Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Conflict detection and resolution for authorization policies in workflow systems*

Chen-hua MA[†1], Guo-dong LU[1], Jiong QIU[2]

([1]*Engineering & Computer Graphics Institute, Zhejiang University, Hangzhou 310027, China*)

([2]*Department of Computer Engineering, Hangzhou Dianzi University, Hangzhou 310008, China*)

[†]E-mail: mchma@zju.edu.cn

Received May 12, 2008; Revision accepted Sept. 19, 2008; Crosschecked Apr. 10, 2009

Abstract: The specification of authorization policies in access control models proposed so far cannot satisfy the requirements in workflow management systems (WFMSs). Furthermore, existing approaches have not provided effective conflict detection and resolution methods to maintain the consistency of authorization polices in WFMSs. To address these concerns, we propose the definition of authorization policies in which context constraints are considered and the complicated requirements in WFMSs can be satisfied. Based on the definition, we put forward static and dynamic conflict detection methods for authorization policies. By defining two new concepts, the precedence establishment rule and the conflict resolution policy, we provide a flexible approach to resolving conflicts.

Key words: Workflow management system (WFMS), Authorization policy, Conflict detection and resolution
doi:10.1631/jzus.A0820366 **Document code:** A **CLC number:** TP39

INTRODUCTION

Workflow management systems (WFMSs) have attracted a lot of interest within both academia and the business community in recent years. A workflow is a collection of tasks which are organized to facilitate some business process specification, re-engineering and automation (Georgakopoulos *et al.*, 1995). WFMSs make it possible to use business processes within a computerized system, thereby gaining significant improvements in efficiency. Access control is an important mechanism required for secure WFMSs. Via the specification of authorization policies, information security can be guaranteed by preventing objects from being granted illegal access rights (permissions). Authorization policies aim to define or constrain the current or future behavior of objects to ensure that their actions are aligned with the objectives of the enterprise.

In the past decade, much work (Ferraiolo *et al.*, 1995; 2001; Sandhu *et al.*, 1996) has been done on role-based access control (RBAC), in which subjects and permissions are separated logically by the introduction of roles to greatly simplify the management of authorization. Authorization policies in RBAC can be represented as a 3-ary tuple $\{r, o, a\}$, indicating that role r can perform operation a on object o. Such authorization policies are passive or static, able to cause immediate activation of permissions but incapable of satisfying the requirements for access control in WFMSs. However, WFMSs need active, dynamic authorization policies. In this case, a permission can be granted to subjects only when the task corresponding to the permission is activated, and the permission should be revoked immediately when the task is finished. To address this issue, considerable related research has been done, such as the workflow authorization model (WAM) (Atluri and Huang, 1996;

* Project supported by the National Natural Science Foundation of China (Nos. 50705084 and 60473129) and the Science and Technology Plan of Zhejiang Province, China (No. 2007C13018)

2000; Huang and Atluri, 1999), task-based authorization control (Thomas and Sandhu, 1997), and task-role based access control (Oh and Park, 2003). These models provide support for active, dynamic authorization policy specification. For example, authorization policies in WAM can be represented as a 4-ary tuple $\{tu, r, o, a\}$, indicating that role r can perform operation a on object o only during the lifetime of task tu. However, several important issues have not been covered by these models:

(1) Context constraints (e.g., time, location, history information, and user qualification) have largely been overlooked in the specification of authorization policies in these models. A context constraint is a logical expression, defining the conditions that need to be satisfied for the activation of an authorization policy. For example, standardization engineers can be granted permissions for drawing standardization during the lifetime of task 'standardize drawing' only when they are not the designer or proof-reader of the drawing.

(2) Only separation-of-duty constraints are specified in these models, such as mutually exclusive roles (Huang and Atluri, 1999) and mutually exclusive tasks (Oh and Park, 2003), insufficient to meet the complicated requirements of security policies in workflow systems.

(3) A workflow system may span multiple organization units, each having its own security requirements. Thus, the number of authorization policies may grow very large and the interactions between them will be complex and difficult to predict. When the objectives of two or more policies cannot be simultaneously met, conflicts will occur, and in this case, there may exist no workflow execution that satisfies all the policies. Therefore, conflict detection and resolution is a significant research challenge. However, solutions to this problem have not been discussed in these models.

Dunlop *et al.*(2002; 2003) studied the conflict detection and resolution in policy-based management systems. By specifying the temporal characteristics of policies and establishing conflict databases, a scalable conflict detection mechanism for policies is presented. He *et al.*(2005) classified the conflicts in policy-based management systems and proposed detection approaches for modality and application correlative conflicts. However, context constraints have also been overlooked in the specification of policies in the above literature. Moreover, these proposed methods of conflict-resolving are to identify which policy involved in a conflict situation will take precedence, such as specific-overrides-general, negative-overrides-positive, and each of them can produce only valid results in some circumstances. How to integrate these methods and establish an order of priority for them is still a research challenge.

All the analysis above shows that existing approaches to the specification of authorization policies cannot satisfy the requirements in WFMSs and the methods proposed so far have not provided effective conflict detection and resolution methods for authorization polices. To address the problems, we propose in this paper the following approaches:

(1) A new type of constraint, context constraint, is proposed since context constraints can meet the complicated requirements of security policies in WFMSs.

(2) We give the definition of authorization policies in WFMSs, in which context constraints are considered.

(3) Effective static and dynamic conflict detection methods for authorization policies in WFMSs are provided.

(4) We bring forth a flexible approach to resolving conflicts by defining two new concepts, the precedence establishment rule and the conflict resolution policy. Security administrators can flexibly decide the methods for conflict resolution according to system requirements.

OVERVIEW OF RBAC96 MODEL CONCEPTS

We will base our discussion on RBAC96 (Sandhu *et al.*, 1996), the most well known role-based access control model. In particular, the RBAC1 model is adopted owing to its support on the use of a role hierarchy. In this section, we provide an overview of the concepts within the model.

