A Formalization of Distributed Authorization with Delegation

Abstract

Trust management is a promising approach for the authorization in distributed environment. There are two key issues for a trust management system: how to design high-level policy language and how to solve the compliance-checking problem [3, 4]. We adopt this approach to deal with distributed authorization with delegation. In this paper, we propose an *authorization language* $\mathcal{AL}$, a human-understandable high level language to specify various authorization policies. We define the semantics of $\mathcal{AL}$ through Answer Set Programming. Language $\mathcal{AL}$ has rich expressive power which can not only specify delegation, threshold structures addressed in previous approaches, but also represent structured resources and privileges, positive and negative authorizations, separation of duty, incomplete information reasoning and partial authorization and delegation. We also demonstrate the application of language $\mathcal{AL}$ through an authorization scenario.

Keywords: Access control, trust management, authorization, delegation, answer set programming, knowledge representation, nonmonotonic reasoning.

1 Introduction

Access control is an important topic in computer security research. It provides availability, integrity and confidentiality services for information systems. The access control process includes identification, authentification and authorization. With the development of Internet, there are increasing applications that require distributed authorization decisions. For example, in the application of electronic commerce, many organizations use the Internet (or large Intranets) to connect offices, branches, databases, and customers around the world. One essential problem among those distributed applications is how to make authorization decisions, which is significantly different from that in centralized systems or even in distributed systems which are closed or relatively small. In these traditional scenarios, the authorizer owns or controls the resources, and each entity in the system has a unique identity. Based on the identity and access control policies, the authorizer is able to make his/her authorization decision. In distributed authorization scenarios, however, there are more entities in the system, which can be both authorizers and requesters, and probably are unknown to each other. Quite often, there is no central authority that everyone trusts. Because the authorizer does not know the requester directly, he/she has to use the information from the third parties who know the requester better. He/She trusts these third parties only for certain things to certain degrees. The trust and delegation issues make distributed authorization different from traditional access control scenarios.

In recent years, the trust management approach, which was initially proposed by Blaze *et al.* in [3], has received a great attention by many researchers [3, 4, 5, 16, 17]. Under this approach public keys are viewed as entities to be authorized and the authorization can be delegated to third parties by credentials or certificates. This approach frames the authorization decision as follows:

> “ Does the set C of *credentials* prove that the *request* r *complies with* the local security *policy* P? ”

from which we can see that there are at least two key issues for a trust management system:

1. Designing a high-level policy language to specify the security policy, credentials, and request. Better it is if the language has richer expressive power and is more human-understandable.

2. Finding well theory foundation for checking proof of compliance.

Several trust-management systems such as PolicyMaker [3], Keynote [6], SPKI/SDSI [7, 8, 9, 10, 19], and DL [17] have been developed. PolicyMaker [3] was the first trust management system. Its access policy and credentials are called *assertion* which can be written in any programming language. It initiates the proof of compliance by creating a “blackboard” for inter-assertion communication, and a proof is achieved if the blackboard contains an acceptance record indicating that a policy assertion approves the request. Keynote [6] is the second generation of trust management systems and was designed according to the same principles as PolicyMaker. Instead of writing policy and credentials in a general-purpose procedural language, It adopts a specific expression. The both systems do not provide the negative authorization and re-delegation control. SPKI (Simple Public Key Infrastructure) [9] and SDSI (Simple Distributed Security Infrastructure) [19] were started independently. Both of them were motivated by the inadequacy of public-key infrastructures based on global name hierarchies, such as X.509 [12] and Privacy Enhanced Mail (PEM) [14]. Later, SPKI and SDSI merged into a collaborative effort, SPKI/SDSI 2.0. SPKI/SDSI 2.0 has two kinds of certificates, name-definition certificates and authorization certificates. A name cert binds a local name to a principal or a more complex name. Name certs are used to resolve names to principals. An auth cert delegates a certain permission from a principal (the cert’s issuer) to the cert’s subject. SPKI/SDSI can deal with the *k-out-of-n* structures and handle certain types of nonmonotic policies based on validity field of auth certificates. It controls whether the authorization should be delegated again or not, but there is no delegation depth control. Delegation Logic, proposed by Li *et al.* [17], is a more expressive formalization. It supports delegation with depth control and static and dynamic threshold structures. Although DL is able to delegate an authorization to a conjunctive-subject structure, it can not deal with the request from conjunctive subjects which is related with *separation of duty*, an important issue in computer security literature. Moreover, it is not suitable to specify the authorization for structured resources.

In our research, we view the problem of a language for representing authorization policy and credentials as a knowledge representation problem. Logic programming approach has been proved very successful in knowledge representation. Some research using logic programming in centralized access control systems has been well developed [2, 13], where underlying languages can support multiple access-control policies and achieve separation of policies from enforcement mechanisms. But their work focuses on centralized systems, and can not be used in distributed systems. Delegation Logic [17], developed by Li *et al.*, is an approach in distributed systems along this line. However the D1LP is based on Definite ordinary logic program, which is less expressive and flexible, and cannot deal with some important issues such as negative authorization, and nonmonotonic reasoning. D2LP extends D1LP to have the nonmonotonic features and bases its syntax and semantics on GCLP (Generalized Courteous Logic Programs). Since it was only briefly mentioned in [15], it was not clear yet how D2LP can handle nonmonotonic reasoning in distributed authorization. In our research, we design a language $\mathcal{AL}$, a nonmontonic language, which is based on Answer Set Programming. We adopt the delegation with depth control and static and dynamic threshold structure from DL approach. Compared to previous work in trust management systems, our language is able to specify positive and negative authorization, the request from conjunctive subjects, structured resources and privileges, incomplete information reasoning, and partial delegation and authorization. The reasons we choose Answer Set Programming as the foundation of language $\mathcal{AL}$ are as follows:

1. Through negation as failure, Answer Set Programming implements nonmonotonic reasoning which is reasoning about incomplete information. Nonmonotoic reasoning was developed to model commonsense reasoning used by humans. A language with nonmontonic feature is easy to specify security policies which is close to the natural language. For example, many systems permit a login request only if they do not find that the requester inputs the password wrong over consecutive three times.

2. The highly efficient solvers for Answer Set Programming have been implemented, such as `Smodels, dlv` etc. This is an important reason that Answer Set Programming has been widely applied in product configuration, planning, constraint programming, cryptanalysis, and so on. We need to indicate that `Smodels` supports some extended literals such as constraint literal and conditional literal which are particularly useful to express the static and dynamic threshold structures.

