSD constraints. Directly mutual exclusive roles are stored in the `ssdconstraints` instance variable of a role object and can be fetched via the `getDirectSSDRoleConstraints` method. The `getTransitiveSSDRoleConstraints` method returns the senior-roles of all directly mutual exclusive roles. With respect to the example in Figure 4a), a method call of R_x `getTransitiveSSDRoleConstraints` would result in a list consisting of R_5, R_6, and R_7. The `getInheritedSSDRoleConstraints` method calls the `getSSDRoleConstraints` method for all junior roles and returns a list of all inherited SSD role constraints. With respect to Figure 4a), the call of R_7 `getInheritedSSDRoleConstraints` would result in a list consisting of R_k, R_l, R_m, and R_x.

5 SOD Constraint Checking in PRA

Figure 6 shows two typical examples for conflicts arising in permission-to-role assignment (PRA). Figure 6a) sketches a situation where a new permission P_n should be assigned to role R_1. However, since P_x is mutual exclusive to P_n this assignment operation cannot be permitted. Otherwise R_3 would acquire both permissions P_x and P_n, causing a model inconsistency. Thus, the conflict checking method for PRA relations has to deny the respective assignment operation. Figure 6b) depicts a more complex situation where $Subject_i$ owns role R_4 and permission P_x, while a new permission P_n should be assigned to R_1. Similar to the example in Figure 6a) the conflict checking method for PRA relations must deny the respective assignment relation. Otherwise $Subject_i$ would acquire both permissions P_x and P_n (via the role hierarchy), causing a model inconsistency. Figure 8 shows the PRA conflict checking method of xoRBAC.

```
RightsManager instproc sodPermConstraintAllowPRA{perm role} {
  # first: check if one or more of the permissions that are
  # (directly or through inheritance) assigned to $role are
  # mutual exclusive to $perm
  foreach rp [$role getAllPerms] {
    if {[$perm isMutualExclusive $rp]} {return 0}
  }
  # now: check if one of the senior-roles of $role already owns
  # a permission that is mutual exclusive to $perm. In this case
  # the assignment of $perm to $role must be denied - otherwise
  # the corresponding senior-role would acquire two mutual
  # exclusive permissions
  foreach sr [$role getAllSeniorRoles] {
    foreach srp [$sr getAllPerms] {
      if {[$perm isMutualExclusive $srp]} {return 0}
    }
  }
  # finally: check if one or more of the subjects currently
  # owning $role do already own a permission that is mutual
  # exclusive to $perm.
  set roleowners [[self] getAllSubjectsOwningRole [$role name]]
  foreach subject $roleowners {
    foreach p [$subject getAllPerms] {
      if {[$p isMutualExclusive $perm]} {return 0}
    }
  }
  # if we get here permission-to-role assignment for $perm
  # and $role is allowable
  return 1
}
```

Figure 8. The PRA conflict checking method

The `sodPermConstraintAllowPRA` method receives two mandatory parameters `perm` and `role`, identifying a `Permission` object and a `Role` object respectively (see Figure 8). First, the method checks if a permission assigned to `role` is mutual exclusive to `perm`. If no mu-

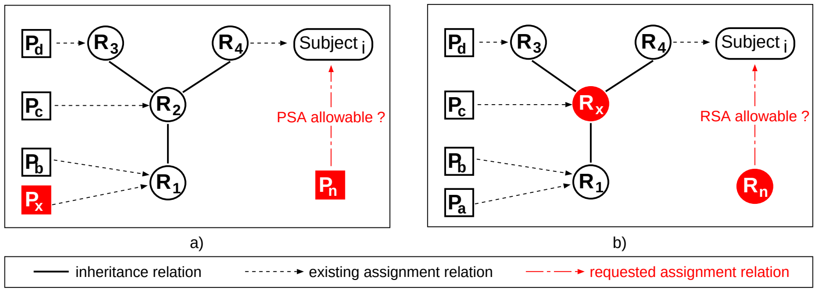

Figure 9. Example conflicts in PSA and RSA

tual exclusion is found, it checks if a senior-role of `role` owns a permission which is mutual exclusive to `perm`. Finally, the `sodPermConstraintAllowPRA` method checks if a subject owning `role` does also possess a permission being mutual exclusive to `perm`. In case none of these checks discovers a conflict, it is allowed to assign `perm` to `role` (with respect to the separation of duty constraints being in effect in that very moment).

