Design and Implementation of a Flexible RBAC-Service in an Object-Oriented Scripting Language

Gustaf Neumann
gustaf.neumann@wu-wien.ac.at

Mark Strembeck
mark.strembeck@wu-wien.ac.at

Department of Information Systems, New Media
Vienna University of Economics and BA
Austria

ABSTRACT

In this paper we present the design and implementation of the xoRBAC component that provides a flexible RBAC service. The xoRBAC implementation conforms to level 4a of the unified NIST model for RBAC and can be reused for arbitrary applications on Unix or Windows with a C or Tcl linkage. xoRBAC runtime elements can be serialized and recreated from RDF data models conforming to a well-defined RDF schema. Furthermore we present our experiences with xoRBAC for the deployment within the HTTP environment for a web-based mobile code system.

Keywords

Role-Based Access Control, Web-Applications, Mobile Code, Object-Orientation, Scripting Language, XOTcl

1. INTRODUCTION

Distributed and/or web-based applications often require the enforcement of complex access control policies. We developed the xoRBAC component especially for web-applications, where a reusable access control component with flexible and customizable access control policies is needed. We tested xoRBAC even for applications that make use of mobile code.

Discretionary access control (DAC) is sometimes criticized as conceding too many liberties to the rights manager while mandatory access control (MAC) commonly is regarded as being too restrictive for most applications [27]. Role-Based Access Control (RBAC) [9, 29, 30] offers a promising alternative and has become very popular in both research and industry. The ACM workshop series on RBAC and its successor the Symposium on Access Control Models and Technologies as well as recent journal publications (e.g. [3, 8, 13]) show the constant interest in this topic. One of the advantages of RBAC is being "policy-neutral". This means that a sophisticated RBAC-service may be configured to enforce many different access control policies including DAC- or MAC-based policies (see [21]).

xoRBAC provides an RBAC service that conforms to level 4a of the NIST model for RBAC and can be reused on Unix and Windows systems within applications providing C or Tcl linkage. xoRBAC is well suited to be used within a component framework. Component frameworks [23, 31] consist of reusable components that can be glued together with application specific semantics to form arbitrary applications.

The xoRBAC component is implemented with XOTcl [19] which offers a dynamic programming environment for rapid application development. XOTcl itself is a Tcl [22] compliant component written in C. Supplementary XOTcl and C components exist that support the development of distributed (esp. web-based) applications [20]. In this paper we present the design and implementation of the xoRBAC component and our experiences with the deployment of xoRBAC within a web-based mobile code system.

1.1 The Notion of Roles

In the domain of computer science researchers of different areas, like object-oriented software construction, database systems or security, often refer to a conceptual entity called "role" to express or model certain phenomenons. Unfortunately no common agreement between these areas exists concerning an exact definition of the semantics of roles that covers all of its usages.

For example, in entity relationship models each end of a relationship corresponds to a "role". In object-oriented development "roles" are used in the modeling and in implementation. In OO-modeling the "roles" are either used explicitly [12, 14] or they are expressed through classes. In the implementation objects can play certain "roles", e.g. in patterns like Observer or Facade [10]. In RBAC "roles" are means for carrying permissions to ease the administration of access rights.

In general roles are a well-known and accepted concept for the modeling of dynamically changing responsibilities and capabilities of objects that represent real-world entities or system-internal entities. A role can enrich the behavioral capabilities and/or the knowledge of the entity it is assigned to. Thus the modeling of roles received much attention in both research and industry (e.g. [14, 25, 26]).

In this paper we are concerned with roles in the context of Role-Based Access Control (RBAC) and object-oriented development. We present an approach to implement RBAC-roles based on dynamic object-oriented language constructs.

CCS'01, November 5-8, 2001, Philadelphia, Pennsylvania, USA.

1.2 Role-Based Access Control

In the context of RBAC, roles are used to model different job-positions and scopes of duty within a particular organization and/or within an information system. These roles are equipped with the permissions that are needed to perform their respective tasks. Users and/or other "active" entities are assigned to roles according to their field of activity.

Within an organization the descriptions of roles tend to change significantly slower than the assignment of individuals to these roles. Establishing roles as an abstraction mechanism for users facilitates the administration of access rights and thus the administration of the access control policy [29]. The actual access control policy is determined by the configuration of the different elements of the RBAC-service.

In [29] a family of RBAC models is introduced. The central concepts are users, roles and permissions. The models of this family range from $RBAC_0$ (Base Model) to $RBAC_3$ (Consolidated Model), where $RBAC_3$ combines the characteristics of the other models. Recently Sandhu et al. [30] proposed the NIST model for RBAC. It is intended to serve as a foundation for the development of a "unified" RBAC standard. Among other things the NIST model bypasses the distinction between roles and groups (as discussed in [28] for instance) by recognizing a sophisticated group-based access control mechanism as the first level of RBAC. For the time being the NIST model distinguishes four levels of RBAC:

- *Flat RBAC:* comprises the base concepts of RBAC, namely roles, users and permissions. User-role assignment and permission-role assignment are defined as many-to-many relations. Furthermore it requires that users are allowed to activate multiple roles simultaneously and that functions for user-role review exist.
- *Hierarchical RBAC:* embodies all features of flat RBAC and additionally demands the ability to define partially ordered role hierarchies.
- *Constrained RBAC:* must provide the functionality to enforce separation of duties (SOD) policies. However, it does not prescribe the exact form of SOD that has to be provided, i.e. both static and/or dynamic SOD are allowed.
- *Symmetric RBAC:* demands the ability to perform permission-role reviews, even (or especially) in distributed systems.

Hierarchical RBAC, constrained RBAC and symmetric RBAC are subdivided in two parts "a" and "b" for the support of arbitrary or limited role-hierarchies respectively. In [30] two alternative interpretations of the NIST model are introduced that differ with respect to the ordering of the different RBAC levels. The first alternative imposes a strict sequential order of the four levels, while the second treats flat and hierarchical RBAC as an ordered sequence and considers constrained and symmetric RBAC as independent and unordered. In appendix A of [30] the second alternative is regarded as the more flexible and preferred approach.

1.3 Approach and Paper Structure

In this paper we present the design and implementation of an RBAC-service called xoRBAC. In order to simplify the implementation of xoRBAC we used a few language supported constructs of XOTcl that were originally designed for the support of design patterns [10]. These constructs are per-object mixins (POMs) and dynamic object aggregation:

- *Per-object mixins* [17] can extend certain objects dynamically with additional behavior or functionality.
- *Dynamic object aggregation* [18] enables the dynamic aggregation and disaggregation of objects at runtime.