The rest of this paper is organized as follows. Section 2 presents the syntax and expressive features of language $\mathcal{AL}$. Section 3 develops an answer set language $\mathcal{L}_{Ans}$, provides the translation from $\mathcal{AL}$ into $\mathcal{L}_{Ans}$, and defines the semantics of $\mathcal{AL}$ based on the translation. Section 4 provides a scenario to demonstrate our research. Finally, Section 5 concludes the paper.

2 An Authorization Language $\mathcal{AL}$

In this section, we define the syntax of the authorization language $\mathcal{AL}$ and illustrate its expressiveness via some examples.

2.1 Syntax of $\mathcal{AL}$

The authorization language $\mathcal{AL}$ consists of *entities, atoms, thresholds, statements, rules* and *queries.* The formal BNF syntax of $\mathcal{AL}$ is given in Figure 1. We explain the syntax in detail as follows.

Entities
In distributed systems, the entities include *subjects* who are authorizers who own or control resources and requesters who make requests, *objects* which are resources and services provided by authorizers, and *privileges* which are actions executed on objects.

We define three types of constant entities, *subject, object* and *privilege.* The constant entity is every element of three disjointed constant symbol sets, *SUB, OBJ*, and *PRIV*, where *SUB* is the set of subject constants, *OBJ* the set of object constants, *PRIV* the set of privilege constants. The constant entity must start with a lower-case character.

Correspondingly the variable entity is every element of three disjointed variable symbol sets, V_{sub}, V_{obj}, and V_{priv} that range over the sets SUB, OBJ, and $PRIV$ respectively. The variable entities are prefixed with an upper-case character.

In the BNF of $\mathcal{AL}$, $\langle$*sub-con*$\rangle$, $\langle$*obj-con*$\rangle$, $\langle$*priv-con*$\rangle$, $\langle$*sub-var*$\rangle$, $\langle$*obj-var*$\rangle$, and $\langle$*priv-var*$\rangle$ represent elements of the sets SUB, OBJ, $PRIV$, V_{sub}, V_{obj}, and V_{priv} respectively.

In language $\mathcal{AL}$, we provide a special subject, *local.* It is the local authorizer which makes the authorization decision based on local policy and credentials from trusted subjects.

Atoms
An atom is a function symbol with n arguments, generally one, two or three constant or variable entities to express a logical relationship between them. There are three types of atoms:

1. $\langle$*relation-atom*$\rangle$. An atom in this type is a 2-ary function symbol and expresses the relationship of two entities. We provide three relation atoms, *neq, eq*, and *below.* The atoms *neq* and *eq* denote two same type entities equal or not equal, and *below* to denote the hierarchy structure for objects and privileges. In most realistic systems, the data information is organized using hierarchy structure, such as file systems and object oriented database system. For example, *below*$(ftp,$*pub-services*$)$ denotes that ftp is one of *pub-services*.

2. $\langle$*assert-atom*$\rangle$. This type of atoms, denoted by $exp(a_1, \ldots, a_n)$, is application dependant function symbol with n arguments, usually one, two or three constant or variable entities and states the property of the subjects, the relationship between entities. It is a kind of flexible atoms in language $\mathcal{AL}$. For example, $isaTutor(alice)$ denotes that *alice* is a tutor.

