Journal of Information Assurance and Security 1 (2008) 38-50

Authorization Constraints Specification and Enforcement

Wei Zhou[1], Christoph Meinel[1], Yidong Xiang[2], and Yang Shao[2]

[1]Potsdam University, Hasso-Plattner-Institute,
Prof.-Dr.-Helmert-Str. 2-3, Potsdam D-14482, Germany
{wei.zhou, meinel}@hpi.uni-potsdam.de

[2]Beijing Shenzhou Aerospace Software Technology Co., Ltd,
No.73 Fu cheng lu, Hai dian Dist., Beijing 100036, China
{xiangyidong, shaoyang}@bjsasc.com

***Abstract*:** Constraints are an important aspect of role-based access control (RBAC) and its different extensions. They are often regarded as one of the principal motivations behind these access control models. There are two important issues relating to constraints: their specification and their enforcement. However, the existing approaches cannot comprehensively support both of them. On the other hand, the early research effort mainly concentrates on separation of duty. In this paper, we introduce two novel authorization constraint specification schemes named prohibition constraint scheme and obligation constraint scheme respectively. Both of them can be used for both expressing and enforcing authorization constraints. These schemes are strongly bound to authorization entity set functions and relation functions that could be mapped to the functions that need to be developed in application systems, so they can provide the system developers a clear view about which functions should be developed in an authorization constraint system. Based on these functions, various constraint schemes can be easily defined. The security administrators can use these functions to create constraint schemes for their day-to-day operations. A constraint system could be scalable through defining new entity set functions and entity relation functions. This approach goes beyond the well known separation of duty constraints, and considers many aspects of entity relation constraints.

***Keywords*:** Access control, authorization constraints, constraints specification, constraints enforcement**.**

1. Introduction

Authorization constraints (also simply called constraints) as an important protection mechanism for handling important business processes or information should be integrated with all access control mechanisms, such as Discretionary Access Control (DAC), Mandatory Access Control (MAC), and especially Role-Based Access Control (RBAC) since most tasks within organizations are performed by roles [1]. In RBAC, access rights are associated with roles, and users are assigned to appropriate roles thereby acquiring the corresponding permissions. It can provide more flexibility to security management over the traditional approach of using user and group identifiers. Constraints as an important aspect of role-based access control are powerful mechanism for laying out higher-level organizational policy. They have been part of most RBAC models of recent years [2, 3, 4, 5, 6, 7].

Separation of duty is an important control principle in management whereby sensitive combinations of duties are partitioned between different individuals in order to prevent the violation of business rules [20]. It is widely considered to be a fundamental principle in computer security [8, 9, 10]. The purpose of this principle is to discourage fraud by spreading the responsibility and authority for an action or task over multiple people, thereby raising the risk involved in committing a fraudulent act by requiring the involvement of more than one individual. A very simple example of this is that checks might require two different signatures.

Although the importance of constraints in RBAC has been recognized for a long time and various approaches have been proposed to model authorization constraints, there are still some issues have not received much attention in the research literature. The early work mainly addresses constraint expression rather than constraint enforcement. Currently there is still no useful approach for both expressing and enforcing constraints. On the other hand, the early research effort mainly concentrates on separation of duty. Other kinds of constraints, such as prerequisite constraints, have received less attention. An example of prerequisite constraint is that a user can be assigned to role *A* only when he/she is already a member of role *B*. To our knowledge, there is still no dedicated work on this topic.

In this paper we propose two authorization constraint specification schemes named prohibition constraint scheme and obligation constraint scheme respectively. They can be used for both expressing and enforcing constraints. Unlike existing approaches, we do not assume that these schemes are confined in RBAC or limited to some predefined entity relations, such as user-role assignments. On the contrary, our constraint schemes could be used in various access control models and the entity relations can be arbitrary. These schemes are strongly bound to entity set and relation functions that can be directly mapped to the functions need to be developed in application systems. So we also call them function-based authorization constraint schemes. These schemes can provide the system designers and developers a clear view about what functions should be defined and developed in an authorization constraint system. Based on these functions, various constraint schemes can be easily

Received January 14, 2008.

defined. The security administrators can use these functions to create constraint schemes for their day-to-day operations. An authorization constraint system is scalable through defining new entity set and relation functions. On the other hand, this approach goes beyond the well known separation of duty constraints, and considers many aspects of the authorization constraints, such as prerequisite constraints.

The rest of this paper is organized as follows. Section 2 presents the background of authorization constraints and introduces the context for the constraint schemes. Section 3 gives the formal definition of the constraint schemes. Section 4 describes these constraint schemes' evaluation processes. Section 5 shows the constraint schemes' expressive power. Section 6 introduces the implementation of the function-based authorization constraint system. Section 7 provides some application cases. Section 8 summarizes the results of this paper.

2. Background

In this section we first discuss the related work and enumerate various constraint forms identified in the literature. We then introduce the TT-RBAC access control model that provides the context for the constraint schemes and enforcement model. Finally, we discuss the constraint classification used by the constraint schemes.

2.1 Related work

Natural language is originally used to describe authorization constraints in the context of RBAC. In RBAC96 [2], the role-based separation of duty is described as "the same user can be assigned to at most one role in a mutually exclusive set". Simon and Zurko [11] also develop a readable rule format to express the constraint policy at architectural level. Natural language specification has the advantage of ease to be understood by human beings, but may be prone to ambiguities, and the specifications do not lend themselves to the analysis of properties of the set of constraints [16]. For example, one may want to check if there are conflicting constraints in a set of authorization constraints for an organization. Another major drawback of using natural language is that constraint specification cannot be automatically dealt by computer systems.

In order to overcome the drawback of the informal definition of constraints, a variety of formal rule-based approaches have been proposed. Giuri and Iglio [12] defined a formal model for constraints on role-activation. Gligor et al. [13] formalize separation of duty constraints enumerated informally by Simon and Zurko [11]. This important theme is also addressed by Kuhn [14], Lupu and Sloman [15], Sandhu et al. [6], and Ferraiolo et al [7]. Unfortunately, rule-based systems, while highly expressive, are harder to visualize and thus to use; thus far they have been avoided by practitioners [19].

Ahn and Sandhu [16] propose a limited logical language called RCL 2000 for expressing separation of duty constraints in the context of RBAC. RCL 2000 reduces the length of the statement of the constraints. However, some constraints require iteration over the members of one set or the other, and the addition of this expression starts to make the constraints complex. The combination of quantification functions and modeling concept functions makes the constraints expressed in the language difficult to visualize. Thus, this approach is an improvement over a completely general logical language, but it is still too complex [19].

Graphical models are also used to express constraints. Nyanchama and Osborn [17] define a graphical model for role-role relationships that includes a combined view of role inheritance and separation of duty constraints based on roles. Osborn and Guo [18] extended the model to include constraints involving users. However, neither the basic model nor the extended model distinguishes between accidental relationships and explicitly constructed relationships. Thus, these models do not support policies with a historical component. Jaeger and Tidswell [19] proposed a graphical constraint model for constraint specification. An access control policy is expressed using a graphical model in which the nodes represent sets (e.g., of subjects, objects, etc.) and the edges represent binary relationships on those sets and constraints are expressed using a few, simple set operators on graph nodes. This model has been designed to be applicable in a general access control model, not just in role-based access control models. The major advantage of a graphical model is as an aid to visualize a system's policy rather than enforce it.

Recently, constraint enforcement has received more attention in the research literature. The rule-based and graph-based approaches still provide significant expressive power in constraint expression, but they are not designed for constraint enforcement. To address this problem, several scheme-based approaches have been proposed.

