Modeling Users in Role-Based Access Control

Sylvia Osborn
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada N6A-5B7
sylvia@csd.uwo.ca

Yuxia Guo
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada N6A-5B7

ABSTRACT

The essential properties of a simple model for users, groups and group hierarchies for role-based access control, embodied in a group graph, are presented. The interaction between the group graph and the role graph model of Nyanchama and Osborn is shown. More complex models of users and their compatibility with the group graph model are discussed.

1. INTRODUCTION

In our previous work on role-based access control ([6]), we have shown how to provide a rich model for a role hierarchy, which we call a role graph. Roles collect privileges or permissions into a single entity which can help simplify the granting of permissions to users. Our model also includes a user/group plane, on which we model collections of users as groups, and a privileges plane, on which implications among privileges are modeled. To date, we have not explored the user/group plane in any detail. It is possible to model users simply as individuals with a unique ID and no other relevant attributes. For the same reason that we gather permissions into roles, it is useful to gather users into groups, that reason being to simplify the work of assigning permissions to users.

We will use the term *group* to refer to a set of users. Groups have been used in operating systems for decades to create sets of users for the convenience of the system managers. Group hierarchies are typically not provided. Sandhu and Ahn have shown how group hierarchies with decentralized management can be provided in Unix ([7]) and in Windows NT ([8]). Our purpose in this paper is to show how groups and a group hierarchy can be used together with roles and the role hierarchy to provide a very rich model for access control. In particular, we will show how a group graph model can be used along side our role graph model.

In some systems, there is a requirement for users to have other attributes, say age for digital library applications or security attributes for a Corba implementation. In other words, there is a requirement for users to be modeled as objects with some structure. We begin by showing how a very simple model of users can be designed and integrated with our role graph model. Then we go on to discuss more complex user models. In the end, we show that these models are not contradictory, and can in fact co-exist easily.

The group graph model is introduced in Section 2. Section 3 contains a discussion of using an object-oriented model for users. A summary is given in Section 4.

2. THE GROUP GRAPH MODEL

2.1 Groups

The role graph model of Nyanchama and Osborn [5, 6] separates users, roles and privileges into three separate planes: a user/group plane, a role plane and a privileges plane. Considerable work has been reported on algorithms for manipulating roles and role graphs [5, 6], to model role-role relationships or role hierarchies[10]. The roles in a role graph are denoted by a role name and set of privileges. Privileges, in turn, consist of a object and some operation on the object. The inclusion of a privilege in a role's privilege set means that any user assigned to that role is granted access to the privilege, i.e. permission to perform the operation on the object. The role graph is an acyclic, directed graph whose edges, say $r_1 \rightarrow r_2$, indicate that the privilege set of r_1 is a proper subset of the privilege set of r_2. We also say that r_1 is *junior* to r_2 (equivalently r_2 is *senior* to r_1).

In this section, we report on a simple model for the user plane [4] and some associated operations and algorithms. In the user/group plane, we represent all the individual users in the system, as well as an arbitrary number of groups, which can vary over time. A *group* is a set of users. For the purpose of this simple model each user must have some unique identifier, say a unique user name or an object ID. No other structure or attributes are assumed for users at this time.

In our group model, there is always one group which contains all users. This group can be used to assign everyone to the role which contains any default privileges (called MinRole in our model). We show group-group relationships as an acyclic directed graph, called the group graph. An edge in the graph between groups g_1 and g_2, denoted $g_1 \rightarrow g_2$ means that $g_1 \subset g_2$[1]. We call g_1 a *subgroup* of g_2. A *group*

[1] more precisely, the membership list of $g_1 \subset$ membership list of g_2

graph, GG(G, S), consists of a set G of groups and subgroup relationships S, the latter shown as the edges in the group graph. G always contains one group for each user, of cardinality one. Each group has a unique name, which, in the case of cardinality one groups can simply be the user name. Each group also has a unique membership list; we insist on this so that we do not have cycles in the group graph which would then need to be collapsed. When we use the notation $g_1 \subset g_2$, we are referring to the membership lists.

To summarize then, the group graph GG(G, S), consists of a set G of groups, each of which has a name and membership list, and subgroup relationships S. The group graph has the following *Group Graph Properties:*

- there exists one group containing all users
- there exists one group for each individual user
- group membership lists are unique
- if the members of $g_1 \subset$ members of g_2, there must be a path from g_1 to g_2 in S.

