CONSTRAINED ROLE-BASED DELEGATION

LONGHUA ZHANG

NuTech Solutions, Inc
Charlotte, NC 28262, USA
Longhua.zhang@nutechsolutions.com

GAIL-JOON AHN
Dept. of Software and Information Systems
University of North Carolina at Charlotte
Charlotte, NC 28223, USA
gahn@uncc.edu

Abstract: Delegation is a promising alternative to traditional role administration paradigms in role-based systems. It empowers users to exercise discretion in how they use resources as it is in discretionary access control (DAC). Unlike the anarchy of DAC, in role-based access control (RBAC) higher-level organizational policies can be specified on roles to regulate user's action. Delegations and revocations are thus governed by these authorization policies. In this paper, we propose a policy approach for specifying and enforcing delegation authorizations. We present a mechanism for constructing authorization policies using a set of rules. Our rule-based language is flexible and powerful to specify and enforce authorization constraints. In addition, rules can also be used to define the exceptions for future actions and resolve possible conflicts.

Key words: Role-based delegation, authorization constraints, access control.

1. INTRODUCTION

Access control model must include an administration policy that regulates the specification of authorizations. In current role-based systems [1, 2], security officers handle this function. This approach is appropriate in centralized systems as well as distributed systems where users, roles, and their assignments are relatively static and stable. However, current dynamic and collaborative work environment often requires users to change their role memberships frequently. It could raise tremendous overhead because of the continuous involvement from security officers. One promising approach is to empower individual user to delegate authorizations. Through delegation, users are trusted to exercise their privileges in how they use resources as it is in discretionary access control (DAC) [4]. We have presented a delegation framework focusing on user-to-user delegation in role-based systems [5].

The framework consists of a delegation model called RDM2000 and a rule-based policy specification language. RDM2000 is our first attempt to address administration of roles through delegation. And the rule-based language is used to specify delegation policies. Administration of roles through delegation is quite feasible and practical. We have demonstrated how RDM2000 can support those features in several role-based systems [19]. However, the original rule-based policy language lacks capabilities to specify and enforce constraints, which are highly desirable features in RBAC. As constraints are a powerful mechanism for laying out organizational policy, constraints enforcement is an important component to administer role-based delegations [1].

In this paper, we enhance the rule-based language to support constraints specification and enforcement. We present a policy approach to implement role-based delegation authorization. We also demonstrate that our authorization policy is flexible and powerful for role-based delegation. We believe that our work provides a flexible mechanism for administration of roles in role-based systems.

The rest of this paper is organized as follows. Section 2 describes delegation requirements and reviews related works. Section 3 describes the RDM2000 model and role-based constraints. Section 4 presents a policy approach for the specification and enforcement of delegation authorizations. We discuss other issues and future directions in section 5. Section 6 concludes this paper.

2. BACKGROUND AND RELATED WORK

Delegation is an important factor for secure distributed computing environment. It has been studied by a number of researchers [5, 8, 9, 10]. In general, it is referred to as one active entity in a system delegates its authority to another entity to carry out some functions. In role-based systems, the delegated authorities are roles. The requirements related to role-based delegation have been identified in the literature [5, 9, 11].

There are many groups working on constraints specification [1, 12, 13]. Sandhu et al. [1] described several types of constraints in RBAC96. Ahn et al. [12] specified role-based separation of duty constraints using RCL2000 language and identified various classes of role-based authorization constraints. Bertino et al. [13] explored authorization constraints in workflow management systems. Bandmann et al. [18] illustrated how to impose controls on handling the delegation tree. However, the enforcement of these constraints has not received much attention. Lupu et al. [6] studied specification of policies in the context of role management. They consider roles as a set of policies and use policies to specify permissions of each role.

In their approach, authorization policies and roles are highly coupled, because roles cannot exist without policies. In our approach, permissions assigned to roles are specified by permission assignment (PA) and authorization policies specify whether a delegation (or revocation) is allowed. So policies and roles are loosely coupled. Another difference is to define a policy as a set of declarative rules instead of a single object. Since rules are easy to understand, create, and modify, it provides flexibility to meet different authorization requirements.

