Journal of Network and Computer Applications (2001) **24**, 87–100
doi:10.1006/jnca.2000.0125, available online at http://www.idealibrary.com on IDEAL®

CONUGA: Constrained User-Group Assignment

Gail-Joon Ahn[*] **and Kwangjo Kim**[†]

[*]*Computer Science Department, University of North Carolina at Charlotte, 9201 University City Blvd., Charlotte, NC 28223, U.S.A. E-mail: gahn@uncc.edu*
[†]*School of Engineering, Information and Communications University, 58-4 Hwaam-dong, Yusong-ku, Taejon, 305–348, Korea (ROK) E-mail: kkj@icu.ac.kr*

(Received 16 November 1999; accepted 9 February 2000)

In role-based access control (RBAC), permissions are associated with roles and users are made members of appropriate roles, thereby acquiring the roles' permissions. The principal motivation behind RBAC is to simplify administration. In this paper, we investigate one aspect of RBAC administration concerning assignment of users to roles. We introduce a constrained user-role assignment model, called CONUGA (CONstrained User-Group Assignment) and describe its implementation in the Windows NT system. Rather than set user and file rights individually for each and every user, the administrator can give rights to various groups, then place users within those groups in Windows NT. Each user within a group inherits the rights associated with that group. We demonstrate how to extend the Windows NT group mechanism supporting our model that is useful in managing group-based access control. © 2001 Academic Press

1. Introduction

Role-based access control (RBAC) has recently received considerable attention as a promising alternative to traditional discretionary and mandatory access controls (see, for example, [1–8]). In RBAC, permissions are associated with roles, and users are made members of appropriate roles, thereby acquiring the roles' permissions. This greatly simplifies management of permissions. Roles are created for the various job functions in an organization and users are assigned to roles based on their responsibilities and qualifications. Users can be easily reassigned from one role to another. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from roles as needed.

In large enterprise-wide systems the number of roles can be in the hundreds or thousands, and users can be in the tens or hundreds of thousands, may be even millions. Managing these roles and users, and their interrelationships is a formidable task that often is highly centralized and delegated to a small team of security administrators. User-role assignment is likely to be the first administrative function that is decentralized and delegated to users rather

[*] Corresponding author

than system administrators. Assigning people to tasks is a normal managerial function. Assigning users to roles should be a natural part of assigning users to tasks. Empowering managers to do this routinely is one way of making security an enabling user-friendly technology rather than an intrusive and cumbersome nuisance as it all too often turns out to be. A manager who can assign a user to perform certain tasks should not have to ask someone else to enrol this user in appropriate roles. This should happen transparently and conveniently.

Sandhu and Bhamidipati [9] recently introduced the URA97 model for decentralized administration of user-role membership (URA97 stands for user-role assignment 1997). They simply focused on user-role assignment without consideration of the important constraints such as separation of duty (SOD) constraints. An example of such a policy may be ‘the patent submitted to a patent authorization agency can be reviewed only by a member of its patent review committee.’ This simple role-based access control may not be adequate for expressing many business policies. An example of such policy is ‘none of the applicants of the patent is eligible to review a patent, even though the applicant is a patent review committee member.’ These policies, also known as SOD constraints should be dealt with user-role assignment.

Constraints are an important aspect of 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. Ahn and Sandhu [10] recently showed that role-based authorization constraints such as separation of duty (SOD) can be expressed by the specification language called RCL 2000. We use the concept of static separation of duty (SSOD) borrowed from their work. The central contribution of this article is to describe how we can achieve this kind of constraints during user-role assignment named constrained user-role assignment as an extension of URA97.

A user-role assignment model can also be used for managing user-group assignment and therefore has applicability beyond RBAC. The notion of a role is similar to that of a group, particularly when we focus on the issue of user-role or user-group membership. For our purpose in this paper we can treat the concepts of roles and groups as essentially identical. The difference between roles and groups was hotly debated at the ACM Workshop. There exists the consensus that a group is a named collection of users (and possibly other groups). Groups serve as a convenient shorthand notation for collections of users and that is the main motivation for introducing them. Roles are similar to groups in that they can serve as a shorthand for collections of users, but they go beyond groups in also serving as a shorthand for a collection of permissions. Assigning users to roles or users to groups are therefore essentially the same function.