There is a many-to-many relationship between groups and roles, called the group-role authorization or group-role assignment, which indicates which groups, and therefore which users, are authorized to perform the operations embodied in a role. Such assignments are made by a role authorization function, which can be modeled as administrative roles [9]. Since groups with duplicate membership lists are not allowed, if a group of users plays two roles in a company, we model this by having just the one group in the group plane, and having two roles to which this one group is assigned.

A simple group graph is shown in Figure 1. As we do with our role graphs, we do not show an edge from a group g_1 to a distinct group g_2 if there is also a path from g_1 to g_2. In other words, we do not show redundant edges in the group graph. For reasons which will become apparent shortly, we put the group representing all users at the bottom of the page, and the groups of cardinality one at the top.

2.2 Operations on the Group Graph

There are a number of operations on the Group Graph which allow groups to be formed and updated. These operations have been implemented and tested in a prototype [4]. Unlike the role graph operations in our previous work ([6]), group-group relationships are determined solely by examining the subset relationships between the groups. (In the role graph case, relationships can be explicitly added or deleted through role administration by adding or deleting edges from the role graph.) The group graph operations are briefly summarized here:

Group Addition(GG, g): adds a new group g to the group graph GG and establishes the appropriate edges in the group graph.

Group Deletion(GG, g): deletes group g from the group graph GG and adjusts the edges in the group graph.

User Addition without Propagation(GG, g, u): adds user u to group g, but does not add this user to any of the group's supergroups.

User Addition with Propagation(GG, g, u): adds user u to group g, and to all of g's supergroups.

User Deletion without Propagation(GG, g, u): deletes user u from group g, but not from any of g's supergroups.

User Deletion with Propagation(GG, g, u): deletes user u from group g, and from all of g's supergroups.

Add User(GG, u): creates a new user in GG.

Delete User from System(GG, u): deletes u from the group graph and from all groups of which it is a member.

Algorithms exist for these operations [4], which perform the indicated action and restore the group graph properties, or report an error if they cannot be carried out. The algorithms are all straightforward; they are not given here in detail.

2.3 Group-Role Assignment

The assignment of groups to roles is a many-to-many relationship (remember that individual users are each represented by a group of cardinality one, so group-role assignment also includes user-role assignment). A sample group-role assignment is shown in Figure 2. This figure shows a group graph on the left, a role graph on the right, and heavy arcs going from the group graph to the role graph indicating group-role assignments. Note that if a group is assigned to a role, such as Engineers to the Engineer role in the example, its members can also perform the permissions in all roles junior to Engineer (i.e. whatever privileges are present in Employee), by the construction of the role graph. Also, all members of subgroups (Quality Engineers and the groups of cardinality 1 representing individual users contained in the Engineers group) can perform the Engineer role, by the structure of the group graph.

There are a few conditions which need to be checked when a new group-role assignment is given. For example, the figure shows that the group Engineers is assigned to the Engineer role. Assigning the Quality Engineers group to the Employee role would be redundant, since these users already have the privileges of the Employee role. This attempt at a group-role assignment is simply not performed. Figure 3 shows this situation with smaller role and group graphs and fewer group-role assignments.

A slightly different case arises if, say, the Engineering Department group is to be assigned to the Project 1 role. A subset of the same users (Engineers) is assigned to a junior role (Engineer). If this latter group-role assignment remains it will be redundant. Therefore it is removed as part of the operation of assigning the Engineering Department group to the Project 1 role. This situation is shown in Figure 4. Again some of the graph nodes and group-role assignment arcs are removed to highlight the situation of concern.