Crampton [20] proposed a simple specification scheme for separation of duty constraints in the context of RBAC. The specification scheme is set-based and has a simpler syntax than the early approaches. This constraint scheme is defined as a triple (s, c, x), where s is the scope set, c is the constraint set and x is the context and takes one of the following values: static, dynamic and historical. But this specification scheme cannot specify those constraints that are based on the aggregation of users and permissions with quantification over sets and members of sets. For example, in the object-based separation of duty constraint, this scheme cannot express that a subject is restricted from performing an operation on a particular object twice. This shortcoming will limit its usages in many cases.

The role-based constraint scheme designed by Li et al. [21] is SMER($\{r_1, \ldots, r_n\}$, m) where r_i is a role, and n and m are integers such that $1 < m \le n$. This constraint forbids a user from being a member of m or more roles in $\{r_1, \ldots, r_n\}$. Chadwick et al. [1] extend this constraint scheme through adding application context for supporting separation of duty constraints among multiple sessions. The extended role-based constraint scheme is MMER($\{r_1, \ldots, r_n\}$, m, BC) where BC identifies the particular business context to which the m mutually exclusive roles apply, in which r_i is a role, and $1 < m \le n$. This constraint forbids a user from activating m or more roles among $\{r_1, \ldots, r_n\}$ in the same business

context. Similarly, the permission-based constraint scheme is defined as MMEP($\{p_1, \ldots, p_n\}$, m, BC). The major drawback of these constraint schemes is that they cannot explicitly specify the scope set and assume the scope set is always a user set. So these constraint schemes cannot express certain constraints, such as mutually exclusive permissions cannot be assigned to the same role.

2.2 RBAC constraints

Various constraint forms have been identified in the literature. In the standard RBAC language, the taxonomy of constraints is summarized by Jaeger and Tidswell [19] are:

- *User-user conflicts* are defined to exist if a pair of users should not be assigned to the same role.
- *Privilege-privilege conflicts* are defined to occur between two privileges (permissions) when they should not both be assigned to the same role.
- *Static user-role conflicts* exclude users from ever being assigned to the specified roles.
- *Static separation of duty* exists if two particular roles should never be assigned to the same person.
- *Simple dynamic separation of duty* disallows two particular roles from being assigned to the same person due to some dynamic event (e.g., Chinese Wall).
- *Session-dependent separation of duty* disallows a principal from activating two particular roles at the same time (e.g., within the same session).
- *Object-based separation of duty* constrains a user never to act on the same object twice. They can also be specified to constrain the same role from acting on the same object twice.
- *Operational separation of duty* breaks a business task into a series of stages and ensures that no single person can perform all stages. Thus, the roles that are entitled to perform each stage may have users in common so long as no user is a member of all the roles entitled to perform each stage of a business task.
- *Order-dependent history constraints* restrict operations on business tasks based on a predefined order in which actions may be taken.
- *Order-independent history constraints* restrict operations on business tasks requiring two distinct actions (such as two distinct signatures) where there is no ordering requirement between the actions.

2.3 TT-RBAC

Role-based access control [2, 7] has emerged as a widely accepted alternative to classical discretionary and mandatory access controls. The essence of RBAC is that permissions are assigned to roles rather than to individual users. Roles are created for various job functions, and users are assigned to roles based on their qualifications and responsibilities. Users can be easily reassigned from one role to another without modifying the underlying access structure. RBAC is thus more scaleable than user-based security specifications and greatly reduces the cost and administrative overhead associated with fine-grained security administration at the level of individual users, objects, or permissions. But subsequent attempts to apply RBAC in collaborative environments revealed some of RBAC's limitations. RBAC lacks the ability to specify a fine-grained access control on individual users in certain roles and on individual object instances. For collaborative environments, it is insufficient to have role permissions based on object types. Rather, it is often the case that a user in an instance of a role might need a specific permission on an instance of an object [22]. On the other hand RBAC does not provide an abstraction to capture a set of collaborative users who operate in different roles. Furthermore, the lack of object typing in RBAC models makes it hard to model workflow constraints [19].

Motivated by these requirements, we defined the Team and Task-based RBAC (TT-RBAC) access control model that extends RBAC model through adding sets of two basic data elements called teams and tasks [23, 24]. TT-RBAC model element sets and relations are defined in Figure 1.

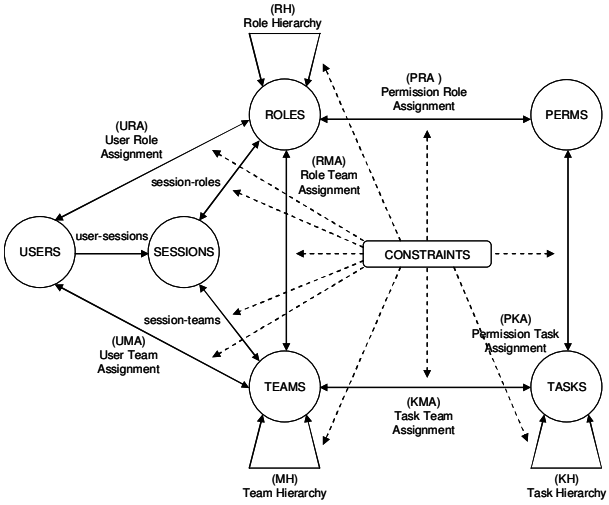

Figure 1. TT-RBAC access control model

A *task* is a fundamental unit of business work or activity. A set of tasks connecting together can form a business process (workflow). Tasks can be defined in any granularity. They can be type-based or instance-based. A *team* encapsulates a collection of users in various roles and a set of roles with the objective of accomplishing specific tasks. The TT-RBAC model as a whole is fundamentally defined in terms of individual users being assigned to teams and roles, tasks being assigned to teams, and permissions being assigned to roles and tasks. The relations among of them are many-to-many. In addition, the TT-RBAC model includes a set of sessions where each session is a mapping onto an activated subset of roles and an activated subset of teams that are assigned to the user. By virtue of team membership, users get access to team's resources specified by assigned tasks. However, for each team member, the exact permissions he/she obtains to a team's resources will determined by his/her role and the current activity of the team. A team member can only activate the roles that are assigned to both him/her and the team. Thus, the team defines a small RBAC application zone, through which we can preserve the advantages of scaleable security administration that RBAC model provides and yet offers the

ability to specify a fine-grained control on individual users in certain roles and on individual object instances.

Central to TT-RBAC is the concept of team relations, around which users, roles and tasks are connected together. Besides the entity relations defined in the RBAC model, Figure 1 illustrates user-team assignment (UMA), role-team assignment (RTA), task-team assignment (KMA) and permission-task assignment (PKA) relations defined in the TT-RBAC model. Similar to the constraints defined in RBAC, some constraints should be defined to restrict the ability to form these relations. For example, conflict roles or conflict tasks cannot be assigned to the same team.

In fact, the added two new entity sets make the relations among the entities in TT-RBAC more complicated than in RBAC. Besides the separation of duty constraints, some other types of constraints need to be considered. For example, at least five users should be assigned to a team, each team must have one team leader and two vice-team leaders, a medical team can be activated only when at least one user with the role of physician is assigned to the team, and so on. Potentially, constraints can exist in any relations among the entities shown in Figure 1.

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 approach for specifying and enforcing various constraints, such as the function-based constraint schemes that can express constraints for arbitrary entity relations.