Central to the model are the concepts of User, Role, Role Hierarchy, Permission, Session, User Assignment, and Permission Assignment. A user is a human being or an intelligent autonomous agent. A role is a job function or job title within an organization with some associated semantics regarding the

authority and responsibility conferred on a member of the role. A role hierarchy is a partially ordered set of roles, able to be represented by a directed acyclic graph as shown in Fig.1, where each role is associated with a node. If role r_i inherits the permissions of role r_j, denoted as $r_i \geq r_j$, we refer to r_i as the senior role of r_j, r_j as the junior role of r_i. The sets of senior and junior roles of r_i are denoted as $Senior(r_i)$ and $Junior(r_i)$ respectively in the following sections.

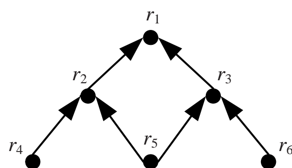

Fig.1 The role hierarchy graph

A permission is an approval of a particular mode of access to one or more objects in the system. A permission p can be formally described as $\{obt, op\}$, where *obt* refers to an object type, and *op* is an access mode for object type *obt*. Each session is a mapping of one user to many possible roles; i.e., a user establishes a session during which the user activates some subset of roles of which he or she is a member. Users are associated with roles using the user assignment relation $UA \subseteq U \times R$, while permissions are linked with roles using the permission assignment relation $PA \subseteq P \times R$.

CONTEXT CONSTRAINTS

Authorization constraints proposed so far can be classified into four classes (Bertino, 2003):

(1) Separation-of-duty constraints: Separation-of-duty constraints aim at reducing the risk of fraud by not allowing any individual to have sufficient authority within the system to perpetrate a fraud on his/her own, such as mutually exclusive roles.

(2) Cardinality constraints: Cardinality constraints aim at providing constraints on some entities, such as role cardinality and task instance cardinality.

(3) Prerequisite constraints: Prerequisite constraints aim at ensuring the authority subjects to have enough competency, knowledge, and capability. For example, a user can be assigned to role *A* only if the user is already a member of role *B*. Role *B* is the prerequisite role of role *A*.

(4) Temporal constraints: Temporal constraints aim at temporally providing constraints on enabling or disabling roles, and are supported by associating time intervals, possibly periodic, with roles. An enabled role can be activated by users, whereas a disabled role cannot be activated by users.

These constraints cannot express complicated requirements of security policies in WFMSs however. For example, auditors and technical managers can approve drawings only when they are on duties and from local computers. In order to address the issue, a new type of constraint, context constraint, is introduced in this paper. Context constraints are conditions that need to be satisfied when subjects are granted related permissions.

Definition 1 (Context predicate) Context is any environmental factor, capable of influencing or controlling when and how an authorization policy is enforced. A context predicate *context* is defined as a 3-ary tuple: *context*: $\{type, predicate, (obj_1, obj_2, \ldots, obj_n)\}$, where *type* refers to the type of *context* such as time and location, $(obj_1, obj_2, \ldots, obj_n)$ the set of objects with which the context is concerned, and *predicate* the character of objects $(obj_1, obj_2, \ldots, obj_n)$ or the relationship among them.

For example, *context*: {time, (between, 09:00, 17:00), *access_time*} is a time context, stating that the access time is between AM 09:00 and PM 17:00, and *context*: {user, ≠, (*accessdrawing_designer*, *access-drawing_proofreader*)} is a user context, indicating that the designer and the proof-reader of an access drawing cannot be the same person.

Definition 2 (Context constraint) A context constraint *ac* is defined as a logical conjunction of context predicates, which is denoted as *ac*: $L_1|L_2|\ldots|L_i|\neg L_{i+1}|\neg L_{i+2}|\ldots|\neg L_n$. L_i represents context predicates defined above, '|' denotes '$\wedge$' or '$\vee$', and '$\neg L_i$' indicates the negative value of L_i. Context constraints can express complicated requirements of security policies in WFMSs. For example, the following context constraint states that the access date is between Monday and Friday, the access time is between AM 09:00 and PM 17:00, and the access address is local:

ac: {date, (between, Monday, Friday), *access_date*}∧{time, (between, 09:00, 17:00), *access_time*}∧{location, (is, local), *access_IP_address*}.

Definition 3 (Intersectant context constraints) For two context constraints ac_i and ac_j, if the contextual

information specified by ac_i overlaps with the contextual information specified by ac_j (e.g., an overlap between two time intervals and another overlap between two locations), then they are intersectant context constraints, denoted as *Inter*(ac_i, ac_j).

Definition 4 (Non-intersectant context constraints) For two context constraints ac_i and ac_j, if one of the following conditions is satisfied, then they are non-intersectant context constraints, denoted as *NonInter*(ac_i, ac_j):

(1) $ac_i \rightarrow \neg ac_j$. The reverse of ac_j is inferable by ac_i; that is, wherever ac_i is satisfied, ac_j is not satisfied.

(2) $ac_j \rightarrow \neg ac_i$. The reverse of ac_i is inferable by ac_j; that is, wherever ac_j is satisfied, ac_i is not satisfied.

For example, consider the following three context constraints: ac_1: {time, (between, 08:00, 10:00), *access_time*}, ac_2: {time, (between, 09:00, 10:00), *access_time*}, ac_3: {time, (between, 11:00, 17:00), *access_time*}. *Inter*(ac_1, ac_2), *NonInter*(ac_1, ac_3); i.e., ac_1 and ac_2 are intersectant constraints, while ac_1 and ac_3 are non-intersectant constraints. A knowledge-based or a rule-based inference mechanism can be employed to perform the above inference.