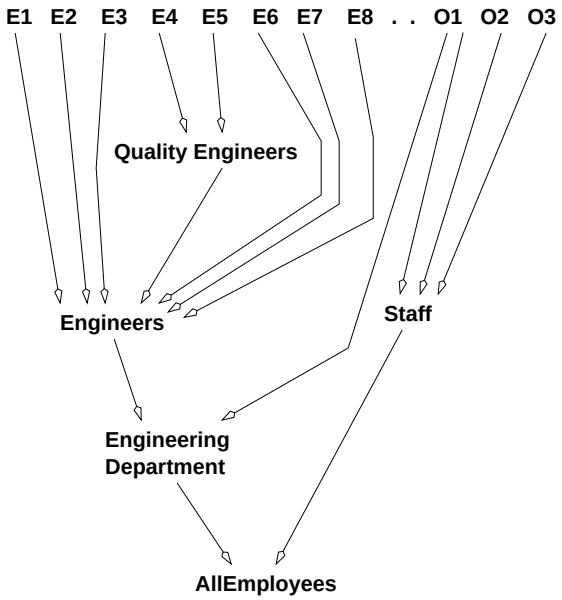

Figure 1: Example Group Graph

The way we have drawn our graphs, with the most senior role at the top in the role graph, and the most all-inclusive group at the bottom in the group graph, means that more exclusive (smaller) groups are near the top of the group graph, opposite the more senior roles in the role graph. Both of the previous cases involve a path in the group graph, a path in the role graph, and an operation which would create a group-role assignment which "crosses over" an existing group-role assignment edge joining another group and role on these paths. When the operation is finished, there should not be a "crossing over" of the group-role assignments in these diagrams. Such redundant assignments are avoided by our algorithm, in keeping with our general policy of avoiding redundant edges in the role graphs and group graphs.

Conflict of interest and separation of duties issues may require that, if a user is assigned to one role, the same user should not also be assigned to a conflicting role. Such conflicts could represent static or dynamic separation of duties – in either case they should be checked when user-role assignment is taking place. These conflicts can be detected and group-role assignments rejected.

An algorithm which summarizes these ideas is given in Figure 5. The algorithm takes a group graph, a role graph, a set of group-role pairs representing the current group-role assignments, and a group-role assignment, (g, r), which is to be added. It begins, in step 1, by making sure such a group-role assignment has not already been made. Then, in step 2, it checks that there is no conflict of interest with existing role assignments for this group. It then checks the two cases discussed above to avoid redundant group-role assignments. Its run time is discussed in Appendix A. Steps 3 and 4 involve traversals up or down from the given g and r in their respective graphs. This cannot be avoided, since the redundant cases being sought can occur anywhere in the graph. A way to cut off the search when one situation is found is discussed briefly in the Appendix.

3. OTHER REQUIREMENTS FOR USERS

There are scenarios in which users need attributes or properties for some other purpose, i.e. in which users need to be modeled with more structure than was assumed in the previous section. In other words, the users need to be modeled as objects. One example is a digital library where the age of the user might be used to decide if he or she should be allowed to view certain materials. Another example occurs when the users might in fact be represented by principals in a Corba system with security attributes [1].

In a class-based object system, objects are created as instances of a class. These, in turn, may be collected into sets according to their class structure, but they do not have to be. There was an extensive debate in the object-oriented database community concerning the modeling of sets of objects. The issue arises because in a database, there are multiple instances of a given class, which need to be collected into sets suitable for querying. In some cases, these sets might correspond to all the instances of a given class. In other cases it might be only some, (e.g. it might be necessary to have a set of all final-year students, rather than just a set of all students). There might even be a need for non-homogeneous sets, such as a student, several faculty and several staff members forming a committee. The result of this debate in the database community is that one can build whatever sets suit the application's needs [2], i.e. sets corresponding to classes, subsets of all the instances of a class, non-homogeneous sets, or a mixture of all these types. Bringing this discussion back to the group plane, if the users are modeled as objects, then the groups are sets of objects. All of the above cases might be appropriate in different applications, so the model should allow one to build whatever groups are appropriate.

Suppose that the class hierarchy for the users shown in Figures 1 and 2 is that given in Figure 6 (with inheritance going down the page). Note that in the group graph shown

