Role-Based Access Control: A Simplified Specification[*]

Yanhong A. Liu Scott D. Stoller
Computer Science Dept., State Univ. of New York at Stony Brook, Stony Brook, NY 11794
{liu,stoller}@cs.sunysb.edu

ABSTRACT

This paper describes a simplified and corrected specification of role-based access control (RBAC) based on the specification in the ANSI standard for RBAC. The simplifications and corrections were made while following a systematic method for deriving efficient implementations from straightforward implementations of clear specifications. The method allows specifications to be written clearly without efficiency concerns. We give a complete specification of core RBAC, illustrating the principles we used in developing it; we give a complete specification of hierarchical RBAC, with an additional option for managing the relationship on roles; we describe a specification of constrained RBAC, making extension relationships among RBAC components clear. We then describe principles for developing clear and simpler specifications more extensively, explain the simplifications and corrections we made in comparison with the standard, and summarize our method and results for generating efficient implementations.

1. INTRODUCTION

Role-based access control (RBAC) is a framework for controlling user access to resources based on roles. It can significantly reduce the cost of access control policy administration and is increasingly widely used in large organizations. The ANSI standard for RBAC was developed "in recognition of a need among government and industry purchasers of information technology products for a consistent and uniform definition of role-based access control features" [2]. It was approved in 2004 after several rounds of public review [18, 11, 8], building on much research during the preceding decade (e.g., [7, 9]), although the idea of roles in access control can be traced back at least two decades [13].

The standard has four components: core RBAC defines core functionalities on permissions, users, roles, and sessions, while the other three components are extensions that add support for a hierarchy of roles, constraints on the roles of a user, and constraints on the active roles in a session, respectively. The functionalities are specified formally and precisely in a set-based specification language, Z [21], which is the standard language of its kind [10].

Because of the importance of RBAC and the ANSI standard for it, we are using them as an application in our research on optimization by incrementalization [15]. The idea is to start with straightforward but likely inefficient implementations of clear specifications and to generate sophisticated and efficient implementations following a systematic optimization method.

Our powerful optimization method allows one to write clear and modular specifications without efficiency concerns, i.e., to just define sets and relations for the given data, and define queries and updates on these data, but not maintain additional data that can be derived from the given data. The method automatically generates efficient, though not clear or modular, implementations that incrementally maintain the results of expensive queries with respect to updates to the data they depend on. This enables us to separate "what" from "how" in specifying the functionalities in a system like RBAC, and to arrive at clearer and simpler specifications that have higher assurance of correctness.

Following this approach, we found a number of complications and errors in the standard, most of which appear to be caused by interaction with efficiency concerns. For example, five of the seven mapping functions defined in the core RBAC reference model are unnecessary, and so are the updates to `assigned_users` and `assigned_permissions` in all core RBAC administrative commands. For another example, maintaining the transitive closure of the given inheritance relation contributed to an error in the `AddInheritance` command in limited hierarchical RBAC, because its precondition, which uses an incorrect definition to retrieve immediate inheritance from transitive inheritance, is always false.

This paper presents a simplified and corrected specification of RBAC. Other than the simplifications and corrections, the specification follows the functionalities of RBAC exactly as defined in the standard [2], building on the results of significant effort of many people [1]. We use the same names and preconditions for commands and functions as in the standard. The specification uses an object-oriented language with high-level operations over sets and relations and with a straightforward execution semantics; the language is also easier to read and write than Z even though it is not a

[*]This work was supported in part by ONR under grants N00014-04-1-0722 and N00014-02-1-0363 and NSF under grant CCR-0306399.

standard as Z is.

We give a complete specification of core RBAC, illustrating the principles we used in developing it; we then give a complete specification of hierarchical RBAC, with an additional option for managing the relationship on roles; we also describe a specification of constrained RBAC, making extension relationships among RBAC components clear. We then describe principles for developing clearer and simpler specifications more extensively, explain the simplifications and corrections we made in comparison with the standard, and summarize our method and results for generating efficient implementations.

The rest of this paper is organized as follows. Section 2 gives an overview of the ANSI standard for RBAC, and defines the language we use for the specification. Sections 3, 4, and 5 specify core RBAC, hierarchical RBAC, and constrained RBAC, respectively. Section 6 discusses the principles and compares with the standard. Section 7 summarizes the simplifications and discusses remaining issues.

2. PRELIMINARIES

ANSI standard for RBAC. The ANSI standard for RBAC has the following four components, where SSD (static separation of duties) and DSD (dynamic separation of duties) are also called constrained RBAC.

- core RBAC: core support for functionalities relating permissions, users, roles, and sessions.
- hierarchical RBAC: added support for a hierarchy of roles.
- SSD: added support for constraints on the roles of a user.
- DSD: added support for constraints on the active roles in a session.

For each component, the standard defines a reference model and a set of functionalities. The reference model refers to the basic data manipulated in the component, such as the set of roles, the set of users, and the relationship between roles and users. The functionalities are divided into four categories:

- administrative commands: commands for administrators to update RBAC sets and relations that are static, i.e., that are independent of user sessions.
- supporting system functions: functions that support user activities which are dynamic, i.e., that are performed as part of user sessions.
- review functions: mandatory functions for administrators to query RBAC sets and relations.
- advanced review functions: additional, optional functions for administrators to query RBAC sets and relations.

The standard uses the Z specification language [10, 21] to formally define the commands and functions, though the reference models and the relationships among components are described in English. We use the following language instead of Z for reasons explained below.

Language. Figure 1 defines the specification language used in this paper, where X^*, X^+, and $X^?$ denote that X occurs 0 or more, 1 or more, and 0 or 1 times, respectively.

A specification is a set of classes, each defining a set of fields and a set of methods, possibly with preconditions. Types may be specified not only for fields but also for variables, method parameters, and return values, although we omit those types from the grammar. We generally omit types when they can be inferred from the specification. Note the types for sets and tuples, the special `for` statement, and the expression for set comprehension. We make substantial use of sets and tuples because they are well suited for high-level specifications.