Certainly the language constructs can be emulated in other languages as well (e.g. traits in SELF [2] are quite similar to POMs) and the design presented in this paper can be used for implementations in other languages too.

Figure 1 depicts the conceptual structure of the xoRBAC component. Permissions, roles and users (or subjects in general) are the basic elements of xoRBAC. The user management provides a means to manage the subjects, i.e. the entities that may actively initiate an operation. The constraint management of xoRBAC is based on the permissions and roles components and enables the definition of static separation of duties constraints. The role hierarchy management uses the constraint management to prevent the creation of role hierarchies that are disallowed by the constraint definitions. The rights management includes the decision component and components for permission/role and user/role assignment. xoRBAC is associated with a metadata service that captures logging and audit information and enables the serialization (and recreation) of xoRBAC runtime instances.

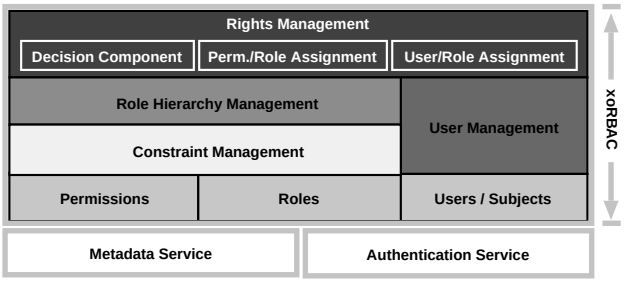


Figure 1: xoRBAC **component: conceptual structure**

To reach an authorization decision for a particular access request, the decision component of xoRBAC expects the following information as parameters: the ID of an authenticated subject, i.e. the user or user-agent who requests an access, the operation to be performed and the name of the object that is the target of the operation. This results in an "subject, operation, object" triple.

A prerequisite for an efficient access control service is the existence of an efficient authentication service. The selection of an authentication mechanism highly depends on the existing infrastructure. Therefore the xoRBAC component presented in this paper does not demand on a particular authentication mechanism. It simply requires that some means exists to authenticate the users and/or user-agents within the system, i.e. xoRBAC can be connected with arbitrary authentication services. It is feasible to use a weak mechanism like HTTP-Basic-Authentication (see RFC 2617) as well as a sophisticated authentication mechanism based on X.509 certificates.

With respect to the categories defined in [30] the current implementation of xoRBAC provides *symmetric RBAC with static separation of duties* (Level 4a). In particular xoRBAC currently provides the following main features:

- many-to-many user-role and permission-role assignment (and revocation).
- user-role review and permission-role review.
- definition of arbitrary role-hierarchies (permission-inheritance interpretation / inheritance hierarchies)
- definition of static SOD (SSD) constraints for both roles and permissions.
- defintion of maximum and minimum cardinalities for both roles and permissions.

xoRBAC offers the functionality for user/role assignment, and permission/role assignment. Furthermore xoRBAC additionally offers the ability to assign permissions directly to certain users/subjects. We feel that this is a sensible feature which allows even more flexibility when defining an access control policy. It can express situations where a certain individual should be granted temporary access rights in addition to the role(s) he occupies (without defining an extra role).

xoRBAC enables the simultaneous activation of multiple roles for every user known to a particular instance of xoRBAC. Every xoRBAC instance does, however, allow exactly one active role-set for each user at a time.

The following sections are structured as follows. In Section 2 we give a brief introduction for the XOTcl language to ease the understanding of the design and the implementation of the xoRBAC component presented in Section 3 and Section 4. Afterwards we present the RDF-based serialization of RBAC definitions and discuss our experiences with the deployment of xoRBAC for web-based applications. Section 7 gives an overview of related work before we conclude the paper and give an outlook on future activities.

2. THE XOTCL LANGUAGE

XOTcl (eXtended Object Tcl) [19], is a general purpose object oriented scripting language that can be dynamically loaded into every Tcl compatible environment such as `tclsh` or `wish` and is embedable in C programs. As a Tcl extension, all Tcl commands [22] are directly accessible in XOTcl. XOTcl is based on the object system of OTcl [34] and supports meta-classes [19, 34]. XOTcl preserves the flexibility of Tcl and adds new language constructs to provide a highly flexible OO programming environment. The new language constructs are: filters, per-object-mixins, per-class mixins, nested classes, dynamic object aggregation and assertions.

In XOTcl all language constructs can be applied in a dynamic fashion. This means that every aspect of a program can be adapted at runtime. It is possible for instance to define new classes at runtime and to insert them into an existing class hierarchy, or to define arbitrary new methods on existing classes. Furthermore XOTcl offers a rich introspection mechanism which allows to inquire nearly all characteristics of XOTcl objects and classes at runtime [19].

XOTcl supports simple and multiple inheritance. Through the superclass-relation classes are arranged in a directed acyclic graph. XOTcl defines a linearized precedence order for class and mixin hierarchies along an unambiguous "next-path" (modeled after CLOS [6]) to avoid potential conflicts during the name resolution. XOTcl is publicly available from [35].

As already mentioned we used POMs and the dynamic object aggregation mechanism to simplify the implementation of xoRBAC.

In XOTcl every object represents its own namespace which may contain other objects. Objects can be aggregated dynamically by an other object at runtime. An aggregation constitutes a part-of-relationship between the corresponding objects. All operations on aggregations are deep operations which affect the corresponding object and all its aggregates. For a more detailed description of the dynamic object aggregation mechanism of XOTcl see [18].

A Per-Object Mixin (POM) is a class which is inserted at the beginning of the precedence order of a particular object. In other words, POMs are inserted in front of the precedence order induced by the class-hierarchy from which the object was instantiated (see Figure 2). Thus POMs are a means to extend every single object with additional behavior or capabilities dynamically at runtime.

The same class could be mixed into the precedence order of multiple objects at the same time. Likewise every object could be associated with any number of POMs simultaneously. If the functionality of a POM is no longer needed, the respective POM can be deleted from the precedence order of this particular object. However, the mixin class is not destroyed, it can still be defined as POM for other objects or serve as abstract type for the instantiation of objects.