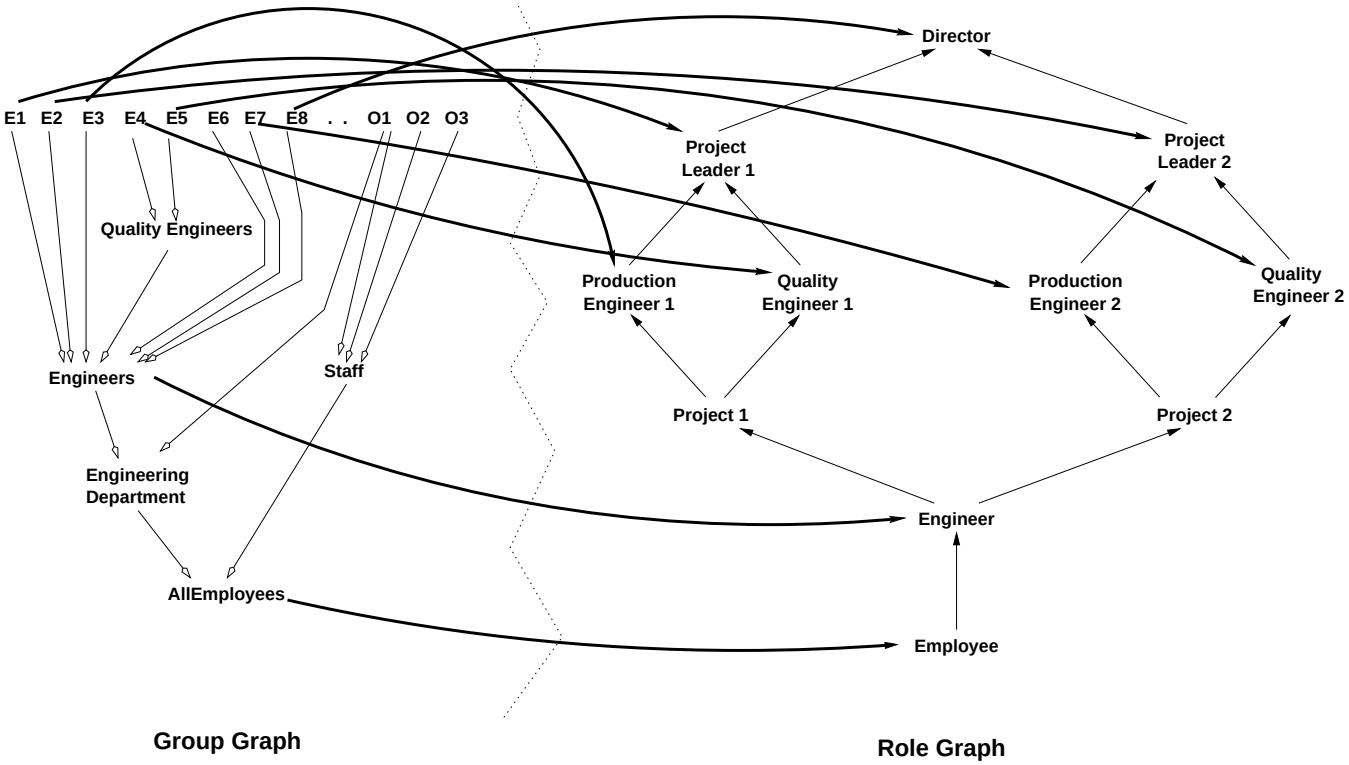

Figure 2: Group graph on left, role graph on right, showing group-role assignments

above (in Figures 1 and 2), there is a group, Engineering Department, which has some members who are Engineers, and some who are not (just Employees). We are assuming that Quality Engineers is a subclass of Engineers, because their instances have some extra qualifications represented by extra object properties.

It is a function of class-based object-oriented systems that all instances of a subclass (say Quality Engineers) are also instances of any superclass(es) of their class (Engineers in this case). Thus, any operations on an object of type Engineers can also be performed on an instance of Quality Engineers because the subclass inherits whatever attributes or properties are required in the carrying out of such operations. In our example, some of the groups correspond to classes, some (the cardinality 1 groups) to instances, and some to a useful grouping of objects (Engineering Department).

When do we perform operations **on** these objects – the users in our system? Mostly, operations in an RBAC system will be performed on users when checking constraints. (Users, in turn, through their role assignments and the permission-role assignments are ultimately authorized to perform operations on other objects.) It then becomes an issue of design of the user/group model for a system. Different scenarios can arise. It may be that no constraints will be checked, in which case the simple model of the previous section is adequate.

At the other extreme, class-supplied properties can be used in a sophisticated way to check constraints. There might be requirements for constraint checking at various levels of detail. Suppose that when a group is assigned to the Quality Engineer 1 role, all members of the group must have some attributes checked (those attributes which are unique to the Quality Engineer class). This operation would fail, possibly in an ungraceful way, if the user is not an instance of class Quality Engineer. It might still fail if the user is a Quality Engineer but does not have the required property value to satisfy the constraint.

To summarize this section, if the designer of the system sees a need to use an object-oriented model for users, and exploit object-oriented properties in some way, our model can accommodate that. In the end, we will build our group graph using the subset relationship among the defined groups, and carry on with the operations and algorithms presented in Section 2.

