Role-based Authorization Constraints Specification

GAIL-JOON AHN
University of North Carolina at Charlotte
and
RAVI SANDHU
George Mason University

Constraints are an important aspect of role-based access control (RBAC) and are often regarded as one of the principal motivations behind RBAC. Although the importance of constraints in RBAC has been recognized for a long time, they have not received much attention. In this paper, we introduce an intuitive formal language for specifying role-based authorization constraints named *RCL 2000* including its basic elements, syntax, and semantics. We give soundness and completeness proofs for *RCL 2000* relative to a restricted form of first-order predicate logic. Also, we show how previously identified role-based authorization constraints such as separation of duty (SOD) can be expressed in our language. Moreover, we show there are other significant SOD properties which have not been previously identified in the literature. Our work shows that there are many alternate formulations of even the simplest SOD properties, with varying degree of flexibility and assurance. Our language provides us a rigorous foundation for systematic study of role-based authorization constraints.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages*; H.2.0 [**Database Management**]: General—*Security, integrity, and protection*; K.6.5 [**Management of Computing and Information System**]: Security and Protection

General Terms: Constraints, Security

Additional Key Words and Phrases: Access control models, authorization constraints, constraints specification, role-based access control

1. INTRODUCTION

Role-based access control (RBAC) has emerged as a widely accepted alternative to classical discretionary and mandatory access controls [Sandhu et al. 1996]. Several models of RBAC have been published and several commercial implementations are available. RBAC regulates the access of users to information and system resources

Authors' address: Gail-Joon Ahn, College of Information Technology, University of North Carolina at Charlotte, 9201 University City Blvd., Charlotte, NC 28223-0001; email: gahn@uncc.edu; url: www.coit.uncc.edu; Ravi Sandhu, Information and Software Engineering Department, Mail Stop 4A4, George Mason University, Fairfax, VA 22030; email: sandhu@gmu.edu; url: www.list.gmu.edu

on the basis of activities that users need to execute in the system. It requires the identification of roles in the system. A role can be defined as a set of actions and responsibilities associated with a particular working activity. Then, instead of specifying all the accesses each individual user is allowed, access authorizations on objects are specified for roles. Since roles in an organization are relatively persistent with respect to user turnover and task re-assignment, RBAC provides a powerful mechanism for reducing the complexity, cost, and potential for error in assigning permissions to users within the organization. Because roles within an organization typically have overlapping permissions RBAC models include features to establish role hierarchies, where a given role can include all of the permissions of another role. Another fundamental aspect of RBAC is authorization constraints (also simply called constraints). Although the importance of constraints in RBAC has been recognized for a long time, they have not received much attention in the research literature, while role hierarchies have been practiced and discussed at considerable length.

In this paper our focus is on constraint specifications, i.e, on how constraints can be expressed. Constraints can be expressed in natural languages, such as English, or in more formal languages. Natural language specification has the advantage of ease of comprehension by human beings, but may be prone to ambiguities. Natural language specifications do not lend themselves to the analysis of properties of the set of constraints. For example, one may want to check if there are conflicting constraints in the set of access constraints for an organization. We opted for a formal language approach to specify constraints. The advantages of a formal approach include a formal way of reasoning about constraints, a framework for identifying new types of constraints, a classification scheme for types of constraints (e.g., prohibition constraints and obligation constraints), and a basis for supporting optimization and specification techniques on sets of constraints.

To specify these constraints we introduce the specification language *RCL 2000* (for Role-based Constraints Language 2000, pronounced *Ríckle* 2000) which is the specification language for role-based authorization constraints. In this paper we describe its basic elements, syntax, and the formal foundation of *RCL 2000* including rigorous soundness and completeness proofs. *RCL 2000* is a substantial generalization of *RSL99* [Ahn and Sandhu 1999] which is the earlier version of *RCL 2000*. It encompasses obligation constraints in addition to the usual separation of duty and prohibition constraints.[1]

Who would be the user of *RCL 2000*? The first reaction might be to say the security officer or the security administrator. However, we feel there is room for a security policy designer distinct from security administrator. The policy designer has to understand organizational objectives and articulate major policy decisions

[1] A common example of prohibition constraints is separation of duty. We can consider the following statement as an example of this type of constraints: if a user is assigned to purchasing manager, he cannot be assigned to accounts payable manager and vice versa. This statement requires that the same individual cannot be assigned to both roles which are declared mutually exclusive. We identify another class of constraints called obligation constraints. In [Sandhu 1996], there is a constraint which requires that certain roles should be simultaneously active in the same session. There is another constraints which requires a user to have certain combinations of roles in user-role assignment. We classify such constraints as obligation constraints.

to support these objectives. The security officer or security administrator is more concerned with day to day operations. Policy in the large is specified by the security policy designer and the actions of the security administrator should be subject to this policy. Thus policy in the large might stipulate what is the meaning of conflicting roles and what roles are in conflict. For example, the meaning of conflicting roles for a given organization might be that no users other than senior executives can belong to two conflicting roles. For another organization the meaning might be that no one, however senior, may belong to two conflicting roles. In another context we may want both these interpretations to coexist. So we have a notion of weak conflict (former case) and strong conflict (latter case), applied to different roles sets. *RCL 2000* is also useful for security researchers to think and reason about role-based authorization constraints.

The rest of this paper is organized as follows. In section 2 we describes the formal language *RCL 2000* including basic elements and syntax. In section 3 we describe its formal semantics including soundness and completeness proofs. Section 4 shows the expressive power of *RCL 2000*. Section 5 concludes this paper.

2. ROLE-BASED CONSTRAINTS LANGUAGE (*RCL 2000*)

RCL 2000 is defined in context of RBAC96 which is a well-known family of models for RBAC [Sandhu et al. 1996]. This model has become a widely-cited authoritative reference and is the basis of a standard currently under development by the National Institute of Standards and Technology [Sandhu et al. 2000]. Here we use a slightly augmented form of RBAC96 illustrated in figure 1. We decompose permissions into operations and objects to enable formulation of certain forms of constraints. Also in figure 1 we drop the administrative roles of RBAC96 since they are not germane to *RCL 2000*.

Intuitively, a user is a human being or an autonomous agent, a role is a job function or job title within an organization with some associated semantics regarding the authority and responsibility conferred on a member of the role, and a permission is an approval of a particular mode of access (operation) to one or more objects in the system. Roles are organized in a partial order or hierarchy, so that a senior role inherits permissions from junior roles, but not vice versa. A user can be a member of many roles and a role can have many users. Similarly, a role can have many permissions and the same permission can be assigned to many roles. Each session relates one user to possibly many roles. Intuitively, a user establishes a session (e.g., by signing on to the system) during which the user activates some subset of roles that he or she is a member of. The permissions available to the users are the union of permissions from all roles activated in that session. Each session is associated with a single user. This association remains constant for the life of a session. A user may have multiple sessions open at the same time, each in a different window on the workstation screen, for instance. Each session may have a different combination of active roles. The concept of a session equates to the traditional notion of a subject in access control. A subject is a unit of access control, and a user may have multiple subjects (or sessions) with different permissions (or roles) active at the same time. RBAC96 does not define constraints formally.