6 SOD Checking in RSA and PSA

Figure 9 shows typical conflicts arising in permission-to-subject assignment (PSA) and role-to-subject assignment (RSA). In Figure 9a), the permissions P_x and P_n are mutual exclusive, and $Subject_i$ owns role R_4. Starting from this position, the new permission P_n should be directly assigned to $Subject_i$. However, this assignment operation must be denied by the respective conflict checking method. Otherwise $Subject_i$ would acquire both permissions P_x and P_n, causing a model inconsistency. Figure 9b) depicts a similar situation with two mutual exclusive roles R_x and R_n whereby the new role R_n should be assigned to $Subject_i$. This assignment operation must also be denied to prevent $Subject_i$ from obtaining both mutual exclusive roles R_n and R_x

```
RightsManager instproc sodPermConstraintAllowPSA{perm subject} {
  foreach sp [$subject getAllPerms] {
    if {[$sp isMutualExclusive $perm]} {return 0}
  }
  return 1
}
```

Figure 10. The PSA conflict checking method

The PSA conflict checking method of XORBAC (see Figure 10) receives two parameters `perm` and `subject` and checks if one of the permissions assigned to `subject` (directly or via roles) is mutual exclusive to `perm`. Figure 11 shows the `sodConstraintAllowRSA` method. This method first checks if one of the roles currently assigned to `subject` is mutual exclusive to `role`. If no mutual exclusion is found, it checks if one of the permissions that are directly assigned to `subject` is mutual exclusive to a permission the subject would acquire via the `Role` object identified through the `role` parameter.

```
RightsManager instproc sodConstraintsAllowRSA{role subject} {
  # first: check if a role assigned to $subject (directly
  # or via a role hierarchy) is mutual exclusive to $role
  foreach cr [$subject getAllRoles] {
    if {[$cr isMutualExclusive $role]} {return 0}
  }
  # now: check if one of the permissions that are directly
  # assigned to $subject are mutual exclusive to one of the
  # permissions assigned to $role.
  set directperms [$subject getAllDirectlyAssignedPerms]
  set newperms [$role getAllPerms]
  if {($directperms != "") && ($newperms != "")} {
    foreach dp $directperms {
      foreach np $newperms {
        if {[$dp isMutualExclusive $np]} {return 0}
      }
    }
  }
  # if we get here role-to-subject assignment of $role to
  # $subject is allowable
  return 1
}
```

Figure 11. The RSA conflict checking method

7 SOD constraint checking in RRA

The specification of a role hierarchy, resp. role-to-role assignment (RRA), influences and is influenced by each of the aforementioned assignment relations. Thus, RRA is the most complex operation with respect to SOD conflict checking. When establishing inheritance relations between two or more roles, SOD conflicts could result from joining together mutual exclusive permissions or roles - which may, again, happen in multiple different ways. Thus, on the source code level, the RRA conflict checking method of XORBAC sequentially calls the conflict checking methods discussed in the preceding sections.

Figure 12 shows typical conflicts arising in RRA. The following examples do not distinguish between SSD role constraints and SSD permission constraints since their respective effects on role-to-role assignment relations are quite similar. Figure 12a) depicts two independent role hierarchies, and R_x and R_z are mutual exclusive. Now, a new role R_n should unite (parts of) these two role hierarchies. The conflict checking method has to decide if the definition of a common senior-role for R_3 and R_6 is acceptable. In Figure 12a) the definition of a common senior-role cannot be allowed. Otherwise R_n (resp. an owner of R_n) would join two mutual exclusive roles, R_x and R_z. In Figure 12b) two role hierarchies exist, each containing one of two mutual exclusive roles R_x and R_z. Role R_n is a junior-role of R_z, and R_3 is a