For the loop `for` v_1 `in` $e_1, ..., v_k$ `in` e_k `|` e `:` s, each variable v_i enumerates elements of the set value of expression e_i, and for each combination of values of v_1 through v_k, if the value of Boolean expression e is `true`, then execute s. We read the entire statement as "for each v_1 in e_1,..., and v_k in e_k such that e, do s". We omit `|` e when e is `true`.

For the set comprehension `{`e_0`:` v_1 `in` $e_1, ..., v_k$ `in` e_k `|` e`}`, each variable v_i enumerates elements of the set value of expression e_i, and for each combination of values of v_1 through v_k, if the value of Boolean expression e is `true`, then the value of expression e_0 is an element of the resulting set. We read the expression as "the set of e_0 where v_1 is from e_1,..., and v_k is from e_k such that e". We omit `|` e when e is `true`.

We use the kinds of expressions in Figure 2 for other operations on sets and tuples. We use standard infix set notation instead of object-oriented notation, although we express the operators in ASCII. They are mostly like expressions on sets and tuples in SETL [20]. We abbreviate the expression `forall` x `in` S `| (forall` y `in` T `| e)` as `forall` x `in` S, y `in` T `| e`, and abbreviate `forall` x `in` S`,` y `in` S `| e` as `forall` x`,` y `in` S `| e`; similarly for `exists`.

Other statements and expressions are as in popular object-oriented languages such as Java. Only side-effect-free methods may be invoked in expressions, and their bodies are always of the form `return` *expr*. We use the convention that `this.`*methodname*(*expr**) and `this.`*fieldname* are abbreviated as *methodname*(*expr**) and *fieldname*, respectively. We abbreviate `and` as a comma. We use indentation to indicate scoping. We use `//` to begin a comment that lasts till the end of the line.

The set operations and other constructs in our language have their standard semantics, allowing formal verification of the specifications. Our language differs from the Z specification language in four main ways: (1) it supports modularity and inheritance, which is needed for describing a system with multiple components where a component may extend others, (2) it is executable—the semantics of a specification corresponds to a straightforward implementation of the specification, (3) it is easy to read, especially set comprehension, since the subexpressions appear in the same order as in a natural English description, and (4) it is easy to write, since it uses only ASCII characters.

3. SPECIFICATION OF CORE RBAC

To write a clear and modular specification, we just define a class for each component, define sets and relations for the given data, as fields, and define queries and updates on these data, as methods. We should not maintain additional data

```
spec  ::= class+
class ::= class classname (extends classname+)?
            (fieldname : type)*
            (methodname(varname*): (precondition: expr ;)? stmt)*
type  ::= set(type) | tuple(type+) | classname | int | ...
stmt  ::= for (varname in expr)+ | expr: stmt
          | expr.methodname(expr*) | return expr
          | expr.fieldname = expr | varname = expr | stmt stmt | ...
expr  ::= {expr: (varname in expr)+ | expr}
          | expr.methodname(expr*)
          | expr.fieldname | varname | expr and expr | ...
classname, fieldname, methodname, varname: identifiers
```

Figure 1: Language.

`{`x_1, ..., x_k`}`	a set with elements x_1, ..., x_k
`[`x_1, ..., x_k`]`	a tuple with elements x_1, ..., x_k
`{[`x_1`,`y_1`]`, ...,`[`x_k`,`y_k`]}`	a binary relation, i.e., a set of 2-tuples, i.e., pairs
S `+` T, S `-` T	union and difference, respectively, of sets S and T
S `subset` T	whether S is a subset of or equal to T
x `in` S, x `notin` S	whether or not, respectively, x is an element of S
`exists` x `in` S `|` e	whether some element x in S satisfies condition e
`forall` x `in` S `|` e	whether every element x in S satisfies condition e
`#`S	number of elements in S

Figure 2: Operations on sets and tuples.

that can be derived from the given data. Derived data would need to be updated at all places where the given data it depends on is updated, which would yield a specification that is not clear or modular.

We specify the core RBAC component as a class `CoreRBAC`, whose fields specify the reference model, and whose methods specify the functionalities.

```
class CoreRBAC:
  ... // reference model
  ... // functionalities
```

We define the following sets and relations for the reference model of core RBAC, explained below.

```
OBJS:     set(Object)    // an operation-object pair
OPS:      set(Operation) // is called a permission.
USERS:    set(User)
ROLES:    set(Role)
PR:       set(tuple(tuple(Operation, Object), Role))
UR:       set(tuple(User, Role))
          // PR subset (OPS * OBJS) * ROLES
          // UR subset USERS * ROLES
SESSIONS: set(Session)
SU:       set(tuple(Session, User))
SR:       set(tuple(Session, Role))
          // SU subset SESSIONS * USERS
          // SR subset SESSIONS * ROLES
```

A system has sets of objects, operations, users, roles, and sessions; their elements are of types `Object`, `Operation`, `User`, `Role`, and `Session`, respectively. A operation-object pair, called a permission, denotes an allowed operation on an object. A permission-role pair in `PR` denotes a permission assigned to a role. A user-role pair in `UR` denotes a user assigned to a role. A session-user pair in `SU` denotes a session and the unique user of the session. A session-role pair in `SR` denotes a session and a role active in the session.

We define below the functionalities summarized in the following table for core RBAC. All administrative commands and supporting system functions except for check-access are updates, and the other functions are queries.