Constraints are an important aspect of role-based access control and are a powerful mechanism for laying out higher level organizational policy. The construc-

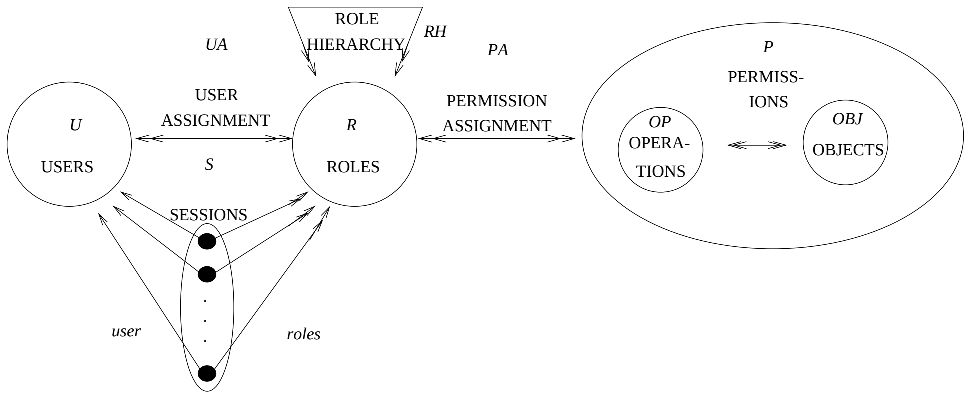


Fig. 1. Basic Elements and System Functions : from RBAC96 Model

tions of [Sandhu 1996; Sandhu and Munawer 1998] clearly demonstrate the strong connection between constraints and policy in RBAC systems. The importance of flexible constraints to support emerging applications has been recently discussed by Jaeger [Jaeger 1999]. Consequently, the specification of constraints needs to be considered. To date, this topic has not received much formal attention in context of role-based access control. A notable exception is the work of Giuri and Iglio [Giuri and Iglio 1996] who defined a formal model for constraints on role-activation. *RCL 2000* considers all aspects of role-based constraints, not just those applying to role activation. Another notable exception is the work of Gligor et al [Gligor et al. 1998] who formalize separation of duty constraints enumerated informally by Simon and Zurko [Simon and Zurko 1997]. *RCL 2000* goes beyond separation of duty to include obligation constraints [Ahn 2000] such as used in the constructions of [Sandhu 1996; Osborn et al. 2000] for simulating mandatory and discretionary access controls in RBAC.[2]

One of our central claims is that it is futile to try to enumerate all interesting and practically useful constraints because there are too many possibilities and variations. Instead, we should pursue an intuitively simple yet rigorous language for specifying constraints such as *RCL 2000*. The expressive power of *RCL 2000* is demonstrated in section 4, where it shown that many constraints previously identified in the RBAC literature and many new ones can be conveniently formulated in *RCL 2000*.

2.1 Basic Components

The basic elements and system functions on which *RCL 2000* is based are defined in figure 2. Figure 1 shows the RBAC96 model which is the context for these definitions. *RCL 2000* has six entity sets called users (`U`), roles (`R`), objects (`OBJ`), oper-

[2]Intuitively, *Prohibition Constraints* are constrains that forbid the RBAC component from doing (or being) something which is not allowed to do (or be). Most of SOD constraints are included in this class of constraints. And *Obligation Constraints* are constraints that force the RBAC component to do (or be) something.

—$\mathtt{U}$ = a set of users, $\{u_1, ..., u_n\}$.

—R= a set of roles, $\{r_1, ..., r_m\}$.

—$\mathtt{OP}$ = a set of operations, $\{op_1, ..., op_o\}$.

—$\mathtt{OBJ}$ = a set of objects, $\{obj_1, ..., obj_r\}$.

—$\mathtt{P} = \mathtt{OP} \times \mathtt{OBJ}$, a set of permissions, $\{p_1, ..., p_q\}$.

—$\mathtt{S}$ = a set of sessions, $\{s_1, ..., s_r\}$.

—$\mathtt{RH} \subseteq \mathtt{R} \times \mathtt{R}$ is a partial order on $\mathtt{R}$ called the role hierarchy or role dominance relation, written as $\preceq$.

—$\mathtt{UA} \subseteq \mathtt{U} \times \mathtt{R}$, a many-to-many user-to-role assignment relation.

—$\mathtt{PA} \subseteq \mathtt{P} \times \mathtt{R} = \mathtt{OP} \times \mathtt{OBJ} \times \mathtt{R}$, a many-to-many permission-to-role assignment relation.

—$\mathtt{user} : \mathtt{S} \rightarrow \mathtt{U}$, a function mapping each session s_i to the single user.
$\mathtt{user} : \mathtt{R} \rightarrow 2^{\mathtt{U}}$, a function mapping each role r_i to a set of users.

—$\mathtt{roles} : \mathtt{U} \cup \mathtt{P} \cup \mathtt{S} \rightarrow 2^{\mathtt{R}}$, a function mapping the set $\mathtt{U}$, $\mathtt{P}$, and $\mathtt{S}$ to a set of roles $\mathtt{R}$.
$\mathtt{roles}^* : \mathtt{U} \cup \mathtt{P} \cup \mathtt{S} \rightarrow 2^{\mathtt{R}}$, extends $\mathtt{roles}$ in presence of role hierarchy.

$\mathtt{roles}(u_i) = \{r \in \mathtt{R} \mid (u_i, r) \in \mathtt{UA}\}$

$\mathtt{roles}^*(u_i) = \{r \in \mathtt{R} \mid (\exists r' \succeq r)[(u_i, r') \in \mathtt{UA}]\}$

$\mathtt{roles}(p_i) = \{r \in \mathtt{R} \mid (p_i, r) \in \mathtt{PA}\}$

$\mathtt{roles}^*(p_i) = \{r \in \mathtt{R} \mid (\exists r' \preceq r)[(p_i, r') \in \mathtt{PA}]\}$

$\mathtt{roles}(s_i) \subseteq \{r \in \mathtt{R} \mid (\mathtt{sessions}\ (s_i), r) \in \mathtt{UA}\}$

$\mathtt{roles}^*(s_i) = \{r \in \mathtt{R} \mid (\exists r' \succeq r)[r' \in \mathtt{roles}(s_i)]\}$

—$\mathtt{sessions} : \mathtt{U} \rightarrow 2^{\mathtt{S}}$, a function mapping each user u_i to a set of sessions.

—$\mathtt{permissions} : \mathtt{R} \rightarrow 2^{\mathtt{P}}$, a function mapping each role r_i to a set of permissions.
$\mathtt{permissions}^* : \mathtt{R} \rightarrow 2^{\mathtt{P}}$, extends $\mathtt{permissions}$ in presence of role hierarchy.

$\mathtt{permissions}(r_i) = \{p \in \mathtt{P} \mid (p, r_i) \in \mathtt{PA}\}$

$\mathtt{permissions}^*(r_i) = \{p \in \mathtt{P} \mid (\exists r \preceq r_i)[(p, r_i) \in \mathtt{PA}]\}$

—$\mathtt{operations} : \mathtt{R} \times \mathtt{OBJ} \rightarrow 2^{\mathtt{OP}}$, a function mapping each role r_i and object obj_i to a set of operations.