3. RDM2000 - THE DELEGATION MODEL

RDM2000 was originally introduced for user-to-user delegation in role-based systems. It formalizes the relationship between two user assignments (UAO/UAD) that form a delegation relation (DLGT), as shown in figure 1. Basic elements and system functions from the RDM2000 model are summarized in definition 1.

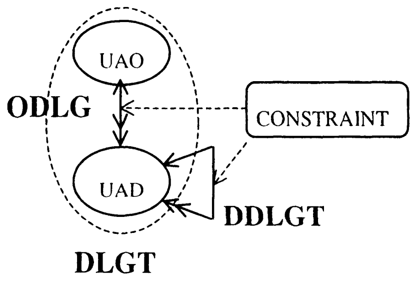

Figure 1. RDM2000

Definition 1. The following is a summary of components defined in RDM2000 model:

- *$UA = UAO \cup UAD \subseteq U \times R$*
- *$UAO \subseteq U \times R$ is an original user to role assignment relation.*
- *$UAD \subseteq U \times R$ is a delegated user to role assignment relation.*
- *$DLGT = ODLGT \cup DDLGT \subseteq UA \times UA$.*
- *$ODLGT \subseteq UAO \times UAD$ is an original user delegation relation.*
- *$DDLGT \subseteq UAD \times UAD$ is a delegated user delegation relation.*
- *$DLGT = ODLGT \cup DDLGT$.*
- *Prior: $UA \rightarrow UA$ is a function that maps a user assignment to another subsequent user assignment that forms a delegation relation.*
- *$DP \subseteq UA \times UA$ represents a delegation path.*
- *$DT \subseteq UA \times UA$ represents a delegation tree.*
- *Path: $UA \rightarrow DP$ is a function that maps a UA to a delegation path.*

In RDM2000, a set of authorization policies is defined. These authorization policies are represented in our policy language as basic authorization rules [5].

Constraints are an important component of RBAC since it can be used for laying out higher-level organizational policies in role-based systems [1,12]. Our objective is to specify and enforce constraints in role-based delegation authorizations. First, we overview some identified constraints in role-based systems [1, 12, 13, 14]: 1) *Static separation of duty*

(SSOD)/Incompatible roles assignment. This constraint states that no common user should be assigned to conflicting roles. A frequently used example is a user cannot be a purchase manager while at the same time being an account payable manager for the same organization. We denote a set of incompatible role assignments as IRA. 2) *Incompatible users.* This constraint states that two conflicting users cannot be assigned to the same role. For example, it might be a company's policy that members from same subdivision should not be assigned to the same steering committee. We denote a set of incompatible users as IU. 3) *Incompatible permissions.* This constraint states that conflicting permissions cannot be assigned to the same role. We denote a set of incompatible permissions as IP. 4) *Cardinality constraints.* This constraint states that a role can have a maximum number of members or a user may belong to a maximum number of roles. For example, there may be only one person in the role of CEO in an organization. As stated in [1], the role cardinality is difficult to implement since the system may not know exactly how many users are still "alive" – some may leave without notifying security officers. We denote the cardinality of x as *cardi(x)*, where x is a role term or a user term.

It is futile to enumerate all role-based constraints, as there are too many possibilities and variants [12]. In the subsequent sections, we show that the enhanced rule-based language is expressive enough to specify a wide range of constraints.

4. AUTHORIZATION POLICIES IN DELEGATION

4.1 Functions, Rules, and Policies

The fundamental element of the policy language is a set of functions. We categorize these functions into three groups: 1) a set of specification functions, expressing information of the RBAC and RDM2000 components; 2) a set of authorization functions, describing an authorization information or decision. We further divide them into a set of basic authorization functions (BAP), a set of derived authorization functions (DAP), and a set of negative authorization functions (NAP); and 3) a set of utility functions, providing supportive functionalities, e.g. comparison and aggregation. These functions and their semantics are listed in appendix.