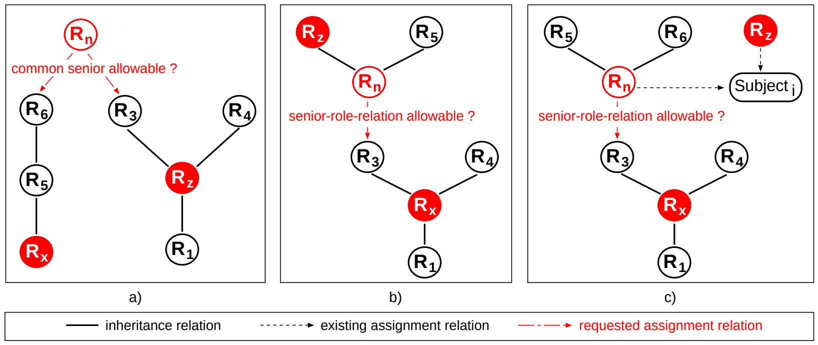

Figure 12. Example conflicts in RRA

senior-role of R_x. Now R_n should be defined as senior-role of R_3. However, the definition of the corresponding inheritance relation must be prevented by the conflict checking method. Otherwise the resulting role hierarchy would include an inheritance relation between R_z and R_x, implying that R_z is mutual exclusive to one of its junior-roles. In Figure 12c) $Subject_i$ owns the roles R_n and R_z, while R_x and R_z are mutual exclusive. Now R_n should be defined as senior-role of R_3. With respect to inter-role inheritance relations this assignment would not cause a conflict and could be allowed. Nevertheless, according to the special case shown in Figure 12c) R_n cannot be defined as senior-role of R_3. Otherwise $Subject_i$ would acquire both mutual exclusive roles R_x and R_z.

8 Conclusion

In this paper, we discussed the conflict checking of separation of duty constraints in role-based access control. We motivated and described the problems arising from the definition and enforcement of separation of duty constraints in the presence of role hierarchies and showed how the corresponding conflict checking methods are affected by these inheritance relations. In particular, we presented the respective conflict checking methods as implemented in the xoRBAC software component. xoRBAC is publicly available from `http://www.xotcl.org`.

References

[1] G.J. Ahn and R. Sandhu. Role-based Authorization Constraints Specification. *ACM Transactions on Information and System Security (TISSEC)*, 3(4), November 2000.

[2] D.D. Clark and D.R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proc. of the IEEE Symposium on Security and Privacy*, April 1987.

[3] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3), August 2001.

[4] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8), August 1976.

[5] C.E. Landwehr. Formal Models for Computer Security. *ACM Computing Surveys*, 13(3), September 1981.

[6] C.E. Landwehr, C.L. Heitmeyer, and J. McLean. A Security Model for Military Message Systems. *ACM Transactions on Computer Systems*, 9(3), August 1984.

[7] G. Neumann and M. Strembeck. Design and Implementation of a Flexible RBAC-Service in an Object-Oriented Scripting Language. In *Proc. of the 8th ACM Conference on Computer and Communications Security (CCS)*, November 2001.

[8] G. Neumann and M. Strembeck. An Approach to Engineer and Enforce Context Constraints in an RBAC Environment. In *Proc. of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2003.

[9] G. Neumann and U. Zdun. XOTcl, an Object-Oriented Scripting Language. In *Proc. of Tcl2k: 7th USENIX Tcl/Tk Conference*, February 2000.

[10] S. Osborn, R. Sandhu, and Q. Munawer. Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(2), February 2000.

[11] P. Samarati and R.S. Sandhu. Access control: Principles and practice. *IEEE Communications*, 32(9), September 1994.

[12] R.S. Sandhu. Access Control: The Neglected Frontier. In *Proc. of the Australasion Conference on Information Security and Privacy*, June 1996.