Categories	Core RBAC Functionalities
administrative commands	add/delete user/role, assign/deassign user, grant/revoke permission
supporting system functions	create/delete session, add/drop active role, check access
review functions	assigned users/roles
advanced review functions	role/user permissions, session roles/perms, role/user ops on obj

Administrative commands. Adding an element to a set of users or roles (`AddUser`, `AddRole`) can be specified in a similar way; the same is true for adding an element to a relation (`AssignUser` adding to `UR`, and `GrantPermission` adding to `PR`, except that the latter uses nested tuples).

```
AddUser(user):
  precondition: user notin USERS;
  USERS = USERS + {user}

AddRole(role):
  precondition: role notin ROLES;
  ROLES = ROLES + {role}

AssignUser(user, role):
  precondition: user in USERS, role in ROLES,
                [user,role] notin UR;
  UR = UR + {[user, role]}
```

```
GrantPermission(operation, object, role):
  precondition: operation in OPS, object in OBJS,
                role in ROLES,
                [[operation,object],role] notin PR;
  PR = PR + {[[operation,object],role]}
```

Deleting an element is symmetric to adding an element, but possibly with two kinds of additional updates. First, if an element is deleted from a set, then from all relations defined using the set, all pairs that contain the deleted element must be deleted. Second, `DeleteUser`, `DeleteRole`, and `DeassignUser` may affect `SESSIONS`, because sessions are created by users and have active roles, and must satisfy the constraint that a session can have a role only if the user of the session is assigned that role. Specifically, `DeleteUser` may either delete associated sessions or leave the sessions to terminate normally; `DeleteRole` and `DeassignUser` have a third option of deleting only the specified role from the sessions. As in the standard, we formally specify only the first option, i.e., deleting all associated sessions, for all three operations, illustrating both kinds of additional updates; the other two options are simpler to specify.

```
DeleteUser(user):
  precondition: user in USERS;
  UR = UR - {[user,r]: r in ROLES} //maintain UR
  for s in SESSIONS                //maintain SESSIONS
    | [s,user] in SU:
    DeleteSession(user,s) //DeleteSession defined below
  USERS = USERS - {user}  //update last for pre-
                          //condition of DeleteSession

DeleteRole(role):
  precondition: role in ROLES;
  PR = PR - {[[op,obj],role]: op in OPS, obj in OBJS}
                                   //maintain PR
  UR = UR - {[u,role]: u in USERS} //maintain UR
  for s in SESSIONS, u in USERS    //maintain SESSIONS
    | [s,u] in SU, [s,role] in SR:
    DeleteSession(u,s)
  ROLES = ROLES - {role}  //update last for update
                          //of SR in DeleteSession

DeassignUser(user, role):
  precondition: user in USERS, role in ROLES,
                [user,role] in UR;
  for s in SESSIONS                //maintain SESSIONS
    | [s,user] in SU, [s,role] in SR:
    DeleteSession(user,s)
  UR = UR - {[user,role]}

RevokePermission(operation, object, role):
  precondition: operation in OPS, object in OBJS,
                role in ROLES,
                [[operation,object],role] in PR;
  PR = PR - {[[operation,object],role]}
```

Supporting system functions. `CreateSession` creates a session for a user with an initial set of active roles; it first checks that the user is assigned those roles, and then adds the appropriate elements to `SU`, `SR`, and `SESSIONS`. `DeleteSession` deletes all elements of `SU`, `SR`, and `SESSIONS` that are associated with the session.

```
CreateSession(user, session, ars):
  precondition: user in USERS, session notin SESSIONS,
                ars subset AssignedRoles(user);
                //AssignedRoles is defined below
  SU = SU + {[session,user]}
  SR = SR + {[session,r]: r in ars}
  SESSIONS = SESSIONS + {session}

DeleteSession(user, session):
  precondition: user in USERS, session in SESSIONS,
                [session,user] in SU;
  SU = SU - {[session,user]}
  SR = SR - {[session,r]: r in ROLES} //maintain SR
  SESSIONS = SESSIONS - {session}
```

Adding and deleting active roles adds to and deletes from `SR`, respectively; adding an active role also first checks that the user of the session is assigned that role. Note the last condition calls `AssignedRoles`, as done in `CreateSession`, but it can also be written as `user in AssignedUsers(role)` or as `[user,role] in UR`.

```
AddActiveRole(user, session, role):
  precondition: user in USERS, session in SESSIONS,
                role in ROLES, [session,user] in SU,
                [session,role] notin SR,
                role in AssignedRoles(user);
  SR = SR + {[session,role]}

DropActiveRole(user, session, role):
  precondition: user in USERS, session in SESSIONS,
                role in ROLES, [session,user] in SU,
                [session,role] in SR;
  SR = SR - {[session,role]}
```

`CheckAccess` checks whether an operation on an object is allowed in a session, i.e., whether the session has an active role that is assigned the operation-object pair as a permission.

```
CheckAccess(session, operation, object):
  precondition: session in SESSIONS,
                operation in OPS, object in OBJS;
  return exists r in ROLES | [session,r] in SR,
                       [[operation,object],r] in PR
```

Review functions and advanced review functions. These functions are queries on the sets and relations. Most of them (`AssignedUsers`, `AssignedRoles`, `RolePermissions`, `SessionRoles`, `RoleOperationsOnObject`) are over one relation, i.e., given a value for the left or right component of a relation, find all associated values for the other component in the relation. For example, the first two are review functions defined by:

```
AssignedUsers(role):
  precondition: role in ROLES;
  return {u: u in USERS | [u,role] in UR}

AssignedRoles(user):
  precondition: user in USERS;
  return {r: r in ROLES | [user,r] in UR}
```