4. SUMMARY

We have shown how a simple model of groups can be designed and integrated with our role-graph model. An algorithm for group-role assignment which avoids redundant group-role assignments is given. It was also shown that when the group graph is drawn with the most inclusive group at the bottom, and the role graph is drawn with the most senior role at the top, a fundamental characterization of redundant group-role assignments is that they cause arcs which cross over existing arcs.

We also discussed how a more complex model for the user plane, involving an object-oriented model of the users, can be used. The system designer may or may not wish to base user groups on the sets of instances of objects belonging to a single class. If this is the case, the algorithm for group-role

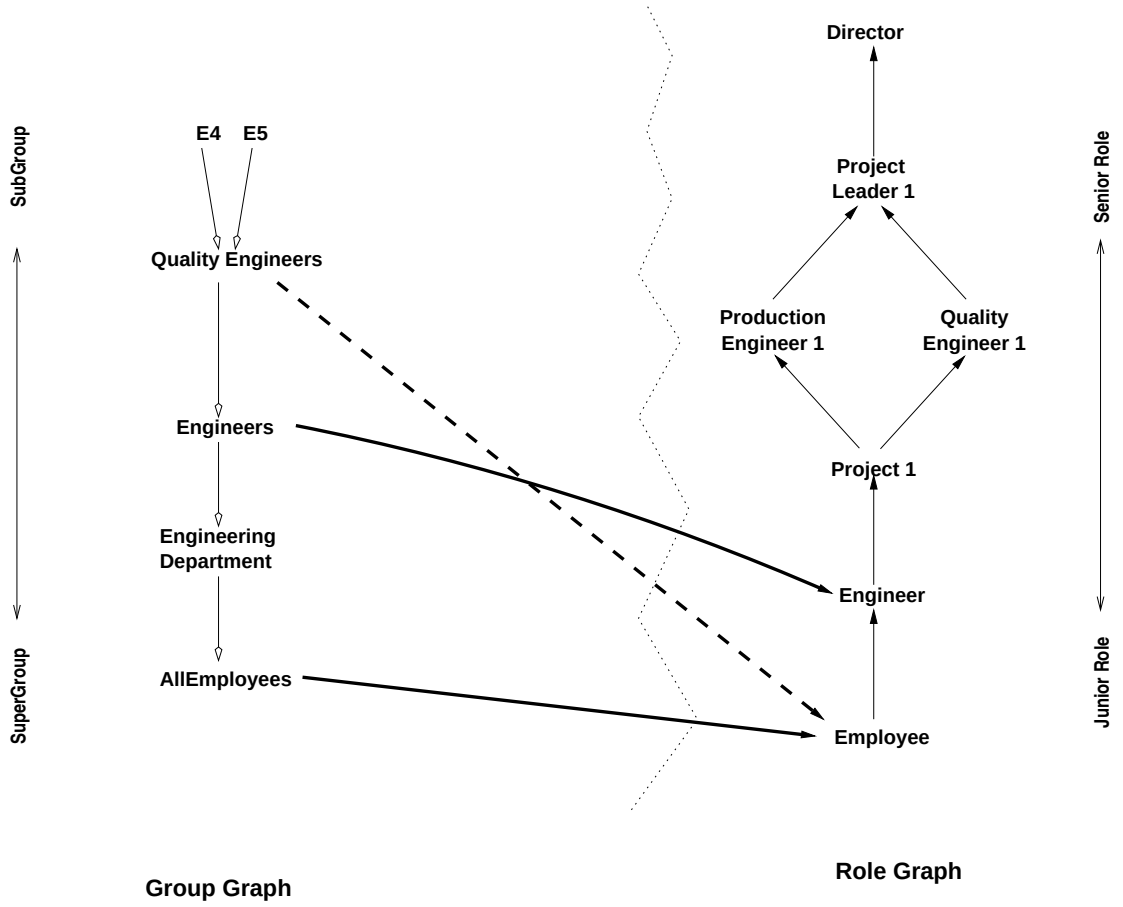

Figure 3: Redundant group-role Assignment

assignment is still valid, as is the observation about avoiding redundant user-role assignments.

5. ACKNOWLEDGMENTS

Sylvia Osborn's research was funded by the Natural Sciences and Engineering Research Council of Canada. Yuxia Guo's research was funded by a University of Western Ontario President's Scholarship for Graduate Studies.

6. REFERENCES

[1] K. Beznosov and Y. Deng. A framework for implementing role-based access control using CORBA security service. In *Proceedings RBAC4 Workshop*, pages 19–30, 1999.