$$
\begin{array}{rcl}
\langle \textit{obj} \rangle & ::= & \langle \textit{obj-con} \rangle \mid \langle \textit{obj-var} \rangle \\
\langle \textit{priv} \rangle & ::= & \langle \textit{priv-con} \rangle \mid \langle \textit{priv-var} \rangle \\
\langle \textit{sub} \rangle & ::= & \langle \textit{sub-con} \rangle \mid \langle \textit{sub-var} \rangle \\
\langle \textit{sub-set} \rangle & ::= & \langle \textit{sub-con} \rangle \mid \langle \textit{sub-con} \rangle,\ \langle \textit{sub-set} \rangle \\
\langle \textit{sub-struct} \rangle & ::= & \langle \textit{sub} \rangle \mid \text{“[”}\langle \textit{sub-set} \rangle\text{“]”} \mid \langle \textit{threshold} \rangle \\
\langle \textit{sub-ext-set} \rangle & ::= & \langle \textit{dth} \rangle \mid \langle \textit{dth} \rangle,\ \langle \textit{sub-ext-set} \rangle \\
\langle \textit{sub-ext-struct} \rangle & ::= & \langle \textit{sub} \rangle \mid \text{“[”}\langle \textit{sub-set} \rangle\text{“]”} \mid \langle \textit{threshold} \rangle \mid \text{“[”}\langle \textit{sub-ext-set} \rangle\text{“]”} \\
\langle \textit{entity} \rangle & ::= & \langle \textit{sub} \rangle \mid \langle \textit{obj} \rangle \mid \langle \textit{priv} \rangle \\
\langle \textit{entity-set} \rangle & ::= & \langle \textit{entity} \rangle \mid \langle \textit{entity} \rangle,\ \langle \textit{entity-set} \rangle \\
\langle \textit{sign} \rangle & ::= & + \mid - \mid \Box \\
\langle \textit{relation-atom} \rangle & ::= & \textit{below}(\langle \textit{obj} \rangle, \langle \textit{obj} \rangle) \mid \textit{below}(\langle \textit{priv} \rangle, \langle \textit{priv} \rangle \mid \\
 & & \textit{neq}(\langle \textit{entity} \rangle, \langle \textit{entity} \rangle) \mid \textit{eq}(\langle \textit{entity} \rangle, \langle \textit{entity} \rangle) \\
\langle \textit{assert-atom} \rangle & ::= & \textit{exp}(\langle \textit{entity-set} \rangle) \\
\langle \textit{auth-atom} \rangle & ::= & \textit{right}(\langle \textit{sign} \rangle, \langle \textit{priv} \rangle, \langle \textit{obj} \rangle) \\
\langle k \rangle & ::= & \langle \textit{natural-number} \rangle \\
\langle \textit{threshold} \rangle & ::= & \langle \textit{sth} \rangle \mid \langle \textit{dth} \rangle \\
\langle \textit{sth} \rangle & ::= & \textit{sthd}(\langle k \rangle, \text{“[”}\langle \textit{sub-set} \rangle\text{“]”}) \\
\langle \textit{dth} \rangle & ::= & \textit{dthd}(\langle k \rangle, \langle \textit{sub-var} \rangle, \langle \textit{assert-stmt} \rangle) \\
\langle \textit{relation-stmt} \rangle & ::= & \text{“}\textit{local}\text{”}\ \textit{says}\ \langle \textit{relation-atom} \rangle \\
\langle \textit{assert-stmt} \rangle & ::= & \langle \textit{sub} \rangle\ \textit{asserts}\ \langle \textit{assert-atom} \rangle \\
\langle \textit{auth-stmt-body} \rangle & ::= & \langle \textit{sub-struct} \rangle\ \textit{grants}\ \langle \textit{auth-atom} \rangle\ \textit{to}\ \langle \textit{sub} \rangle \\
\langle \textit{auth-stmt-head} \rangle & ::= & \langle \textit{sub} \rangle\ \textit{grants}\ \langle \textit{auth-atom} \rangle\ \textit{to}\ \langle \textit{sub-ext-struct} \rangle \\
\langle \textit{delegate-stmt-body} \rangle & ::= & \langle \textit{sub-struct} \rangle\ \textit{delegates}\ \langle \textit{auth-atom} \rangle \\
 & & \textit{with depth}\ \langle k \rangle\ \textit{to}\ \langle \textit{sub} \rangle \\
\langle \textit{delegate-stmt-head} \rangle & ::= & \langle \textit{sub} \rangle\ \textit{delegates}\ \langle \textit{auth-atom} \rangle \\
 & & \textit{with depth}\ \langle k \rangle\ \textit{to}\ \langle \textit{sub-struct} \rangle \\
\langle \textit{head-stmt} \rangle & ::= & \langle \textit{relation-stmt} \rangle \mid \langle \textit{assert-stmt} \rangle \mid \\
 & & \langle \textit{auth-stmt-head} \rangle \mid \langle \textit{delegate-stmt-head} \rangle \\
\langle \textit{body-stmt} \rangle & ::= & \langle \textit{relation-stmt} \rangle \mid \langle \textit{assert-stmt} \rangle \mid \\
 & & \langle \textit{auth-stmt-body} \rangle \mid \langle \textit{delegate-stmt-body} \rangle \\
\langle \textit{list-of-body-stmt} \rangle & ::= & \langle \textit{body-stmt} \rangle \mid \langle \textit{body-stmt} \rangle, \langle \textit{list-of-body-stmt} \rangle \\
\langle \textit{rule} \rangle & ::= & \langle \textit{head-stmt} \rangle\ [\ \textit{if}\ [\ \langle \textit{list-of-body-stmt} \rangle\ ] \\
 & & [\ \textit{with absence}\ \langle \textit{list-of-body-stmt} \rangle\ ]\ ] \\
\langle \textit{query} \rangle & ::= & \langle \textit{sub} \rangle\ \textit{requests}\ (+, \langle \textit{priv} \rangle, \langle \textit{obj} \rangle) \mid \\
 & & \text{“[”}\langle \textit{sub-set} \rangle\text{“]”}\ \textit{requests}\ (+, \langle \textit{priv} \rangle, \langle \textit{obj} \rangle)
\end{array}
$$

Figure 1. BNF for the Authorization Language $-\mathcal{AL}$

3. $\langle \textit{auth-atom} \rangle$. The *auth-atom* is of the form,
   $\textit{right}(\langle \textit{sign} \rangle, \langle \textit{priv} \rangle, \langle \textit{obj} \rangle)$.
   It states the *positive* or *negative* privilege executed on the *object* based on its

arguments, $\langle sign \rangle$, $\langle obj \rangle$, and $\langle priv \rangle$. When an auth atom is used in delegation statement, the $\langle sign \rangle$ is $\Box$ to denote both positive and negative authorizations.

Statements

There are four types of statements, *relation statement, assert statement, auth statement*, and *delegation statement*. Only the local authorizer can issue the *relation statement* to denote the structured resources and privileges. We provide the body and head forms for auth statements and delegation statements.

Threshold

There are two types of threshold structures, static threshold and dynamic threshold.

The static threshold structure is of the form,

$sthd(k, [s_1, s_2, \ldots, s_n])$,

where k is the threshold value, $[s_1, s_2, \ldots, s_n]$ is the static threshold pool, and we require $k \leq n$ and $s_i \neq s_j \ for \ 1 \leq i \neq j \leq n$. This structure states that we choose k subjects from the threshold pool.

The dynamic threshold structure is of the form,

$dthd\ (k, S, \langle sub \rangle\ assert\ exp(\ldots, S, \ldots))$,

where S is a subject variable and we require that S is one argument of assert atom *exp*. This structure denotes we choose k subjects who satisfy the assert statement.

Rules

The rule is of the form,

$$\begin{aligned}&\langle head\text{-}stmt \rangle\ \ if\ \langle list\text{-}of\text{-}body\text{-}stmt \rangle\\ &\qquad with\ absence\ \langle list\text{-}of\text{-}body\text{-}stmt \rangle.\end{aligned}$$

The basic unit of a rule is a statement. Let h be a head statement and b a body statement, a rule is as follows,

$$\begin{aligned}&h_0,\ if\ b_1,\ b_2, \ldots, b_m,\\ &\qquad with\ absence\ b_{m+1}, \ldots, b_n.\end{aligned}$$

In language $\mathcal{AL}$, a rule is a local authorization policy or a credential from other subjects and the issuer of the rule is the issuer of the head statement h_0. That is the reason why we limit the issuer structure in head statements.

Query

Language $\mathcal{AL}$ supports single subject query and group subject query. They are of the forms,

sub requests $right(+, p, o)$, and
$[s_1, s_2, \ldots, s_n]$ requests $right(+, p, o)$.

Through group subject query, we implement *separation of duty* which is an important security concept. It ensures that a critical task cannot be carried out by one subject. If we grant an authorization to a group subject, we permit it only when the subjects in the group request the authorization at the same time.

2.2 Characteristics of $\mathcal{AL}$

In this subsection, we present some examples to show the expressive power of $\mathcal{AL}$.

Structured resources

In the file system of a server in a university, there is a directory *postgraduate* which has one subdirectory for each postgraduate student, such as *alice*, *bob*, and so on.

$local$ says $below(alice,\ postgraduate)$.

$local$ says $below(bob,\ postgraduate)$.