The other ones (`UserPermissions`, `SessionPermissions`, `UserOperationsOnObject`) are over two relations, i.e., given a value for one component of a relation, equate the other component of the relation with one component of a second relation, and find all associated values for the other component of the second relation. Two of the functions (`RoleOperationsOnObject`, `UserOperationsOnObject`) involve lookups over nested tuples but are otherwise similar to the other functions. All advanced review functions are defined below:

```
RolePermissions(role):
  precondition: role in ROLES;
  return {[op,obj]: op in OPS, obj in OBJS
          | [[op,obj],role] in PR}

UserPermissions(user):
  precondition: user in USERS;
  return {[op,obj]: r in ROLES, op in OPS, obj in OBJS
          | [user,r] in UR, [[op,obj],r] in PR}

SessionRoles(session):
  precondition: session in SESSIONS;
  return {r: r in ROLES | [session,r] in SR}

SessionPermissions(session):
  precondition: session in SESSIONS;
  return {[op,obj]: r in ROLES, op in OPS, obj in OBJS
          | [session,r] in SR, [[op,obj],r] in PR}

RoleOperationsOnObject(role, object):
  precondition: role in ROLES, object in OBJS;
  return {op: op in OPS | [[op,object],role] in PR}

UserOperationsOnObject(user, object):
  precondition: user in USERS, object in OBJS;
  return {op: r in ROLES, op in OPS
          | [user,r] in UR, [[op,object],r] in PR}
```

4. SPECIFICATION OF HIERARCHICAL RBAC

Hierarchical RBAC allows a role to inherit permissions from other roles without being granted those permissions directly. The ANSI standard for hierarchical RBAC has two sub-components: general hierarchy, which allows multiple inheritance, and limited hierarchy, which allows only single inheritance. For both of them, the standard requires that the inheritance relation be acyclic.

We consider the same two subcomponents, for consistency with the standard. However, we see little motivation for limited hierarchy. Single inheritance in object-oriented languages avoids the problem of a class inheriting conflicting definitions of a method, but that problem does not arise in role hierarchy. While inheriting from more than one role may give a role too much power, a role may acquire too much power from other operations anyway, so other controls, such as separation of duties in constrained RBAC, are used to prevent this.

We also consider a third option, which we call unrestricted inheritance, where the inheritance relation is unrestricted and thus may contain cycles. Although managerial hierarchies are acyclic, roles and their relationships do not always mimic managerial hierarchies, and the extra flexibility from allowing cycles may be useful. A cycle simply means that all the roles in the cycle are in an equivalence class and indeed have the same permissions. This is useful compared with forcing all the roles in the cycle to be one role because an inheritance edge in the cycle may be removed later and the remaining inheritance edges can still be used.

For general and limited hierarchical RBAC, we define the following two classes, one for each of the subcomponents, where the new and redefined parts are specified below.

```
class GeneralHierarchicalRBAC extends CoreRBAC:
  ... // new inheritance relation in reference model
  ... // new and redefined functionalities
```

```
class LimitedHierarchicalRBAC extends
                                GeneralHierarchicalRBAC:
  ... // redefined functionalities
```

The third option discussed above could be defined as a new subcomponent that extends core RBAC, and general hierarchical RBAC could extend it instead of core RBAC.

We define the inheritance relation INH to be a set of role pairs given explicitly by administrators, not the reflexive-transitive closure of the role pairs given.

```
INH: set(tuple(Role, Role))
```

A pair [r1,r2] in INH, where $r1 \neq r2$, denotes that r1 inherits from r2; the implication is that a user assigned r1 can activate not only r1 but also r2, without having to be assigned r2 directly. We call r1 the heir and r2 the bearer. We use INH* to denote the transitive-reflexive closure of INH. The acyclicity requirement is

```
forall r1, r2 in ROLES |
  [r1,r2] in INH*, [r2,r1] in INH* ⇒ r1=r2
```

and the single inheritance requirement is

```
forall r, r1, r2 in ROLES |
  [r,r1] in INH, [r,r2] in INH ⇒ r1=r2
```

The functionalities of general hierarchical RBAC are the same as those of core RBAC except for the changes summarized in the following table, where + indicates additional functionalities, and ∓ indicates redefined functionalities.

Categories	General Hierarchical RBAC Functionalities
administrative commands	+ add/delete inheritance, + add ascendant/descendant
supporting system functions	∓ create session, ∓ add active role
review functions	+ authorized users/roles
advanced review functions	∓ role/user permissions, ∓ role/user ops on obj

The functionalities of limited hierarchical RBAC are the same as those of general hierarchical RBAC except for a modification to the administrative command AddInheritance. In fact, all functionalities except for AddInheritance are the same for unrestricted inheritance, general hierarchy, and limited hierarchy, so we describe them together below.

The command AddInheritance simply adds a new pair to INH for unrestricted inheritance; for general hierarchy, its precondition checks acyclicity, and for limited hierarchy, also checks single inheritance. DeleteInheritance simply removes a pair from INH. AddAscendant and AddDescendant are self explanatory, although we would call them AddHeir and AddBearer, respectively.

```
AddInheritance(heir, bearer): //for unrestricted
                              //inheritance
  precondition: heir in ROLES, bearer in ROLES,
               [heir,bearer] notin INH, heir!=bearer;
  INH = INH + {[heir,bearer]}

AddInheritance(heir, bearer): //for general hierarchy
  ... //same as above except to add precondition
      //[bearer,hier] notin INH*, to check acyclicity
```

```
AddInheritance(heir, bearer): //for limited hierarchy
  ... //same as above except to also add precondition
      //not exists r in ROLES | [heir,r] in INH,
      //to check single inheritance

DeleteInheritance(heir, bearer):
  precondition: heir in ROLES, bearer in ROLES,
                [heir,bearer] in INH;
  INH = INH - {[heir,bearer]}

AddAscendant(heir, bearer):
  AddRole(heir)
  AddInheritance(heir,bearer)

AddDescendant(bearer, heir):
  AddRole(bearer)
  AddInheritance(heir,bearer)
```

The supporting system functions `CreateSession` and `AddActiveRole` are minimally modified to call a new review function; this change allows a user to activate inherited roles. Following the standard, the function `CheckAccess` is inherited from core RBAC and does not use the inheritance relation; to use a permission from inherited roles, a user must find an authorized role that is assigned that permission and explicitly activate that role for the session during `CreateSession` or using `AddActiveRole`.

```
CreateSession(user, session, ars):
  ... //same as in CoreRBAC except in precondition,
      //AssignedRoles is replaced with AuthorizedRoles,
      //which is defined below

AddActiveRole(user, session, role):
  ... //same change as for CreateSession above
```