2.4 Constraint classification

Simon and Zurko [11] use two broadest categories of separation of duty variations: static separation of duty and dynamic separation of duty. Static separation of duty prevents mutually exclusive roles from assigning to the same user and conflict privileges from assigning to the same role. Dynamic separation of duty prevents some roles from being activated at the same time. History-based separation of duty is classified into this category. One example of history-based constraints is that one user cannot perform all the steps in a workflow instance. This kind of constraint classification is widely adopted in research literature (see, for example, Gligor et al. [13], Bertino et al. [5] and Li et al. [21]).

NIST RABC [7] also classifies the constraints into static separation of duty and dynamic separation of duty two categories. But their static separation of duty only consider that mutually exclusive roles cannot be assigned to the same user and their dynamic separation of duty only consider that conflict roles cannot be activated at the same time. History-based separation of duty is not supported by this model.

Crampton [20] systematically discusses the history-based constraints that are classified into static historical constraints and dynamic historical constraints two categories. One example of static historical constraint is that once u has been assigned to r_1, then u can never be assigned to r_2. One example of dynamic historical constraint is that once u has activated r_1, then u can never activate r_2.

Chadwick et al. [1] propose multi-session separation of duty to model the business processes which include multiple tasks enacted by multiple users over many user access control sessions in dynamic virtual organization environment. Basically, the multi-session separation of duty belongs to the history-based separation of duty.

In this paper we adopt two axes on which to classify authorization constraints. The first axis is the objective of constraints. The second axis is the enforcement context of constraints.

According to the objective of constraints, we classify constraints into two categories: *prohibition constraints* and *obligation constraints*. Their definitions are taken from [16]. The prohibition constraints are constraints that forbid the entities from being (or doing) something which it is not allowed to be (or do). Separation of duty constraints belong to this category. Obligation constraints are constraints that force the entities to do (or be) something. In some literature, the obligation constraint is also called as prerequisite constraint. An example of obligation constraint is that a user can be assigned to one role only when he/she is already a member of some other roles. We designed two authorization constraint schemes that are used to express prohibition constraints and obligation constraints, respectively.

According to the enforcement context of constraints, we classify constraints into three categories: *static constraints*, *dynamic constraints* and *historical constraints*. Static constraints are enforced in privilege assignment stage; for example, a user is prevented from being assigned to mutually exclusive roles. Dynamic constraints are enforced at runtime within or across a user's sessions; for example, a user is allowed to be authorized for two or more roles that do not create a conflict of interest when acted on independently, but produce policy concerns when activated simultaneously. Historical constraints allow the individual access of each user to be constrained based on what they have done (or been); for example, the same user cannot access the same object a certain number of times. In the workflow environment, one user may have the privileges to perform several steps, but the same user can only perform one step in the same workflow instance. Such kind of constraints is classified into historical constraints. Both prohibition constraint scheme and obligation constraint scheme can express static, dynamic and historical constraints.

3. Constraint schemes

To be able to use constraints to ensure safety, we must find a suitable formalism to express constraints and then enforce these constraints. Jaeger and Tidswell [19] identifies that constraints in an access control environment are set comparisons. There are two steps in expressing a set comparison: (1) expressing the sets to be compared and (2) expressing the comparison to be made. We designed two types of constraint schemes named *prohibition constraint scheme* and *obligation constraint scheme* to express the sets and their comparisons for various authorization constraints.

These constraint schemes are strongly bound to some functions used to represent the sets to be compared. These functions can be classified into two categories: *entity set functions* and *entity relation functions*.

Entity set function is used to get a set of entities according to some data query criterion. There is no limitation about how the set functions should be implemented. The set functions are used to represent and get data when it is impossible or not suitable to enumerate all data items, e.g. all the staff of a university. Set functions are used to represent entity sets in constraint schemes and obtain the entity sets at runtime. For example, the set function *get_account_users* can be used to represent and obtain all the users belonging to the financial department.

Entity relation function is used to get entity relation between different types of entities. For example, the relation function *assigned_user_roles* returns the set of roles assigned to a given user, and the relation function *assigned_role_users* returns the set of users assigned to a given role.

Now we introduce the function naming rules adopted in this paper. For static assignment functions we use the prefix *assigned_*, e.g. *assigned_user_roles*; for single session functions we use the prefix *session_*, e.g. *session_user_roles*; for multiple sessions functions we use the prefix *sessions_*, e.g. *sessions_user_roles*; for historical assignment functions we use the prefix *ever_assigned_*, e.g. *ever_assigned_user_roles*. In the presences of entity hierarchies we use the prefixes *authorized_* or *ever_authorized_* to replace the prefixes *assigned_* and *ever_assigned_*, respectively. All the entity relation functions are set valued functions. As a general notational device we have the following convention. For any set valued function f defined on set X, We understand

$$f(X) = f(x_1) \cup f(x_2) \cup ... \cup f(x_n), \text{ where } X = \{x_1, x_2, ..., x_n\}.$$

For example, suppose we want to get all roles that are assigned to a set of users $U = \{u_1, u_2, u_3\}$. We can express this using the function *assigned_user_roles*(U) as equivalent to

$$assigned\_user\_roles(u_1) \cup assigned\_user\_roles(u_2) \cup assigned\_user\_roles(u_3)$$

3.1 Prohibition constraint scheme

Prohibition constraint scheme can be formatted as a triple (*S*, *C*, *T*), where *S* is the *scope element* that specifies what entities are applicable to the constraint scheme, *C* is the *constraint element* that specifies what constraints should be applied to each entity defined in the scope element, and *T* is the *constraint context* that takes one of the following values: *SPC*, *DPC* and *HPC*, which denote *Static Prohibition Constraint*, *Dynamic Prohibition Constraint*, and *Historical Prohibition Constraint*, respectively.

The scope element *S* is further defined as a 4-tuple (*SS*, *SF*, *SO*, *SN*). *SS* is the *scope set* that needs to be constrained, e.g. a user set. The scope set can be represented by an entity set function that is also called *scope set function*. For example, the function *get_users* returns a set of users. *SF* is the *scope relation function* that maps a value belonging to the constraint set defined in the constraint element to a set of values that have the same type with the scope set. For example, the function *assigned_role_users* returns a set of users assigned to a given role. *SO* is a relational operator that can be “>”, “>=”, “<”, “<=”, “=” or “≠”. *SN* as the right operand of *SO* is a natural number. The (*SO*, *SN*) pair expresses the cardinality constraint to the scope set. If *SF* is not specified, then all the members in the scope set are applicable to the constraint element. An example of scope element is ($\{u_1, u_2, u_3\}$, *assigned_role_users*, <, 3) that states that less than three users defined in the scope set can be assigned to a given role.

The constraint element *C* is further defined as a 4-tuple (*CS*, *CF*, *CO*, *CN*). *CS* is the *constraint set* that expresses the constraint entities applied to the scope set, e.g. a role set. The constraint set can be represented by an entity set function that is also called *constraint set function*. For example, the function *get_roles* returns a set of roles. *CF* is the *constraint relation function* that maps a value belonging to the scope set defined in the scope element to a set of values that have the same type with the constraint set. For example, the function *assigned_user_roles* returns a set of roles assigned to a given user. *CO* is a relational operator which can be “>”, “>=”, “<”, “<=”, “=” or “≠”. *CN* as the right operand of *CO* is a natural number. The (*CO*, *CN*) pair expresses the cardinality constraint to the constraint set. An example of constraint element is ($\{r_1, r_2, r_3\}$, *assigned_user_roles*, <, 2) that states that less than two roles defined in the constraint set can be assigned to a given user.

In TT-RBAC, the scope set and constraint set are a subset of the following data sets: *USERS*, *ROLES*, *PERMS*, *TEAMS*, *TASKS* and *OBJS*. Any constraints can be defined among these entity sets if the corresponding entity set functions and entity relation functions are defined. An example of prohibition constraint scheme is shown as:

($\{u_1, u_2, u_3\}$, *assigned_role_users*, <, 3), ($\{r_1, r_2, r_3\}$, *assigned_user_roles*, <, 2), *SPC*).