Partial delegation and authorization

A firewall system protects the *allServices*, including *ssh, ftp*, and *http.* The administrator permits *ipA* to access all the services except *ssh* and delegates this right to *ipB* and allow it redelegated within 2 steps.

$local$ grants $right(+,\ access,\ X)$ to ipA if

$local$ says $below(X,\ allServices)$, $local$ says $neq(X,\ ssh)$.

$local$ delegates $right(\Box,\ access,\ X)$ with depth 2 to ipB if

$local$ says $below(X,\ allServices)$, $local$ says $neq(X,\ ssh)$.

Separation of duty

A company chooses to have multiparty control for emergency key recovery. If a key needs to be recovered, three persons are required to present their individual PINs. They are from different departments, *managerA*, a member of management, *auditorB*, an individual from auditing, and *techC*, one individual from IT department.

$local$ grants $right(+,\ recovery,\ k)$ to $[managerA,\ auditorB,\ techC]$.

Negative authorization

In a firewall system, the administrator *sa* does not permit *ipB* to access the *ftp* services.

sa grants $right(-,\ access,\ ftp)$ to ipB.

Incomplete information reasoning

In a firewall system, the administrator *sa* permit a *person* to access the *mysql* service if the human resource manager *hrM* asserts the person is a *staff* and not in holiday.

sa grants $right(+,\ access,\ mysql)$ to X if

$hrM\ asserts\ isStaff(X)$, with absence $hrM\ asserts\ inHoliday(X)$.

3 Semantics of $\mathcal{AL}$

In this section, we define the semantics for language $\mathcal{AL}$ through translating it to Answer Set Programming based language $\mathcal{L}_{Ans}$. We first present the definition for the *domain description* $\mathcal{D}_{\mathcal{AL}}$ and how to answer queries $\mathcal{Q}_{\mathcal{AL}}$ of language $\mathcal{AL}$. Queries are the requests in $\mathcal{AL}$. In subsection 3.1, we introduce the language $\mathcal{L}_{Ans}$ briefly. In the following subsection, we define function $TransRules(\mathcal{D}_{\mathcal{AL}})$ to translate $\mathcal{D}_{\mathcal{AL}}$ into program $\mathcal{P}$ of $\mathcal{L}_{Ans}$, and function $TransQuery(\mathcal{Q}_{\mathcal{AL}})$ to translate query $\mathcal{Q}_{\mathcal{AL}}$ into program Π and ground literals $\varphi(+)$ and $\varphi(-)$. We use $\varphi(+)$ to denote positive right and $\varphi(-)$ to denote negative right. There is detailed description for them in section 3.2.3. We solve a query based on $\mathcal{P}$, Π and φ via *Smodels.*

An answer set program may have one, more than one, or no answer sets at all. For a given program Π and a ground atom φ, we say Π *entails* φ, denoted by $\Pi \models \varphi$, iff φ is in every answer set of Π.

Definition 1 *A* domain description $\mathcal{D}_{\mathcal{AL}}$ *of language* $\mathcal{AL}$ *is a finite set of rules.*

Definition 2 *Given a domain description* $\mathcal{D}_{\mathcal{AL}}$ *and a query* $\mathcal{Q}_{\mathcal{AL}}$ *of language* $\mathcal{AL}$*, there are* $\mathit{TransRules}(\mathcal{D}_{\mathcal{AL}}) = \mathcal{P}$ *and* $\mathit{TransQuery}(\mathcal{Q}_{\mathcal{AL}}) = \Pi \cup \varphi(+) \cup \varphi(-)$*. We say that query* $\mathcal{Q}_{\mathcal{AL}}$ *is* permitted, denied, *or* unknown *by the domain description* $\mathcal{D}_{\mathcal{AL}}$ *iff* $(\mathcal{P} \cup \Pi) \models \varphi(+)$*,* $(\mathcal{P} \cup \Pi) \models \varphi(-)$*, or* $(\mathcal{P} \cup \Pi) \not\models \varphi(+)$ *and* $(\mathcal{P} \cup \Pi) \not\models \varphi(-)$ *respectively.*

3.1 An overview of language $\mathcal{L}_{Ans}$

In this subsection, we first briefly introduce language $\mathcal{L}_{Ans}$, and then give the propagation rules, authorization rules, and conflict resolution and decision rules in $\mathcal{L}_{Ans}$.

Language $\mathcal{L}_{Ans}$ is based on Answer Set Programming [1] and we use *Smodels* as the solver of $\mathcal{L}_{Ans}$ which has some extended features such as constraint and conditional literals to express the threshold structures [20]. The alphabet of language $\mathcal{L}_{Ans}$ includes *entity sorts, function symbols* and *predicates symbols.* $\mathcal{L}_{Ans}$ has constant entities beginning with a lowercase letter, and variable entities beginning with a uppercase letter. For both of them, there are three types of entities respectively, *subject, privilege*, and *object.* We define two function symbols, $right(sign, priv, obj)$ and $exp(a_1, \ldots, a_n)$, where *priv* is of privilege sort, *obj* of object sort, and a_i of any entity sort. In *Smodels*, the both functions are symbolic functions which just defines a new constant as an argument for the predicates in the application. After grounding, there are no any variables in symbolic functions and they are just ordinary constants. We also define predicates for $\mathcal{L}_{Ans}$, including *below, assert, auth, delegate, grant, ggrant*, and so on. The detailed description for language $\mathcal{L}_{Ans}$ is listed in Appendix A.

For an access control system, the authorization policy is the key component. We need to indicate that it is easy and flexible for $\mathcal{L}_{Ans}$ to specify different types of policies. In the following subsections, we will only present some parts of rules for authorization policies to demonstrate the expressiveness of $\mathcal{L}_{Ans}$ because of a space limitation. Readers are referred to our full paper for the complete set of rules [21].

3.1.1 Propagation rules

In most real world situations, the work to assign all the authorizations is burdensome and not necessary. The security officer prefers to assign them partly and propagate them based on propagation policy. There are various different propagation policies in real world application. Basically, there are three types: (1) No propagation; (2) Propagation without considering whether there are conflicts with previous authorization; (3) Propagation with considering preferences. Here we choose the second one as an example to show how to write propagation rules using $\mathcal{L}_{Ans}$ (We also use them for our scenario in section 4). We leave the conflicts to be solved by conflict resolution and decision rules.

$$\begin{aligned} &auth(S_1, S_2, right(Sign, P, Obj_2), T) \leftarrow \\ &\quad auth(S_1, S_2, right(Sign, P, Obj_1), T),\ below(Obj_1, Obj_2). \end{aligned} \tag{1}$$

$$below(A_1, A_3) \leftarrow\ below(A_1, A_2),\ below(A_2, A_3). \tag{2}$$

The rule (1) is for object propagation. We have a same rule for privilege propagation. The rule (2) is for structured data propagation.