The rest of the paper is organized as follows. In Section 2, we review the URA97 grant model. Section 3 describes constrained user-group assignments. In Section 4 we discuss implementation details. Section 5 concludes the paper.

2. Overview of the URA97 model

This section reviews URA97. We often use the term group as an identical notion of role. Our description of URA97 is informal and intuitive, but a formal statement of URA97 is given in [9]. In this section we simply give a quick overview of the grant model which is dealing with granting a user membership in a group.

2.1 *User-group grant model*

URA97 imposes restrictions on which users can be added to a group by whom. URA97 requires a hierarchy of groups (such as in Figure 1) and a hierarchy of administrative groups (such as in Figure 2). The set of groups and administrative groups are required to be disjoint. Senior groups are shown toward the top and junior ones toward the bottom. Senior groups inherit permissions from junior groups. We write $x > y$ to denote x is senior to y with obvious extension to

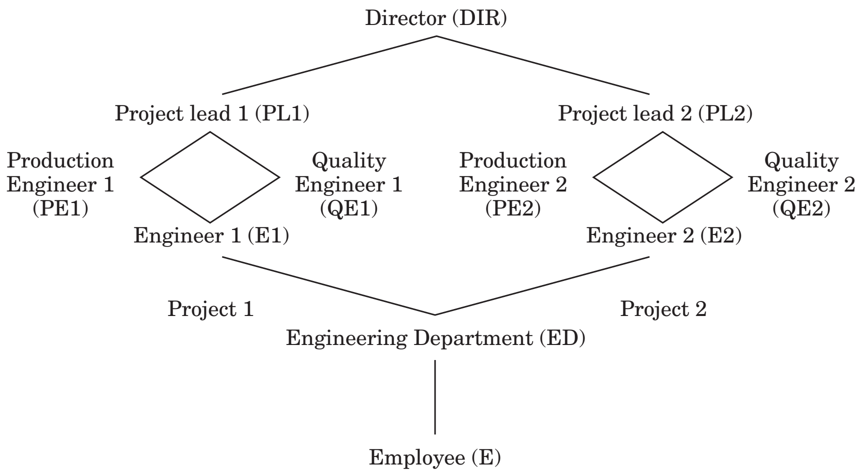


Figure 1. An example group hierarchy.

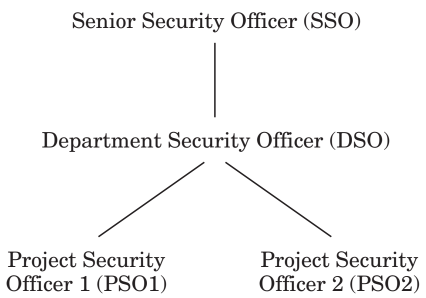

Senior Security Officer (SSO)
Department Security Officer (DSO)
Project Security Officer 1 (PSO1)
Project Security Officer 2 (PSO2)

Figure 2. An example administrative group hierarchy.

$x \geq y$. The notion of prerequisite condition is a key part of URA97. User-group assignment is authorized in URA97 by the *can-assign* relation.

Definition 1: A *prerequisite condition* is a boolean expression using the usual $\wedge$ and $\vee$ operators on terms of the form x and $\bar{x}$ where x is a regular role (i.e., $x \in R$). A prerequisite condition is evaluated for a user u by interpreting x to be true if $(\exists x' \geq x)(u, x') \in UA$ and $\bar{x}$ to be true if $(\forall x' \geq x)(u, x') \notin UA$. For a given set of roles R let CPR denote all possible prerequisite conditions that can be formed using the roles in R. □

Definition 2: The URA97 model controls user-role assignment by means of the relation *can-assign* $\subseteq AR \times CPR \times 2^R$. □

The meaning of *can-assign* $(x, y, \{a, b, c\})$ is that a member of the administrative role x (or a member of an administrative role that is senior to x) can assign a user whose current membership, or non-membership, in regular roles satisfies the prerequisite condition y to be a member of regular roles a, b or c.