We borrow the notion of two non-deterministic functions from RCL2000 [12]: *one_element* and *all_other* (originally as *OE* and *AO*). These functions are introduced to replace explicit quantifiers, thus keep the language simple and intuitive. The *one_element(X)* function allows us to get one element x_i from set X. Multiple occurrences of *one_element (X)* in a single rule statement select the same element x_i. With *all_other(X, x_i)* we can get a subset of X by taking out one element x_i.

The policy language is a rule-based language with a clausal logic [5]. A rule is the form:

$H \leftarrow B.$

Where H stands for rule head and B stands for rule body.

A successful inference of B triggers H to be true. This provides the mechanism for constraints specification and enforcement. A constraint is similar to an assertion. If the condition defined in the constraints is true, then it triggers some actions (restrictions). Thus, the condition information of a constraint can be encoded in a rule body; and the restrictions can be encoded in the rule head.

Since it is natural to represent restrictions as negative authorizations, we include the negative authorization concept in the rule-based policy language. A negative authorization defines what a user is forbidden to do. Given an authorization *can_do* that defines what can do, there can always be a negative authorization *cannot_do* that defines what cannot do. Although negative authorization has been introduced in discretionary access control for a long time, it has not been studied in role-based context. In [14], Bertino et al. explored negative authorization in relational data management systems (RDMS). They proposed strong and weak enforcement for positive and negative authorizations. Their approach provides a flexible mechanism to express a number of DAC policies. Because of the existence of role hierarchies, negative authorizations in RBAC are more complex than in DAC. For example, a negative authorization on a role needs to consider an impact on its senior roles. In this paper, we consider negative authorization can be inherited in role hierarchies: if a user cannot be assigned to a role *r*, he cannot be assigned to any roles that are senior to *r*. We do not distinguish a strong or a weak enforcement of authorization for the brevity. An override rule determines whether positive or negative authorization needs to be enforced. In this paper we limit the application of negative authorizations to constraints enforcement and exception handling. In our future work, we would general use of negative role-based authorizations in depth.

There are three sets of rules in RDM2000: basic authorization rules, authorization derivation rules, and override rules. Bodies of basic authorization rules are empty. In other words, they are always true. Basic authorization rules are predefined security policies or facts specified within RBAC components. An authorization derivation rule expresses authorization on an individual user. The rule body describes an inference logic that consists of basic authorization, specification and utility functions. The result can be either true (authorized) or false (denied). However, the result might be overridden by other rules in certain situations. An override rule specifies exceptions and conflict resolution policy. Having discussed different functions and rules, we define the following authorization policy:

Definition 2. (**Authorization Policy**) An authorization policy consists of a finite number (possibly zero) of rules.

An authorization policy is a logic program that defines authorization for individual users to exercise privileges in role-based systems. Each authorization policy has one or more authorization derivation rules representing the goal. The conclusion of a goal is the result of the logic program execution.

4.2 Constraints Specification

In order to represent constraints, we define rules that are extremely suited for constraints specification as well as enforcement. We articulate several constraints and specify them using our rule-based language.

A **static separation of duty (SSOD): incompatible roles assignment constraint** states that no common user can be assigned to conflicting roles in the incompatible role set *ira* = $\{r_1, r_2, \ldots\}$. This constraint can be represented as:

cannot_assign(u, r) ←senior(r, one_element(ira)) &
member_of(u, one_element(all_other(ira, one_element(ira)))).

where cannot_assign, equals, one_element, member_of, all_other are functions defined in our rule-based language; $u \in U$, $r \in R$, *and* $ira \in IRA$.

The rule says if *r* equals one element of a set of the incompatible role assignments *ira*, and a user *u* is already member of another role other than *r* in the incompatible role set, then *u* cannot be assigned role *r*.