DEFINITION OF AUTHORIZATION POLICY IN WFMSs

The specification of authorization policies in previous research cannot satisfy the requirements in WFMSs. Accordingly we propose the following definition of authorization policy:

Definition 5 (Authorization policy) An authorization policy *ap* in WFMSs is defined as a 6-ary tuple *ap*: {*tu*, *R*, *P*, *type*, *inheritable*, *ac*}. Therein *tu* is the task associated with *ap*. A task is an atomic action within the context of a workflow instance. *R* represents the set of roles that can be granted permissions in *P* during the lifetime of *tu*, *P* is the set of permissions associated with *ap*, *type* denotes one of the policy types belonging to either positive '+' or negative '−', *inheritable* is used to identify whether *ap* is inheritable, and *ac* is the context constraint associated with *ap*, specifying the situation at which the authorization policy is active.

Definition 6 (Positive authorization policy) *ap*: {*tu*, *R*, *P*, +, *inheritable*, *ac*} as a positive authorization policy states that roles in *R* can be granted permissions in *P* during the lifetime of *tu* only if the context constraint *ac* is satisfied.

Definition 7 (Negative authorization policy) *ap*: {*tu*, *R*, *P*, −, *inheritable*, *ac*} as a negative authorization policy means that roles in *R* are forbidden to be granted permissions in *P* during the lifetime of *tu* if the context constraint *ac* is satisfied.

Definition 8 (Inheritable and non-inheritable authorization policies) If senior roles of each role in *R* can also be granted permissions in *P* during the lifetime of *tu*, then *ap* is inheritable. If senior roles of each role in *R* cannot be granted permissions in *P* during the lifetime of *tu*, then *ap* is non-inheritable.

In the following sections, let *tu*(*ap*), *R*(*ap*) and *P*(*ap*) represent the task, the set of roles and the set of permissions associated with *ap*, respectively. If *ap* is inheritable, then $R(ap) = R \cup (\bigcup_{r_i \in R} Senior(r_i))$; otherwise, *R*(*ap*)=*R*. Let *U*(*ap*) represent the set of users associated with *ap*, $U(ap) = \bigcup_{r_j \in R(ap)} AU(r_j)$, where $AU(r_j)$ is the set of users assigned to role r_j.

The example WFMS we will use throughout this work is a drawing management system. Fig.2 presents the workflow specification and Fig.3 shows roles and the role hierarchy. Authorization policies in the system are defined as in Table 1.

Definition 9 (Correlative authorization policies) For two authorization policies ap_i and ap_j, if with an overlapping scope specified by the following pre-condition: $(tu(ap_i)=tu(ap_j)) \wedge (R(ap_i) \cap R(ap_j) \neq \varnothing) \wedge (P(ap_i) \cap P(ap_j) \neq \varnothing)$, then they are correlative authorization policies, denoted as *correlate*(ap_i, ap_j).

For example, the policies ap_5 and ap_6 defined in Table 1 are correlative policies.

Definition 10 (Valid role set and valid user set of authorization policy) For an authorization policy ap_i, let *ValidRole*(ap_i) represent the valid role set of ap_i, composed of the roles in *R*(ap_i) that can satisfy the context constraint associated with ap_i; let *ValidUser*(ap_i) represent the valid user set of ap_i, consisting of the users in *U*(ap_i) that can satisfy the context constraint associated with ap_i.

For example, suppose that user-role assignments in the example system are determined as shown in Table 2.

Table 1 Authorization policies in example system

Authorization policy	Description
ap_1: {design drawing, designer, (drawing, design), +, inheritable}	ap_1 is a positive policy, stating that role 'designer' and its senior roles can be granted permission '(drawing, design)' during the lifetime of task 'design drawing'
ap_2: {design drawing, (proof-reader, standardization engineer, auditor), (drawing properties, read), +, inheritable}	ap_2 is a positive policy, stating that role 'proof-reader', 'standardization engineer', 'auditor' and their senior roles can be granted the permission for reading drawing properties during the lifetime of task 'design drawing'
ap_3: {proof-read drawing, proof-reader, (drawing, proof-read), +, non_inheritable, ac_1}, where ac_1: {user, ≠, (*accessdrawing_designer*, *accessdrawing_proofreader*)}	ap_3 is a positive policy, stating that role 'proof-reader' can be granted the permission for proof-reading drawings during the lifetime of task 'proof-read drawing' only if the proof-reader is not the designer of the access drawing
ap_4: {standardize drawing, standardization engineer, (drawing, standardize), +, non_inheritable, ac_2}, where ac_2: {user, ≠, (*accessdrawing_designer*, *accessdrawing_proofreader*, *accessdrawing_standardization_engineer*)}	ap_4 is a positive policy, stating that role 'standardization engineer' can be granted the permission for drawing standardization during the lifetime of task 'standardize drawing' only if the designer, proof-reader and standardization engineer of the access drawing are not the same user
ap_5: {approve drawing, (auditor, technical manager), (drawing, approve), +, non_inheritable, ac_3}, where ac_3: {user, ≠, (*accessdrawing_designer*, *accessdrawing_auditor*)}	ap_5 is a positive authorization policy, stating that technical managers and auditors can be granted permission '(drawing, approve)' during the lifetime of task 'approve drawing' if they are not the designer of the access drawing
ap_6: {approve drawing, auditor, (drawing, approve), −, non_inheritable, ac_4}, where ac_4: {user, (≥, 2), *accessdrawing_designer*}	ap_6 is a negative authorization policy, stating that auditors cannot be granted permission '(drawing, approve)' during the lifetime of task 'approve drawing' if the drawing is designed collaboratively