3.1.2 Authorization rules

In this subsection, we present the authorization rules for the following authorization policy: if there is only positive authorization and no negative authorization, we conclude positive authorization; if there is no positive authorization, we grant negative authorization; if there are positive and negative authorizations at the same time, We leave the decision problem to conflict resolution and decision policy.

$$\begin{aligned} &grant(X, right(+, P, O)) \leftarrow auth(local, X, right(+, P, O), T), \\ &\quad not\ exist\_neg(X, right(-, P, O)),\ not\ exist\_subneg(X, right(-, P, O)). \end{aligned} \tag{3}$$

$$\begin{aligned} &ggrant(l, right(+, P, O)) \leftarrow auth(local, X, right(+, P, O), T), \\ &\quad match(X, right(+, P, O)),\ not\ exist\_neg(X, right(-, P, O)), \\ &\quad not\ exist\_subneg(X, right(-, P, O)). \end{aligned} \tag{4}$$

We provide positive authorization rules (3) and (4). In rule (4), l is a special group subject entity to represent the set of subjects who make a request together. We have rules for *exit_neg, exit_pos, exit_subneg, exit_subpos, match*, and negative authorizations [21].

3.1.3 Conflict resolution and decision rules

When both positive and negative authorizations are permitted, the conflict occur. Existing approaches for handling conflicts include: (1) no conflict policy. It relies on the security administrator to write the consistent authorization rules. If there are conflicts, errors happen; (2) no decision policy. When conflicts occur, the system neither permits nor denies the request; (3) a policy based on relative authorization or specification. For example, when conflicts occur, the system chooses denial-take-preference or permission-take-preference; (4) a decision based on ordered authorization rules. In this paper, we consider *delegation* as an action and get the step for each authorization which is decided by the delegation step. All the authorizations arise from *local* originally and then the step number denotes how far the authorization is away from *local*. We take the smallest step authorization preference. If the conflict occurs with the same step, we deny the request. Our approach belongs to the third category. The following are some of rules for our conflict resolution and decision policy.

$$\begin{aligned}grant(X, right(+,P,O)) \leftarrow\ & auth(local, X, right(+,P,O), T1),\\ & auth(local, X, right(-,P,O), T2),\ neg\_far(X, right(-,P,O), T2),\\ & not\ pos\_far(X, right(+,P,O), T1),\ not\ exist\_subneg(X, right(-,P,O)).\end{aligned} \tag{5}$$

$$\begin{aligned}ggrant(l, right(+,P,O)) \leftarrow\ & auth(local, X, right(-,P,O), T2),\\ & neg\_far(X, right(-,P,O), T2),\ match(X, right(+,P,O)),\\ & auth(local, X, right(+,P,O), T1),\ not\ pos\_far(X, right(+,P,O), T1),\\ & not\ exist\_subneg(X, right(-,P,O)).\end{aligned} \tag{6}$$

The rule (5) and (6) specify the policy we take positive authorization if positive and negative authorizations coexist and positive authorization has smaller step than negative authorization.

3.2 Transformation from $\mathcal{AL}$ to $\mathcal{L}_{Ans}$

A rule $r_{\mathcal{D}}$ in the domain description $\mathcal{D}_{\mathcal{AL}}$ is of the following form,

$$h_0\ if\ b_1,\ b_2, \ldots, b_m,\ with\ absence\ b_{m+1}, \ldots,\ b_n. \tag{7}$$

where h_0 is *head statement* denoted by $head(r_{\mathcal{D}})$ and b_is are *body statements* denoted by $body(r_{\mathcal{D}})$. We call the set of statements, $\{b_1,\ b_2, \ldots, b_m\}$, *positive body statements*, denoted by $pos(r_{\mathcal{D}})$ and the set of statements, $\{b_{m+1},\ b_{m+2}, \ldots, b_n\}$ *negative body statements*, denoted by $neg(r_{\mathcal{D}})$. If there is no confusion in context, we use *positive statements* and *negative statements* to express them respectively. In (7), if $m = 0$ and $n = 0$, the rule is h_0 called a *fact*.

In the next subsections we provide translation functions for $\mathcal{D}_{\mathcal{AL}}$ and $\mathcal{Q}_{\mathcal{AL}}$. The function $TansRules(\mathcal{D}_{\mathcal{AL}})$ translates the rules in the domain description $\mathcal{D}_{\mathcal{AL}}$ into an answer set program $\mathcal{P}$. We divide the process into three phases, body translation, head translation, and adding rules in section 3.1.1, 3.1.2, and 3.1.3. For a query in language $\mathcal{AL}$, we provide $TransQuery(\mathcal{Q}_{\mathcal{AL}})$ to translate it into a program Π and ground literals $\varphi(+)$ and $\varphi(+)$.

In language $\mathcal{AL}$, there are function symbols, *assert-atom* and *auth-atom*. Correspondingly there are functions *exp* $(a_1, \ldots, a_n)$ and *right* $(sign, priv, obj)$ in language $\mathcal{L}_{Ans}$. In our translation, if there is no confusion in the context, we use *exp* and *right* to denote them in both languages.

3.2.1 Body transformation

In language $\mathcal{AL}$, there are four types of body statements, *relation statement, assert statement, auth statement*, and *delegation statement.* As auth statement and delegation statement have similar structure, we give their transformation together. For each rule $r_{\mathcal{D}}$, its body statement b_i is one of the following cases.

1. Relation statement:
   local says $below(arg_1,\ arg_2)$,
   local says $neq(arg_1,\ arg_2)$, and *local* says $eq(arg_1,\ arg_2)$.

Replace them respectively in program $\mathcal{P}$ using:

$$below(arg_1,\ arg_2), \tag{8}$$

where arg_1 and arg_2 are of *object or privilege entity sort*,
$neq(arg_1,\ arg_2)$ and $eq(arg_1,\ arg_2)$,
where arg_1 and arg_2 are of same type entity sort to specify they are equal or not equal. In *Smodels*, *neq* and *eq* are internal function and work as a constraint for the variables in the rules.