New review functions (`AuthorizedUsers`, `AuthorizedRoles`) and redefined advanced review functions (`RolePermissions`, `UserPermissions`, `RoleOperationsOnObject`, and `UserOperationsOnObject`) use the inheritance relation together with `UR` and `PR`. Note that `SessionRoles` and `SessionPermissions` are inherited from core RBAC and do not use the inheritance relation, consistent with the definition of `CheckAccess`.

```
AuthorizedUsers(role):
  precondition: role in ROLES;
  return {u: heir in ROLES, u in USERS
          | [heir,role] in INH*, [u,heir] in UR}

AuthorizedRoles(user):
  precondition: user in USERS;
  return {r: heir in ROLES, r in ROLES
          | [user,heir] in UR, [heir,r] in INH*}

RolePermissions(role):
  precondition: role in ROLES;
  return {[op,obj]: bearer in ROLES,
                    op in OPS, obj in OBJS
          | [role,bearer] in INH*,
            [[op,obj],bearer] in PR}

UserPermissions(user):
  precondition: user in USERS;
  return {[op,obj]: heir in ROLES, bearer in ROLES,
                    op in OPS, obj in OBJS
          | [user,heir] in UR, [heir,bearer] in INH*,
            [[op,obj],bearer] in PR}

RoleOperationsOnObject(role, object):
  precondition: role in ROLES, object in OBJS;
  return {op: bearer in ROLES, op in OPS
          | [role,bearer] in INH*,
            [[op,object],bearer] in PR}

UserOperationsOnObject(user, object):
  precondition: user in USERS, object in OBJS;
  return {op: heir in ROLES, bearer in ROLES,op in OPS
          | [user,heir] in UR, [heir,bearer] in INH*,
            [[op,object],bearer] in PR}
```

5. SPECIFICATION OF CONSTRAINED RBAC

Constrained RBAC supports separation of duty, whose purpose is to reduce fraud by limiting the power of individual users (statically constrained) or individual sessions (dynamically constrained), so fraud can be perpetrated only through collusion among multiple users or multiple sessions.

Static separation of duty. A static separation of duty (SSD) constraint is characterized by a name, used to identify it in administrative commands, a set `rs` of roles, and a natural number `n`, called the cardinality, such that `1 <= n <= #rs-1`. In our specification, the meaning of an SSD constraint is that a user can be assigned to `n` or fewer roles from `rs`. We find this interpretation more intuitive than the one in the standard, which says that `2 <= n <= #rs` and that a user can be assigned to fewer than `n` roles from `rs`. Our formal model of an SSD constraint includes a set `SsdNAMES` of SSD constraint names, a relation `SsdNR` that relates a name in `SsdNAMES` to a role in the associated role set, and a relation `SsdNC` that relates a name in `SsdNAMES` to its unique associated cardinality.

```
SsdNAMES: set(SsdName)
SsdNR:    set(tuple(SsdName, Role))
SsdNC:    set(tuple(SsdName, int))
```

SSD constraints can be added to core RBAC or general or limited hierarchical RBAC. In core RBAC with SSD constraints, the assignment of roles to users must satisfy

```
forall u in USERS, [name,n] in SsdNC |
  #{r: r in AssignedRoles(u) | [name,r] in SsdNR} <= n
```

In general or limited hierarchical RBAC with SSD constraints, the user assignment and inheritance relation must satisfy the same constraints except with `AssignedRoles` replaced with `AuthorizedRoles`.

The functionalities are described below. We omit the detailed definitions because they are straightforward aside from the points explained.

Core RBAC with SSD constraints extends core RBAC. All administrative commands are inherited except `AssignUser` is redefined to also check that the updated user assignment would satisfy the SSD constraints. New administrative commands (`Create/DeleteSsdSet`, `Add/DeleteSsdRoleMember`, `SetSsdSetCardinality`) are added to create, modify, and delete SSD constraints; the non-deletion commands check that the new or updated SSD constraint would be satisfied, and that the cardinality would be in the required range. New review functions (`SsdRoleSets`, `SsdRoleSetRoles`, SsdRoleSetCardinality) are introduced to query SSD constraints. Supporting system functions and advanced review functions are simply inherited.

General hierarchical RBAC with SSD constraints is defined similarly, except that (1) it extends general hierarchical RBAC and redefines also command AddInheritance to check that the SSD constraints would be satisfied, and (2) it also extends core RBAC with SSD constraints and just redefines non-deletion commands. These redefinitions simply use AuthorizedRoles in place of AssignedRoles.

Limited hierarchical RBAC with SSD constraints is defined by extending general hierarchical RBAC with SSD constraints and just redefining AddInheritance to add a check for single inheritance.

Dynamic separation of duty. A dynamic separation of duty (DSD) constraint is also characterized by a name, a set rs of roles, and a cardinality n such that 1 <= n <= #rs-1. In our specification, the meaning is that a session can have n or fewer roles from rs that are active. Our interpretation of the cardinality is different than the interpretation in the standard, just like for SSD constraints. Our formal model of DSD constraints is very similar to our model of SSD constraints.

```
DsdNAMES: set(DsdName)
DsdNR:    set(tuple(DsdName, Role))
DsdNC:    set(tuple(DsdName, int))
```

DSD constraints can be added to core RBAC, general or limited hierarchical RBAC. In all of them, the following condition must be satisfied.

```
forall s in SESSIONS, [name,n] in DsdNC |
  #{r: r in SessionRoles(s) | [name,r] in DsdNR} <= n
```

Core RBAC with DSD constraints extends core RBAC. All administrative commands are inherited. New administrative commands are added to create, modify, and delete DSD constraints, just as for SSD above, except that DSD constraints are checked instead of SSD constraints. All supporting system functions are inherited except that CreateSession and AddActiveRole are redefined to also check that the DSD constraints would be satisfied. New review functions are introduced to query DSD constraints, similar to those for SSD above. Advanced review functions are simply inherited.