[2] R. G. G. Cattell. *Object Data Management, Object-oriented and Extended Relational Systems, revised edition*. Addison-Wesley, 1994.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[4] Y. Guo. User/group administration for rbac. Master's thesis, Dept. of Computer Science, The University of Western Ontario, 1999.

[5] M. Nyanchama and S. L. Osborn. Access rights administration in role-based security systems. In J. Biskup, M. Morgenstern, and C. E. Landwehr, editors, *Database Security, VIII, Status and Prospects WG11.3 Working Conference on Database Security*, pages 37–56. North-Holland, 1994.

[6] M. Nyanchama and S. L. Osborn. The role graph model and conflict of interest. *ACM TISSEC*, 2(1):3–33, 1999.

[7] R. Sandhu and G.-J. Ahn. Decentralized group hierarchies in unix: An experiment and lessions learned. In *National Information Systems Security Conference*, 1998.

[8] R. Sandhu and G.-J. Ahn. Group hierarchies with decentralized user assignment in windows nt. In *Proceedings of IASTED Conference on Software Engineering*, 1998.

[9] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Trans. on Information and Systems Security*, 2(1):105–135, Feb. 1999.

[10] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *Computer*, 29:38–47, Feb. 1996.

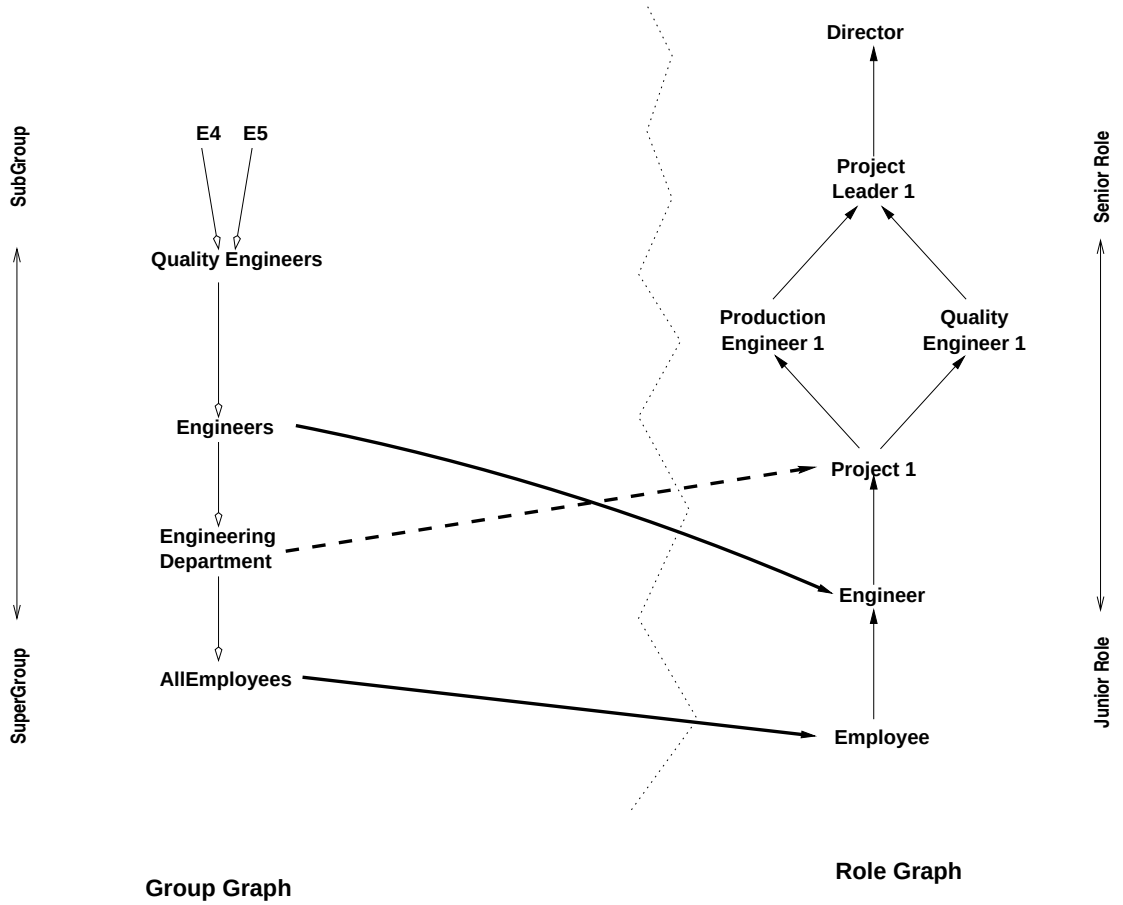

Figure 4: Redundant group-role Assignments should be Deleted

APPENDIX

A. RUN TIME ANALYSIS

We present here a discussion of some more detail and a run time analysis of the algorithm in Figure 5.