2. Assert body statement:
   issuer asserts *exp*.

   Replace it in program $\mathcal{P}$ using,

$$assert(issuer, exp), \tag{9}$$

   where *issuer* is a subject constant or variable, and *exp* is an assertion.

3. Auth body statement or delegation statement:
   issuer grants *right* to *grantee*, or
   issuer delegates *right* with depth k to *delegatee*.

   If *issuer* is a subject constant or variable, we replace the statements in program $\mathcal{P}$ using,

$$auth(issuer, grantee, right, T),\ or \tag{10}$$

$$delegate(issuer, delegatee, right, k, Step), \tag{11}$$

   where T is a step variable that means how many steps the right has gone through from *issuer* to *grantee*, k delegation depth, and *Step* length variable that the delegation has gone through.

   If *issuer* is a set of subjects, $[s_1, \ldots, s_n]$, for auth statements, we replace them in program $\mathcal{P}$ by conjunction forms of (10) as,
   $auth(s_1, grantee, right, T_1), \ldots\ auth(s_n, grantee, right, T_n)$.

   If *issuer* is a static threshold structure, $sthd(k, [s_1, s_2, \ldots, s_n])$, we use choice rule to replace them as follows,
   $k\{auth(s_1, grantee, right, T_1), \ldots\ auth(s_n, grantee, right, T_n)\}k$.

   If *issuer* is a dynamic threshold structure, $dthd(k, S, assert(sub, exp(S))$, we use choice rule including constraint literal to replace them using,
   $k\{auth(S, grantee, right, T_i) :\ assert(sub, exp(S))\}k$.

The translation for delegation body statements is to replace (10) by (11) in previous forms.

We translate the positive statements as above steps, and for the negative body statements, we do the same translation and just add *not* before them.

3.2.2 Head transformation

In language $\mathcal{AL}$, there are four types of head statements, *relation statement, assert statement, auth statement*, and *delegation statement*. If the head statement h_0 is a *relation statement* or an *assert statement*, the translation is same as the body statements. We adopt the rules (8), (9) to translate them respectively. In relation head statements, there are no statements for atom *neq* and *eq* that just be used as a variable constraints in body statements. Here we present the translation for *assert head statement*, and *delegation head statement*.

1. Auth head statement:

   issuer grants *right* to *grantee*.

   If *grantee* is a subject constant or variable, we replace it by,

   $auth(issuer, grantee, right, 1)$,

   where 1 means the *right* is granted from *issuer* to *grantee* directly.

   If *grantee* is a complex structure, *subject set, threshold*, or *subject extent set*, we introduce group subject entity l_{new} to denote the subjects in complex subject structures, and replace its head in program $\mathcal{P}$ as follows,

   $auth(issuer, l_{new}, right, 1)$.

   We add different rules for different structures.

   case 1: $[s_1, \ldots, s_n]$

   $match(l_{new}, right) \leftarrow\ auth(issuer, l_{new}, right, 1),\ n\{req(s_1, right), \ldots, req(s_n, right)\}n.$

   case 2: $sthd(k, [s_1, s_2, \ldots, s_n])$

   $match(l_{new}, right) \leftarrow\ auth(issuer, l_{new}, right, 1),\ k\{\ req(s_1, right), \ldots,\ req(s_n, right)\}k.$

   case 3: $dthd\ (k, S, sub\ assert\ exp(S)\ )$

   $match(l_{new}, right) \leftarrow\ auth(issuer, l_{new}, right, 1),\ k\{\ req(S, right) :\ assert(sub, exp(S))\ \}k.$

   case 4: $[dthd\ (k_1, S, s_1\ assert\ exp_1(S)\ ), \ldots,\ dthd\ (k_n, S, s_n\ assert\ exp_n(S)\ )]$.

   $holds(l_{new}, exp_1(S)) \leftarrow\ auth(issuer, l_{new}, right, 1),\ assert(s_1, exp_1(S)),\ req(S, right).$

   $\vdots$

   $holds(l_{new}, exp_n(S)) \leftarrow\ auth(issuer, l_{new}, right, 1),\ assert(s_n, exp_n(S)),\ req(S, right).$

   $match(l_{new}, right) \leftarrow\ k_1\{holds(l_{new}, exp_1(S))\}k_1, \ldots\ k_n\{holds(l_{new}, exp_n(S))\}k_n.$

2. Delegation head statement:

   issuer delegates *right* with depth k to *delegatee*

   If *delegatee* is a subject constant or variable, we replace the statement in program P using,

   $delegate(issuer, delegatee, right, k, 1)$.

   where k is the delegation depth, and 1 means the *issuer* delegates the *right* to *delegatee* directly.

   Moreover, we need to add the following implied rules for it in program $\mathcal{P}$.

   Auth-delegation rules: When the issuer delegates a right to the delegatee, the issuer will agree with the delegatee to grant the right to other subjects within delegation depth. The authorization step increases 1. Since we consider structured resources and privileges, there are three auth-delegation rules.

$$\begin{aligned}&auth(issuer, S, right(Sn, P, O), T+1) \leftarrow\\&\quad delegate(issuer, delegatee, right(\Box, P, O), k, 1),\\&\quad auth(delegatee, S, right(Sn, P, O), T).\end{aligned}$$

$$\begin{aligned}&auth(issuer, S, right(Sn, P, SO), T+1) \leftarrow\\&\quad delegate(issuer, delegatee, right(\Box, P, O), k, 1),\\&\quad auth(delegatee, S, right(Sn, P, SO), T),\ below(SO, O).\end{aligned}$$

$$\begin{aligned}&auth(issuer, S, right(Sn, SP, O), T+1) \leftarrow\\&\quad delegate(issuer, delegatee, right(\Box, P, O), k, 1),\\&\quad auth(delegatee, S, right(Sn, SP, O), T),\ below(SP, P).\end{aligned}$$

Dele-chain rules: The delegation can be redelegated within delegation depth. We also have three dele-chain rules for structured resources and privileges. Here we just give one of them.

$$\begin{aligned}&delegate(issuer, S, right(\Box, P, O), min(k\text{-}Step, Dep), 1+Step) \leftarrow\\&\quad delegate(issuer, delegatee, right(\Box, P, O), k, 1),\\&\quad delegate(delegatee, S, right(\Box, P, O), Dep, Step),\ Step\ <\ k.\end{aligned}$$