General hierarchical RBAC with DSD constraints is defined similarly, except that (1) it extends general hierarchical RBAC and simply inherits all definitions, and (2) it also extends core RBAC with DSD constraints and just redefines CreateSession and AddActiveRole to use AuthorizedRoles in place of AssignedRoles.

Limited hierarchical RBAC with DSD constraints is defined by extending general hierarchical RBAC with DSD constraints and just redefining AddInheritance to be the same as in limited hierarchical RBAC.

6. DISCUSSION AND COMPARISON WITH THE ANSI STANDARD

We discuss the principles we use for developing clearer and simpler specifications with higher assurance for correctness. Compared with the standard, we eliminated a number of complications and redundancies and corrected a number of errors, most of which appear to be caused by efficiency considerations.

Maintaining basic, not derived, sets and relations. For clarity and simplicity, only basic data should be maintained in specifications, where basic data refers to data given and modified externally, in contrast to derived data that can be computed from basic data. Following this principle, we found unnecessary complications in core RBAC and hierarchical RBAC, a subtle error in hierarchical RBAC, and omissions in the standard.

The main unnecessary complications in core RBAC are that five of the seven mapping functions defined in the reference model (assigned_users, assigned_permissions, Op, Ob, and avail_session_perms) are unnecessary, and so are all updates to assigned_users and assigned_permissions in all administrative commands, so we eliminated them. These mapping functions are not used in the rest of the specification. The functions corresponding to assigned_users, assigned_permissions, and avail_session_perms are also defined as review or advanced review functions, called AssignedUsers, RolePermissions, and SessionPermissions, respectively; the other two can be defined as review functions too if needed. Note that incrementally updating the result of a query, such as AssignedUsers, is needed only for efficiency reasons, because the result can always be computed from scratch; such updates, if needed for efficiency reason, can be derived from the query and how the data it depends on are updated, as we do in [15].

Similarly, we removed the two mapping functions, authorized_permissions and authorized_users, in the reference model of hierarchical RBAC. Another small simplification we made to core RBAC is that we removed the set PERMS , which equals OPS $\times$ OBJS , so the few uses of PERMS are replaced with uses of OPS and OBJS .

The error is that the precondition of the AddInheritance command in limited hierarchical RBAC is always false, due to the incorrect definition of the inheritance relation in terms of its reflexive-transitive closure. The standard maintains the derived relation INH* (denoted RH in the standard) instead of the basic relation INH. This makes some functionalities more efficient, but it requires giving a definition of INH in terms of INH*. The definition given in Section 5.2 of the standard is incorrect, because it does not remove the reflexive relationships and therefore does not completely undo the effects of the reflexive-transitive closure. This makes the precondition of AddInheritance in limited hierarchical RBAC always false, and could also affect other uses of the direct inheritance relation. The most straightforward remedy is to maintain the basic relation INH and use INH* as needed, as done in Section 4 of this paper. Fixing the incorrect definition is not as good a remedy, because it results in an unnecessarily complicated specification. Maintaining INH instead of INH* yields much simpler definitions of AddInheritance and DeleteInheritance.

Maintaining INH instead of INH* also provides a more natural semantics for DeleteInheritance. For example, consider this sequence of calls: AddInheritance(r1,r2), AddInheritance(r1,r3), AddInheritance(r2,r3), and DeleteInheritance(r2,r3). With our specification, the last two calls cancel each other out exactly; in other words, AddInheritance and DeleteInheritance are inverses. With the definition in the standard, the call to DeleteInheritance also removes the inheritance relation between r1 and r3.

Omissions of functionalities were found because our design principle allows one to freely update and query basic data as needed; we did not add them to the specifications earlier to avoid distraction from the main concepts. First, basic data is given and updated externally, so functionalities should be provided for appropriate updates. The standard lacks commands to add and delete objects and operations. These commands can be defined similarly to commands for adding and deleting users and roles. Similarly, some query functions seem needed but not provided. For example, in hierarchical RBAC, since a user must explicitly activate an inherited role to use its permissions, one needs a function PermissionRoles to find the set of roles granted a given permission and, furthermore, a function UserPermissionRoles to find the set of roles authorized to the user and granted the given permission:

```
PermissionRoles(operation, object):
  precondition: operation in OPS, object in OBJS;
  return {r: r in ROLES | [[operation,object],r] in PR}

UserPermissionRoles(user, operation, object):
  precondition: user in USERS,
                operation in OPS, object in OBJS;
  return {r: r in ROLES | r in AuthorizedRoles(user),
                          [[operation,object],r] in PR}
```

Query functions should probably also include most of the mapping functions in the standard that do not have corresponding review or advanced review functions. For example, one other mapping function in core RBAC (besides the five eliminated), called session_users (though could perhaps be called session_user), maps a session to its unique user; this could be a useful review function.

There are two related problems. First, some of the query functions are needed by users as well as administrators, so it is not clear whether they should be classified as supporting system functions or review or advanced review functions. Second, it is not clear what criteria are used in the standard to distinguish review functions from advanced review functions. For example, SessionPermissions is essentially the operation needed for efficient CheckAccess (discussed with other simplifications below), but it is only an optional advanced review function, not a mandatory review function.

Using relations instead of mapping functions. Relations should generally be used instead of mapping functions. Relations can be updated more readily, and information in them can easily be used as mappings from any components to other components. In particular, a binary relation can be easily used as two mapping functions—from left to right, and right to left. Following this principle, we replaced the two other mapping functions in core RBAC (besides the five eliminated), session_user and session_roles, with relations SU and SR, respectively. This simplified maintenance of these values in all supporting system functions and uses of these values in deletion commands, as discussed below. We also replaced mapping functions with relations in constrained RBAC, which gives similar benefits.