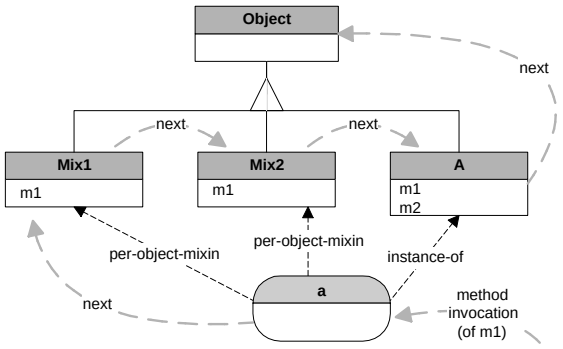


Figure 2: Next-Path with per-object mixins

Figure 2 depicts the classes `Mix1` and `Mix2` which are registered as POMs for the object `a` which is an instance of the class `A`. Through the next-path each method call to `a` is at first directed to the classes `Mix1` and `Mix2` prior to invoking the method in the original class (`A`). This feature is called method combination or *method chaining* [19]. Method chaining is performed along the next-path and without explicit naming the class containing the method. Method chaining is a requirement for dynamic class structures.

Now we give a short example to illustrate the definition and usage of POMs.

First we define the class `A` with two methods `m1` and `m2` which simply print the actual class identifier by using the `puts` command and then pass the call to the successor in the precedence order (`next`) - inheritable methods are defined with `instproc` in XOTcl. Afterwards the classes `Mix1` and `Mix2` are defined. For each of them an instproc `m1` is defined. Finally an instance `a` of class `A` is created and the classes `Mix1` and `Mix2` are registered as `mixin` classes for the object `a`. Note that POMs are always registered for individual objects (instances).

```
Class A
A instproc m1 {} {puts "::A"; next}
A instproc m2 {} {puts "::A"; next}

Class Mix1
Mix1 instproc m1 {} {puts "::Mix1"; next}

Class Mix2
Mix2 instproc m1 {} {next; puts "::Mix2"}

A a -mixin {Mix1 Mix2}
```

The method `m1` defined in `Mix2` first passes the call to the `next` successor in the precedence order. This means that after the corresponding methods of all successors have been executed the call returns to `m1` of `Mix2` and prints its class identifier. Therefore the call "`a m1`" results in the following output:

```
::Mix1
::A
::Mix2
```

Whereas the call "`a m2`" simply results in the output of "`::A`" since none of the mixin classes defines a method with the name `m2`.

Here we demonstrated the mixin of independent classes only. Though XOTcl enables to register whole class-hierarchies as POMs. The procedure to achieve this is quite simple. One can register a class which is part of a class-hierarchy (i.e. has several superclasses) as mixin for an object. The object is thereby extended with the abilities offered by any class included in the class hierarchy. Note that one may also register classes from different class-hierarchies simultaneously as POMs for the same object. In other words: it is possible to mixin multiple class hierarchies. For a more detailed description of POMs see [16, 17].

3. XORBAC: DESIGN DECISIONS

The xoRBAC component consists of five classes: `RoleManager`, `Role`, `Permission`, `User` and `Audit`. The essential design level relations between these classes are depicted in Figure 3.

Since roles should be independent from the user-objects to which they could be assigned, we suggest to define association relationships between actual `Role` objects and `User` objects rather than inheritance relationships. Moreover we think that it should be possible to manipulate each runtime object of an RBAC service (roles, permissions, etc.) individually to achieve an extensive flexibility. Therefore `Role` objects in xoRBAC do not inherit attributes and/or behavior from a common "User" superclass. In xoRBAC instances of `Role` are allowed to have many-to-many association-relations to instances of the `Permission` and `User` classes. As already mentioned, `User` and `Permission` objects may be also directly associated with each other.

In xoRBAC permissions are always positive, i.e. a permission always grants a certain access right and does not deny it. The abstraction level on which permissions are defined highly depends on the actual application context and on the access control policy that should be realized. xoRBAC does not constrain the granularity of permissions, i.e. the permissions may be coarse- as well as fine-grained. Thereby the respective engineer is free to define permissions for primitive operations like "read" or "write" as well as for abstract operations like "transfer-money" for instance.

The essence of access control is to decide if a certain subject (active component, initiator) is allowed to perform a certain operation on a certain object (passive component, target). Simplified: in xoRBAC permissions are "operation object" pairs and subjects are represented by `User` objects. The access control function `grantAccess` decides whether a particular access could be authorized by inspecting the roles (and permissions) of the `User` object under consideration. xoRBAC does not impose a restriction that subjects need to be real-life persons (of course). Instead xoRBAC allows to register different kinds of subjects as "users" (`User` objects), e.g. long-running programs, like mobile-agents.

The revocation semantic of xoRBAC designates that all revocation operations immediately come into effect. This means, for example, that a role may be revoked from a `User` object even if the corresponding real-life person has currently activated this particular role. The same is true for the revocation of permissions from roles and/or users. Moreover the revocation of a senior-role r_i also includes the revocation of all junior-roles of r_i. We chose this semantic since changes of the access control policy should, in our opinion, be distributed without delay to avoid inconsistencies and possibly resulting security violations.

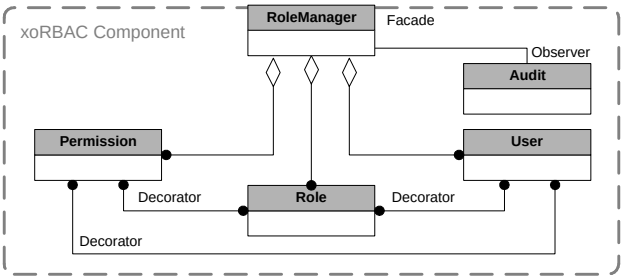


Figure 3: The xoRBAC component: class relations

Every instance of xoRBAC contains exactly one `RoleManager` object. This object aggregates all instances of `User`, `Role` and `Permission`, and manages their relations and interactions. The `RoleManager` object therefore hides the internal structure of the xoRBAC component. In order to use the xoRBAC-service other components and subsystems thus need to access the API offered by the `RoleManager` class. Therefore Figure 3 depicts the `RoleManager` class as the external interface of xoRBAC. The `RoleManager` object serves as facade (see [10]) for the other xoRBAC objects.

At runtime the `RoleManager` controls its aggregated objects and enforces the integrity rules and constrains, e.g. cardinality- and SSD-constraints on roles and/or permissions. Within the same `RoleManager` instance the names of `Role`, `Permission` and `User` objects must be unique.