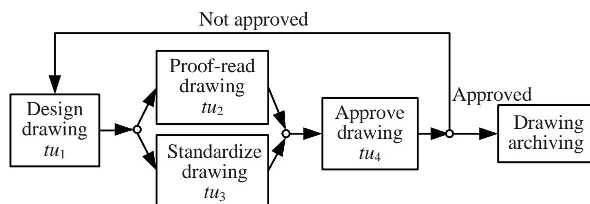

Fig.2 Example workflow specification

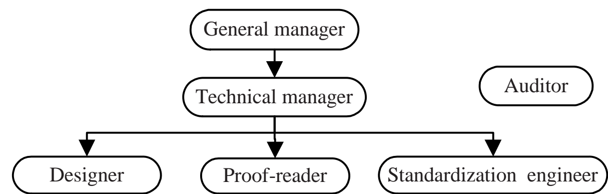

Fig.3 Example roles and role hierarchy

Table 2 Example user-role assignments

Role	User(s) assigned
General manager	Lu
Technical manager	Li
Standardization engineer	Fei, Cheng
Designer	Ma, Lei, Cheng
Proof-reader	Xu, Lei
Auditor	Liu, Yi

Let us assume that the designer, proof-reader and standardization engineer of drawing *t* are Li, Xu and Fei, respectively. In such a situation, the valid role set and the valid user set of policy ap_5 are {auditor} and {Liu, Yi}, respectively.

CONFLICT DETECTION FOR AUTHORIZATION POLICIES

Policy conflicts occur when the objectives of two or more policies cannot be simultaneously met. Effective conflict detection methods for authorization policies should be provided to ensure and maintain the consistency of all policies. Current conflict detection methods primarily focus on the authorization policies in which context constraints are overlooked, hence not suitable for our policies. In this section, we are mainly concerned with the provision of effective static and dynamic conflict detection rules for the above defined authorization policies in WFMSs.

We believe that conflicts can be classified into two broad categories as defined below:

Definition 11 (Policy-policy conflict) Policy-policy conflicts occur when two or more authorization policies are deemed incompatible with each other. For example, if positive and negative policies have a triple overlap of tasks, roles and permissions, there is a potential for policy-policy conflicts.

Definition 12 (Policy-constraint conflict) Policy-constraint conflicts occur when the performance of two or more authorization policies will lead to situations that are forbidden by other constraints (e.g., separation-of-duty constraints) in the system. For example, when two positive authorization policies have an overlap of roles and the permissions of the two policies are defined as conflicting by authorization constraints, there is a potential for policy-constraint conflicts.

Consider the following two authorization policies: ap_1: {design drawing, designer, (drawing, design), +, inheritable}; ap_2: {approve drawing, designer, (drawing, approve), +, inheritable}. The performance of the two policies will introduce conflicts into the system since the same person must not be allowed both to design a drawing and to approve the drawing for the separation of duties. The detection of policy-constraint conflicts depends on the understanding of the semantics of authorization policies. Due to the complexity and diversity of constraints, detecting policy-constraint conflicts is a complex topic, which is outside the scope of this study and will be addressed in our future work. In this work, we focus on providing effective static and dynamic conflict detection rules for policy-policy conflicts. In the following sections, 'conflict' refers to 'policy-policy conflict'.

Static conflict detection

Conflict detection can be classified into static detection and dynamic detection. The goal of static detection is to identify an actual conflict that has occurred and can be resolved statically. Static conflict detection can be enforced during the definition of policies.

1. Static conflict detection algorithm

For conflict detection, an authorization policy ap: $\{tu, R, P, type, inheritable, ac\}$ can be denoted as ap: $(ac \rightarrow (tu(ap), R(ap), P(ap)), sign)$, where ac is the left-hand side of ap, $(tu(ap), R(ap), P(ap))$ is the right-hand side of ap, and $sign$ is the type of ap (for positive authorization policies, $sign$ is '+'; for negative ones, $sign$ is '−').

In the following sections, the left-hand sides of authorization policies are unified as F, and the right-hand sides are unified as B.

Definition 13 (Graph for authorization policies) The graph for authorization policies is an undirected one, denoted as $G(APB)=(V, E)$, where V is the set of vertices in $G(APB)$, composed of left- and right-hand sides of all authorization policies, i.e., $V=\{F, B | ap: (F \rightarrow B, sign) \in AP\}$. AP denotes the set of authorization policies, E is the set of edges in $G(APB)$, and each edge is not directed. An edge in E can be denoted as e: $(v_1, v_2, sign)$, where v_1 and v_2 are the two vertices corresponding to e, and $sign$ is the identifier of e belonging to either '+' or '−'. For an edge e, it may be one of the following cases:

(1) e: $(F, B, sign)$, ap: $(F \rightarrow B, sign) \in AP$. That is, the two vertices corresponding to e are the left- and right-hand sides of an authorization policy ap, and the identifier of e is the type of ap.

(2) e: $(B_i, B_j, +)$, $correlate(ap_i: (F_i \rightarrow B_i, sign_i), ap_j: (F_j \rightarrow B_j, sign_j))$, $ap_i, ap_j \in AP$. That is, the two vertices of e are the right-hand sides of two correlative policies, and the identifier of e is positive.

(3) e: $(F_i, F_j, +)$, $correlate(ap_i: (F_i \rightarrow B_i, sign_i), ap_j: (F_j \rightarrow B_j, sign_j)) \wedge Inter(F_i, F_j)$, $ap_i, ap_j \in AP$. That is, the two vertices of e are the left-hand sides of two correlative policies and the left-hand sides of the two policies are intersectant context constraints. The identifier of e is positive.

(4) e: $(F_i, F_j, -)$, $correlate(ap_i: (F_i \rightarrow B_i, sign_i), ap_j: (F_j \rightarrow B_j, sign_j)) \wedge NonInter(F_i, F_j)$, $ap_i, ap_j \in AP$. That is, the two vertices of e are the left-hand sides of two correlative policies and the left-hand sides of the two policies are non-intersectant context constraints. The identifier of e is negative.

The static conflict detection for authorization policies can be enforced by the construction of $G(APB)$.