$\mathtt{operations}(r_i, obj_i) = \{op \in \mathtt{OP} \mid (op, obj_i, r_i) \in \mathtt{PA}\}$

—$\mathtt{object} : \mathtt{P} \rightarrow 2^{\mathtt{OBJ}}$, a function mapping each permission p_i to a set of objects.

Fig. 2. Basic Elements and System Functions : from the RBAC96 Model

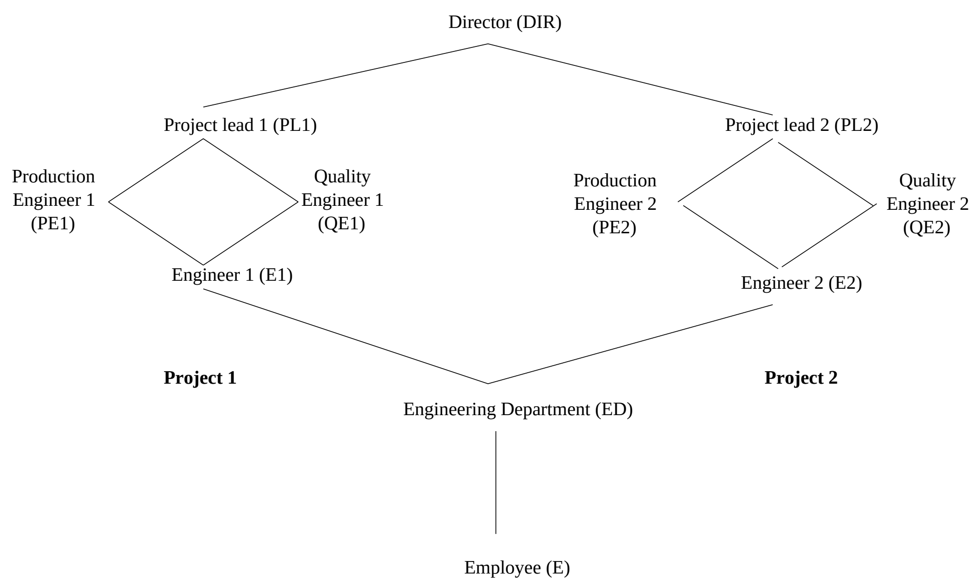


Fig. 3. Example of role hierarchies

ations (`OP`), permissions (`P`), and sessions (`S`). These are interpreted as in RBAC96 as discussed above. `OBJ` and `OP` are not in RBAC96. `OBJ` is the passive entities that contain or receive information. `OP` is an executable image of a program, which upon execution causes information flow between objects. `P` is an approval of a particular mode of operation to one or more objects in the system.

The function `user` gives us the user associated with a session and `roles` gives us the roles activated in a session. Both functions do not change during the life of a session. This is a slight simplification from RBAC96 which does allow roles in a session to change. *RCL 2000* thus builds in the constraint that roles in a session cannot change.

Hierarchies are a natural means for structuring roles to reflect an organization's lines of authority and responsibility (see Figure 3). By convention, senior roles are shown toward the top of this diagram and junior roles toward the bottom. Mathematically, these hierarchies are partial orders. A partial order is a reflexive, transitive, and antisymmetric relation, so that if $x \succ y$ then role x inherits the permissions of role y, but not vice versa. In figure 3, the junior-most role is that of Employee. The Engineering Department role is senior to Employee and thereby inherits all permissions from Employee. The Engineering Department role can have permissions besides those it inherited. Permission inheritance is transitive, for example, the Engineer1 role inherits permissions from both the Engineering Department and Employee roles. Engineer1 and Engineer2 both inherit permissions from the Engineering Department role, but each will have different permissions directly assigned to it.

The user assignment relation `UA` is a many-to-many relation between users and roles. Similarly the permission-assignment relation `PA` is a many-to-many relation

—$\texttt{CR}$ = a collection of conflicting role sets, $\{cr_1, ..., cr_s\}$, where $cr_i = \{r_i, ..., r_t\} \subseteq \texttt{R}$
—$\texttt{CP}$ = a collection of conflicting permission sets, $\{cp_1, ..., cp_u\}$, where $cp_i = \{p_i, ..., p_v\} \subseteq \texttt{P}$
—$\texttt{CU}$ = a collection of conflicting user sets, $\{cu_1, ..., cu_w\}$, where $cu_i = \{u_i, ..., u_x\} \subseteq \texttt{U}$
—**oneelement**(X) = x_i, where $x_i \in$ X
—**allother**(X) = X - {OE(X)}

Fig. 4. Basic Elements and Non-deterministic Functions: beyond RBAC96 Model

between permissions and roles. Users are authorized to use the permissions of roles to which they are assigned. This is the essence of RBAC.

The remaining functions defined in figure 2 are built from the sets, relations and functions discussed above. In particular, note that the **roles** and **user** functions can have different types of arguments so we are overloading these symbols. Also the definition of **roles*** is carefully formulated to reflect the role inheritance with respect to users and sessions going downward and with respect to permissions going upward. In other words a permission in a junior role is available to senior roles, and activation of a senior role makes available permissions of junior roles. This is a well-accepted concept in the RBAC literature and is a feature of RBAC96. Using a single symbol **roles*** simplifies our notation so long as we keep this duality of inheritance in mind.

Additional elements and system functions used in *RCL 2000* are defined in figure 4. The precise meaning of conflicting roles, permissions and users will be specified as per organizational policy in *RCL 2000*. For mutually disjoint organizational roles such as those of purchasing manager and accounts payable manager, the same individual is generally not permitted to belong to both roles. We defined these mutually disjoint roles as conflicting roles. We assume that there is a collection CR of sets of roles which have been defined to conflict.

The concept of conflicting permissions defines conflict in terms of permissions rather than roles. Thus the permission to issue purchase orders and the permission to issue payments are conflicting, irrespective of the roles to which they are assigned. We denote sets of conflicting permissions as CP. As we will see defining conflict in terms of permissions offers greater assurance than defining it in terms of roles. Conflict defined in terms of roles allows conflicting permissions to be assigned to the same role by error (or malice). Conflict defined in terms of permissions eliminates this possibility. In the real world, conflicting users should be also considered. For example, for the process of preparing and approving purchase orders, it might be company policy that members of the same family should not prepare the purchase order, and also be a user who approves that order.

RCL 2000 has two non-deterministic functions, **oneelement** and **allother**. The **oneelement**(X) function allows us to get one element x_i from set X. We usually write **oneelement** as OE. Multiple occurrences of OE(X) in a single *RCL 2000* statement all select the same element x_i from X. With **allother**(X) we can get a set by taking out one element. We usually write **allother** as AO. These two non-deterministic functions are related by context, because for any set S, $\{\texttt{OE}(S)\} \cup \texttt{AO}(S) = S$, and at the same time, neither is a deterministic function.

In order to illustrate how to use these two functions to specify role-based constraints, we take the requirement of static separation of duty (SOD) property which is the simplest variation of SOD. For simplicity assume there is no role hierarchy (otherwise replace `roles` by `roles*`).