Every `Role` and `Permission` object "knows" the roles or permissions it is mutually exclusive to, and its maximum and minimum cardinality (if defined). When assigning a role to a user the `RoleManager` checks whether the new role is mutual exclusive to one of the `Role` objects that are already associated with the corresponding `User` object. A similar check is performed when a permission is assigned to a role. xoRBAC supports the definition of maximum cardinalities ≥ 0 and minimum cardinalities ≥ 1. A maximum cardinality of 0 is sensible for the definition of private roles [29]. The `RoleManager` prevents violations of cardinality constraints

and issues predefined error messages if the performance of a (otherwise regular) method-call would do so.

Situations may occur where the maximum and the minimum cardinality for a particular role (or permission) are equal. Thence xoRBAC offers so called `safeReplace` methods for both roles and permissions that enable to replace a role (or permission) owner without violating the cardinality constraints.

The `RoleManager` class realizes the role-hierarchy management and the user management functionality depicted in Figure 1. While the rights management and the constraint management are distributed among the `RoleManager`, `Role`, `Permission` and `User` classes.

Every instance of `RoleManager` is associated with exactly one `Audit` object. The corresponding `Audit` object acts as observer (see [10]) for this particular instance of `RoleManager`. The `Audit` object records all API calls that cause changes of the `RoleManager` object itself or one of its aggregated objects. The information recorded by the `Audit` object ease the analysis of irregularities like policy violations for example.

4. XORBAC: IMPLEMENTATION ISSUES

Figure 4 depicts a `RoleManager` object at runtime, it shows the dynamic object aggregation of the `User`, `Role` and `Permission` instances and their encapsulation within a corresponding namespace. To assign a certain permission to a particular role (or user) the respective `Permission` instance is registered as POM for the corresponding `Role` (or `User`) instance. Analogously a role is assigned to a user by registering the `Role` object as POM for a `User` object (Figure 4). Therefore the `Permission` objects act as decorators (see [10]) for the `Role` and `User` objects. Equally, the `Role` instances act as decorators for `User` objects. For the sake of distinctness Figure 4 exemplary shows the assignment relations only for one role and two permissions.

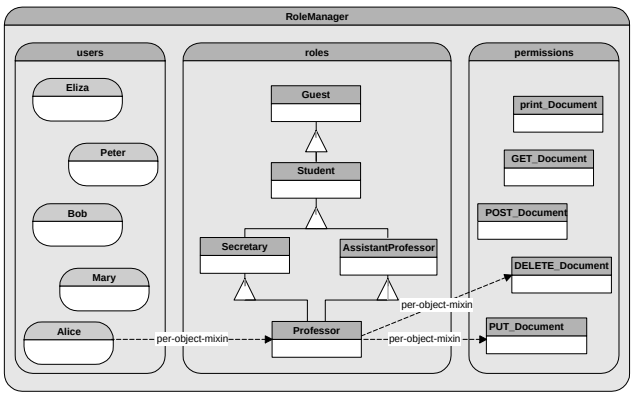


Figure 4: A `RoleManager` object at runtime

The usage of POMs for user-role and permission-role assignment provides a clear separation between users, roles and permissions. Besides, it eases the tasks of assignment and revocation on the programming level since one only needs to register or deregister the respective `Role` or `Permission` instance as POM.

In xoRBAC the classes `Role` and `Permission` (see Figure 3) are defined as meta-classes. In essence, there were three main reasons that led to this design:

1. Instances of meta-classes are "ordinary" classes which are allowed to be part of a class-hierarchy (an inheritance relation). Since `Role` is a meta-class its instances may thus be arranged within a class-hierarchy. We utilize this feature to define role-hierarchies in a natural way as XOTcl class-hierarchies. In other words: role-hierarchies in xoRBAC are class-hierarchies (directed acyclic graphs) which may, in principle, consist of an arbitrary number of `Role` instances. Defining role-hierarchies in this manner has the consequence that junior-roles (in general) are represented by superclasses while senior-roles, i.e. more powerful roles, are represented by subclasses. Within such a hierarchy each class represents one particular role which transitively occupies the permissions of its superclasses.

2. Only classes may be registered as POMs. POMs are a suitable means to realize the decorator pattern in a straightforward manner [17]. As mentioned above we register `Permission` instances as decorators for `Role` (and `User`) instances, and `Role` instances as decorators for `User` objects. Therefore the `Role` and `Permission` classes are defined as meta-classes so that their instances (actual roles or permissions) can be registered as POMs.

3. Through the usage of POMs it can be guaranteed that two `User` objects which should be assigned to the same role are virtually assigned to the same `Role` instance and not only to roles having the same name. This holds equivalently for the role-permission assignment.

In contrast to OO-languages where classes are purely types (such as C++ or Java) XOTcl-classes are runtime objects and are therefore able to interact with other runtime objects (without being explicitly instantiated). For a detailed discussion of classes and meta-classes within XOTcl we refer the interested reader to [17, 19, 34].

We like to recall that in XOTcl all language constructs can be applied in a dynamic fashion (see Section 2). This means that we can define arbitrary new classes, e.g. `Role`s, at runtime and insert them into an existing class hierarchy (used as role-hierarchies in xoRBAC).

4.1 The Decision Component

As already mentioned, the decision component of xoRBAC is distributed among the `RoleManager`, `Role`, `Permission` and `User` classes. In this subsection we illustrate the realization of the access control function of xoRBAC which represents the core element of the decision component. This access control function is represented by the `grantAccess` method, which is the most frequently invoked method of xoRBAC after the setup/configuration phase is completed. Figure 5 depicts a simple example of a next-path resulting from a call of `grantAccess` on the `User` object `user1`. The object `user1` is assigned to the role `role1` which in turn owns the permissions `permission1` and `permission2`.

At first, the call of `grantAccess` follows the next-path depicted in light grey. This means that `user1` passes the call to its POM `role1`. From here the call is passed to `permission1` and `permission2` which are registered as POMs for `role1`. Then the call is forwarded along the next-path depicted in dark grey, i.e. back to `role1` and the `User` class. In reality there is of course one unambiguous next-path, we used different shades of grey to ease the description of Figure 5.