2.1.1 *Range notation*. URA97 also defines *can-assign* by identifying a range within the role hierarchy by means of the familiar closed and open interval notation.

Definition 3: Role sets are specified in the URA97 model by the notation below:

$$[x, y] = \{r \in R \mid x \geq r \wedge r \geq y\}$$
$$(x, y] = \{r \in R \mid x > r \wedge r \geq y\}$$
$$[x, y) = \{r \in R \mid x \geq r \wedge r > y\}$$
$$(x, y) = \{r \in R \mid x > r \wedge r > y\}.$$

□

2.1.2 *Prerequisite conditions*. Let us consider the PSO1 tuples (analysis for PSO2 is exactly similar). The first tuple authorizes PSO1 to assign users with prerequisite role ED into E1. The second one authorizes PSO1 to assign users with prerequisite condition $\mathrm{ED} \wedge \overline{\mathrm{QE1}}$ to PE1. Similarly, the third tuple authorizes PSO1 to assign users with prerequisite condition $\mathrm{ED} \wedge \overline{\mathrm{PE1}}$ to QE1. Taken together the second and third tuples authorize PSO1 to put a user who is a member of ED into one but not both of PE1 and QE1. This illustrates how mutually exclusive roles can be enforced by URA97. PE1 and QE1 are mutually exclusive with respect to the power of PSO1. However, for the DSO and SSO these are not mutually exclusive. Hence, the notion of mutual exclusion is a relative one in URA97. The fourth tuple authorizes PSO1 to put a user who is a member of both PE1 and QE1 into PL1. Of course, a user could have become a member of both PE1 and QE1 only by actions of a more powerful administrator than PSO1. Table 1 is an example of a *can-assign* relation table.

Table 1. *Example of can-assign*

Admin. role	Prereq. condition	Role range
PSO1	ED	[E1, E1]
PSO1	$ED \wedge \overline{QE1}$	[PE1, PE1]
PSO1	$ED \wedge \overline{PE1}$	[QE1, QE1]
PSO1	$PE1 \wedge QE1$	[PL1, PL1]
PSO2	ED	[E2, E2]
PSO2	$ED \wedge \overline{QE2}$	[PE2, PE2]
PSO2	$ED \wedge \overline{PE2}$	[QE2, QE2]
PSO2	$PE2 \wedge QE2$	[PL2, PL2]
DSO	ED	(ED, DIR)
SSO	E	[ED, ED]
SSO	ED	[ED, DIR]

3. Constrained User-Group Assignment

RBAC regulates the access of users to information and system resources on the basis of activities that users need to execute in the system. Instead of specifying all the access each user is allowed to execute, access authorizations on objects are specified for roles. RBAC provides a powerful mechanism for reducing the complexity, cost, and potential for error of assigning users permissions 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). This issue has not received much attention in the research literature, while role hierarchies have been practiced and discussed at considerable length. Constraints are an important aspect of role-based access control and are a powerful mechanism for laying out higher level organizational policy. Most of role-based constraints work has focused on separation of duty constraints which is a foundational principle in computer security.

As a security principle, separation of duty (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. 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.

Separation of duty constraints can be determined by the assignment of individuals to roles at user-assignment time. Consider the case of initiating and authorizing payments. The separation of duty constraints could require that no individual who can serve as payment initiator could also serve as payment authorizer. This could be implemented by ensuring that no one who can perform the initiator role could also be assigned to the authorizer role. This static separation of duty can apply to the user-role assignment. Therefore, we adapt the grant model in URA97. User u can be explicitly assigned to role r_i where $(u, r_i) \in UA$. Also, user u can be implicitly assigned to role r_j where $(\exists r_i \preceq r_j)[(u, r_j) \in UA]$. Let *CR* be a set of roles which are needed to be in static SOD. *CR* is said to be a conflicting role set. The static SOD requirement is that the same user cannot be assigned explicitly or implicitly to more than one role in *CR*.