> `Requirement`: *No user can be assigned to two conflicting roles.* In other words, conflicting roles cannot have common users. We can express this requirement as below.
> `Expression`: $\mid \texttt{roles(OE(U))} \cap \texttt{OE(CR)} \mid \leq 1$

$\texttt{OE(CR)}$ means a conflicting role set and the function $\texttt{roles(OE(U))}$ returns all roles which are assigned to a single user $\texttt{OE(U)}$. Therefore this statement ensures that a single user cannot have more than one conflicting role from the specific role set $\texttt{OE(CR)}$. We can interpret the above expression as saying that if a user has been assigned to one conflicting role, that user cannot be assigned to any other conflicting role. We can also specify this property in many different ways using *RCL 2000*, such as $\texttt{OE(OE(CR))} \in \texttt{roles(OE(U))} \Longrightarrow \texttt{AO(OE(CR))} \cap \texttt{roles(OE(U))} = \phi$ or $\texttt{user(OE(OE(CR)))} \cap \texttt{user(AO(OE(CR)))} = \phi$.

The expression $\mid \texttt{roles(OE(}\mathit{sessions}\texttt{(OE(U))))} \cap \texttt{OE(CR)} \mid \leq 1$ specifies dynamic separation of duties applied to active roles in a single session as opposed to static separation applied to user-role assignment. Dynamic separation applied to all sessions of a user is expressed by $\mid \texttt{roles(}\mathit{sessions}\texttt{(OE(U)))} \cap \texttt{OE(CR)} \mid \leq 1$.

A permission-centric formulation of separation of duty is specified as $\texttt{roles(OE(OE(CP)))} \cap \texttt{roles(AO(OE(CP)))} = \phi$. The expression $\texttt{roles(OE(OE(CP)))}$ means all roles which have a conflicting permission from say cp_i, and $\texttt{roles(AO(OE(CP)))}$ stands for all roles which have other conflicting permissions from the same conflicting permission set cp_i. This formulation leaves open the particular roles to which conflicting permissions are assigned but requires that they be distinct. This is just a sampling of the expressive power of *RCL 2000* to be discussed in section 4.

RCL 2000 system functions do not include a time or state variable in their structure. So we assume that each function considers the current time or state. For example, the `sessions` function maps a user u_i to a set of current sessions which are established by user u_i. Elimination of time or state from the language simplifies its formal semantics. *RCL 2000* thereby cannot express history or time-based constraints. It will need to be extended to incorporate time or state for this purpose.

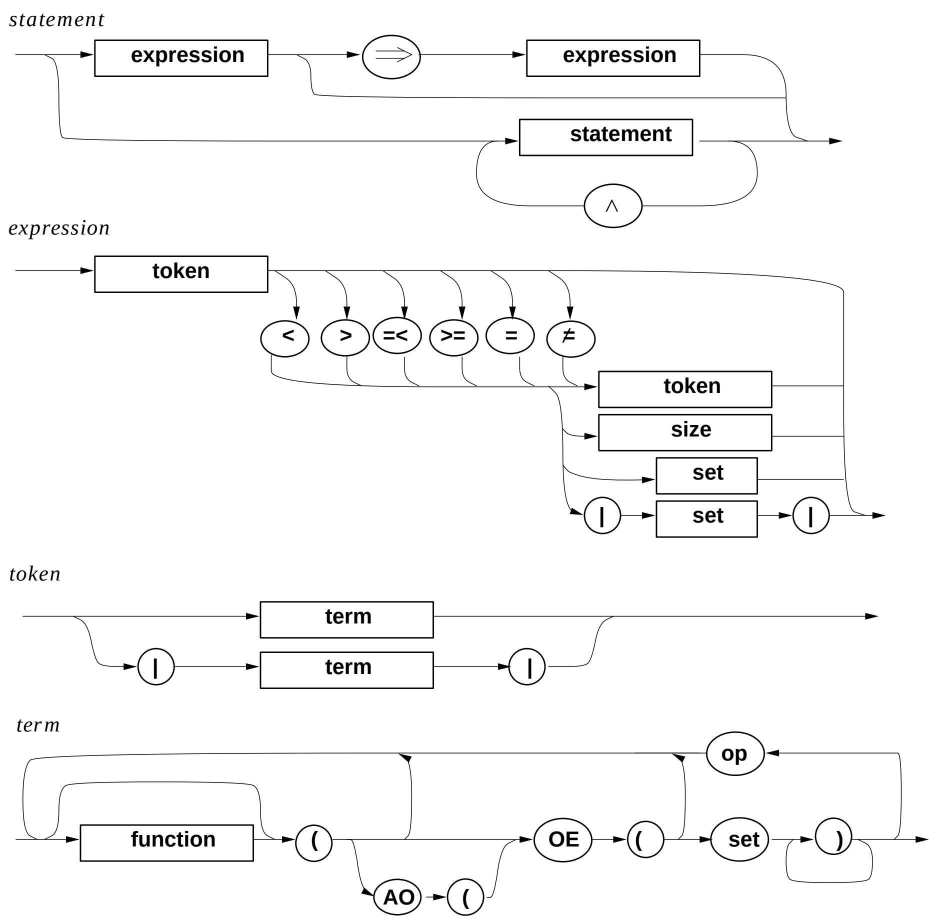
As a general notational device we have the following convention.

—For any set valued function f defined on set X,
We understand $f(\mathrm{X}) = f(x_1) \cup f(x_2) \cup ... \cup f(x_n)$, where X=$\{x_1, x_2, x_3, ..., x_n\}$.

For example, suppose we want to get all users who are assigned to a set of roles $\texttt{R} = \{Employee, Engineer1, Engineer2\}$. We can express this using the function $\texttt{user(R)}$ as equivalent to $\texttt{user(}Employee\texttt{)} \cup \texttt{user(}Engineer1\texttt{)} \cup \texttt{user(}Engineer2\texttt{)}$.

2.2 Syntax of *RCL 2000*

The syntax of *RCL 2000* is defined by the syntax diagram and grammar given in figure 5. The rules take the form of flow diagrams. The possible paths represent the possible sequence of symbols. Starting at the beginning of a diagram, a path is followed either by transferring to another diagram if a rectangle is reached or by reading a basic symbol contained in a circle. Backus Normal Form (BNF) is also



op ::= $\in$ | $\cap$ | $\cup$

$size$ ::= ϕ | 1 | ... | N

set ::= `U` | `R` | `OP` | `OBJ` | `P` | `S` | `CR` | `CP` | `CU`

$function$ ::= `user` | `roles` | `roles*` | `sessions` | `permissions` | `permissions*` | `operations` | `object` | `OE` | `AO`

Fig. 5. Syntax of Language

used to describe the grammar of *RCL 2000* as shown in the bottom of figure 5. The symbols of this form are: ::= meaning "is defined as" and | meaning "or." Figure 5 shows that *RCL 2000* statements consist of an expression possibly followed by implication ($\Longrightarrow$) and another expression. Also *RCL 2000* statements can be recursively combined with logical AND operator ($\wedge$). Each expression consists of a token followed by a comparison operator and token, size, set, or set with cardinality. Also token itself can be an expression. Each token can be just a term or a term with cardinality. Each term consists of functions and sets including set operators. The sets and system functions described earlier in section 2.1 are allowed in this syntax. Also, we denote `oneelement` and `allother` as `OE` and `AO` respectively.