This constraint scheme specifies that no user defined in the scope set $\{u_1, u_2, u_3\}$ can be assigned to more than one role defined in the constraint set $\{r_1, r_2, r_3\}$, and less than three users defined in the scope set can be assigned to any role defined in the constraint set.

3.2 Obligation constraint scheme

The obligation constraint scheme can be formatted as a 4-tuple (*S*, *R*, *C*, *T*), where *S* is the *scope element* that defines the entities needs to be constrained, *R* is the *request element* that defines the entities requested by the entities defined in the scope element, *C* is the constraint element that expresses what kind of constraints should be applied to each entity defined in the scope element, and *T* is the *constraint context* that takes one of the following values: *SOC*, *DOC* and *HOC*, which denote *Static Obligation Constraint*, *Dynamic Obligation Constraint* and *Historical Obligation Constraint*,

respectively.

The scope element *S* only contains the *scope set SS*, in which the entities need to be constrained, e.g. a user set. The request element *R* only contains the *request set RS*, in which the entities defined in the scope set want to be assigned or activate, e.g. a role set. The scope set and request set can be represented by entity set functions called *scope set function* and *request set function* respectively. The constraint element *C* has the same structure and meaning as in the prohibition constrain scheme.

In TT-RBAC, the scope set, request set and constraint set are a subset of one of the following sets: *USERS*, *ROLES*, *PERMS*, *TEAMS*, *TASKS* and *OBJS*. An example of obligation constraint scheme is shown as:

($\{u_1, u_2, u_3\}$, $\{r_3, r_4\}$, ($\{r_1, r_2\}$, *assigned_user_roles*, >, 1), *SOC*).

This constraint scheme states that any user defined in the scope set $\{u_1, u_2, u_3\}$ can be assigned to any role defined in the request set $\{r_3, r_4\}$ if and only if he/she is already be assigned to more than one role defined in the constraint set $\{r_1, r_2\}$.

4. Constraint scheme evaluation

The prohibition constraint schemes and obligation constraint schemes are not only used for expressing authorization constraints, but also used for enforcing authorization constraints. In this section, we investigate how these constraint schemes are evaluated at runtime.

4.1 Functions for constraint scheme evaluation

Authorization constraint request can be expressed as: (*s*, *o*, *a*), where *s* is the request subject, *o* is the request object, and *a* is the request action. For example, (u_1, r_1, *RUA*) is a role-user assignment request. The *RUA* is the request action that denotes the action of assigning a role to a user. We define *CRS*, *CRO* and *CRA* three functions to get constraint request subject, object and action, respectively. Other functions related to the constraint scheme evaluation are defined as follows.

- *SRF*(*X*) represents the scope relation function that maps a set value *X* to another set value that have the same type with the scope set. At runtime, the *SRF* is replaced by the scope relation function of a constraint scheme, and the value *X* is replaced by the constraint set of the constraint scheme. For example, *assigned_role_users*($\{r_1, r_2\}$) = $\{u_1, u_2, u_3\}$.
- *CRF*(*x*) represents the constraint relation function that maps a value to a set value that have the same type with the constraint set. At runtime, the *CRF* is replaced by the constraint relation function of a constraint scheme, and the value *x* is replaced by the constraint request subject. For example, *assigned_user_roles*(*u*) = $\{r_1, r_2\}$.
- *EN*(*X*) is a function to get the element number of a set value *X*. For example, *EN*($\{r_1, r_2, r_3\}$) = 3.

4.2 Prohibition constraint scheme evaluation

The prohibition constraint scheme evaluation comprises two steps. One is the *scope element evaluation* that is composed of *scope element applicable check* and *scope element cardinality check*. The other is the *constraint element evaluation* that is composed of *constraint element applicable check* and *constraint element cardinality check*. When ((*SS*, *SF*, *SO*, *SN*), (*CS*, *CF*, *CO*, *CN*), *T*) is the prohibition constraint scheme and *q* is the constraint request, the prohibition scope element evaluation can be formulated as:

$$ESE(q) = \begin{cases} CRS(q) \in SS & \text{if } SF = null \\ CRS(q) \in SS \wedge EN((SRF(CS) \cup CRS(q)) \cap SS) & \\ \text{satisfy}(SO, SN) & \text{if } SF \neq null \end{cases}.$$

ESE is the function for the scope element evaluation. There are two cases for this evaluation. In the case of scope relation function is *null*, we only need to do the scope element applicable check. The scope element is "Applicable" if and only if the constraint request subject is defined in the scope set; otherwise the scope element is "NotApplicable". If the scope element is applicable, the *ESE* then does the scope element cardinality check. In the case of *SF* is *null*, the *ESE* returns the value of "Permit". In the case of *SF* is not *null*, the *ESE* then checks if it still satisfies the scope element cardinality constraint after the object is assigned to the subject or activated by the subject. If so, the *ESE* returns the value of "Permit", otherwise returns the value of "Deny".

The prohibition constraint element evaluation can be formulated as:

$$\begin{aligned} &ECE(q) = CRO(q) \in CS \wedge \\ &EN((CRF(CRS(q)) \cup CRO(q)) \cap CS)\,\text{satisfy}(CO, CN) \end{aligned}.$$

ECE is the function for the constraint element evaluation. Here we need to do the constraint element applicable check and constraint element cardinality check. The constraint element is "Applicable" if and only if the constraint request object is defined in the constraint set; otherwise the constraint element is "NotApplicable". If the constraint element is applicable, the *ECE* then checks if it still satisfies the constraint element cardinality constraint after the object is assigned to the subject or activated by the subject. If so, the *ECE* returns the value of "Permit", otherwise returns the value of "Deny".

A prohibition constraint scheme evaluation result is "Permit" if and only if both scope element check and constraint element check return "Permit". The prohibition constraint scheme evaluation can be formulated as:

$$EPC(q) = ESE(q) \wedge ECE(q).$$

The *EPC* is the function for the prohibition constraint scheme evaluation. The prohibition constraint scheme true table is shown in Table 1.

Value of ESE(q)	Value of ECE(q)	Value of EPC(q)
"Permit"	"Permit"	"Permit"
Do not care	"Deny"	"Deny"
"Deny"	Do not care	"Deny"
"NotApplicable"	"Permit", "NotApplicable" or "Indeterminate"	"NotApplicable"
"Permit", "NotApplicable" or "Indeterminate"	"NotApplicable"	"NotApplicable"
"Indeterminate"	"Permit", "Applicable" or "Indeterminate"	"Indeterminate"
"Permit", "Applicable" or "Indeterminate"	"Indeterminate"	"Indeterminate"

Table 1. Prohibition constraint scheme truth table.

4.3 Obligation constraint scheme evaluation

The obligation constraint scheme evaluation comprises three steps. The first is the *scope element evaluation* that is composed of *scope element applicable check*. The second is the *request element evaluation* that is composed of *request element applicable check*. The third is the *constraint element evaluation* that is composed of *constraint element cardinality check*. When (*SS*, *RS*, (*CS*, *CF*, *CR*, *CN*), *T*) is the obligation constraint scheme and *q* is the constraint request, the obligation scope element evaluation can be formulated as:

$$ESE(q) = CRS(q) \in SS \cdot$$

ESE is the function for the scope element evaluation. It only needs to do the scope element applicable check. The scope element is "Applicable" if and only if the constraint request subject is defined in the scope set; otherwise the scope element is "NotApplicable".