We can enforce static SOD as we check each assignment task with a given *CR*. This enforcement framework is based on the basic conceptual structure illustrated in Figure 3. We have AT-SET (assignment time set) table which includes SOD sets used to enforce SOD requirements at assignment time. The example of AT-SET table with *CR* is described below. This table tells us that roles *pay-initiator* and *pay-authorizer* are conflicting each other so a user cannot be assigned to

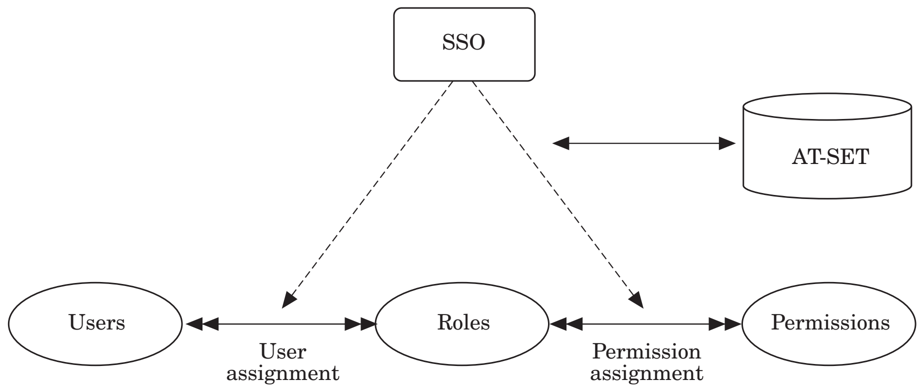


Figure 3. Constrained user assignment.

both roles.

SET-NAME	ELEMENT
CR_1	$\{pay-initiator, pay-authorizer\}$

Whenever System Security Officer (SSO) does assignment tasks, each assignment task should be checked with AT-SET table and satisfy the constraints in the table. Figure 4 describes an algorithm which achieves desired behavior of CONUGA. There are two procedures called `Membership` and `JuniorList`. Membership procedure allows us to have all assigned roles to a user and `JuniorList` procedure returns all junior roles to a specified role by walking down the hierarchy. This grant algorithm checks *can-assign* table and AT-SET table to enforce constrained user assignment.

4. Implementation of CONUGA

Every account in Windows NT's user database contains a group membership list indicating which groups the account belongs to [11].* Users belonging to a group are explicitly displayed with the administrative tool. Windows NT notably lacks a facility for including one group in another. Many commercial database management systems, such as Informix, Oracle and Sybase, provide facilities for hierarchical groups (or roles). Commercial operating systems, however, provide limited facilities at best for this purpose.

Let $x > y$ signify that group x is *senior* to y, in the sense that a member of x is also automatically a member of y but not vice versa. Note that a member of x has the power of a member of y and may have additional power, hence a member of x is considered senior to a member of y. It is natural to require that seniority is a partial ordering, i.e., $>$ is irreflexive, transitive and asymmetric. The irreflexive property is obviously required since every member of x is already a member of x. Transitivity is certainly an intuitive assumption and perhaps even inevitable. After all, if $x > y$ and $y > z$ then a member of x is a member of y and so should also be a member of z. The asymmetric requirement eliminates redundancy by excluding groups which would otherwise be equivalent. We write $x > y$ to mean $x > y$ or $x = y$. If x is senior to y we also say that y is *junior* to x. For convenience we use the term hierarchy to mean a partial order. As we have mentioned, Windows NT does not have the notion of hierarchy between groups.† To maintain the group hierarchy we use the file `grouphr.txt` to store the children and parents

* For simplicity, we just focused on global groups in Windows NT.
† Like Windows NT, group policy in Windows 2000 permits administrators to define customized rules about virtually every facet of a user's computer environment such as security, user rights, desktop settings, applications, and resources, minimizing the likelihood of misconfiguration. Group policy in Windows 2000, however, works in conjunction with the Active Directory service which might be able to construct relationships between groups.