3. FORMAL SEMANTICS OF *RCL 2000*

In this section we discuss the formal semantics for *RCL 2000*. We do so by identifying a restricted form of first order predicate logic called RFOPL which is exactly equivalent to *RCL 2000*. Any property written in *RCL 2000*, called a *RCL 2000* expression, can be translated to an equivalent expression in RFOPL and vice versa. The syntax of RFOPL is described later in this section. The translation algorithm, namely *Reduction*, converts a *RCL 2000* expression to an equivalent RFOPL expression. This algorithm is outlined in figure 6. *Reduction* algorithm eliminates `AO` function(s) from *RCL 2000* expression in the first step. Then we translate `OE` terms iteratively introducing universal quantifiers from left to right. If we have nested `OE` functions in the *RCL 2000* expression, translation will start from innermost `OE` terms. This algorithm translates *RCL 2000* expression to RFOPL expression in time $\mathcal{O}(\text{n})$, supposing that the number of `OE` term is n.

For example, the following expression can be converted to RFOPL expression according to the sequences below.

Example 1
$\mathtt{OE(OE(CR))} \in \mathtt{roles(OE(U))} \Longrightarrow \mathtt{AO(OE(CR))} \cap \mathtt{roles(OE(U))} = \phi$

---

(1) $\mathtt{OE(OE(CR))} \in \mathtt{roles(OE(U))} \Longrightarrow (\mathtt{OE(CR)} - \{\mathtt{OE(OE(CR))}\}) \cap \mathtt{roles(OE(U))} = \phi$

(2) $\forall\, cr \in \mathtt{CR}$: $\mathtt{OE}(cr) \in \mathtt{roles(OE(U))} \Longrightarrow (cr - \{\mathtt{OE}(cr)\}) \cap \mathtt{roles(OE(U))} = \phi$

(3) $\forall\, cr \in \mathtt{CR}$, $\forall\, r \in cr$: $r \in \mathtt{roles(OE(U))} \Longrightarrow (cr - \{r\}) \cap \mathtt{roles(OE(U))} = \phi$

(4) $\forall\, cr \in \mathtt{CR}$, $\forall\, r \in cr$, $\forall\, u \in \mathtt{U}$: $r \in \mathtt{roles}(u) \Longrightarrow (cr - \{r\}) \cap \mathtt{roles}(u) = \phi$

Example 2
$|\, \mathtt{roles(OE(U))} \cap \mathtt{OE(CR)} \,| \leq 1$

---

(1) $\forall\, u \in \mathtt{U} : |\, \mathtt{roles}(u) \cap \mathtt{OE(CR)} \,| \leq 1$

(2) $\forall\, u \in \mathtt{U}, \forall\, cr \in \mathtt{CR} : |\, \mathtt{roles}(u) \cap cr \,| \leq 1$

The resulting RFOPL expression will have the following general structure.

(1) The RFOPL expression has a (possibly empty) sequence of universal quantifiers as a left prefix, and these are the only quantifiers it can have. We call this sequence the *quantifier part.*

Reduction **Algorithm**
Input: *RCL 2000* expression ; Output: RFOPL expression

Let `Simple-OE` term be either `OE`(set), or `OE`(`function`($element$)), where
set is an element of {`U`, `R`, `OP`, `OBJ`, `P`, `S`, `CR`, `CU`, `CP`, cr, cu, cp} and
`function` is an element of {`user`, `roles`, `roles*`, `sessions`, `permissions`, `permissions*`, `operations`, `object`}

1. `AO` elimination
    replace all occurrences of `AO`($expr$) with ($expr$ - {`OE`($expr$)});
2. `OE` elimination
    While There exists `Simple-OE` term in *RCL 2000* expression
        choose `Simple-OE` term;
        call `reduction` procedure;
    End
    Procedure `reduction`
        case (i) `Simple-OE` term is `OE`(set)
            create new variable x;
            put $\forall x \in set$ to right of existing quantifier(s);
            replace all occurrences of `OE`(set) by x;
        case (ii) `Simple-OE` term is `OE`(`function`($element$))
            create new variable x;
            put $\forall x \in$ `function`($element$) to right of existing quantifier(s);
            replace all occurrences of `OE`(`function`($element$)) by x;
    End

Fig. 6. Reduction

Construction **Algorithm**
Input: RFOPL expression ; Output: *RCL 2000* expression

1. Construction *RCL 2000* expression from RFOPL expression
    While There exists a quantifier in RFOPL expression
        choose the rightmost quantifier $\forall x \in$ X;
        pick values x and X from the chosen quantifier;
        replace all occurrences of x by `OE`(X);
    End
2. Replacement of `AO`
    if there is ($expr$ - {`OE`($expr$)}) in RFOPL expression
        replace it with `AO`($expr$);

Fig. 7. Construction

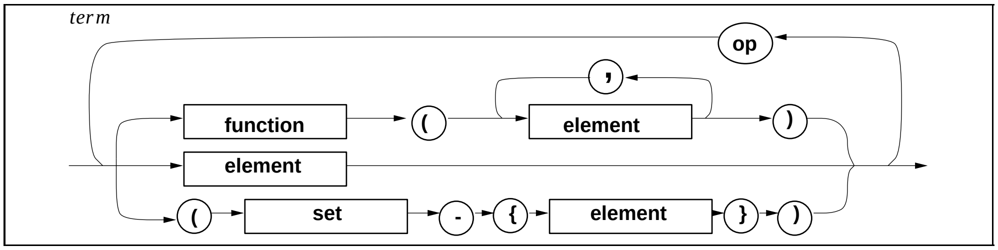


Fig. 8. Syntax of restricted FOPL expression

(2) The quantifier part will be followed by a predicate separated by a colon (:), i.e., *universal quantifier part* **:** *predicate*

(3) The predicate has no free variables or constant symbols. All variables are declared in the quantifier part, e.g., $\forall\, r \in \texttt{R}$, $\forall\, u \in \texttt{U} : r \in \texttt{roles}(u)$.

(4) The order of quantifiers is determined by the sequence of OE elimination. In some cases this order is important so as to reflect the nesting of OE terms in the *RCL 2000* expression. For example, in $\forall\, cr \in \texttt{CR}$, $\forall\, r \in cr$,$\forall\, u \in \texttt{U}$: *predicate*; the set cr, which is used in the second quantifier, must be declared in a previous quantifier as an element, such as cr in the first quantifier.

(5) *Predicate* follows most of rules in the syntax of *RCL 2000* except *term* syntax in figure 5. Figure 8 shows the syntax which *predicate* should follow to express *term*.

Because the reduction algorithm has non-deterministic choice for reduction of OE term, we may have several RFOPL expressions that are translated from a *RCL 2000* expression. As we will see in lemma 4 these expressions are logically equivalent, so it does not matter semantically which one is obtained.