Self-delegation rule: The delegatee can delegate the right to himself/herself within k depth.

$$\begin{aligned}&delegate(delegatee, delegatee, right, Dep, 1) \leftarrow\\&\quad delegate(issuer, delegatee, right, k, 1),\ Dep \leq k.\end{aligned}$$

Weak-delegation rule: If there is a delegation with k steps, we can get the delegation with steps less than k.

$$\begin{aligned}&delegate(issuer, delegatee, right, Dep, 1) \leftarrow\\&\quad delegate(issuer, delegatee, right, k, 1),\ Dep < k.\end{aligned}$$

If *delegatee* is a complex structure, *subject set, static threshold*, or *dynamic threshold*, we introduce a new group subject l_{new} to denote the subjects in complex structures, and replace the statement in program $\mathcal{P}$ using,

$$delegate(issuer, l_{new}, right, k, 1).$$

We need to add *auth-delegation* and *dele-chain* rules for them. There are similar rules for them, and here we present the rules for *subject set* structure.

Auth-delegation rule:

$$\begin{aligned}&auth(issuer, S, right, T+1) \leftarrow\ delegate(issuer, l_{new}, right, k, 1),\\&\quad auth(s_1, S, right, T_1),\ \ldots,\ auth(s_n, S, right, T_n),\ T = max(T_1, \ldots, T_n).\end{aligned}$$

Dele-chain rule:

$$\begin{aligned}&delegate(sub, S, right, T_1, T_2) \leftarrow\ delegate(sub, l_{new}, right, k, 1),\\&\quad delegate(s_1, S, right, Dep_1, Step_1),\ \ldots,\ delegate(s_n, S, right, Dep_n, Step_n),\\&\quad T_1 = min(k\text{-}Step_1, \ldots, k\text{-}Step_n, Dep),\ T_2 = max(1+Step_1, \ldots, 1+Step_n),\\&\quad T_1 > 0.\end{aligned}$$

3.2.3 Query Transformation

In language $\mathcal{AL}$, there are two kinds of queries, single subject query and group subject query. We present the function $\mathit{TransQuery}(\mathcal{Q}_{\mathcal{AL}})$ for both of them and this function returns program Π and ground literals $\varphi(+)$ and $\varphi(-)$.

If $\mathcal{Q}_{\mathcal{AL}}$ is a single subject query,
s *requests* $right(+, p, o)$,
TransQuery returns program Π and ground literals $\varphi(+)$ and $\varphi(-)$ as follows respectively,
$\{req(s, right(+, p, o))\}$, $grant(s, right(+, p, o))$ and $grant(s, right(-, p, o))$.

If $\mathcal{Q}_{\mathcal{AL}}$ is a group subject query,
$[s_1, s_2, \ldots, s_n]$ *requests* $right(+, p, o)$.
TransQuery returns program Π and ground literals $\varphi(+)$ and $\varphi(-)$ as follows respectively,
$\{req(\ s_i, right(+, p, o)) \mid i = 1, \ldots, n\ \}$, $ggrant(l, right(+, p, o))$ and $ggrant(l, right(-, p, o))$, where l is a group subject entity to denote the set of subjects, $[s_1, \ldots, s_n]$.

4 A Scenario

In this section we represent a specific authorization scenario to demonstrate the features of language $\mathcal{AL}$.

Scenario: A company chooses to have multiparty control for emergency key recovery. If a key needs to be recovered, three persons are required to present their individual PINs. They are from different departments, a member of management, an individual from auditing, and one individual from IT department. The system trusts the manager of Human Resource Department to identify the staff of the company. The domain description $\mathcal{D}_{\mathcal{AL}}$ for this scenario is the following rules represented using language $\mathcal{AL}$.

$local$ grants $right(+, recover, key)$ to
$[\ dthreshold(1, X, hrM$ asserts $isAManager(X))$,
$dthreshold(1, Y, hrM$ asserts $isAnAuditor(Y))$,
$dthreshold(1, Z, hrM$ asserts $isATech(Z))\ ]$.
hrM asserts $isAManager(alice)$.
hrM asserts $isAnAuditor(bob)$.
hrM asserts $isAnAuditor(carol)$.
hrM asserts $isATech(david)$.

We translate them into language $\mathcal{L}_{Ans}$,

$auth(local, l_{key}, right(+, recovery, key), 1)$.
$holds(l_{key}, isAManager(X)) \leftarrow\ auth(local, l_{key}, right(+, recovery, key), 1),$
$assert(hrM, isAManager(X)),\ req(X, right(+, recovery, key))$.
$holds(l_{key}, isAnAuditor(X)) \leftarrow\ auth(local, l_{key}, right(+, recovery, key), 1),$
$assert(hrM, isAnAuditor(X)),\ req(X, right(+, recovery, key))$.
$holds(l_{key}, isATech(X)) \leftarrow\ auth(local, l_{key}, right(+, recovery, key), 1),$

$assert(hrM, isATech(X)),\ req(X, right(+, recovery, key)).$
$match(l_{key}, right(+, recovery, key)) \leftarrow$
$1\{holds(l_{key}, isAManager(X))\}1,$
$1\{holds(l_{key}, isAnAuditor(Y))\}1,$
$1\{holds(l_{key}, isATech(Z))\}1.$

In this scenario, the program $\mathcal{P}$ consists of the above translated rules, and those authorization rules we specified in section 3.1.2[1]. If Alice, Bob, and David make a request to recover a key together, that is,

$[alice, bob, david]$ requests $right(+, recovery, key)$.

After translation, we get program Π,
$req(alice, right(+, recovery, key)),$
$req(bob, right(+, recovery, key)),$
$req(david, right(+, recovery, key)),$
and the ground literal $\varphi(+)$ is,
$grant(l, right(+, recovery, key)),$
where l is a group subject entity to represent the set of subjects, $[alice, bob, david]$.

Then program $\mathcal{P} \cup \Pi$ (Refer to the Appendix B for complete program) has only one answer set, and $ggrant(l, right(+, recovery, key))$ is in the answer set. Therefore the request is permitted.

Now if we consider that Alice, Bob, and Carol make the same request, they cannot satisfy the rule for $match$, then $ggrant(l, right(+, recovery, key))$ is not in the answer set. Instead, we get $ggrant(l, right(-, recovery, key))$, then the request will be denied.