An **incompatible users constraint** states that two conflicting users in the incompatible user set $iu=\{u_1, u_2, \ldots\}$ cannot be assigned to the same role. This constraint can be represented as:

cannot_assign(u, r)←
equals(u', one_element(all_other(iu, u))) & member_of(u', r).

An **incompatible permissions constraint** states that two conflicting permissions in the incompatible user set $ip=\{p_1, p_2, \ldots\}$ cannot be assigned to the same role. This constraint can be represented as:

cannot_assignp(r, p)←equals(p', one_element(all_other(ip, p))) &
in(p', permissions_role(r)).

A **role cardinality constraint** states that a role can have a maximum number *N* of user members. This constraint can be represented as:

cannot_assign(u, r)←greater_than(cardi(r), maxcardi(r)-1).

A **user cardinality constraint** states that a user can be member of a maximum number *N* of roles. This constraint can be represented as:

cannot_assign(u, r)←greater_than(cardi(u), maxcardi(u)-1).

We have demonstrated how different constraints can be specified using rules. Next we show how exceptions can be specified and how to resolve possible conflicts in rules.

4.3 Exception Handling and Conflict Resolution

Exceptions allow users to state overridden policies for previous specified and enforced authorizations in some cases. For example, a security officer may need to suspend Bob's role membership of account manager role *AcctMgr* without revoking him from the role. This can be achieved by issuing following rule:

cannot_activate(Bob, AcctMgr, _)←.

where _ stands for anonymous variable. An anonymous variable can be anything. In this case, it can be any session.

This rule enforces *Bob* cannot activate his role *AcctMgr* in any session.

Another exception example could be the duration-triggered revocation. Revocation using duration-restriction constraints was proposed by Barka and Sandhu [9]. In such a revocation, a duration constraint is attached to each delegation such that the delegation expires when the assigned time expires. Duration-restriction revocation is a simple self-triggered process that ensures the revocation of role membership. Suppose there is a delegation relation *(Linda, AcctMgr, Alice, AcctPart) ∈ DLGT* with an associated duration constraint *t*. The duration-triggered revocation can be represented as:

der_can_revokeGD(Linda, AcctMgr, Alice, AcctPart, rvk_opt)
← expires(Alice, AcctPart, t).

Therefore, role-based systems can revoke Alice from the *AcctPart* role, which is a part-time worker in an accounting department based on the duration *t*.

It is worth emphasizing that negative authorizations are the main mechanism to specify constraints and exceptions. Enabling negative authorization may inevitably cause inconsistency: whenever a user holds both a positive and a negative assertion for the same authorization, conflicts arise. A conflict resolution policy is then needed to determine whether the negative or positive assertion should be enforced. There are many different approaches that could be taken [16]: 1) *No conflicts allowed.* The presence of a conflict is considered as an error; 2) *Negative authorization takes precedence.* The negative always overrides the positive; 3) *Positive authorization takes precedence.* The positive always overrides the negative. 4) *A mixed approach.* Neither authorization is considered as prevailing over the other.

In our approach, we specify the override rules to resolve possible conflicts. The decision on whether a negative or a positive authorization takes precedence is determined by an override rule.

override(can_activate(Bob, AcctMgr, _), false)←
can_activate(Bob, AcctMgr, _)&
cannot_activate(Bob, AcctMgr, _).

This rule explicitly states that *Bob* cannot activate *AcctMgr* if conflict exists. Override rules themselves may have conflicts. Although another override rule can be defined to resolve the conflict, this may cause a loop in override rule definition. A default organizational policy can be specified to solve it. For example,

override(X, false)← override(X, true)&override(X, false).

It means that if there exist conflicting override rules, the negative takes precedence.