Grant algorithm

Let *invoker* be an initiator of user-role assignment and let *assign_DB* have three attributes such as *assign_DB*.`admin`, *assign_DB*.`cond` and *assign_DB*.`range` to construct a table as shown in Table 1.

```
$invoker\_role\_set \leftarrow$ Membership($invoker$)
$target\_role \leftarrow$ role to be assigned
$user \leftarrow$ user to which $target\_role$ is assigned
$assign\_DB \leftarrow can\text{-}assign$ relation table
$CR\_set \leftarrow$ AT_SET table
$grant\_Flag \leftarrow$ false
$assign\_role\_set = \phi$

While($assign\_DB \neq$ EOF)
  if $invoker\_role\_set$ exists in $assign\_DB$.admin then
    if $target\_role$ exists in $assign\_DB$.range then
    $user\_role\_set \leftarrow$ Membership($user$);
      if $user\_role\_set$ exists in $assign\_DB$.cond then
      $grant\_Flag =$ true;
      return;
      endif
    endif
  endif
End

if $grant\_Flag =$ true then
  $assign\_role\_set \leftarrow$ JuniorList($target\_role$);
  if $assign\_role\_set \cap CR\_set = \phi$ then
   do the assignment of role in $assign\_role\_set$;
  else
   exit;
  endif
endif
```

Procedure Membership *(user)*
Take all assigned roles to a user

Procedure JuniorList *(role)*
Take all junior roles to a specified role in role-hierarchies

Figure 4. Grant algorithm in CONUGA.

Table 2. *The example group hierarchy of Figure 1*

Group name	Parent group(s)	Child group(s)
DIR	—	PL1, PL2
PL1	DIR	PE1, QE1
PL2	DIR	PE2, QE2
PE1	PL1	E1
QE1	PL1	E1
PE2	PL2	E2
QE2	PL2	E2
E1	PE1, QE1	ED
E2	PE2, QE2	ED
ED	E1, E2	E
E	ED	—

of each group. The group hierarchy of Figure 1 is represented in `grouphr.txt` as shown in Table 2. The first column gives the group name, the second column gives the (immediate) parent groups of that group, and the third column gives the (immediate) children. The null symbol '–' means that the group has no parent or child as the case may be. Using `grouphr.txt`, we can find all seniors and juniors for a group by respectively chasing the parents and children.

We say a user is an *explicit* member of a group if the user is explicitly designated as a member of the group. A user is an *implicit* member of a group if the user is an explicit member of some senior group. To simulate a group hierarchy we use information about explicit and implicit membership in `accountDB`.* If Alice belongs explicitly or implicitly to a group she will be added to that group's member list in `accountDB`. However, `accountDB` is not sufficient to distinguish the case where Alice is both an explicit and implicit member of some group from the case where she is only an implicit member of the group. For this purpose we introduce another file `explicit.txt` that keeps information about explicit membership only.

Another limitation of Windows NT groups is that membership is exclusively controlled by built-in administrator groups which does not scale gracefully to systems with large numbers of groups and users. More generally, it is possible to decentralize user-group assignment by allowing administrators to selectively delegate authority to assign certain users to certain groups. Effective decentralization of user-group assignment is one step towards making security more acceptable to end users as an enabling and empowering technology, rather than as the general nuisance it is often perceived to be [12].

We use Microsoft RPC to enforce desired behavior of CONUGA with respect to different administrative groups. The RPC mechanism uses the Windows NT

* For ease of reference, we call Windows NT's user account database `accountDB`. Windows NT's user account database is managed via User Manager Tool only by a user with administrative powers. Through this tool, new user or group can be created and security policies can be specified.

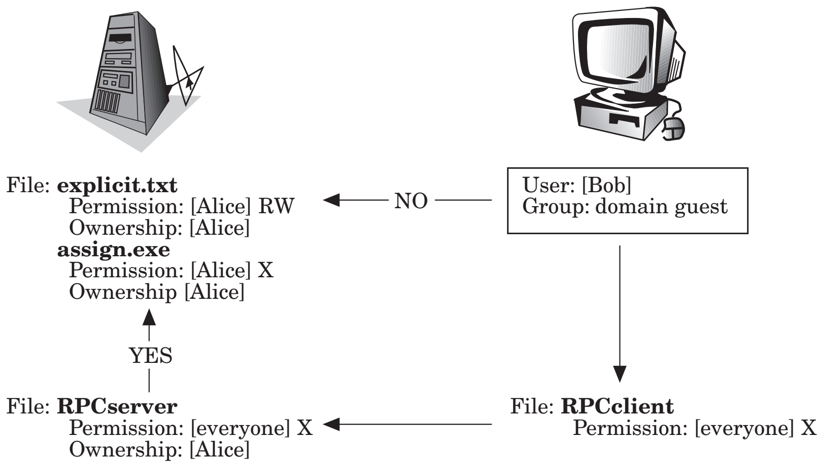


Figure 5. RPC Mechanism.