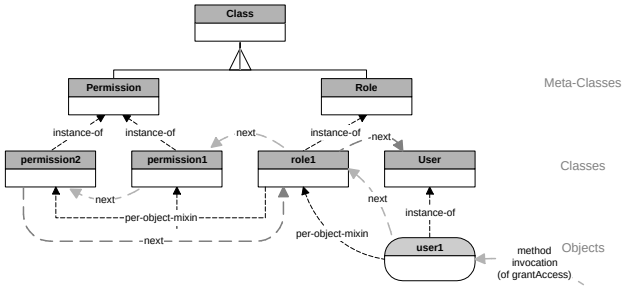


Figure 5: Next-path for the call of `grantAccess`

To reduce complexity the small example above shows only one role with two permissions registered as POM for the `user1` object. Nevertheless one can also register classes that are part of a class-hierarchy (i.e. have several superclasses) and/or multiple POMs simultaneously (see Section 2). Since in xoRBAC class-hierarchies serve as role-hierarchies, a `User` instance thereby transitively occupies the permissions of all junior-roles (superclasses) also. The unique next-path always includes all roles and permissions which are registered as POM (directly or transitively).

Regarding the `grantAccess` method, the different instances of `Role` and `Permission` form a chain of responsibility (see [10]). A `grantAccess` call is passed along the next-path until a `Permission` object declares itself responsible and handles the request by returning `true`, i.e. the permission authorizes the respective access-request. Figure 6 depicts a message sequence chart (MSC) for an example where the requested access is denied, i.e. could not be authorized by one of the permissions assigned to `role1`.

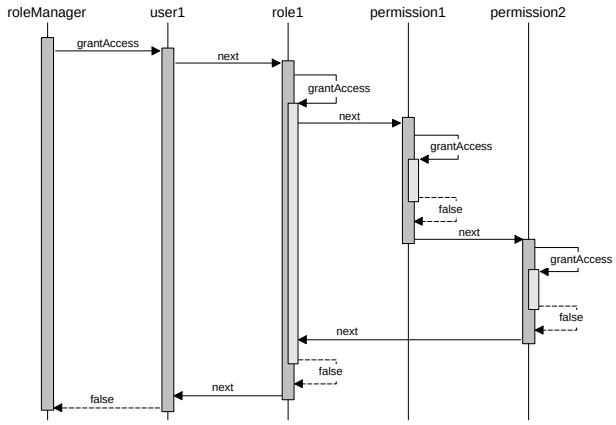


Figure 6: `grantAccess` MSC for the return of `false`

For a call of `grantAccess`, the respective `User` object is always the last object within the next-path and therefore the last object within the chain of responsibility. This means: if no `Permission` object previously authorizes the requested access (returns `true`), the call is finally passed back to the `User` object. The `User` object then returns `false` since none of its permissions allows the requested access. In other words: if no `Permission` object within the chain of responsibility returns `true` beforehand, every `User` object delivers `false` as default result of a `grantAccess` call (see Figure 6).

A `grantAccess` call that returns `true` - which means that `user1` is allowed to perform the requested operation - has a quite similar flow of events to the one depicted in Figure 6. However, the message passing along the next-path is stopped as soon as a specific permission (e.g. `permission1`) grants the access by returning `true`. A further search through the remaining permissions (and roles) is not necessary.

4.2 User-Role and Permission-Role Review

To realize user-role review and permission-role review functionalities we especially made use of the introspection capabilities offered by XOTcl. In essence, xoRBAC provides the following review interfaces:

- `getAllRoles {user}`: returns all roles which are directly or transitively assigned to the `User` object `user`.
- `getAllRoleMembers {role}`: returns all `User` objects which directly or transitively possess the role represented by the object `role`.
- `getAllPermissions {role}`: returns all permissions which are directly or transitively assigned to `role`.
- `getAllPermissionOwners {perm}`: returns all roles to which the permission `perm` is directly or transitively assigned.

In the current implementation the methods `getAllRoleMembers` and `getAllPermissionOwners` methods are relatively expensive with respect to performance (in the range of milliseconds). However, these methods are by far not invoked as often as the `grantAccess` method. Nevertheless, the performance can be easily improved through redundant information.

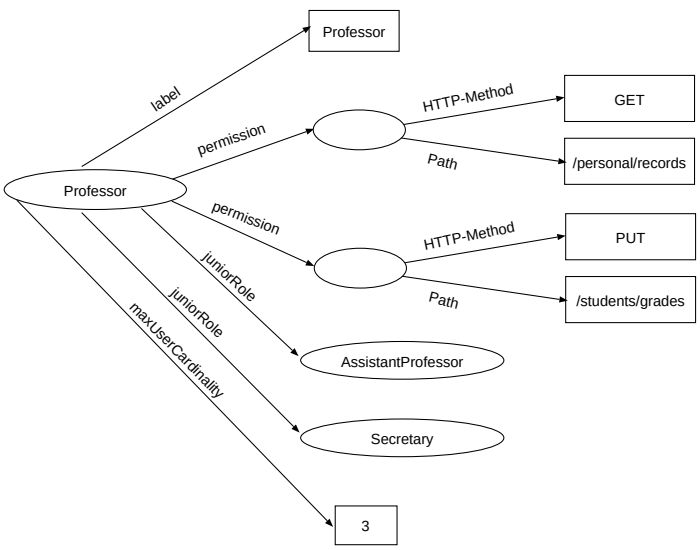
4.3 Definition of SSD-Constraints

Two mutual exclusive permissions are not allowed to be assigned to the same role, while two mutual exclusive roles are not allowed to be assigned to the same user. xoRBAC provides interfaces for defining static separation of duties constraints (SSD-constraints) on both permissions and roles:

- `setSSDRoleConstraint {role mutlexcl}`: defines a list of roles (`mutlexcl`) as mutual exclusive to `role`.
- `setSSDPermConstraint {perm mutlexcl}`: defines a list of permissions (`mutlexcl`) as mutual exclusive to `perm`.

Since the `mutlexcl` parameter above is defined as Tcl-list one can define an arbitrary number of roles (or permissions) as mutual exclusive to `role` (or `permission`) with a single method call. For both `set`-methods mentioned above a corresponding `unset`-method exists to delete the SSD-constraints. Moreover xoRBAC provides methods to query/review which roles (or permissions) are currently defined as mutual exclusive to a given role (or permission).