4.4 Delegation Authorization Policies

As defined in section 4.1, authorization policies are a set of rules regulating whether or not a user is allowed to exercise privileges. To support constraints and exceptions, the delegation authorization includes a single rule. For example, a user-to-user delegation is authorized by the user-user delegation authorization derivation rule in [5]:

der_can_delegate(u, r, u', r', dlg_opt) ← can_delegate(r", cr, n)&
active(u, r, s)&delegatable(u, r)&
senior(r, r")&satisfy(u', cr)&
junior(r', r")&in(depth(u, r), n). (1)

To enforce constraints and exceptions, more rules are needed for authorizing the delegation. For example, if we consider SSOD, incompatible users, incompatible roles, and cardinality constraints, we need to add the following rules:

der_cannot_delegate(u, r, u', r', dlg_opt) ← cannot_assign(u', r'). (2)

cannot_assign(u', r')← senior(r', one_element(ira)) &
member_of(u', one_element(
all_other(ira, one_element(ira)))). (3)

cannot_assign(u', r')← equals(u", one_element(
all_other(iu, u'))) & member_of(u", r'). (4)

cannot_assignr(r', r")← senior(r", one_element(
all_other(irr, r'))). (5)

cannot_assign(u', r')← greater_than(cardi(r'), maxcardi(r')-1). (6)

cannot_assign(u', r')← greater_than(cardi(u'), maxcardi(u')-1). (7)

override(der_can_delegate(u,r,u',r' dlg_opt), false))←
der_can_delegate(u,r,u',r' dlg_opt) &
der_cannot_delegate(u,r,u',r' dlg_opt). (8)

where u, u', and u" are elements of users; r, r', and r" are elements of roles; cr and s are elements of prerequisite condition and sessions respectively; dlg_opt is a Boolean term, if it is true, then delegatable (u', r') is true; ira is any conflict role assignment set that contains r'; iu is any incompatible user set that contains u'; irr is any incompatible role set that contains r'; maxcardi(r') and), maxcardi(u') are maximum allowed cardinalities for r' and u' respectively.

Rule 2 specifies that if there is any constraint that forbids the user assignment (u', r') then the delegation should be denied. Rule 3, 4, 5, 6, and 7 specify constraints. Rule 8 deals with the resolution of conflicts. These rules authorize a user-to-user delegation. Next, we consider handling exceptions in the delegation authorization policy. The basic authorization rule allows users to delegate roles only if users satisfy the prerequisite condition. Otherwise, the authorization decision should be reconsidered. Since there may be multiple basic authorization rules that could be applied to infer the delegation authorization, we may specify the following exception handling rule:

der_can_revokeGD(u, r, u', r', rvk_opt) ←can_revokeGD (r') &
not(satisfy(u', cr)) & not(der_can_delegate(u, r, u', r', dlg_opt)). (9)

This rule says whenever user *u'* cannot satisfy the prerequisite condition *cr* after an authorized delegation, the system needs to re-infer *der_can_delegate(u, r, u', r', dlg_opt)*. If the inference fails, then *u'* is revoked from *r'*.

The delegating user can also specify a duration-triggered revocation:

der_can_revokeGD(u, r, u', r', rvk_opt) ←
can_revokeGD (r') & expires(u', r', t). *(10)*

The exception handling rules do not affect the initial decision of a delegation authorization. Rather, they define the actions that would be taken if the conditions embedded in the rule bodies are triggered after the delegation.

We summarize the delegation authorization policy as follows: a *goal*, which is specified by an authorization derivation rule; *rules* that enforce role-based constraints; *override rules* to resolve possible conflicts; and *exception handling rules* to specify sub-goals that process post-delegation authorizations.

Similarly, we can construct revocation authorization policies for grant-dependent (GD) and grant-independent (GI) revocations. The revocation authorization policies only consist of goals. Although we mainly illustrated specification of delegation and revocation authorization policies using rules in this section, the concept of building a policy is equally applicable to any other authorization scenarios in role-based systems.