security built in as part of the operating system. The main mechanism is illustrated in Figure 5. In Figure 5, let's assume that Alice owns the RPC server program and two files `assign.exe` and `explicit.txt`. Alice can execute the executable file `assign.exe` which should refer `explicit.txt` file to accomplish its function. Alice also can read and write the text file `explicit.txt`. And Alice allows everyone to execute RPC server program by assigning such permission set to RPC server program. According to this configuration, a user (say Bob) cannot access Alice's two files. That is, no one can access `assign.exe` and `explicit.txt`, except for Alice. However, by means of using Microsoft RPC named pipes, Bob can execute `assign.exe` and access `explicit.txt`. The execution of RPC server program allows us to provide users access to `explicit.txt` but only by way of `assign.exe`. In other words `assign.exe` is a protected subsystem which runs with different permissions than the user who invokes it. When the RPC server program is executed, the effective user of the process is the owner (Alice) of the file, acquiring that user's access rights for duration of the program contained in this file. Therefore, a user who is executing RPC server program through RPC client program can invoke assign.exe and access `explicit.txt`. Thereby, a user who is working as an administrative group can read and write reference files: `explicit.txt`, `grouphr.txt` and `can-assign.txt`.* Using this feature, we can enforce desired behaviour of URA97 with constrained conditions.

Our implementation follows the basic steps shown in Figure 6. The diagram shows the data flow and the relationship between assignment processes. There is a procedure for assigning a user to a group. The procedure call is as follows.

- `assign (user, tgroup)`

* This is a table as described in Table 1.

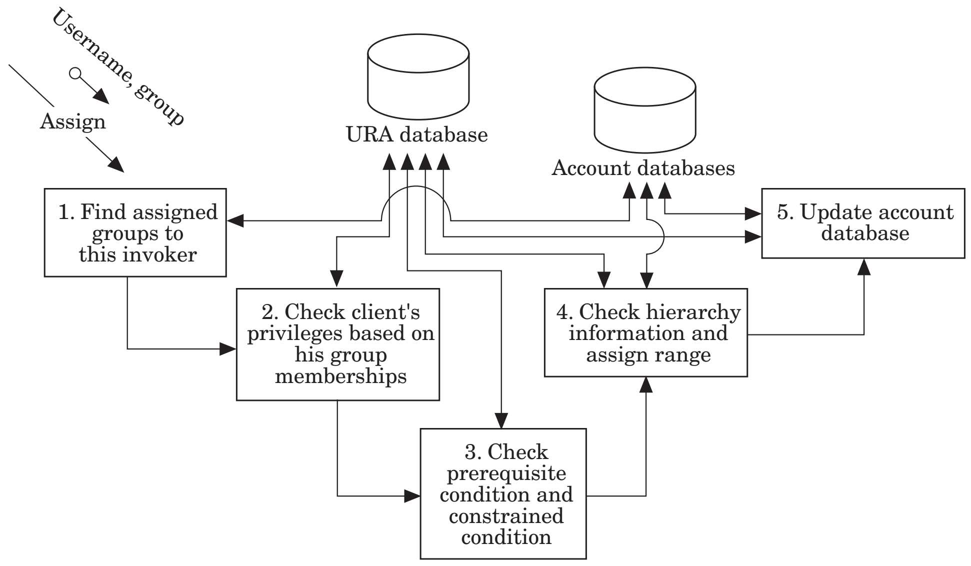

Username, group
Assign
URA database
Account databases
1. Find assigned groups to this invoker
2. Check client's privileges based on his group memberships
3. Check prerequisite condition and constrained condition
4. Check hierarchy information and assign range
5. Update account database

Figure 6. Data flow and its relationship diagram.