Rule 1 (Static conflict detection for authorization policies) In $G(APB)$, if there is a loop where there exists one and only one edge identified as '−', and the vertices of the loop are the left- and right-hand sides of two authorization policies, then authorization policies in $G(APB)$ are not statically consistent and conflicts will occur.

For instance, Fig.4a is the graph for authorization policies constructed for the example policies listed in Table 1. No static conflict is detected for these policies according to Rule 1. If we add a new policy ap_7: {design drawing, technical manager, (drawing, design), −, non_inheritable}, the graph for authorization policies will be reconstructed (Fig.4b).

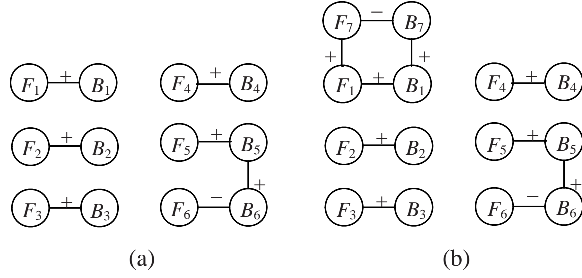

Fig.4 (a) Example graph for authorization policies; (b) Reconstructed graph with a new policy

The policy ap_7 states that role 'technical manager' cannot be granted the permission for designing drawings during the lifetime of task 'design drawing'. Since the policy ap_1 defined in Table 1 shows that role 'designer' and its senior roles, including role 'technical manager', can be granted the permission to design drawings during the lifetime of task 'design drawing', the two policies ap_1 and ap_7 exhibit a static conflict. In such a situation, there are a loop in the graph and an edge identified as '−' in the loop. The vertices of the loop are the left- and right-hand sides of the two policies.

Algorithm 1 Static conflict detection for authorization policies

Input: The graph for authorization policies $G(APB)=(V, E)$, a new policy *new_ap*: (*new_F*, *new_B*, *new_sign*).
Output: The set of authorization policies exhibiting static conflicts *Conflict_AP*.

```
AuthorizationPolicyStaticConflictDetection(G(APB), new_ap)
    V←V∪new_F∪new_B;
    E←E∪(new_F, new_B, new_sign);
    for each policy ap: (F→B, sign)∈AP do
      if correlate(ap, new_ap) then
        E←E∪(B, new_B, +);
        if Inter(F, new_F) then
          E←E∪(F, new_F, +);
        end if
        if NonInter(F, new_F) then
          E←E∪(F, new_F, −);
        end if
      end if
    end for
    L←all loops in G(APB);
    for all l_i∈L (l_i contains only one edge signed '−') do
      if (the vertices of l_i are the left- and right-hand sides of
        new_ap and an existing policy) then
          Conflict_AP=Conflict_AP∪(ap_i, new_ap), ap_i∈l_i;
      end if
    end for
```

2. Proof of completeness and correctness of Algorithm 1

In this subsection, we try to show how a proof of completeness and correctness of Algorithm 1 may be given.

Lemma 1 For two authorization policies ap_i: (ac_i→($tu(ap_i)$, $R(ap_i)$, $P(ap_i)$), $sign_i$) and ap_j: (ac_j→($tu(ap_j)$, $R(ap_j)$, $P(ap_j)$), $sign_j$), only if one of the following conditions holds, would the two policies exhibit a static conflict:

(1) $correlate(ap_i, ap_j)\wedge different_type(ap_i, ap_j)\wedge Inter(ac_i, ac_j)$. Therein $different_type(ap_i, ap_j)$ represents that ap_i and ap_j have different types; i.e., one is positive and the other is negative.

(2) $correlate(ap_i, ap_j)\wedge positive(ap_i, ap_j)\wedge NonInter(ac_i, ac_j)$. Therein $positive(ap_i, ap_j)$ represents that ap_i and ap_j are both positive policies.

Proof Suppose that ap_i is positive and ap_j is negative, and they both satisfy the first condition. The policy ap_i states that roles in $R(ap_i)$ can be granted permissions in $P(ap_i)$ during the lifetime of $tu(ap_i)$ only if the context constraint ac_i is satisfied. The policy ap_j states that roles in $R(ap_j)$ cannot be granted permissions in $P(ap_j)$ during the lifetime of $tu(ap_j)$ if the context constraint ac_j is satisfied. These two policies have an overlapping scope specified by the pre-condition $(tu(ap_i)=tu(ap_j))\wedge(R(ap_i)\cap R(ap_j)\neq\varnothing)\wedge(P(ap_i)\cap P(ap_j)\neq\varnothing)$ and their context constraints ac_i and ac_j are intersectant; thus, we cannot decide whether the roles in the set $R(ap_i)\cap R(ap_j)$ can or cannot be granted the permissions in the set $P(ap_i)\cap P(ap_j)$ under the situation in which ac_i and ac_j are simultaneously met during the lifetime of task $tu(ap_i)$. We would therefore be able to identify that the two policies exhibit a static conflict. For example, the two policies ap_1 and ap_7 defined above satisfy the first condition and exhibit a static conflict.

Suppose that ap_i and ap_j are both positive policies, and satisfy the second condition. The policy ap_i states that roles in $R(ap_i)$ can be granted permissions

in $P(ap_i)$ during the lifetime of $tu(ap_i)$ only if the context constraint ac_i is satisfied. The policy ap_j states that roles in $R(ap_j)$ can be granted permissions in $P(ap_j)$ during the lifetime of $tu(ap_j)$ only if the context constraint ac_j is satisfied. Since ac_i and ac_j are non-intersectant constraints, we cannot decide whether the roles in the set $R(ap_i)\cap R(ap_j)$ can or cannot be granted the permissions in the set $P(ap_i)\cap P(ap_j)$ during the lifetime of task $tu(ap_i)$. Therefore, in such a case, the two policies would exhibit a static conflict. For example, consider the following two policies: ap_1: {approve drawing, auditor, (drawing, approve), +, non_inheritable, ac_1}, where ac_1: {location, (is, local), *access_IP_address*}; ap_2: {approve drawing, auditor, (drawing, approve), +, non_inheritable, ac_2}, where ac_2: {location, (is not, local), *access_IP_address*}. The policy ap_1 states that auditors can approve drawings only when their access IP addresses are local. Whereas, the policy ap_2 states that auditors can approve drawings only when their access IP addresses are not local. The two policies are incompatible with each other and exhibit a static conflict.