Next, we discuss the algorithm *Construction* that constructs a *RCL 2000* expression from an RFOPL expression. The algorithm is described in figure 7. This algorithm repeatedly chooses the rightmost quantifier in RFOPL expression and constructs the corresponding OE term by eliminating the variable of that quantifier. After all quantifiers are eliminated the algorithm constructs AO terms according to the formal definition of AO function. The running time of the algorithm obviously depends on the number of quantifiers in RFOPL expression.

For example, the following RFOPL expression can be converted to *RCL 2000* expression according to the sequence described below.
RFOPL expression:
$\forall\, cr \in \texttt{CR}$, $\forall\, r \in cr$, $\forall\, u \in \texttt{U}$: $r \in \texttt{roles}(u) \Longrightarrow (cr - \{r\}) \cap \texttt{roles}(u)=\phi$

RCL 2000 expression :

(1) $\forall\, cr \in \texttt{CR}$, $\forall\, r \in cr$: $r \in \texttt{roles(OE(U))} \Longrightarrow (cr - \{r\}) \cap \texttt{roles(OE(U))}=\phi$

(2) $\forall\, cr \in \texttt{CR}$: $\texttt{OE}(cr) \in \texttt{roles(OE(U))} \Longrightarrow (cr - \{\texttt{OE}(cr)\}) \cap \texttt{roles(OE(U))}=\phi$

(3) $\texttt{OE(OE(CR))} \in \texttt{roles(OE(U))} \Longrightarrow (\texttt{OE(CR)} - \{\texttt{OE(OE(CR))}\}) \cap \texttt{roles(OE(U))}=\phi$

(4) $\texttt{OE(OE(CR))} \in \texttt{roles(OE(U))} \Longrightarrow \texttt{AO(OE(CR))} \cap \texttt{roles(OE(U))}=\phi$

Unlike the reduction algorithm we can observe the following lemma, where $\mathcal{C}(expr)$ denotes the *RCL 2000* expression constructed by *Construction* algorithm.

LEMMA 1. *$\mathcal{C}(\beta)$ always gives us the same* RCL 2000 *expression α.*

Proof: Construction algorithm always choose the rightmost quantifiers to construct *RCL 2000* expression from RFOPL expression. This procedure is deterministic. Therefore, given RFOPL expression β, we will always get the same *RCL 2000* expression α. □

We introduced two algorithms, namely *Reduction* and *Construction*, that can reduce and construct *RCL 2000* expression. Next we show the soundness and completeness of this relationship between *RCL 2000* and RFOPL expressions.

3.1 Soundness Theorem

Let us define the expressions generated during reduction and construction as intermediate expression collectively called *IE*. These expressions have mixed form of *RCL 2000* and RFOPL expressions, that is, they contain quantifiers as well as `OE` terms. Note that *RCL 2000* and RFOPL expressions are also intermediate expressions.

In order to show the soundness of *RCL 2000*, we introduce the following lemma.

LEMMA 2. *If the intermediate expression γ is derived from* RCL 2000 *expression α by reduction algorithm in k iterations then construction algorithm applied to γ will terminate in exactly k iterations.*

Proof: It is obvious that γ has k quantifiers because the reduction algorithm generates exactly one quantifier for each iteration. Now the construction algorithm eliminates exactly one quantifier per iteration, and will therefore terminate in k iterations. □

This leads to the following theorem, where $\mathcal{R}(expr)$ denotes the RFOPL expression translated by *Reduction* algorithm. We define all occurrences of same `OE` term in an intermediate expression as a distinct `OE` term.

THEOREM 1. *Given* RCL 2000 *expression α, α can be translated into RFOPL expression β. Also α can be reconstructed from β. That is, $\mathcal{C}(\mathcal{R}(\alpha)) = \alpha$.*

Proof: Let us define $\mathcal{C}^n$ as n iterations of reduction algorithm, and $\mathcal{R}^n$ as n iterations of reduction algorithm. We will prove the stronger result that $\mathcal{C}^n(R^n(\alpha))$ $= \alpha$ by induction on the number of iterations in reduction $\mathcal{R}$ (or, $\mathcal{C}$ under the result of lemma 2).

Basis: If the number of iterations n is 0, the theorem follows trivially.

Inductive Hypothesis: We assume that if n=k, this theorem is true.

Inductive Step: Consider the intermediate expression γ translated by reduction algorithm in $k+1$ iterations. Let $\gamma^{'}$ be the intermediate expression translated by reduction algorithm in the k^{th} iteration. γ differs from $\gamma^{'}$ in having an additional rightmost quantifier and one less distinct `OE` term. Applying the construction algorithm to γ, eliminates this rightmost quantifier and brings back the same `OE` term in all its occurrences. Thus the construction algorithm applied to γ gives us $\gamma^{'}$. From this intermediate expression $\gamma^{'}$, we can construct α due to the inductive hypothesis. This completes the inductive proof. □

3.2 Completeness Theorem

In order to show the completeness of *RCL 2000* relative to RFOPL, we introduce the following lemma analogous to lemma 2.

LEMMA 3. *If the intermediate expression γ is derived from RFOPL expression β by construction algorithm in k iterations then reduction algorithm applied to γ will terminate in exactly k iterations.*

Proof: It is obvious that γ has k distinct `OE` terms because the construction algorithm generates exactly one distinct `OE` term for each iteration. Now the reduction algorithm eliminates exactly one distinct `OE` term per iteration, and will therefore terminate in k iterations. □

Next we prove our earlier claim that even though the reduction algorithm is non-deterministic, all RFOPL expressions translated from the same *RCL 2000* expression will be logically equivalent. More precisely we prove the following result.

LEMMA 4. *Let α be an intermediate expression. If $\mathcal{R}(\alpha)$ gives us β_1 and β_2, $\beta_1 \neq \beta_2$ then $\beta_1 \equiv \beta_2$.*

Proof: The proof is by induction on the number n of `OE` terms in α.
Basis: If n is 0 the lemma follows trivially.
Inductive Hypothesis: We assume that if $n{=}k$, this lemma is true.
Inductive Step: Let $n = k+1$. By definition $\mathcal{R}$ reduces a simple `OE` term. Clearly the choice of variable symbol used for this term is not significant. The choice of term does not matter so long as it is a simple term. Thus all choices for reducing a simple `OE` term are equivalent. The Lemma follows by induction hypothesis. □

The final step to our desired completeness result is obtained below.

LEMMA 5. *There exists an execution of $\mathcal{R}$ such that $\mathcal{R}(\mathcal{C}(\beta)) = \beta$*

Proof: We prove the stronger result that there is an execution of $\mathcal{R}$ such that $\mathcal{R}^n(C^n(\beta)) = \beta$ by induction on the number of iterations in construction $\mathcal{C}$ (or, $\mathcal{R}$ under the result of lemma 3).
Basis: If the number of iterations n is 0, the theorem follows trivially.
Inductive Hypothesis: We assume that if $n{=}k$, this theorem is true.
Inductive Step: Consider the intermediate expression γ constructed by construction algorithm in $k+1$ iterations. Let $\gamma^{'}$ be intermediate expression after the k^{th} iteration. γ differs from $\gamma^{'}$ in having one less quantifier and one more distinct `OE` term. Applying one iteration of the reduction algorithm to γ, we can choose to eliminate this particular `OE` term and introduce the same variable in the new rightmost quantifier. This gives us $\gamma^{'}$. By inductive hypothesis from $\gamma^{'}$ there is an execution of $\mathcal{R}^k$ that will give us β. □

Putting these facts together, we obtain the theorem which shows the completeness of *RCL 2000*, relative to RFOPL.