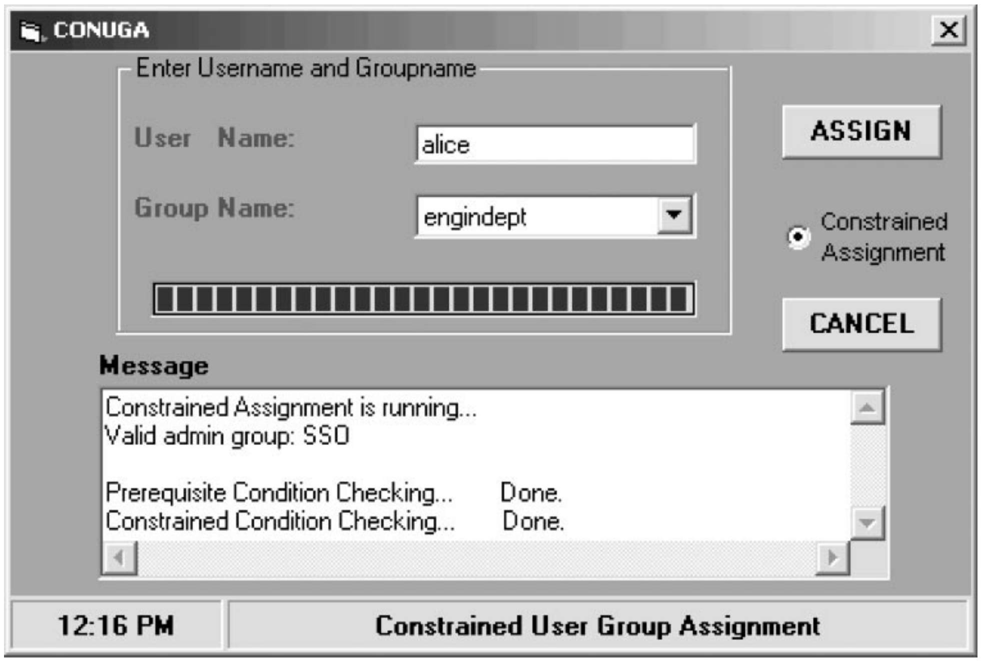
The parameters user and tgroup (target group) specify which user is to be assigned to tgroup. This procedure is called at the Windows NT command line prompt (which is actually DOS prompt) as follows.

```
C:\> assign username target-group
```

There is a built-in Windows NT command line utility `net.exe`. To assign a user to group(s) the `assign` function generates batch files which include this command line utility with several parameters based on its syntax. The `net.exe` function enables us to check whether or not a user is assigned to a specified group. The `assign` function also has a module that calls this system call and uses the results from it. Basic syntax and an example of this system call are as follows.

```
C:\> net
The syntax of this command is:
NET [ACCOUNTS |COMPUTER |CONFIG |CONTINUE |FILE |GROUP |HELP |
      HELPMSG |LOCALGROUP |NAME |PAUSE |PRINT |SEND |SESSION |
      SHARE |START |STATISTICS |STOP |TIME |USE |USER |VIEW]
C:\> net user
User accounts for \\gahn
.............................................................................
Administrator            Guest
The command completed successfully.
```

The `assign` function has five main modules. Each module accesses URA database or Windows NT's user account database for its purpose. URA database maintains



User Name: alice
Group Name: engindept
[✓ Constrained Assignment]

```
Constrained Assignment is running...
Valid admin group: SSO

Prerequisite Condition Checking...      Done.
Constrained Condition Checking...       Done.
```

12:16 PM — Constrained User Group Assignment

Figure 7. CONUGA interface.

group hierarchy information, explicit membership, *can-assign* relation table and AT-SET table. This `assign` function is based on grant algorithm described in Figure 4.

In order to make our implementation more convenient we developed a graphical user interface which interacts with this procedure to do user-group assignment. The graphical user interface is illustrated in Figure 7. This interface was developed using Visual Basic programming and is used to initiate user-group assignment instead of typing the above as command line procedure call. This implementation is convenient for administrative groups since they only need to define the group hierarchy and the relations *can-assign*.