As Moffett and Sloman (1994) pointed out, overlap is crucial to the discussion of policy conflicts. Without certain overlap between the objects in two policies, there can be no conflict between them. If having no triple overlap of tasks, roles, and permissions (i.e., they are not correlative), the two policies would not exhibit static conflicts. For example, the policies ap_1 and ap_2 defined in Table 1 are not corrective, thus exhibiting no static conflict.

For two correlative policies, if not satisfying the two conditions defined above, they would exhibit no static conflict. For example, consider the following two policies: ap_1: {approve drawing, auditor, (drawing, approve), +, non_inheritable, ac_1}, where ac_1: {time, (between, 08:00, 17:00), *access_time*}; ap_2: {approve drawing, auditor, (drawing, approve), −, non_inheritable, ac_2}, where ac_2: {time, (between, 17:00, 17:30), *access_time*}. ap_1 and ap_2 exhibit no static conflict.

Lemma 2 Only if two authorization policies exhibit a static conflict, would there exist a loop in $G(APB)$, where there exists one and only one edge identified as '−'. The vertices of the loop are the left- and right-hand sides of the two policies.

Proof According to Lemma 1, if exhibiting a static conflict, two authorization policies ap_i and ap_j must satisfy one of the two conditions specified in Lemma 1. In such a situation, we would discover that there must exist a loop in $G(APB)$, where there exists one and only one edge identified as '−', and the vertices of the loop are the left- and right-hand sides of the two policies. If ap_i and ap_j satisfy the first condition specified in Lemma 1, the loop formed is shown as Fig.5a. If ap_i and ap_j satisfy the second condition specified in Lemma 1, the loop formed is shown as Fig.5b.

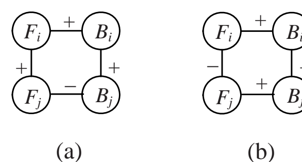

Fig.5 Loops formed by conflicting policies
(a) Loops formed by policies of different types; (b) Loops formed by policies of the same type

If two authorization policies ap_i and ap_j exhibit no static conflict, there would not exist a loop in $G(APB)$, where there exists one and only one edge identified as '−', and the vertices of the loop are the left- and right-hand sides of the two policies. For example, the following two policies exhibit no static conflict and the graph for authorization policies constructed for them is as shown in Fig.6 (note that the loop has no edge identified as '−'): ap_1: {approve drawing, auditor, (drawing, approve), +, non_inheritable, ac_1}, where ac_1: {time, (between, 08:00, 17:00), *access_time*}; ap_2: {approve drawing, auditor, (drawing, approve), +, non_inheritable, ac_2}, where ac_2: {time, (between, 08:00, 18:00), *access_time*}.

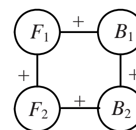

Fig.6 Example graph for two policies exhibiting no static conflict

We can conclude from Lemmas 1, 2 that static conflicts can be detected by constructing the graph for authorization policies. Only if two authorization policies exhibit a static conflict, would there exist a loop in the graph, where there exists one and only one edge identified as '−', and the vertices of the loop are the left- and right-hand sides of the two policies. If a new policy statically conflicts with an existing policy,

the two policies will be added to the set of authorization policies exhibiting static conflicts by the computation of Algorithm 1; if the two policies exhibit no static conflict, they will not be added to the set.

Dynamic conflict detection

The goal of dynamic detection is to predict a dynamic, potential conflict that may, or may not, occur in the future, and more specifically, what circumstances will expose that conflict. Dynamic detection is enforced at run-time. Dynamic conflict detection for authorization policies is a complicated problem, in that the dynamic conflict is potential and quite unpredictable; that is, it may, or may not, proceed to an actual conflict, and will be exposed only under specific circumstances.

For example, we would be able to predict that the policies of ap_5 and ap_6 defined in Table 1 would be in a state of potential conflict if a drawing is designed collaboratively by more than two designers and one of them is the technical manager. In such a case, policy ap_5 becomes active and the drawing can only be approved by the auditor, whereas policy ap_6 also turns active and the drawing cannot be approved by the auditor. Therefore, the occurrence of this particular event means that the dynamic, potential conflict between ap_5 and ap_6 has been realized.

Note that when a dynamic conflict between ap_5 and ap_6 occurs, the valid role set of ap_5 is {auditor}; it means that in such a situation only the auditor is valid and can be granted the permission for approving the drawing. If the valid role set of ap_6 is {auditor}, the auditor cannot be granted the permission for approving the drawing. In such a case, $ValidRole(ap_5) \subseteq ValidRole(ap_6)$, $ValidUser(ap_5) \subseteq ValidUser(ap_6)$.

Note that from the above analysis, when the dynamic conflict between two correlative authorization policies occurs, the requirements of the two policies become incompatible, and this can be detected by comparing the valid role sets or the valid user sets of the two policies. Therefore, the dynamic conflict between two correlative authorization policies can be detected by the calculation of their valid role sets and valid user sets.