Let g denote $|G|$, the number of groups in the group graph, and r denote $|R|$, the number of roles in the role graph. Assume that current assignments are stored as a list of pairs (g, r), and that there are a of them. The upper bound on a is $g \times r$. Also, assume that there may be role-role conflicts, which are stored as a list of pairs (r_1, r_2). Assume there are c such conflicts. The upper bound on c is r^2.

Step 1 of the algorithm takes $O(a)$ comparisons with the current group-role assignments.

Step 2 has a loop which is executed r times, and which involves a comparisons with the group-role assignments list, and a traversal of the conflicts list; i.e., this step takes $O(r \times (a + c))$ steps.

Steps 3 and 4 refer to subgroups/supergroups in the group graph, and to seniors/juniors in the role graph. The subgroups of a group can be found by a depth first traversal of the group graph, starting at the group in question, going up. Similarly, supergroups can be found by a depth first traversal going down. In the role graph, one goes up to get all seniors, and down to get all juniors. In the worst case, any of these traversals could visit all of the nodes of the respective graph. The problem being sought could happen near the starting node, or not until one reaches the top or bottom of the graph. So, a more efficient algorithm, on average, would involve traversing up/down until the problem is found or we reach the top/bottom. However, the worst case for all 4 of these traversals is that we must visit every node in the appropriate graph. Depth-first traversal can be done in $O(n + e)$, where n is the number of nodes in a graph and e is the number of edges [3]. Let e_G be the number of edges in the group graph, and e_R be the number of edges in the role graph. Then, both steps 3 and 4 take $O((g + e_G) \times (r + e_R) \times a)$ operations. In other words, the algorithm takes time polynomial in the sizes of the two graphs, the list of current assignments, and the list of role conflicts.

Algorithm Group-Role Assignment
Inputs: GG /* the group graph */
RG /* the role graph */
Assignments(G,R) /* a set of group-role pairs */
g /* the group to be assigned */
r /* the role to which g should be assigned */
Outputs: Assignments′(G,R) /* a possibly modified group-role assignment */

Method:

```
if (g,r) ∈ Assignments(G,R)                                                    (1)
    then abort with message "this group-role assignment already exists"

else /* check for conflict of interest */                                      (2)
    for all roles r′ in R do
        if (g, r′) ∈ Assignments(G,R) and r and r′ have a conflict
            then abort with message "role conflict"

else /* check if a supergroup of g has been assigned to a senior role of r */  (3)
    for all groups g″ which are supergroups of g do
        for all roles r″ which are senior to r do
            if (g″, r″) ∈ Assignments(G, R)
                then abort with message "a supergroup has been assigned to a senior role"

else /* check if a subgroup of g is assigned to r or a junior of r */          (4)
    for all groups g″ which are subgroups of g do
        for all roles r″ which are junior to r, ∪ r do
            if (g″, r″) ∈ Assignments(G, R)
                then Assignments(G,R) := Assignments(G,R) - {(g″, r″)}

/* Assign g to r */
Assignments′(G,R) := Assignments(G,R) ∪ {(g,r)}
```

Figure 5: Algorithm for group-role assignment

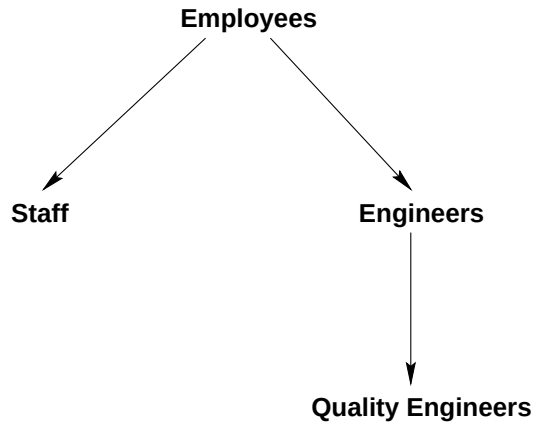

Figure 6: Possible Class Hierarchy for Users