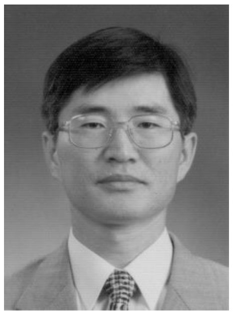
5. Conclusion

In this paper we have described our experiment to provide constrained user-group assignment. When a user is assigned to a group the system checks constraints including prerequisite conditions and conflicting role set, and *automatically* adds the user to all junior groups to the group. We have extended the URA97 model and implemented it in Windows NT by means of Microsoft RPC programs. Our result indicates that (static) separation of duty constraints can be determined by the assignment of individuals to groups at user-group assignment time and this behaviour can be achieved by accommodating sophisticated access control model to some extent.

References

1. S. H. von Solms and I. van der Merwe 1994. The management of computer security profiles using a role-oriented approach. *Computers & Security*, **13**(8):673–680.
2. M. Y. Hu, S. A. Demurjian and T. C. Ting. User-role based security in the ADAM object-oriented design and analyses environment. In (J. Biskup, M. Morgernstern and C. Landwehr, eds) *Database Security VIII: Status and Prospects*. North-Holland.
3. M. Nyanchama and S. Osborn 1995. Access rights administration in role-based security systems. In (J. Biskup, M. Morgernstern and C. Landwehr, eds) *Database Security VIII: Status and Prospects*. North-Holland 1998.
4. D. Ferraiolo, J. Cugini and R. Kuhn 1995. Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th Annual Computer Security Application Conference, New Orleans, LA, U.S.A., December 11–15 1995*, 241–248.
5. L. Giuri 1995. A new model for role-based access control. In *Proceedings of 11th Annual Computer Security Application Conference, New Orleans, LA, U.S.A., December 11–15 1995*, 249–255.
6. L. Giuri and P. Iglio 1996. A formal model for role-based access control with constraints. In *Proceedings of IEEE Computer Security Foundations Workshop 9*, Kenmare, Ireland, June 1996, 136–145.
7. R. S. Sandhu, E. J. Coyne, H. I. Feinstein and C. E. Youman 1996. Role-based access control models. *IEEE Computer* **29**(2):38–47.
8. C. Youman, E. Coyne and R. Sandhu, eds 1997. *Proceedings of the 1st ACM Workshop on Role-Based Access Control, Nov 31-Dec. 1, 1995*. ACM.
9. R. Sandhu and V. Bhamidipati 1997. The URA97 model for role-based administration of user-role assignment. In (T. Y. Lin and X. Qian) eds, *Database Security XI: Status and Prospects*. North-Holland.
10. G. -J. Ahn and R. Sandhu 1999. The RSL99 language for role-based separation of duty constraints. In *Proceedings of 4th ACM Workshop on Role-Based Access Control, Fairfax, VA, U.S.A., October 28–29*, 43–54. ACM.
11. S. A. Sutton 1997. *Windows NT Security Guide*. Addison Wesley Developers Press.
12. R. Sandhu and G. -J. Ahn 1998. Decentralized group hierarchies in unix: An experiment and lessons learned. In *Proceedings of 21st NIST-NCSC National Information Systems Security Conference, Arlington, VA, U.S.A., October 5–8 1998*.

Gail-Joon Ahn is an Assistant Professor of the Computer Science Department at the University of North Carolina at Charlotte. His principal research and teaching interests are in information and systems security. He received PhD and MS degrees from George Mason University, Fairfax, Virginia, and a BS degree in Computer Science from SoongSil University, Seoul, Korea. He was a research associate at the Laboratory for Information Security Technology, George Mason University. His research interests include access control, security architecture for distributed objects, and secure e-commerce systems. Gail-Joon Ahn is a member of ACM and the IEEE Computer Society.



Kwangjo Kim is an Associate Professor of the School of Engineering at the Information and Communications University, Korea. He received BS and MS degrees in Electronic Engineering from Yonsei University, Seoul, Korea in 1980 and 1983 respectively. In 1991, he received a PhD degree in Electrical and Information Engineering from Yokohama National University, Japan. He worked in cryptographic technology for the Electronics and Telecommunications Research Institute prior to joining ICU from 1979–1998. His research interests include all fields of cryptography and information security. Kwangjo Kim is a director of the International Association for Cryptologic Research and is a member of KIISC, IEICE and IEEE.