In xoRBAC SSD-constraints are inherited within a role-hierarchy. To avoid inconsistencies or unreasonable configurations xoRBAC performs a number of tests prior to setting a SSD-constraint. It is not sensible for example to define a role as mutual exclusive to one of its junior- or senior-roles. Moreover xoRBAC prevents that two `Role` (or `Permission`) instances that are already assigned to the same `User` (or `Role`) instance are defined as mutual exclusive.




```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns="http://nm.wu-wien.ac.at/rdf-schema/xoRBAC#">

    <Role ID="Professor>
        <rdfs:label>Professor</rdfs:label>

        <Permission rdf:parseType="Resource">
            <HTTP-Method>GET</HTTP-Method>
            <Path>/personal/records</Path>
        </Permission>

        <Permission rdf:parseType="Resource">
            <HTTP-Method>PUT</HTTP-Method>
            <Path>/students/grades</Path
        </Permission>

        <juniorRole rdf:resource="#AssistantProfessor">
        <juniorRole rdf:resource="#Secretary">

        <maxUserCardinality>3</maxUserCardinality>
    </Role>

    <Role ID="AssistantProfessor">
        ...
    </Role>

    <Role ID="Secretary">
        ...
    </Role>
</rdf:RDF>
```

Figure 7: Example for the representation of xoRBAC roles in RDF and XML

5. XORBAC: SERIALIZATION

Object serialization is a process where the complete state information of an object is written to an output stream. This enables to send the object-state over a network and/or to save it permanently. Afterwards copies of the serialized object can be recreated at arbitrary times by reading the serialized state information from an input stream.

xoRBAC provides methods that enable the serialization of all xoRBAC runtime elements as RDF metadata. The Resource Description Framework (RDF) is a W3C recommendation for the definition, description and processing of structured metadata, esp. in a web-context [15]. The main purpose of RDF is to provide a generic mechanism for the description of resources from arbitrary application domains. Every person or organization may define own RDF-vocabularies that perfectly fit their respective needs by specifying a so called RDF-schema [7]. RDF itself is syntax-neutral, thus RDF data models can be represented in every suitable interchange syntax. In [15] two syntaxes are defined for the XML-based description of RDF data models.

For the serialization of xoRBAC elements we defined a RDF-schema for RBAC definitions. xoRBAC provides methods to serialize a single `User`, `Role` or `Permission` object or to serialize a `RoleManager` object with all aggregated instances as a whole. The exported RDF documents are encoded in XML serialization syntax [15] and conform to the schema or parts of the RDF-schema we defined. Furthermore xoRBAC can import xoRBAC elements from an RDF document in XML serialization syntax that conform to this schema. Figure 7 shows an example for the representation of xoRBAC entities in RDF. It depicts an RDF graph and the corresponding XML serialization for a role "Professor".

The definition of an access control policy and the correct configuration of the corresponding access control service is a major effort. The serialized xoRBAC elements can be used, for instance, to permanently save a configuration which represents a particular access control policy. Moreover a new xoRBAC instance can be smoothly initialized by importing the corresponding RDF data model(s).

We chose RDF and XML as interchange and storage format for xoRBAC runtime elements since these are open internet standards which facilitate data interchange with other components that rely on open standards as many web-based applications and environments do. Furthermore RDF and XML are platform independent and allow to define, query, and reuse RBAC definitions in the semantic web context.

6. EXPERIENCES WITH XORBAC

In this section we give a summary of our experiences with the deployment of xoRBAC in real application contexts. We applied xoRBAC together with the ActiWeb environment for the development of web applications. ActiWeb [20] is a mobile code system that enables the development of distributed web applications with active web objects. The web objects of ActiWeb reside on and are accessed through so-called places. A place provides the runtime environment for active web objects. Every ActiWeb place is a HTTP server, and web objects (the methods they provide) can be accessed through an URL. Thus ActiWeb places and the corresponding web objects can be accessed from every web-browser or other clients. ActiWeb is publicly available from [35].

Figure 8 shows how we integrated xoRBAC in the ActiWeb environment. xoRBAC uses the metadata service of ActiWeb for the serialization of xoRBAC entities in RDF and XML. Moreover xoRBAC can be bound to the mobile object system (MOS) of ActiWeb. ActiWeb places are a part of the MOS, and every web-object (mobile as well as stationary) is accessed via a place. With the ActiWeb binding of xoRBAC one could decide individually: if a certain place should use a local instance of xoRBAC, if it should be connected to a remote instance of xoRBAC located on another place, or if it should not use xoRBAC at all.

When using xoRBAC all access requests on ActiWeb web-objects are captured and passed to the respective xoRBAC instance which decides if this access could be granted. In particular the `grantAccess` function of the xoRBAC decision component receives a "subject, operation, object" triple and decides - according to the active role-set of the respec-

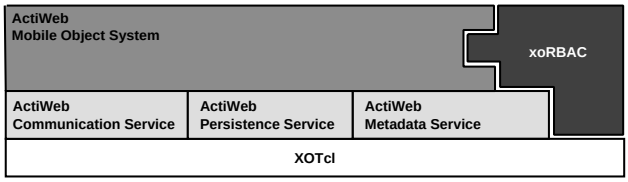


Figure 8: Connection of xoRBAC to ActiWeb

tive subject - if the subject is allowed to perform the requested operation on the specified object. ActiWeb web-objects could be both the subject or the object of an access. A certain active web object may access methods offered by other active web objects and in turn may itself be accessed by human-users or other active web objects.

We utilized xoRBAC in the ActiWeb environment for two main tasks:

- Access control for web resources that are provided by a web-server and (primary) accessed by human users, as HTML documents, pictures or sound-files.
- Access control for services provided by the ActiWeb runtime environment and for web objects that are accessed by human users and/or long-living computer programs esp. by (other) active web objects.

We defined permissions for the corresponding services and web objects and assigned them to roles which were in turn assigned to human users and active web objects.

For authentication purposes we applied HTTP basic and digest authentication in both experiments. The subject (in terms of access control) must authenticate itself for the respective realm by transmitting its ID and password. This weak authentication mechanism is usually sufficient within a trusted intranet or for non-critical applications. Primarily we applied HTTP authentication because it is easy to use and directly available in ActiWeb. As mentioned earlier, xoRBAC does not demand on a specific authentication mechanism, it only assumes that some authentication service is in place. Therefore xoRBAC can also be used with a sophisticated authentication environment based on X.509 certificates (and SSL) for instance.

In xoRBAC each `User`, `Role` and `Permission` object has an explicit name. This name is unique within the corresponding xoRBAC instance (at least). Thus the actual user IDs provided by the authentication service, can be directly mapped on the names of the `User` objects within xoRBAC. This means that a `User` object may have a globally unique X.500 distinguished name for instance. The same is true for the names of `Role` and `Permission` objects.