To illustrate the benefit of easy maintenance, consider the command AddActiveRole(user,session,role). In the standard, it updates session_roles by retrieving the current value of session_roles(session), inserting role in the returned set to obtain the new set of active roles, removing the current entry for this session from session_roles, and then adding an entry to session_roles that maps this session to the new set of active roles. The formula to do this is too long to fit on one line. In our specification, the update is simply SR = SR + {[session,role]}.

To illustrate the benefit of binary relations being easily usable in both directions, consider DeleteRole, which calls DeleteSession to delete sessions in which the role being deleted is active. There is an error in the standard here: DeleteSession has two parameters, a user and a session, but the call site in DeleteRole passes only one argument, namely, the session. To fix this, user_sessions (which is not defined but is used in multiple functionalities in the standard) needs to be used in reverse to find the associated user. Using a function in reverse is awkward. In our specification, the relation SU can easily be used in both directions.

For statically constrained RBAC, the standard represents SSD constraints as a set SSD of names together with mapping functions ssd_set and ssd_card that map each name to the associated role set and cardinality, respectively. We replaced the two mapping functions with two relations, SsdNR and SsdNC. This leads to simplified specifications of the functionalities for manipulating SSD constraints; the simplifications are similar to those described above. We did a similar replacement for DSD constraints in dynamically constrained RBAC.

Other simplifications. One kind of simplification is to replace a complicated formula with a more straightforward, shorter, and logically equivalent formula. Another kind of simplification is to call other defined functions, instead of repeating the bodies of those functions. One can also combine these two kinds of simplifications.

We did the first kind of simplification for all the more complicated functionalities of statically constrained RBAC. Specifically, in administrative commands, both redefined commands (AssignUser and AddInheritance) and three of the five new commands (CreateSsdSet, AddSsdRoleMember, SetSsdSetCardinality) contain similar conditions that check whether the attempted operation would lead to the user assignment violating the SSD constraints. As written in the standard, all of these conditions are hard to read, because they involve a triple or quadruple subscript and an implicit universal quantification over subset, and because they are structured much differently than the informal explanation of SSD constraints in Section 5.3.1 of the standard. In our specification, all of these conditions are replaced with simpler ones based closely on the informal explanation, like our formula for SSD constraints in Section 5 of this paper.

We did the second kind of simplification in the preconditions of CreateSession and AddActiveRole in core RBAC and hierarchical RBAC. For example, the precondition of CreateSession(user,session,ars) in core RBAC checks whether user is assigned all of the roles in ars. In the standard, this condition includes repeating the body of the review function AssignedRoles. In our specification, that expression is replaced with a call to AssignedRoles; this generally makes the specification shorter and easier to read. The benefit shows up more when CreateSession in hierarchical RBAC needs a more complicated precondition, and one can simply replace AssignedRoles with AuthorizedRoles in the definition. Similarly, also for consistency, we call

AssignedRoles and AuthorizedRoles in the precondition of AddActiveRole in core RBAC and hierarchical RBAC, respectively. Note that the standard uses the mapping function authorized_users in the precondition of AddActiveRole in hierarchical RBAC; user in authorized_users(role) equals user in AuthorizedUsers(role), which equals our role in AuthorizedRoles(user).

For combining both kinds of simplifications, consider CheckAccess(session,operation,object), which returns the Boolean value of the following expression.

```
exists r in ROLES | [session,r] in SR,
            [[operation,object],r] in PR
```

This expression equals at least the following three expressions that use advanced review functions, although, here, the bodies of those functions are not exactly the same as the replaced fragments; the last one clearly looks neatest:

```
exists r in ROLES | [session,r] in SR,
             [operation,object] in RolePermissions(r)
exists r in SessionRoles(session)
           | [operation,object] in RolePermissions(r)
[operation,object] in SessionPermissions(session)
```

One could also use a review or advanced review function in the definition of another function; even when such rewrite does not yield a more concise definition, one might find that in an extended component, only the first function has to be redefined instead of both. For example, if in core RBAC one uses RolePermissions and RoleOperationsOnObject in the definitions of UserPermissions and UserOperationsOnObject, respectively, then in hierarchical RBAC, only the first two have to be redefined, not all four. We did not do these rewrites, since we felt that they are not more straightforward, even though they look neater—one must understand the functions called to understand the caller.

Implementation. Our specification can be translated straightforwardly into a programming language that supports objects and classes as well as sets and tuples. We chose Python (http://www.python.org/), which has excellent support for these. This provides an executable specification, useful for validation and prototyping. This straightforward implementation is inefficient, because it always computes expensive queries, such as the set of roles satisfying the conditions in CheckAccess, from scratch.

We have developed a powerful method for optimization by incrementalization [15], which analyzes programs to identify expensive queries and updates to their parameters, i.e., values on which the query result depends, and transforms the programs to incrementally maintain the results of expensive queries with respect to updates to their parameters. We applied a prototype implementation of this method for Python to our straightforward implementation of core RBAC, and obtained efficient implementations [16]. The incrementalized implementations incrementally maintain the results of expensive queries, such as the set of roles satisfying the conditions in CheckAccess. The experiments reported in [15, 16] confirm the effectiveness of this method, showing improvements from polynomial time in the straightforward implementation, to constant time in the incrementalized implementations, for CheckAccess.

An executable specification of core RBAC is at ftp://ftp.cs.sunysb.edu/pub/liu/coreRBAC.py. We plan to make an executable specification of the entire RBAC available.

7. CONCLUSION

To summarize, this paper shows how a simplified specification can be developed following important principles. In addition to correcting a number of errors, we made the following main simplifications to the specification in the standard. For core RBAC, we eliminated five of the seven mapping functions and maintenance of assigned_users and assigned_permissions in all administrative commands, replaced the two other mapping functions with relations and simplified their maintenance in all supporting system functions. For hierarchical RBAC, we eliminated the two mapping functions, maintain the direct inheritance relation instead of the transitive inheritance relation, and simplified AddInheritance and DeleteInheritance. In constrained RBAC, we simplified the SSD constraints, and thus all the commands and functions that check them, and we replaced two mapping functions with relations, which also leads to simplified definitions of some functionalities.