5. DISCUSSIONS AND FUTURE WORK

We are attracted by some virtues of the rule-based language: they are highly expressive; they can easily be extended to facilitate additional requirements; and they can handle conflicts during creating and updating of rule sets. However, there are several challenges in rule-based policy implementation. One difficulty is the validation of semantics. Although the rule declaration is intuitively simple that non-experts can feel comfortable in specifying the rule sets, a formal definition and proof of soundness and completeness are still necessary. Another difficulty is the complexity of implementing inferences in the authorization policy. As an ongoing effort, we have been developing security architectures for delegation in distributed role-based systems. Policy service is one of the major components in this architecture. It consists of a rule compiler and a policy engine: the compiler runs as a preprocessor to convert declarative rules to logic program, the output is fed to the policy engine, which may infer the authorization decision and process exceptions for post-delegation actions.

6. CONCLUSIONS

In this paper, we presented a policy approach for constrained delegation authorization. We have shown that our authorization policy is flexible and powerful to regulate role-based delegation. Not only can it enforce role-based constraints, but also be used to define the exceptions for future actions. We reiterate that our work makes it easier to administer roles through delegation in role-based systems. By specifying authorization policy, the delegation is adapted to meet users' needs, while organizational policies can still be specified to impose restrictions. We believe that specifying and implementing complex delegation authorization policies are critical and challenging tasks in large, distributed role-based systems.

REFERENCE

[1] R. Sandhu, E. Coyne, H. Feinstein and C. Youman. Role-based access control model. IEEE Computer, 29(2), Feb. 1996.

[2] D. Ferriaolo, J. Cugini, and R. Kuhn. Role-based access control (RBAC): Features and Motivations. Proceeding s of 11th Annual Computer Security Application Conference, pages 241-248, New Orleans, LA, Dec 11-15 1995

[3] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 Model for Role-based Administration of Roles. ACM Transactions on Information and System Security. Vol.2, No.1, Feb. 1999, pages 105-135

[4] S. Osborn, R. Sandhu and Q. Munawer. Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. ACM Transactions on Information and Systems Security, Vol.3, No. 2, (2000).

[5] L. Zhang, G. Ahn, and B. Chu. A Rule-Based Framework for Role-Based Delegation. Proceedings of ACM Symposium on Access Control Models and Technologies, pages 153–162. Chantilly, VA, May 3-4, 2001

[6] E. Lupu, D. Marriott, M. Sloman and N. Yialelis. A Policy Based Role Framework for Access Control. Proceedings of the First ACM Workshop on Role Based Access Control, Gaithersburg, Maryland, USA, Nov. 1995, ACM, ISBN: 0-89791-759-6.

[7] E. Lupu and M. Sloman. Towards a Role Based Framework for Distributed Systems Management. Journal of Network and Systems Management, 5(1):5-30, Plenum Press Publishing, 1997.

[8] N. Li and B. N. Grosof. A practically implementation and tractable delegation logic. Proceedings of IEEE Symposium on Security and Privacy. May 2000.

[9] E. Barka and R. Sandhu. Framework for Role-Based Delegation Model. Proceedings of 23rd National Information Systems Security Conference, pages 101-114, Baltimore, Oct. 16-19, 2000

[10] M. Gasser, E. McDermott. An Architecture for Practical Delegation a Distributed System. Proceedings of IEEE Computer Society Symposium on Research in Security and Privacy. Oakland, CA, May 7-9,1990

[11] A. Hagstrom, S. Jajodia, F. P. Presicce, D. Wijesekera, Revocations - a classification. Proceedings of 14th IEEE Computer Security Foundations Workshop, Nova Scotia, Canada, June 2001, pages 44-58.

[12] G. Ahn and R. Sandhu. Role-based Authorization Constraints Specification. ACM Transactions on Information and System Security, pages 207-226, Vol. 3, No. 4, ACM, November 2000

[13] E. Bertino, E. Ferrari, V. Atluri. The specification and enforcement of authorization constraints in workflow management systems, ACM Transactions on Information and System Security (TISSEC), Vol.2 No.1, p.65-104, Feb. 1999

[14] E. Bertino, S. Jajodia, P. Samarati, "A Flexible Authorization Mechanism for Relational Data Management Systems", ACM Trans. Information Systems, Vol. 17, No. 2, April 1999, pages 101-140.