Figure 9 depicts the three principle ActiWeb configurations that we tested xoRBAC with. Subsequently we describe each of these configurations in more detail:

- Figure 9 a) shows a configuration where every single ActiWeb place is connected to a local instance of xoRBAC. This means that the places are autonomous in respect of access control. This configuration is useful in an environment where the different places are hosted by independent parties for example and require independent access control policies.
- In Figure 9 b) we have the situation that several ActiWeb places exist which form a so-called "area", i.e. a part of a network consisting of different places which share the same xoRBAC instance and therefore the same access control policy. One place provides its local xoRBAC service as remote service to the other places. This configuration eases access control administration in an intranet for example and is sensible if several distributed places are hosted by the same authority.
- Figure 9 c) shows a configuration where two (or more) places provide their local xoRBAC service to other places. Furthermore at least one place exists that is connected to a local instance of xoRBAC and has an additional remote connection to another xoRBAC instance. This configuration could be useful to avoid redundancies, when two (or more) independent xoRBAC instances should share a number of roles and/or permissions but are different enough to exist on their own. Because of its realization on the programming level we call this configuration "cascading xoRBAC". If the first instance of xoRBAC cannot grant an access, the request is passed to the next instance. This cascade is not limited to two instances but can be applied in principle for an arbitrary number of xoRBAC instances. A shortcoming of this is the reduced performance for `grantAccess` if the cascaded xoRBAC instances reside on different physical nodes.

In principle xoRBAC may be applied in two ways within ActiWeb: transparent or non-transparent for human users and other subjects. If used transparently the subjects authenticate themselves when entering a place (or area) and the roles they posses in this realm are automatically activated. In this case the subjects do not know that an RBAC service is in place and thus cannot control the (de)activation of their roles. This may be a sensible option within a mobile code environment. Alternatively one could allow some (or all) subjects to manage their active role-set in their own right, e.g. all human users may be allowed to manage their active role-set while mobile objects or other non-human subjects are not. Note that in our approach for web-based RBAC no information is stored on the client side, e.g. by using cookies or extended URLs. Since all relevant information is held by the server each subject needs to re-authenticate itself when leaving and re-entering a place or area.

Our experiences show that the performance of the xoRBAC prototype is sufficient for a number of web-based applications. We defined a few simple test cases where about one hundred roles, five hundred permissions and one hundred users were involved. A `grantAcess` run for directly assigned roles ranges between 0.5 and 0.6 ms. While a `grantAccess` run with inherited roles and permissions costs about 0.8 ms. These measurements where performed on a 400 MHZ Pentium II machine with Red Hat Linux 6.2.

7. RELATED WORK

In [4] Barkley introduced an approach for an implementation of RBAC in an object oriented programming language. Each role is represented by a different role class. At runtime, role objects (instances of role classes) serve as proxies for the applications that want to access certain methods. The approach in [4] does not consider role-hierarchies and constraints.

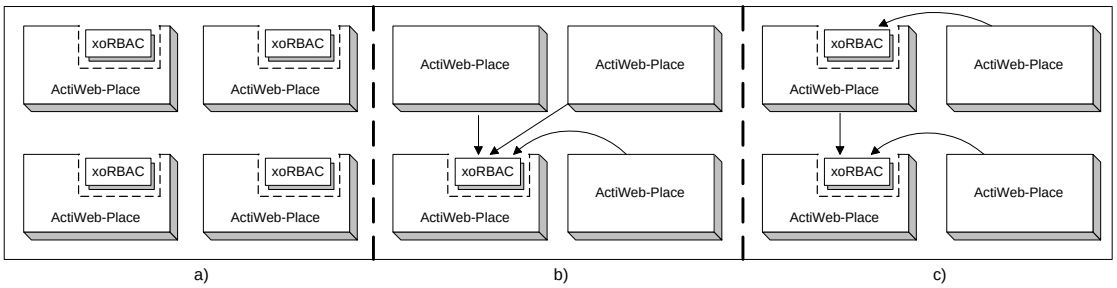


Figure 9: Usage of xoRBAC in ActiWeb

The hyperDRIVE system [5] uses the LDAP standard to implement authentication and authorization services. The system makes use of the Java Applet technology. The hyperDRIVE Java Applet is loaded into a Java-enabled web-browser and provides an interface to access protected web resources. An LDAP Server serves as central repository for all authentication and access control information. The definition of role hierarchies and mutually exclusive roles is supported by special purpose LDAP attributes.

The I-RBAC system [32] suggests an approach for the realization of RBAC within intranets. A characteristic of I-RBAC is the parallel use of local and global role hierarchies. Local roles own permissions for the local resources of one particular server. Global roles possess permissions for so-called global resources which have an unique identifier within an intranet and can be "globally" accessed. I-RBAC does not enable the definition of mutual exclusive roles.

In [11] Giuri suggests an RBAC mechanism based on Java servlets. He proposes the so called JRBAC-WEB architecture which relies on the execution of HTTP requests (GET, PUT, etc.) through a secure Java servlet. The implementation supports the definition of role hierarchies and separation of duties constraints. JRBAC-WEB uses HTTP authentication. Optionally the approach allows to keep RBAC-specific session data using cookies or an URL-rewriting technique for Java servlets.

RBAC/Web [8] is a comprehensive implementation of an RBAC service for web-servers. RBAC/Web does not prescribe a specific authentication mechanism and supports role-hierarchies, cardinality constraints and dynamic and static separation of duties. A web-server may use an extension module to directly access the RBAC/Web API or forward the calls to designated RBAC/Web CGI scripts.

Park and Sandhu describe an approach to provide RBAC with role hierarchies through extended X.509 certificates [24]. A role server authenticates the user and issues so called smart certificates that include the roles for this particular user. When accessing a secured web server the user chooses one of his smart certificates and sends it to the server. The web server uses the role information enclosed in this certificate to perform RBAC until the certificate expires.

In [13] Gutzmann introduces a security services architecture for HTTP-based environments. He uses LDAP to realize an RBAC service. Gutzmann suggests two LDAP schema extensions to implement $RBAC_0$ as proposed in [29]. This implementation thus neither supports role hierarchies nor (SOD-)constraints. Nevertheless Gutzmann notes that the object class hierarchy of LDAP may be used to augment the approach with role hierarchies ($RBAC_1$).