The obligation request element evaluation can be formulated as:

$$ERE(q) = CRO(q) \in RS \cdot$$

ERE is the function for the request element evaluation. It only needs to do the request element applicable check. The request element is "Applicable" if and only if the constraint request object is defined in the request set; otherwise the request element is "NotApplicable".

The obligation constraint element evaluation can be formulated as:

$$ECE(q) = EN((CRF(CRS(q)) \cup CRO(q)) \cap CS) \text{satisfy}(CO, CN).$$

ECE is a function for the constraint element evaluation. It only needs to do the constraint element cardinality check. It checks if the constraint element still satisfies the constraint element cardinality constraint after the object is assigned to the subject or activated by the subject. If so, the *ECE* returns the value of "Permit", otherwise returns the value of "Deny".

An obligation constraint scheme evaluation result is "Permit" if and only if both scope element applicable check and request element applicable check return the value of "Applicable" and constraint element cardinality check returns the value of "Permit". The obligation constraint scheme evaluation can be formulated as:

$$EOC(q) = ESE(q) \wedge ERE(q) \wedge ECE(q) \cdot$$

EOC is the function for the obligation constraint scheme evaluation. The obligation constraint scheme true table is shown in Table 2.

Value of ESE(q)	Value of ERE(q)	Value of ECE(q)	Value of EOC(q)
"Applicable"	"Applicable"	"Permit"	"Permit"
"Applicable"	"Applicable"	"Deny"	"Deny"
"Applicable"	"Applicable"	"Indeterminate"	"Indeterminate"
"NotApplicable"	Do not care	Do not care	"NotApplicable"
Do not care	"NotApplicable"	Do not care	"NotApplicable"
"Indeterminate"	"Applicable" or "Indeterminate"	Do not care	"Indeterminate"
"Applicable" or "Indeterminate"	"Indeterminate"	Do not care	"Indeterminate"

Table 2. Obligation constraint scheme truth table.

4.4 Example of constraint scheme evaluation

In this subsection we show the prohibition constraint scheme evaluation process via an example. The prohibition constraint scheme is defined as:

(({u_1, u_2, u_3}, *assigned_role_users*, <, 3), ({r_1, r_2, r_3}, *assigned_user_roles*, <, 2), *SPC*).

This constraint scheme specifies that less then three users defined in the scope set {u_1, u_2, u_3} can be assigned to any role defined in the constraint set{r_1, r_2, r_3}, and each user defined in the scope set can only be assigned to less than two roles defined in the constraint set. We assume that user u_1 is already assigned to role r_1. Thus, there are:

$$assigned\_user\_roles(u_1) = \{r_1\}.$$
$$assigned\_role\_users(r_1) = \{u_1\}$$

Now we use this prohibition constraint scheme to check a series of constraint requests.

Constraint request1: $q = (u_2, r_2, RUA)$.

Scope element evaluation:

$$u_2 \in \{u_1, u_2, u_3\} \Rightarrow \text{scope applicable check} = \text{Applicable}$$
AND
$$EN((SRF(CS) \cup CRS(q)) \cap SS) =$$
$$EN((assigned\_role\_users(\{r_1, r_2, r_3\}) \cup \{u_2\}) \cap \{u_1, u_2, u_3\}) =$$
$$EN((\{u_1\} \cup \{u_2\}) \cap \{u_1, u_2, u_3\}) = EN(\{u_1, u_2\}) = 2 < 3$$
$$\Rightarrow ESE(q) = \text{Permit}$$

Constraint element evaluation:

$$r_2 \in \{r_1, r_2, r_3\} \Rightarrow \text{constraint applicable check} = \text{Applicable}$$
AND
$$EN((CRF(CRS(q)) \cup CRO(q)) \cap CS) =$$
$$EN((assigned\_user\_roles(u_2) \cup \{r_2\}) \cap \{r_1, r_2, r_3\}) =$$
$$EN((\{\} \cup \{r_2\}) \cap \{r_1, r_2, r_3\}) = EN(\{r_2\}) = 1 < 2$$
$$\Rightarrow ECE(q) = \text{Permit}$$

Constraint scheme evaluation:

$$ECE(q) = ESE(q) \wedge ECE(q) = \text{Permit} \wedge \text{Permit} \Rightarrow \text{Permit}$$

So this request is permitted. After r_2 is assigned to u_2, there are:

$$assigned\_user\_roles(u_1) = \{r_1\}$$
$$assigned\_user\_roles(u_2) = \{r_2\}.$$
$$assigned\_role\_users(r_1) = \{u_1\}$$
$$assigned\_role\_users(r_2) = \{u_2\}$$

Constraint request2: $q = (u_1, r_2, RUA)$.

Scope element evaluation:

$u_1 \in \{u_1, u_2, u_3\} \Rightarrow$ scope applicable check = Applicable
AND
$EN((SRF(CS) \cup CRS(q)) \cap SS) =$
$EN((assigned\_role\_users(\{r_1, r_2, r_3\}) \cup \{u_1\}) \cap \{u_1, u_2, u_3\}) =$
$EN((\{u_1, u_2\} \cup \{u_1\}) \cap \{u_1, u_2, u_3\}) = EN(\{u_1, u_2\}) = 2 < 3$
$\Rightarrow ESE(q) =$ Permit

Constraint element evaluation:

$r_2 \in \{r_1, r_2, r_3\} \Rightarrow$ constraint applicable check = Applicable
AND
$EN((CRF(CRS(q)) \cup CRO(q)) \cap CS) =$
$EN((assigned\_user\_roles(\{u_1\}) \cup \{r_2\}) \cap \{r_1, r_2, r_3\}) =$
$EN((\{r_1\} \cup \{r_2\}) \cap \{r_1, r_2, r_3\}) = EN(\{r_1, r_2\}) = 2$
$\Rightarrow ECE(q) =$ Deny

Constraint scheme evaluation:

$ECE(q) = ESE(q) \wedge ECE(q) =$ Permit $\wedge$ Deny $\Rightarrow$ Deny

So this request is denied, and role r_2 cannot be assigned to user u_1.

5. Expressive power of constraint schemes

In this section, we demonstrate the expressive power of our constraint schemes by showing how they can be used to express a variety of constraints. For comparative purposes, we indicate the correspondence between our examples and those in the paper by Jaeger and Tidswell [19], which provides probably the most comprehensive set of examples in the literature.

Example 1. A user-user conflict separation of duty constraint. It is forbidden for two users to both be assigned to any common authorization type (role). In this case, it belongs to the user-based separation of duty and the constraint set is a subset of *USERS*. This constraint is expressed as:

((*R*, , ,), ({u_1, u_2}, *assigned_role_users*, <, 2), *SPC*).

It requires that no role defined in the scope set *R* can be assigned to both u_1 and u_2.

Example 2. A privilege-privilege conflict separation of duty constraint. It is forbidden for two permissions to both be assigned to a common authorization type (role). It belongs to the permission-based separation of duty constraint and the constraint set is a subset of *PERMS*. This constraint is expressed as:

((*R*, , ,), ({p_1, p_2}, *assigned_role_permissions*, <, 2), *SPC*).

It requires that no role defined in the scope set *R* can be assigned to both p_1 and p_2.

Example 3. A role-role conflict separation of duty constraint. It is forbidden for two roles to both be assigned to the same user. It belongs to the role-based separation of duty constraint and the constraint set is a subset of *ROLES*. This constraint is expressed as:

((*U*, , ,), ({r_1, r_2}, *assigned_user_roles*, <, 2), *SPC*).

It requires that no user defined in the scope set *U* can be assigned to both r_1 and r_2.