[15] S. Jajodia, P. Samarati, M. L. Sapino, V. S. Subrahmanian, ``Flexible support for multiple access control policies," ACM Trans. on Database Systems, June 2001, pages 214-260

[16] S. Jajodia, P. Samarati, V. S. Subrahmanian, E. Bertino, A unified framework for enforcing multiple access control policies, ACM SIGMOD Record, Vol.26 No.2, pages 474-485, June 1997

[17] A. Herzberg, Y. Mass, J. Michaeli, D. Naor, and R. Ravid. Access control meets public key infrastructure, or: assigning roles to strangers. Proceedings of IEEE Symposium on Security and Privacy, Oakland, May 2000

[18] O. Bandmann, B. S. Firozabadi, and M. Dam. Constrained Delegation. Proceedings of IEEE Symposium on Security and Privacy. May 12 - 15, 2002. Berkeley, California

[19] Longhua Zhang, Gail-Joon Ahn, Bei-Tseng Chu: A role-based delegation framework for healthcare information systems, Proceedings of ACM Symposium on Access Control Models and Technologies, pages 153–162, June 2002.

APPENDIX

Some functions and their semantics are listed in table 1, 2, and 3. We use UT, RT, PT, ST, UAT, PAT, DLGTT, DPT, CRT, IRAT, IUT, IRRT, BT, DT, and NT to indicate set of users, roles, permissions, sessions, user assignments, permission assignments, delegations, delegation paths, prerequisite conditions, incompatible role assignments, incompatible users, incompatible roles, booleans, durations, and natural numbers, respectively.

Table 1. Utility functions

Predicate	Arity	Argument	Return	Meaning
All_other	2	set of XT, XT	set of XT	all_other(X, x) = X-{x}.
equals	2	XT, XT	BT	If equals(x, y) is true, then x = y.
In	2	XT, set of XT	BT	If in(x, y) is true, then $x \in y$.
Not	1	BT	BT	not(true) = false and not(false)=true.
one_element	1	set of XT	XT	oneelement(X) returns one element in set X.
satisfy	2	UT, CR	BT	If satisfy(u, cr) is true, then user u satisfies cr.

Table 2. Specification functions

Predicate	Arity	Argu.	Return	Meaning
cardi	1	RT	NT	cardi(r) returns current number of users in role r.
delegatable	1	UAT	BT	If delegatable(ua) is true, then ua can further delegate.
expires	3	UT, RT, DT	BT	expires(u, r, t) returns true if duration t assigned to (u, r) expires.
maxcardi	1	RT	NT	maxcardi(r) returns the maximum cardinality allowed for role r.
parent	2	UAT, UAT	BT	parent(ua, ua') ← equals(ua, prior(ua')).
predecessor	2	UAT, UAT	BT	predecessor(ua, ua') ← predecessor(ua, ua"), parent(ua", ua'). predecessor(ua, ua') ← parent(ua, ua').
revoked_cascade	1	UAT	BT	If revokedcas(ua) is true, then ua will be revoked cascadingly.

Table 3. Authorization functions

Predicate	Arity	Argu.	Type	Meaning
override	2	Rule head, BT	BAP	override(H, b) means let H=b. It states conflict resolution policy for H←B.
cannot_assign	2	UT, RT	NAP	cannot_assign(u, r) means user u cannot be assigned role r.
cannot_assignr	2	RT, RT	NAP	cannot_assignr(r, r') means role r' cannot be assigned to role r.
cannot_activate	3	UT, RT, ST	NAP	cannot_activate(u, r, s) means u cannot activate r in sessions. If session s is an anonymous variable, cannot_activate(u, r, _) means u cannot activate r in any session.
der_cannot_delegate	5	UT, RT, UT, RT, BT	NAP	der_cannot_delegate(u, r, u', r', b) means user u with role r cannot delegate role r' to user u'.