Rule 2 (Dynamic conflict detection for authorization policie) For two correlative authorization policies ap_i: $(ac_i \rightarrow (tu(ap_i), R(ap_i), P(ap_i)), sign_i)$ and ap_j: $(ac_j \rightarrow (tu(ap_j), R(ap_j), P(ap_j)), sign_j)$, if one of the following conditions is satisfied at time t, then ap_i and ap_j are not consistent and a dynamic conflict will occur:

(1) $(sign_i=\text{'+'}) \wedge (sign_j=\text{'−'}) \wedge ((ValidRole(ap_i) \subseteq ValidRole(ap_j)) \vee (ValidUser(ap_i) \subseteq ValidUser(ap_j)))$; i.e., if ap_i and ap_j are different in type, and the valid role set of the positive policy ap_i is included in the valid role set of the negative policy ap_j or the valid user set of ap_i is included in the valid user set of ap_j, then the two policies are dynamically inconsistent.

(2) $(sign_i=sign_j=\text{'+'}) \wedge ((ValidRole(ap_i) \cap ValidRole(ap_j)=\varnothing) \vee (ValidUser(ap_i) \cap ValidUser(ap_j)=\varnothing))$; i.e., if ap_i and ap_j are both positive in type and the intersection of the valid role sets or the valid user sets of the two policies is empty, then the two policies are dynamically inconsistent.

CONFLICT RESOLUTION FOR AUTHORIZATION POLICIES

If conflicts occur, it becomes necessary to apply methods to conflict resolution. A practical method of resolving conflicts is to identify which policy involved in a conflict situation will take precedence. Some methods that can be used to establish precedence in conflict situations have been studied, including specific-overrides-general and negative-overrides-positive, but each of them can produce only valid results in some circumstances. For example, consider the following two policies: ap_1: {design drawing, employee, (technical drawing properties, read), +, inheritable}; ap_2: {design drawing, proof-reader, (drawing properties, read), −, non_inheritable}. The policy ap_1 states that the role employee can read the properties of technical drawings during the lifetime of task 'design drawing'. The policy ap_2 states that the role proof-reader cannot read the properties of drawings during the lifetime of task 'design drawing'. The role of ap_2 is more specific than that of ap_1, whereas the permission of ap_1 is more specific than that of ap_2. Therefore, we cannot decide which policy is more specific.

Precedence based on modality, e.g., negative-overrides-positive, resolves conflicts in a deterministic way. For example, consider the following two policies: ap_1: {design drawing, employee, (drawing, design), −, inheritable}; ap_2: {design drawing, designer, (drawing, design), +, inheritable}. The policy

ap_1 states that all employees cannot be granted the permission for designing drawings during the lifetime of task 'design drawing'. If negative policies take precedence over positive ones, then none of the designers can design drawings during the lifetime of task 'design drawing'.

To address the issue, the concepts of 'precedence establishment rule' and 'conflict resolution policy' are introduced in this study, and security administrators can flexibly decide the methods for conflict resolution according to system requirements.

Definition 14 (Precedence establishment rule) A precedence establishment rule defines a method to identify which authorization policy involved in a conflict situation will take precedence. It can be formally described as: $Priority_Rule$: $\{ap_i, (relation_1, relation_2, \ldots, relation_t), ap_j\} \rightarrow Priority(ap_i) > Priority(ap_j)$, where ap_i and ap_j are two authorization policies involved in a conflict situation, and $(relation_1, relation_2, \ldots, relation_t)$ denotes the possible relationships between them. For example, $(ap_i, >_{\text{timestamp}}, ap_j)$ represents that the creation date of ap_i is later than that of ap_j, and $\{ap_i, (>_{\text{role}}, NAP), ap_j\}$ states that roles in ap_i are more specific than those in ap_j (ap_i is a negative policy while ap_j is a positive one). The possible relationships between two policies can be defined flexibly according to the conflict resolution requirements of the system. The rule states that if two authorization policies ap_i and ap_j satisfy the relationship specified by $(relation_1, relation_2, \ldots, relation_t)$, then ap_i precedes ap_j.

Definition 15 (Conflict resolution policy) A conflict resolution policy is a total order of some precedence establishment rules. The policy can be formally described as $Conflict_Resolution_Policy$: $(Priority_Rule_1 > Priority_Rule_2 > \ldots > Priority_Rule_n)$, where $Priority_Rule_t$ (t=1, 2, …, n) are the precedence establishment rules integrated in the conflict resolution policy, and '>' denotes the order of priority established for the rules. Conflict resolution is a step-by-step process. Sequentially, a precedence establishment rule of a higher priority is selected to resolve remaining conflicts at each step, until no conflict remains.

For example, there is a conflict resolution policy defined as $Conflict_Resolution_Policy$: $(Priority_Rule_1 > Priority_Rule_2 > Priority_Rule_3)$. Therein $Priority_Rule_1$: $\{ap_i, >_{\text{timestamp}}, ap_j\} \rightarrow Priority(ap_i) > Priority(ap_j)$ states that if the creation date of ap_i is later than that of ap_j, then ap_i will take precedence over ap_j; $Priority_Rule_2$: $\{ap_i, >_{\text{granter_level}}, ap_j\} \rightarrow Priority(ap_i) > Priority(ap_j)$ states that if ap_i is issued by a higher authority and ap_j by a lower authority, then ap_i will take precedence over ap_j; $Priority_Rule_3$: $\{ap_i, NAP, ap_j\} \rightarrow Priority(ap_i) > Priority(ap_j)$ means that if ap_i is a negative policy and ap_j a positive one, then ap_i will take precedence over ap_j.

The resolution policy represents that when two authorization policies ap_i and ap_j exhibit a conflict, first we will compare the timestamps of the two conflict policies, and a new policy will override the old policy. If the two conflict policies have the same timestamp, then compare the level of the two granters, in which a higher-level granter overrides the lower-level granter. If the level of the two granters cannot be compared, then $Priority_Rule_3$ will be used for conflict resolution.

COMPUTATIONAL COMPLEXITY

In this section we provide the computational complexity for the proposed static, dynamic conflict detection and resolution methods.