Example 4. A user can access one permission or the other, but not both. This constraint can be used to enforce a Chinese Wall restriction. That is, the history of users granted permission p_1 must not overlap with the history of users granted permission p_2. It belongs to the permission-based separation of duty constraint and the constraint set is a subset of *PERMS*. This constraint can be expressed as:

((*U*, , ,), ({p_1, p_2}, *ever_assigned_user_permissions*, <, 2), *HPC*).

It requires that no user defined in the scope set *U* can be assigned to both p_1 and p_2.

Example 5. A object-object conflict separation of duty constraint. It is forbidden for two objects to both be assigned to a common authorization type (role). It belongs to the object-based separation of duty constraint and the constraint set is a subset of *OBS*. This constraint can be expressed as:

((*R*, , ,), ({o_1, o_2}, *assigned_role_objects*, <, 2), *SPC*).

It requires that no role defined in the scope set *R* can be assigned to both o_1 and o_2.

Example 6. A user is restricted from accessing an object more than once. It belongs to the object-based separation of duty constraint and the constraint set is a subset of *OBS*. This constraint can be expressed as:

((*U*, , ,), ({o_1}, *ever_assigned_user_objects*, <, 2), *HPC*).

It requires that each user defined in the scope set *U* can only be assigned to o_1 less than two times.

Example 7. An alternative interpretation of the user-user conflict constraint expressed in Example 1 in which two sets of users are restricted form being assigned to any common authorization type (role). This constraint can be expressed with two prohibition constraint schemes specifying that two teams are restricted from being assigned to any common user or authorization type (role). In both constraint schemes, the constraint sets are the subsets of *TEAMS*. The constraint scheme

(({u_1, u_2, …, u_n}, , ,), ({m_1, m_2}, *assigned_user_teams*, <, 2), *SPC*)

specifies that no user is assigned to both m_1 and m_2. The constraint scheme

(({r_1, r_2, …, r_n}, , ,), ({m_1, m_2}, *assigned_role_teams*, <, 2), *SPC*)

specifies that no role is assigned to both m_1 and m_2.

Example 8. Another user-user conflict separation of duty constraint. Here two users are restricted from sharing any authorization type (role) in the set of restricted types (roles). In this case, it belongs to the user-based separation of duty and the constraint set is a subset of *USERS*. The constraint can be expressed as:

(({r_1, r_2, …, r_n}, , ,), ({u_1, u_2}, *assigned_role_users*, <, 2), *SPC*).

It requires that no role defined in the scope set {r_1, r_2, …, r_n} can be assigned to both u_1 and u_2.

Example 9. A user-role conflict separation of duty constraint. A user or set of users are prohibited from being assigned to any authorization type (role) in a set. It belongs to the role-based separation of duty constraint and the constraint set is a subset of *ROLES*. This constraint can be expressed as:

(({u_1, …, u_n}, , ,), ({r_1, r_2, …, r_m}, *assigned_user_roles*, <, 1), *SPC*).

It requires that no user defined in the scope set {u_1, …, u_n} can be assigned to any role defined in the constraint set{ r_1, r_2, …, r_m}.

Example 10. Operational separation of duty. In this constraint, no user is permitted to obtain all the permissions necessary to perform all the tasks in a process. Typically, each task in a process is represented by an authorization type (role), and the execution of a task is assumed to be equivalent to the invocation of a sequence of permissions assigned to an authorization type (role). In this case, it belongs to the role-based separation of duty constraint and the constraint set is a subset of *ROLES*. This constraint can be expressed as:

((U, , ,), ({r_1, …, r_n}, *assigned_user_roles*, <, m), *SPC*).

It requires that each user defined in the scope set U can only be assigned to less than m roles defined in the constraint set {r_1, …, r_n}, where $1 < m \le n$.

Example 11. A session-dependent separation of duty constraint. In this constraint, all the users in an aggregate are prevented from being assigned to all the authorization types (roles) during their sessions. In this case, it belongs to the role-based dynamic separation of duty constraint and the constraint set is a subset of *ROLES*. This constraint can be expressed as:

((U, , ,), ({r_1, …, r_n}, *session_user_roles*, <, m), *DPC*).

It requires that each user defined in the scope set U can only activate less than m roles defined in the constraint set {r_1, …, r_n} in a session, where $1 < m \le n$.

Example 12. A universal quantification. In this constraint, all users are restricted from being assigned to more than one conflicting authorization type (role). In this case, it belongs to the role-based separation of duty constraint and the constraint set is a subset of *ROLES*. The constraint can be expressed as:

((*USERS*, , ,), ({r_1, …, r_n}, *assigned_user_roles*, <, 2), *SPC*).

It requires that no user can be assigned to more than one role defined in the constraint set {r_1, …, r_n}.

Example 13. Reconsider the Example 10 in the context of authorization type (role) hierarchy. In this case, it belongs to the static role-based separation of duty constraint and the constraint set is a subset of *ROLES*. The constraint can be expressed as:

((U, , ,), (*authorized_role_roles*({r_1, …, r_n}), *authorized_user_roles*, <, m), *SPC*).

It requires that each user defined in the scope set U can only be assigned to less than m roles defined in the constraint set {r_1, …, r_n} and the roles inherited by these roles. The function *authorized_role_roles* returns the roles that are inherited by a given role.

Example 14. Kuhn [14] identified that there may exist a mutual exclusion between authorization types whereby some permissions not involved in the mutual exclusion may be shared. In this constraint a mutual exclusion is set between types A' and B', but the inheritance of this constraint only excludes the permissions of B' from A and A' from B. It is possible for A and B to each inherit permissions from another authorization type C as long as the permissions inherited from C are disjoint from those in A' and B'. It means only part of the roles gotten through role hierarchies are mutually exclusive. This kind of constraints cannot be directly supported by our schemes. But we can use several simple constraint schemes to accomplish this complex constraint. For the previous example, the constraint can be expressed with next four constraint schemes:

((PERMS, , ,), ({A', B'}, *authorized_user_roles*, <, 2), SPC),
((*A*, , ,), ({B' }, *authorized_role_roles*, <, 1), SPC),
((*B*, , ,), ({A'}, *authorized_role_roles*, <, 1), SPC),
((*C*, , ,), ({A', B'}, *authorized_role_roles*, <, 1), SPC).

Example 15. Both order-dependent and order-independent history constraints specify that a certain history must have taken place before an operation can be executed. For example, two *sign signature* tasks must be executed before the *approve task* can be performed in a workflow instance. In this case, it belongs to the order-independent historical task-based obligation constraint and the constraint set is a subset of *TASKS*. Here we assume that the sign signature tasks are t_1 and t_2, the approve task is t_3. The constraint can be expressed as:

(*USERS*, {t_3}, ({t_1, t_2}, *ever_performed_tasks*, >, 2), *HOC*).

It is a historical obligation constraint that requires that any user who can perform t_3 only when t_1 and t_2 have been performed. The function *ever_performed_tasks* is used to obtain the performed tasks in a workflow instance. For this function input parameters, the user-id is not important, but the workflow-instance-id must be provided. If we want the

performing sequence is t_1, t_2 and t_3, then we can use two obligation constraint schemes to specify the execution order. The constraint scheme

(*USERS*, {t_2}, ({t_1}, *ever_performed_tasks*, >, 0), *HOC*)

requires that any user can perform t_2 only when t_1 has been performed. The constraint scheme

(*USERS*, {t_3}, ({t_2}, *ever_performed_tasks*, >, 0), *HOC*)

requires that any user can perform t_3 only when t_2 has been performed.