There has been much effort by many people on development of access control models and on specifications of these models [1]. This includes not only the specification of RBAC in the standard, but also specifications of various variants, extensions, and subsets of RBAC, notably [17, 5, 3, 12, 19, 6, 4, 14], among others, some as parts of larger works. We are not aware of any work that yields the same simplified specification as in this paper. Furthermore, the simplifications follow from a set of general principles. We believe that the principles can also be used to write clear specifications for new variants and extensions of access control models, and write them more easily and faster.

Remaining issues. There are several ways that our current specification and the RBAC model could be further improved.

First, for specifying administrative commands DeleteUser, DeleteRole, and DeassignUser, the standard informally describes multiple acceptable alternatives but formally specifies only one of them, and we did the same. The other alternatives should also be formally specified, but more importantly, the challenge is to find the best way to specify all the alternatives declaratively, instead of calling DeleteSession in a loop.

A related issue is that most administrative commands perform a primary update accompanied by secondary updates needed to preserve consistency among the sets and relations. For example, in DeleteUser, the update to USERS is accompanied by an update to UR and a sequence of calls to DeleteSession, which updates SU, SR, and SESSIONS. A better approach would be to express the consistency constraints explicitly and declaratively, and systematically derive the secondary updates needed to preserve them.

Another issue is that there seems to be a conceptual inconsistency between hierarchical RBAC and core RBAC in the standard. In core RBAC, a session is deleted if a role active in the session is deleted, or if the user of the session is deassigned from a role that is active in the session. In hierarchical RBAC, a session is not deleted even if a role active in the session was authorized by an inheritance and the inheritance is deleted.

In many applications, separation of duty constraints are most naturally expressed in terms of permission to perform multiple specified operations on a single object or in a single transaction. It would be useful to extend the specification to support this. Such separation of duty constraints cannot be reliably enforced by the static or dynamic separation of duty constraints in the current specification, because they can be circumvented by updates to the user-role assignment and use of multiple sessions, respectively.

More powerful security policies may require the handling of attributes, trust, information flow, etc. We believe that a unified framework that incorporates all these aspects is attainable, and that this is an important subject for further study.

Finally, formal verification of correctness properties of the specification, similar as done for variants of RBAC [19], would provide even higher assurance of its correctness.

Acknowledgment

Michael Gorbovitski implemented the method for optimization by incrementalization; he also first used `AssignedRoles` in the precondition of `CreateSession`. Tom Rothamel helped with experiments for optimization by incrementalization. Hongxu Cai implemented a clean version of the core RBAC and hierarchical RBAC standard in Python and described a number of inconsistencies in the standard. They all helped confirm some of the errors and complications in the standard as described in this paper. Amit Sasturkar provided helpful comments on a draft of this paper. Chen Wang found an error in the specification in a draft of this paper. Zongyan Qiu pointed out the conceptual inconsistency between hierarchical RBAC and core RBAC discussed in the conclusion.

8. REFERENCES

[1] ACM SACMAT. ACM Symposium on Access Models and Technologies. http://www.sacmat.org/.

[2] American National Standards Institute, Inc. Role-Based Access Control. ANSI INCITS 359-2004. Approved Feb. 3, 2004.

[3] J. Bacon, K. Moody, and W. Yao. A model of oasis role-based access control and its support for active security. *ACM Trans. Inf. Syst. Secur.*, 5(4):492–540, 2002.

[4] S. Barker and P. J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. Inf. Syst. Secur.*, 6(4):501–546, 2003.

[5] E. Bertino, P. A. Bonatti, and E. Ferrari. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.

[6] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, 6(1):71–127, 2003.

[7] D. Ferraiolo and R. Kuhn. Role-based access control. In *Proceedings of the NIST-NSA National Computer Security Conference*, pages 554–563, 1992.

[8] 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 Systems Security*, 4(3):224–274, 2001.

[9] A. Gavrila and J. Barkley. Formal specification for RBAC user/role and role relationship management. In *Proceedings of the 3rd ACM Workshop on Role Based Access Control*, pages 81–90, 1998.

[10] International Organization for Standardization. Z formal specification notation – Syntax, type system and semantics. ISO/IEC 13568:2002.

[11] T. Jaeger and J. Tidswell. Rebuttal to the NIST RBAC model proposal. In *Proceedings of the 5th ACM Workshop on Role Based Access Control*, pages 66–66, Berlin, Germany, July 2000.

[12] M. Koch, L. V. Mancini, and F. Parisi-Presicce. A graph-based formalism for rbac. *ACM Trans. Inf. Syst. Secur.*, 5(3):332–365, 2002.

[13] C. E. Landwehr, C. L. Heitmeyer, and J. McLean. A security model for military message systems. *ACM Trans. Comput. Syst.*, 2(3):198–222, 1984.

[14] N. Li and M. V. Tripunitara. Security analysis in role-based access control. In *Proc. Ninth ACM Symposium on Access Control Models and Techniques (SACMAT)*, June 2004.

[15] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In *Proceedings of the 20th ACM Conference Object-Oriented Programming, Systems, Languages, and Applications*, pages 473–486, San Diego, California, Oct. 2005.

[16] Y. A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. Core role-based access control: Efficient implementations by transformations. In *Proceedings of the ACM SIGPLAN 2006 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Jan. 2006.

[17] M. Nyanchama and S. Osborn. The role graph model and conflict of interest. *ACM Trans. Inf. Syst. Secur.*, 2(1):3–33, 1999.

[18] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*, pages 47–63, Berlin, Germany, July 2000.

[19] A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, pages 13–22, New York, New York, 2002.

[20] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.

[21] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.