THEOREM 2. *Given RFOPL expression β, β can be translated into* RCL 2000 *expression α. Also any $\beta^{'}$ retranslated from α is logically equivalent to β. That is, $\mathcal{R}(\mathcal{C}(\beta)) \equiv \beta^{'}$.*

Proof: Lemma 1 states that $\mathcal{C}(\beta)$ gives us a unique result. Let us call it α. Lemma 5 states there is an execution of $\mathcal{R}$ that will go back exactly to β from α. Lemma 4 states that all executions of $\mathcal{R}$ for α will give an equivalent RFOPL expression. The theorem follows. □

In this section we have given a formal semantics for *RCL 2000* and have demonstrated its soundness and completeness. Any property written in *RCL 2000* could be translated to an expression which is written in a restricted form of first order predicate logic, which we call RFOPL. During the analysis of this translation, we proved two theorems which support the soundness and completeness of the specification language *RCL 2000* and RFOPL respectively.

4. EXPRESSIVE POWER OF *RCL 2000*

In this section we demonstrate the expressive power of *RCL 2000* by showing how it can be used to express a variety of separation of duty (SOD) properties. In [Ahn 2000] it is further shown how the construction of [Sandhu 1996] and [Osborn et al. 2000] to respectively simulate mandatory and discretionary access controls in RBAC can be expressed in *RCL 2000*. As a security principle, SOD is a fundamental technique for prevention of fraud and errors, known and practiced long before the existence of computers. It is used to formulate multi-user control policies, requiring that two or more different users be responsible for the completion of a transaction or set of related transactions. The purpose of this principle is to minimize fraud by spreading the responsibility and authority for an action or task over multiple users, thereby raising the risk involved in committing a fraudulent act by requiring the involvement of more than one individual. A frequently used example is the process of preparing and approving purchase orders. If a single individual prepares and approves purchase orders, it is easy and tempting to prepare and approve a false order and pocket the money. If different users must prepare and approve orders, then committing fraud requires a conspiracy of at least two, which significantly raises the risk of disclosure and capture.

Although separation of duty is easy to motivate and understand intuitively, so far there is no formal basis for expressing this principle in computer security systems. Several definitions of SOD have been given in the literature. For the purpose of this paper we use the following definition.

> **Separation of duty** *reduces the possibility for fraud or significant errors (which can cause damage to an organization) by partitioning of tasks and associated privileges so cooperation of multiple users is required to complete sensitive tasks.*

We have the following definition for interpreting SOD in role-based environments.

> **Role-Based separation of duty** *ensures SOD requirements in role-based systems by controlling membership in, activation of, and use of roles as well as permission assignment.*

There are several papers in the literature over the past decade which deal with separation of duty. During this period various forms of SOD have been identified. Attempts have been made to systematically categorize these definitions. Notably, Simon and Zurko [Simon and Zurko 1997] provide an informal characterization, and Gligor et al. [Gligor et al. 1998] provide a formalism of this characterization. However, this work has significant limitations. It omits important forms of SOD

Properties	Expressions
1. SSOD-CR	$\mid \texttt{roles}^*(\texttt{OE(U)}) \cap \texttt{OE(CR)} \mid \leq 1$
2. SSOD-CP	$\mid \texttt{permissions}(\texttt{roles}^*(\texttt{OE(U)})) \cap \texttt{OE(CP)} \mid \leq 1$
3. Variation of 2	$(2) \wedge \mid \texttt{permissions}^*(\texttt{OE(R)}) \cap \texttt{OE(CP)} \mid \leq 1$
4. Variation of 1	$(1) \wedge \mid \texttt{permissions}^*(\texttt{OE(R)}) \cap \texttt{OE(CP)} \mid \leq 1$ $\wedge\ \texttt{permissions}(\texttt{OE(R)}) \cap \texttt{OE(CP)} \neq \phi \Longrightarrow \texttt{OE(R)} \cap \texttt{OE(CR)} \neq \phi$
5. SSOD-CU	$(1) \wedge \mid \texttt{user}(\texttt{OE(CR)}) \cap \texttt{OE(CU)} \mid \leq 1$
6. Yet another variation	$(4) \wedge (5)$

Table 1. Static Separation of Duty

including session-based dynamic SOD needed for simulating lattice-based access control and Chinese Walls in RBAC [Sandhu 1993; Sandhu 1996]. It also does not deal with SOD in the presence of role hierarchies. Moreover, as will see, there are additional SOD properties that have not been identified in the previous literature.

Here, we take a different approach to understand SOD. Rather than simply enumerating different kinds of SOD we show how *RCL 2000* can be used to specify the various separation of duty properties.

4.1 Static SOD

Static SOD (SSOD) is the simplest variation of SOD. In table 1 we show our expression of several forms of SSOD. These include new forms of SSOD which have not previously been identified in the literature. This demonstrates how *RCL 2000* helps us in understanding SOD and discovering new basic forms of it.

Property 1 is the most straightforward property. The SSOD requirement is that no user should be assigned to two roles which are conflicting each other. In other words, it means that conflicting roles cannot have common users. *RCL 2000* can clearly express this property. This property is the classic formulation of SSOD which is identified by several papers including [Gligor et al. 1998; Kuhn 1997; Sandhu et al. 1996]. It is a role-centric property.

Property 2 follows the same intuition as property 1, but is permission-centric. Property 2 says that a user can have at most one conflicting permission acquired through roles assigned to the user. Property 2 is a stronger formulation than property 1 which prevents mistakes in role-permission assignment. This kind of property has not been previously mentioned in the literature. *RCL 2000* helps us discover such omissions in previous work. In retrospect property 2 is an "obvious property" but there is no mention of this property in over a decade of SOD literature. Even though property 2 allows more flexibility in role-permission assignment since the conflicting roles are not predefined, it can also generate roles which cannot be used at all. For example, two conflicting permissions can be assigned to a role. Property 2 simply requires that no user can be assigned to such a role or any role senior to it, which makes that role quite useless. Thus property 2 prevents certain kinds of mistakes in role-permissions but tolerates others.

Property 3 eliminates the possibility of useless roles with an extra condition, $|\texttt{permissions}^*(\texttt{OE(R)}) \cap \texttt{OE(CP)} \mid \leq 1$. This condition ensures that each role can have at most one conflicting permission without consideration of user-role assignment.

With this new condition, we can extend property 1 in presence of conflicting permissions as property 4. In property 4 we have another additional condition that conflicting permissions can only be assigned to conflicting roles. In other words, non-conflicting roles cannot have conflicting permissions. The net effect is that a user can have at most one conflicting permission via roles assigned to the user.