6. Implementation

In this section we first introduce the constraint schema that organizes a set of constraint schemes for authorization constraint check, and then describe the implementation of the function-based authorization constraint system.

6.1 Constraint schema

Constraint schema is the basic unit of managing constraint schemes and checking constraint requests. The combining algorithm for these constraint schemes are "deny-overrides" that is described in the following.

In the entire set of schemes in a schema, if any scheme evaluates to "Deny", then the result of the scheme combination is "Deny". If any scheme evaluates to "Permit" and all other schemes evaluate to "NotApplicable", then the result of the scheme combination is "Permit". If all schemes evaluate to "NotApplicable", then the scheme combination is "NotApplicable". If an error occurs while evaluating a scheme, then the scheme combination is "Indeterminate".

Now we illustrate how to use prohibition constraint schemes and obligation constraint schemes to construct a constraint schema through an example. The application scenario is: all the users (*U*) with the role "Staff" can be assigned to the role "President" or "Vice-President", there is only one user can be assigned to the role "President", there are no more than two users can be assigned to the role "Vice-President", and role "President" and role "Vice-President" are mutually-exclusive roles. In order to implement these constraints, the corresponding constraint schema should include the following schemes:

(*U*, {*President*, *Vice-President*}, ({*Staff*}, *assigned_user_roles*, >, 0), *SOC*);

((*U*, *assigned_role_user*, <, 2), ({*President*}, *assigned_user_roles*, <, 2), *SPC*);

((*U*, *assigned_role_user*, <, 3), ({*Vice-President*}, *assigned_user_roles*, <, 2), *SPC*);

((*U*, , ,), ({*President*, *Vice-President*}, *assigned_user_roles*, <, 2), *SPC*).

The first scheme specifies only users with the role "Staff" can be assigned to roles "President" and "Vice-President". The second scheme specifies less than two users can be assigned to the role "President". The third scheme specifies less than three users can be assigned to the role "Vice-President". The fourth scheme specifies that "President" and "Vice-President" are mutually-exclusive roles. In order to enforce these constraint schemes, the functions that should be implemented are *get_employees*, *assigned_user_roles* and *assigned_role_users*.

6.2 Implementation

We have implemented the function-based authorization constraint system. This system is developed with Java and named *authorization constraint monitor*. The essential class relations of the authorization constraint monitor is shown Figure 2. *ConstraintSPC*, *ConstraintDPC*, *ConstraintHPC*, *ConstraintSOC*, *ConstraintDOC* and *ConstraintHOC* are classes used to hold and enforce the six types of constraint schemes, respectively. All the prohibition constraint scheme classes inherit from the superclass *ConstraintPC*. All the obligation constraint scheme classes inherit from the superclass *ConstraintOC*. *ConstraintPC* and *constraintOC* further inherit from the superclass *Constraint*. *ConstraintSchema* is the class for holding and enforcing constraint schemas. Their functionalities are described as follows:

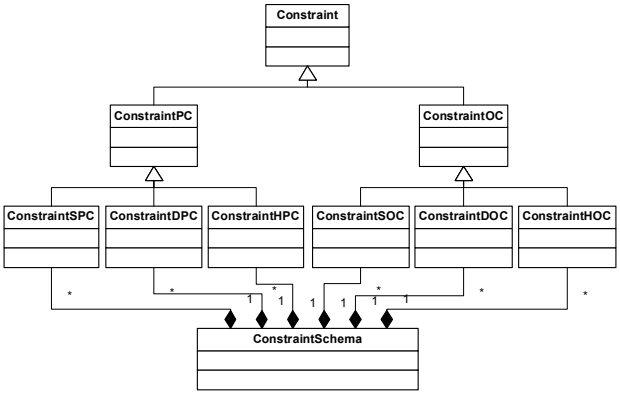

Figure 2. Essential class relations of constraint monitor

- *ConstraintSchema* serves as an interface for the constraint monitor, that is, it hides the internal structures from other components that use this service. Thus, each external component uses constraint service through a well-defined API offered by the *ConstraintSchema*. Each *ConstraintSchema* instance holds only one constraint schema that specifies which constraint schemes must be satisfied before agreeing one constraint request.
- *ConstraintSPC*, *ConstraintDPC*, *ConstraintHPC* are used to hold and evaluate static prohibition schemes, dynamic prohibition schemes and historical prohibition schemes, respectively. Each class instance holds one scheme. They are responsible for extracting data from the constraint request and invoking the methods defined in the superclass to complete the corresponding prohibition constraint scheme evaluation.
- *ConstraintSOC*, *ConstraintDOC*, *ConstraintHOC* are used to hold and evaluate static obligation schemes, dynamic obligation schemes and historical obligation schemes, respectively. Each class instance holds one scheme. They are responsible for extracting data from the constraint request and invoking the methods defined

in the superclass to complete the corresponding obligation constraint scheme evaluation.

- *ConstraintPC* defines the variables used to hold a prohibition constraint scheme and the methods used to implement the logic for evaluating a prohibition scheme.
- *ConstraintOC* defines the variables used to hold an obligation constraint scheme and the methods used to implement the logic for evaluating an obligation scheme.
- *Constraint* defines the common variables and methods used by both *ConstraintPR* and *ConstraintSD*. The entity set functions and entity relation functions are also implemented or registered in this class.

Constraint request is evaluated by a constraint monitor against to a given constraint schema that contains a set of constraint schemes. The evaluation result could be "Deny", "Permit", "NotApplicable" or "Indeterminate".

7. Applications

Our function-based authorization constraints system has been integrated into some real application systems. In this section, we briefly introduce two application cases. Both of them are from a R&D project that cooperates with the Beijing Shenzhou Aerospace Software Technology Co., Ltd (www.bjsasc.com) that is a subcompany of the China Aerospace Science and Technology Corporation (CASC).

Case 1. Constraints in management information system. The first project is "CASC Material Management Information System Integration Tool". As a whole, the "Material Management Information System" comprises many subsystems. Currently, each subsystem has its own authentication and authorization systems. In order to access different functionalities, users have to logon different subsystems separately. This integration tool will provide *single sign on* function for all the subsystems. The authorization mechanism adopted by the integration tool is role-based access control. Due to a very big system and the requirement of fine-grained access control, many roles and subsystems have been defined. Because their businesses are special and sensitive, one of the most important issues that must be considered is information security, such as separation of duty. Basically, the authorization constraints to the privilege management are summarized as follows.

- Mutually exclusive roles cannot be assigned to the same user.
- Mutually exclusive permissions (subsystems) cannot be assigned to the same role.
- Mutually exclusive permissions (subsystems) cannot be assigned to the same user.
- Some roles have cardinality constraints.
- Prerequisite roles are required in some user-role assignments and prerequisite permissions (subsystems) are required in some role-permission assignments.

Here both prohibition constraints and obligation constraints are involved. Obviously, it is better to have some mechanism that can automatically detect any improper privilege assignments. This issue is addressed by our function-based authorization constraints. In this project we developed an authorization constraint module that is invoked by the integration tool through its API. All the user-role assignments, role-permission assignments and user-permission assignments will be checked by the module according to some authorization constraint rules. With the help of the authorization constraint module, any intentional or unintentional privilege assignment that could cause a violation of an authorization constraint will be avoided.

Case 2. Constraints in workflow management system. The second project is "CASC Material Management Information System". This system manages the materials' whole lifecycle such as planning, purchase, storage, distributing, and so forth. The system development adopts the Actionsoft Workflow Suite (AWS) that is a Business Process Management (BPM) application developing platform (http://www.actionsoft.com.cn). AWS provides comprehensive functionalities for workflow design, running and maintenance. But some complex application scenarios cannot be directly implemented by AWS, e.g. authorization constraints in a workflow instance.