8. CONCLUSION AND FUTURE WORK

The xoRBAC component presented in this paper provides a flexible RBAC service that conforms to level 4a of the NIST RBAC model introduced in [30]. It can be reused for arbitrary applications providing C or Tcl linkage. xoRBAC provides a generic serialization feature. Its runtime elements can be serialized as and recreated from RDF data models (in XML syntax) conforming to a well defined RDF-schema. In conjunction with the ActiWeb environment xoRBAC can be used for mobile code and supports the definition of distributed access control policies through the usage of multiple cascading xoRBAC instances. In addition to roles xoRBAC enables the definition of subject specific rights. The component offers a well defined API to other components and does not prescribe the authentication mechanism it is used with. The current implementation of xoRBAC is about 2500 lines of code (without comments and blank lines).

For the implementation of xoRBAC we used the OO scripting language XOTcl. XOTcl offers per-object mixins and dynamic object aggregation as native language features which are suitable for the implementation of an RBAC service as described in this paper. Moreover the highly dynamic object system of XOTcl eases the implementation of very flexible programs. At runtime roles, permissions and users (or other subjects) are represented by corresponding programming objects which can be dynamically associated with each other. The xoRBAC component performs certain tests to ensure consistency and integrity of the different objects and their relations and the keeping of constraints.

Since XOTcl is an interpreted language it has a relatively low performance compared with pure C implementations but offers from our experiences equal or higher performance than Java. We suggest the usage of the current version of xoRBAC especially for small and mid-size environments with hundreds or a few thousands of parallel active subjects as well as hundred or less roles and a few hundred permissions.

Currently, we are developing a graphical administration tool for xoRBAC. This tool should be applied to monitor and modify runtime instances of xoRBAC as well as to specify RDF data models conforming to the RDF schema for RBAC definitions that we defined. Together with the adoption of the administration tool we plan to incorporate administrative RBAC into xoRBAC.

With regard to our experiences we will further investigate the inclusion of history information [1] and the context-dependent (de)activation of access rights [33] for distributed applications in general and mobile code systems in specific. We especially go into the matter how xoRBAC (and RBAC

in general) can be efficiently applied in the context of digital rights management, particularly for the passing and/or transfer of access rights for mobile digital goods (e.g. music or video files, pictures or digital periodicals).

xoRBAC is publicly available from [35]

9. REFERENCES

[1] A. Acharya, V. Chaudhary, and G. Edjlali. History-based access control for mobile code. In *Proc. of the Fifth ACM Conference on Computer and Communications Security*, November 1998.

[2] O. Agesen, L. Bak, C. Chambers, B. Chang, U. Hoelzle, J. Maloney, R. Smith, D. Ungar, and M. Wolczko. *The SELF 4.0 Programmer's Reference Manual*. Sun Microsystems, 1995.

[3] G. Ahn and R. Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and Systems Security*, 3(4), November 2000.

[4] J. Barkley. Implementing role based access control using object technology. In *Proc. of ACM Workshop on Role Based Access Control*, November 1995.

[5] L. Bartz. hyperDRIVE: Leveraging LDAP to implement RBAC on the web. In *Proc. of the ACM workshop on Role-Based Access Control*, 1997.

[6] D. Bobrow, R. DeMichiel, S. Keene, G. Kiczales, and D. Moon. Common lisp object system specification. *Sigplan Notices*, 23(9), 1988.

[7] D. Brickley and R. Guha. Resource description framework (RDF) schema specification 1.0. http://www.w3.org/TR/rdf-schema/, March 2000. W3 Consortium Candidate Recommendation.

[8] D. Ferraiolo, J. Barkley, and D. Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security*, 2, February 1999.

[9] D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proc. of the 15th NIST-NCSC National Computer Security Conference*, October 1992.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] L. Giuri. Role-based access control on the web using java. In *Proc. of the ACM Workshop on Role-Based Access Control*, 1999.

[12] G. Gottlob, M. Schrefl, and B. Rck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3), July 1996.

[13] K. Gutzmann. Access control and session management in the HTTP environment. *IEEE Internet Computing*, January/February 2001.

[14] B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory & practical language issues. *Theory and Practice of Object Systems*, 2(3), 1996.

[15] O. Lassila and R. R. Swick. Resource description framework (RDF) model and syntax specification. http://www.w3.org/TR/REC-rdf-syntax/, February 1999. W3 Consortium Recommendation.

[16] G. Neumann and U. Zdun. Enhancing object-based system composition through per-object mixins. In *Proc. of Asia-Pacific Software Engineering Conference (APSEC)*, December 1999.

[17] G. Neumann and U. Zdun. Implementing object-specific design patterns using per-object mixins. In *Proc. of Second Nordic Workshop on Software Architecture (NOSA)*, August 1999.

[18] G. Neumann and U. Zdun. Towards the usage of dynamic object aggregations as a form of composition. In *Proc. of Symposium of Applied Computing (SAC'00)*, March 2000.

[19] G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proc. of Tcl2k: 7th USENIX Tcl/Tk Conference*, February 2000.

[20] G. Neumann and U. Zdun. Distributed web application development with active web objects. In *Proc. of the 2nd International Conference on Internet Computing*, June 2001.

[21] 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 Systems Security*, 3(2), February 2000.

[22] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[23] J. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3), March 1998.

[24] J. Park and R. Sandhu. RBAC on the web by smart certificates. In *Proc. of the ACM Workshop on Role-Based Access Control*, 1999.

[25] T. Reenskaug, P. Wold, and O. Lehne. *Working with objects*. Manning Publications, 1996.

[26] D. Riehle and T. Gross. Role model based framework design and integration. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1998.

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

[28] R. Sandhu. Roles versus groups. In *Part I of Proc. ACM Workshop on Role-Based Access Control*, 1995.

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

[30] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Proc. of ACM Workshop on Role-Based Access Control*, 2000.

[31] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[32] Z. Tari and S. Chan. A role-based access control for intranet security. *IEEE Internet Computing*, September/October 1997.

[33] R. Thomas and R. Sandhu. Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented authorization management. In *Proc. of the IFIP WG11.3 Workshop on Database Security*, August 1997.

[34] D. Wetherall and C. Lindblad. Extending Tcl for dynamic object-oriented programming. In *Proc. of the Tcl/Tk Workshop 95*, July 1995.

[35] XOTcl homepage. http://www.xotcl.org.