Property 4 can be viewed as a reformulation of property 3 in a role-centric manner. Property 3 does not stipulate a concept of conflicting roles. However, we can interpret conflicting roles to be those that happen to have conflicting permissions assigned to them. Thus for every cp_i we can define $cr_i = \{r \in \mathtt{R} \mid cp_i \cap \mathtt{permissions}(r) \neq \phi\}$. With this interpretation, properties 3 and 4 are essentially identical. The viewpoint of property 3 is that conflicting permissions get assigned to distinct roles which thereby become conflicting, and therefore cannot assigned to the same user. Which roles are deemed conflicting is not determined a priori but is a side-effect of permission-role assignment. The viewpoint of property 4 is that conflicting roles are designated in advance and conflicting permissions must be restricted to conflicting roles. These properties have different consequences on how roles get designed and managed but essentially achieve the same objective with respect to separation of conflicting permissions. Both properties achieve this goal with much higher assurance than property 1. Property 2 achieves this goal with similar high assurance but allows for the possibility of useless roles. Thus, even in the simple situation of static SOD, we have a number of alternative formulations offering different degrees of assurance and flexibility.

Property 5 is a very different property and is also new to the literature. With a notion of conflicting users, we identify new forms of SSOD. Property 5 says that two conflicting users cannot be assigned to roles in the same conflicting role set. This property is useful because it is much easier to commit fraud if two conflicting users can have different conflicting roles in the same conflicting role set. This property prevents this kind of situation in role-based systems. A collection of conflicting users is less trustworthy than a collection of non-conflicting users, and therefore should not be mixed up in the same conflicting role set. This property has not been previously identified in the literature.

We also identify a composite property which includes conflicting users, roles and permissions. Property 6 combines property 4 and 5 so that conflicting users cannot have conflicting roles from the same conflict set while assuring that conflicting roles have at most one conflicting permission from each conflicting permission sets. This property supports SSOD in user-role and role-permission assignment with respect to conflicting users, roles, and permissions.

4.2 Dynamic SOD

In RBAC systems, a dynamic SOD (DSOD) property with respect to the roles activated by the users requires that no user can activate two conflicting roles. In other words, conflicting roles may have common users but users can not simultaneously activate roles which are conflicting each other. From this requirement we can express user-based Dynamic SOD as property 1. We can also identify a Session-based Dynamic SOD property which can apply to the single session as property 2. We can also consider these properties with conflicting users such as property 1-1 and 2-1. Additional analysis of dynamic SOD properties based on conflicting permissions can also be pursued as was done for static SOD.

Properties	Expressions
1. User-based DSOD	$\mid$ roles*(sessions(OE(U))) $\cap$ OE(CR) $\mid \leq 1$
1-1. User-based DSOD with CU	$\mid$ roles*(sessions(OE(OE(CU)))) $\cap$ OE(CR) $\mid \leq 1$
2. Session-based DSOD	$\mid$ roles*(OE(sessions(OE(U)))) $\cap$ OE(CR) $\mid \leq 1$
2-1. Session-based DSOD with CU	$\mid$ roles*(OE(sessions(OE(OE(CU))))) $\cap$ OE(CR) $\mid \leq 1$

Table 2. Dynamic Separation of Duty

5. CONCLUSION

In this paper we have described the specification language *RCL 2000*. This language is built on RBAC96 components and has two non-deterministic functions OE and AO. We have given a formal syntax and semantics for *RCL 2000* and have demonstrated its soundness and completeness. Any property written in *RCL 2000* could be translated to an expression which is written in a restricted form of first order predicate logic, which we call RFOPL. During the analysis of this translation, we proved two theorems which support the soundness and completeness of the specification language *RCL 2000* and RFOPL respectively.

RCL 2000 provides us a foundation for studying role-based authorization constraints. It is more natural and intuitive than RFOPL. The OE and AO operators were intuitively motivated by Chen and Sandhu [Chen and Sandhu 1995] and formalized in *RCL 2000*. They provide a viable alternative to reasoning in terms of long strings of universal quantifiers. Also the same *RCL 2000* expression has multiple but equivalent RFOPL formulations indicating that there is a unifying concept in *RCL 2000*.

There is room for much additional work with *RCL 2000* and similar specification languages. The language can be extended by introducing time and state. Analysis of *RCL 2000* specifications and their composition can be studied. The efficient enforcement of these constraints can also be investigated. A user-friendly front-end to the language can be developed so that it can be realistically used by security policy designers.

Acknowledgement

This work is partially supported by the National Science Foundation and the National Security Agency.

REFERENCES

Ahn, G.-J. January 2000. *The RCL 2000 Language for Specifying Role-Based Authorization Constraints.* PhD Thesis, George Mason University (Adviser: Ravi Sandhu).

Ahn, G.-J. and Sandhu, R. 1999. The RSL99 language for role-based separation of duty constraints. In *Proceedings of 4th ACM Workshop on Role-Based Access Control* (Fairfax, VA, October 28-29 1999), pp. 43–54. ACM.

Chen, F. and Sandhu, R. 1995. Constraints for role based access control. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control* (Gaithersburg, MD, November 30-December 1 1995), pp. 39–46. ACM.

Giuri, L. and Iglio, P. 1996. A formal model for role-based access control with constraints. In *Proceedings of 9th IEEE Computer Security Foundations Workshop* (Kenmare, Ireland, June 1996), pp. 136–145. IEEE.

GLIGOR, V. D., GAVRILA, S. I., AND FERRAIOLO, D. 1998. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of IEEE Symposium on Research in Security and Privacy* (Oakland, CA, May 1998), pp. 172–183. IEEE.

JAEGER, T. 1999. On the increasing importance of constraints. In *Proceedings of 4th ACM Workshop on Role-Based Access Control* (Fairfax, VA, October 28-29 1999), pp. 33–42. ACM.

KUHN, D. R. 1997. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *Proceedings of 2nd ACM Workshop on Role-Based Access Control* (Fairfax, VA, November 6-7 1997), pp. 23–30. ACM.

OSBORN, S., SANDHU, R., AND MUNAWER, Q. 2000. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security 3*, 2 (May).

SANDHU, R. 1993. Lattice-based access control models. *IEEE Computer 26*, 11 (November), 9–19.

SANDHU, R., FERRAIOLO, D., AND KUHN, R. 2000. The nist model for role-based access control: Towards a unified standard. In *Proceedings of 5th ACM Workshop on Role-Based Access Control* (Berlin, Germany, July 2000). ACM.

SANDHU, R. AND MUNAWER, Q. 1998. How to do discretionary access control using roles. In *Proceedings of 3rd ACM Workshop on Role-Based Access Control* (Fairfax, VA, October 22-23 1998), pp. 47–54. ACM.

SANDHU, R. S. 1996. Role hierarchies and constraints for lattice-based access controls. In E. BERTINO Ed., *Proc. Fourth European Symposium on Research in Computer Security*. Rome, Italy: Springer-Verlag. Published as *Lecture Notes in Computer Science, Computer Security–ESORICS96*.

SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. 1996. Role-based access control models. *IEEE Computer 29*, 2 (February), 38–47.

SIMON, R. AND ZURKO, M. 1997. Separation of duty in role-based environments. In *Proceedings of 10th IEEE Computer Security Foundations Workshop* (Rockport, Mass., June 1997), pp. 183–194. ACM.