Getting enforced by constructing the graph for authorization policies, the detection of static conflicts is required when a new policy is added. We gradually add authorization policies, the number of which initially defined is denoted by n_{ap}. The complexity of establishing the initial graph for authorization policies is $O(n_{\text{ap}}^2)$, and the complexity of detecting all static conflicts for these authorization policies is $O(n_{\text{ap}}^2)$. After the graph for authorization policies is initially constructed, only when a new policy is added or a policy is updated, will static conflicts need to be detected. In such a case, the complexity of reconstructing the graph for authorization policies is $O(n_{\text{ap}})$, and the complexity of detecting static conflicts for the new policy and existing policies is $O(n_{\text{ap}})$.

Dynamic conflicts are detected by calculating the valid role sets and valid user sets of authorization polices. For an authorization policy ap: $\{tu, R, P, type, inheritable, ac\}$, let $n_{\text{ap_r}}$ represent the number of roles in R. If ap is inheritable, the complexity of calculating its valid role set and valid user set is $O(n_{\text{ap_r}} n_{\text{r}})$ and

$O(n_{ap_r}n_rn_u)$, respectively, since senior roles of each role in R also can or cannot be granted permissions in P during the lifetime of tu if ac is satisfied. Here n_r is the number of roles and n_u is the number of users in the system. If ap is non-inheritable, the complexity of calculating its valid role set and valid user set is $O(n_{ap_r})$ and $O(n_{ap_r}n_u)$, respectively. Since the maximum value of n_{ap_r} is n_r, the complexity of detecting dynamic conflicts for two authorization policies is $O(n_r^2n_u)$.

Suppose that n_{apt} is the number of all authorization policies in the system. The maximum number of conflicts is $n_{apt}(n_{apt}-1)/2$. The complexity of resolving all these conflicts using the proposed conflict resolution method is $O(m\,n_{apt}^2)$, where m is the number of precedence establishment rules in the resolution policy. By the definition of precedence establishment rules and conflict resolution policies, our resolution method can resolve conflicts flexibly according to system requirements. However, existing conflict resolution methods such as specific-overrides-general can produce only valid results in some circumstances and fail to resolve all types of conflicts. The resolution method proposed in this study is therefore more effective and efficient.

CONCLUSION AND FUTURE WORK

In this paper, we provide approaches to help workflow designers construct a flexible, consistent workflow authorization schema. We introduce a new type of authorization policy language that is more suitable and flexible for WFMSs. A set of static and dynamic conflict detection rules is defined for authorization policies in WFMSs to guarantee the exemption of inconsistency and ambiguities within these policies. Furthermore, a flexible and novel conflict resolution method is proposed by the introduction of two new concepts, the precedence establishment rule and the conflict resolution policy. Security administrators can flexibly decide the methods for conflict resolution according to system requirements. Our future work will address the following issues:

(1) The detection and resolution methods for policy-constraint conflicts are in want of provision.

(2) Validity detection rules for authorization policies need to be proposed, as conflicts may exist in an authorization policy itself.

(3) Authorization policies expect further refinement, as policies are considered to exist at many different levels of abstraction.

References

Atluri, V., Huang, W.K., 1996. An Authorization Model for Workflows. Proc. 5th European Symp. on Research in Computer Security, p.44-64. [doi:10.1007/3-540-61770-1]

Atluri, V., Huang, W.K., 2000. A petri net based safety analysis of workflow authorization models. *J. Comput. Secur.*, **8**(2):209-240.

Bertino, E., 2003. RBAC models—concepts and trends. *Comput. & Secur.*, **22**(6):511-514. [doi:10.1016/S0167-4048(03)00609-6]

Dunlop, N., Indulska, J., Raymond, K., 2002. Dynamic Conflict Detection in Policy-based Management Systems. Proc. 6th Int. Enterprise Distributed Object Computing Conf., p.15-26. [doi:10.1109/EDOC.2002.1137693]

Dunlop, N., Indulska, J., Raymond, K., 2003. Methods for Conflict Resolution in Policy-based Management System. 7th IEEE Int. Enterprise Distributed Object Computing Conf., p.98-109. [doi:10.1109/EDOC.2003.1233841]

Ferraiolo, D.F., Cugini, J.A., Kuhn, D.R., 1995. Role-Based Access Control (RBAC): Features and Motivations. Proc. 11th Annual Computer Security Application Conf., p.11-15.

Ferraiolo, D.F., Sandhu, R.S., Gavrila, S., Kuhn, D.R., Chandramouli, R., 2001. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, **4**(3):224-274. [doi:10.1145/501978.501980]

Georgakopoulos, D., Hornick, M., Sheth, A., 1995. An overview of workflow management: from process modelling to workflow automation infrastructure. *Distrib. Parall. Databases*, **3**(2):119-153. [doi:10.1007/BF01277643]

He, Z.L., Tian, J.D., Zhang, Y.S., 2005. Analysis, detection and resolution of policy conflict. *J. Lanzhou Univ. Technol.* **31**(5):83-86 (in Chinese).

Huang, W.K., Atluri, V., 1999. SecureFlow: A Secure Web-enabled Workflow Management System. Proc. 4th ACM Workshop on Role-based Access Control, p.83-94. [doi:10.1145/319171.319179]

Moffett, J.D., Sloman, M.S., 1994. Policy conflict analysis in distributed system management. *Ablex Publish. J. Organ. Comput.*, **4**(1):1-22.

Oh, S., Park, S., 2003. Task-role-based access control model. *Inf. Syst.*, **28**(6):533-562. [doi:10.1016/S0306-4379(02)00029-7]

Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E., 1996. Role-based access control models. *IEEE Comput.*, **29**(2):38-47. [doi:10.1109/2.485845]

Thomas, R.K., Sandhu, R.S., 1997. Task-Based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-oriented Authorization Management. Proc. IFIP WG11.3 Workshop on Database Security, p.11-13.