In order to automatically search the operators for different nodes in a workflow instance, the workflow designers need to set a route policy for each node. The AWS platform predefines many route schemes. For example, one route scheme permits the system automatically looks for a valid operator according to users' roles. But these route schemes do not consider any kind of authorization constraints, such as separation of duty. When the role-based route scheme is adopted, it is possible that all the steps are performed by the same user in a workflow instance if he/she has all the required roles. Our authorization constraint mechanism can be used to avoid such kind of situations happening.

In this project we developed an authorization constraint monitor that is responsible for enforcing authorization constraints in workflow systems at runtime. Each constrained request that could potentially cause a violation of an authorization constraint is passed to the constraint monitor. The constraint monitor checks whether granting the request would violate an authorization constraint rules and takes appropriate action. This authorization constraint monitor is integrated into the workflow systems through AWS developing interface. With the help of authorization constraint monitor, various authorization constraints, such as separation of duty in workflow instances, can be realized.

8. Conclusion

The major contribution of this paper is providing two novel authorization constraint specification schemes that can be used for both expressing authorization constraints and enforcing authorization constraints in different access control models. To our knowledge ours is the first constraint specification language that can be used for both expression and enforcement aims. These constraint schemes are strongly bound to some functions that could be directly mapped to the functions that should be developed in the application systems. Thus, these schemes can provide the system designers and security administrators a clear view

about which functions should be defined in an authorization constraint system. Based on these functions, various constraint schemes can be easily defined, and then enforced. We believe that our approach is far simpler to understand, has a much less cumbersome syntax and more closer to the real world. Moreover, the implementation can be based on the functions that already exist in an application system, and an authorization constraint system could be scalable through adding new entity set functions and entity relation functions. On the hand, this approach goes beyond the well known separation of duty constraints, and considers many aspects of entity relation constraints.

References

[1] D. W. Chadwick, W. Xu, S. Otenko, R. Laborde, B. Nasser. "Multi-Session Separation of Duties (MSoD) for RBAC". In *Proceedings of the First International Workshop on Security Technologies for Next Generation Collaborative Business Applications (SECOBAP'07)*, Istanbul, Turkey, April 2007.

[2] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. "Role-based access control models", *IEEE Computer*, vol. 29, pp. 38-47, Feb. 1996.

[3] E. C. Lupu, M. Sloman. "A policy based role object model". In *Proceedings of the 1st IEEE Enterprise Distributed Object Computing Workshop*, Calif, Oct. 1997.

[4] L. Giuri, P. Iglio. "Role templates for content-based access control". In *Proceedings of the 2nd Workshop on Role-Based Access Control*, 1997.

[5] E. Bertino, E. Ferrari, V. Atluri. "The specification and enforcement of authorization constraints in workflow management systems", *ACM Trans. Inf. Syst. Sec. (TISSEC)* 1, 2, Feb. 1999.

[6] R. S. Sandhu, V. Bhamidipati, Q. Munawer. "The ARBAC97 model for role-based administration of roles", *ACM Trans. Inf. Syst. Sec*. 1, 2, Feb. 1999.

[7] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, R. Chandramouli. "Proposed NIST standard for role-based access control", *ACM Transactions on Information and System Security*, vol. 4, pp. 224-274, Aug. 2001.

[8] J. H. Saltzer, M. D. Schroeder. "The protection of information in computer systems". In *Proceedings of the IEEE*, 63(9):1278-1308, September 1975.

[9] D. D. Clark, D. R. Wilson. "A comparision of commercial and military computer security policies". In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, pp. 184-194, May 1987.

[10] M. Bishop. *Computer Security — Art and Science*, Addison-Wesley, 2003.

[11] R. Simon, M. E. Zurko. "Separation of duty in role based access control environments". In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, pp. 183-194, Rockport, MA, June 1997.

[12] L. Giuri, P. Iglio. "A formal model for role-based access control with constraints". In *Proceedings of 9th IEEE Workshop on Computer Security Foundations*, pp. 136-145, Kenmare, Ireland, June 1996.

[13] V. D. Gligor, S. Gavrila, D. Ferraiolo. "On the formal definition of separation of duty policies and their composition". In *Proceedings of the 1998 IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 172-183, Oakland, CA, May 1998.

[14] R. Kuhn. "Mutual exclusion as a means of implementing separation of duty requirements in role based access control systems". In *Proceedings of the Second ACM Workshop on Role Based Access Control*, pp. 23-30, 1997.

[15] E. Lupu, M. Sloman. "Conflicts in policy-based distributed systems management", *IEEE Trans. Softw. Eng*. 25, 6, Nov./Dec., 1999.

[16] G. Ahn and R. Sandhu. "Role-based authorization constraint specification", *ACM Trans. Inf. Syst. Sec*. 3, 4 (Nov.), 2000.

[17] M. Nyanchama, S. Osborn. "The role graph model and conflict of interest", *ACM Transactions on Information and System Security* 2, 1, pp. 3-33, 1999.

[18] S. Osborn, Y. Guo. "Modelling users in role-based access control". In *Proceedings of the 5th ACM Role-Based Access Control Workshop*, July 2000.

[19] T. Jaeger and J. Tidswell. "Practical safety in flexible access control models", *ACM Transactions on Information and System Security* 4, 2, pp. 158-190, 2001.

[20] J. Crampton, "Specifying and enforcing constraints in role-based access control". In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, pp. 43-50, Como, Italy, June 2003.

[21] N. Li, Z. Bizri, M. V. Tripunitara. "On mutually exclusive roles and separation of duty". In *Proceedings of the CCS'04*, pp. 42-51, Washington, DC, USA, October 2004.

[22] W. Tolone, G. Ahn, T. Pai, "Access Control in Collaborative Systems", *ACM Computing Surveys*, Vol. 37, No. 1, pp. 29-41, March 2005.

[23] W. Zhou, V. H. Raja, C. Meinel, M. Ahmad. "A Framework for Cross-Institutional Authentication and Authorisation". In *Proceedings of the eChallenges e-2005 Conference (e-2005)*, pp. 1259-1266, Ljubljana, Slovenia, October 2005.

[24] W. Zhou, C. Meinel. "Team and Task Based RBAC Access Control Model". In *Proceedings of the 5th Latin American Network Operations and Management Symposium (LANOMS 2007)*, pp. 84-94, Petrópolis, Brazil, September 2007.

Author Biographies

Wei Zhou received his BS and MS degrees in computer science in 1992 and 1997, respectively, both from the Jilin University of Technology, China. He is a PhD student at the Hasso-Plattner-Institute (HPI), University of Potsdam, Germany. He has rich work experience as a computer engineer in enterprises and government (China Customs). From 2004 to 2006, he worked at the University of Warwick, United Kingdom. His research interests include e-commerce, e-government, information security, and Internet applications.

Prof. Dr. Christoph Meinel is director of the Hasso-Plattner-Institute (HPI), and head of the chair "Internet Technologies and System". His main work concerns security engineering technologies and methods, telemedicine applications and innovative forms of teaching (Teleteaching) and learning (E-Learning).

Yidong Xiang received his BS degree in computer science in 2004 from the Northeast Dianli University, China. Currently, he is a senior J2EE Architect at Beijing Shenzhou Aerospace Software Technology Co., Ltd, China.

Yang Shao received his BS degree in applied physics in 1999 from the University of Tianjin, China. Currently, he is chief technology officer at the Division of Integration Information Administration, Beijing Shenzhou Aerospace Software Technology Co., Ltd, China.