5 Conclusion and Future Work

In this paper, we developed an expressive authorization language $\mathcal{AL}$ to specify the distributed authorization with delegation. We used Answer Set Programming as a foundational basis for its semantics. As we have showed earlier, $\mathcal{AL}$ has a rich expressive power which can represent positive and negative authorization, structured resources and privileges, partial authorization and delegation, and separation of duty. It is worth mentioning that language $\mathcal{AL}$ can represent all the scenarios discussed by Delegation Logic [17]. Moreover, as we have illustrated in section 4, $\mathcal{AL}$ can also represent complex authorization scenarios which Delegation Logic cannot.

We should indicate that our formulation also has implementation advantages due to recent development of Answer Set Programming technology in AI community.[2] The scenario in section 4 has been fully implemented through Answer Set Programming (See Appendix B).

Our paper leave space for future work. One issue we plan to investigate is using preference of policy rules for conflict resolution which is more reasonable and flexible in some real applications. We also plan to investigate how to find the authorization path (Trust path) based on answer sets.

[1]A complete answer set logic program for this translation is referred to Appendix B.
[2]Please refer to http://www.tcs.hut.fi/Software/smodels/index.html

References

[1] C. Baral. Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, 2003. ISBN 0521818028.

[2] E. Bertino, F. Buccafurri, E. Ferrari, and P. Rullo. A Logical Framework for Reasoning on Data Access Control Policies. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop(CSFW-12)*, pages 175-189, IEEE Computer Society Press, Los Alamitos, CA, 1999.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the Symposium on Security and Privacy*, IEEE Computer Society Press, Los Alamitos,1996, pages 164-173.

[4] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance-checking in the PolicyMaker trust management system. In *Proceedings of Second International Conference on Financial Cryptography (FC'98)*, volume 1465 of Lecture Notes in Computer Science, pages 254-274. Springer, 1998.

[5] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The Role of Trust Management in Distributed Systems. *Secure Internet Programming*, Lecture Note of Computer Science, vol. 1603, pages 185-210, Springer, Berlin, 1999.

[6] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust-Management System, Version 2, Internet Engineering Task Force RFC 2704, September 1999. http://www.ietf.org/rfc/rfc2704.txt

[7] D. Clarke, J. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate Chain Discovery in SPKI/SDSI, manuscript, Nov 1999.

[8] J. Elien. Certificate Discovery Using SPKI/SDSI 2.0 Certificates. Masters Thesis, MIT LCS, May 1998, http://theory.lcs.mit.edu/ cis/theses/elien-masters.ps.

[9] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. Internet Engineering Task Force RFC 2693, September 1999. http://www.ietf.org/rfc/rfc2693.txt

[10] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Simple Public Key Certificate, Internet Draft (Work in Progress), July1999. http://world.std.com/ cme/spki.txt

[11] M.Gelfond and V.Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. of the Fifth Int'L Conf. and Symp.*, pages 1070-1080. MIT Press, 1988.

[12] ITU-T Rec. X.509 (revised), The Directory - Authentication Framework, International Telecommunication Union, 1993.

[13] S. Jajodia, P. Samarati, and V. S. Subrahmanian. Flexible Support for Multiple Access Control Policies. In*ACM Transactions on Database Systems*, Vol.26, No.2, June 2001, Pages 214-260.

[14] S. T. Kent. Internet Privacy Enhanced Mail, Communications of the ACM, 36:8, pages 48-60, August 1993.

[15] N. Li, J. Feigenbaum, and B.N. Grosof. A logic-based knowledge representation for authorization with delegation (extended abstract). In *Proceedings of the IEEE Computer Security Foundations Workshop (CSFW-12)*(June). IEEE Computer Society Press, Los Alamitos, Calif., pages 162-174.

[16] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. In *Journal of Computer Security*, volume 11, number 1, pages 35-86, February 2003.

[17] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. In*ACM Transactions on Information and System Security (TISSEC)*, February 2003.

[18] I. Niemela and P. Simons, and T. Syrjanen, Smodels: A system for answer set programming. In *proceedings of the 8th International Workshop on Non-monotonic Reasoning.* USA 2000.

[19] R. L. Rivest, and B. Lampson. SDSI - A Simple Distributed Security Infrastructure, October 1996. Available at http://theory.lcs.mit.edu/ rivest/sdsi11.html

[20] T. Syrjänen. Lparse 1.0 User's Mannual. http://www.tcs.hut.fi/Software/smodels.

[21] S. Wang, and Y. Zhang. Handling Distributed Authorization with Delegation through Answer Set Programming(manuscript). 2005.

Appendix A

The language alphabet of $\mathcal{L}_{Ans}$

1. Entity Sort:
   There are three types of constant entities, *subject*, *object*, and *privilege.* The subject entity sort includes group subject entities introduced in translation process to denote a set of subjects. l is a particular group subject to denote the set of subjects who make a request together. All the constant entities start with a lowercase characters.

   There are three disjointed variable sets, the sets of subject variables, object variables, and privilege variables that range over the constant entities respectively. The variable entities begin with a uppercase characters.

2. Function symbols:
   right($sign$, $priv$, obj), where $sign$ is $+$, $-$ or $\square$, $priv$ privilege sort, obj object sort.

   exp($a_1, \ldots, a_n$), where a_i is one of the entity sort.

   In *Smodels*, the above both functions are symbolic functions which just defines a new constant as an argument for the predicates in the application. We define them just to combine the related arguments together to express a right or an assertion which are parameters for predicates *auth*, *delegate*, and *assert*. After the rules in the program are grounded, there are no any variables in both functions and they are just ordinary constant arguments for the related predicates.

3. Predicate symbols:
   below($arg1$, $arg2$), where $arg1$ and $arg2$ are of the same kind of *entity sort* to denote partial order relationship in a hierarchy structure.

   assert(*issuer*, $exp(a_1, \ldots, a_n)$), where *issuer* is of *subject sort*, *exp* is an application dependant function of n arguments that are of *entity sort*.

   auth(*issuer*, *grantee*, $right(sign, priv, obj), time$), where *issuer* and *grantee* are both of *subject entity* sort, *time* is a natural number or variable which means how many steps the *right* goes through from *issuer* to *grantee*.

   delegate(*issuer***,** *delegatee***,***right*(*sign*, *priv*, *obj*) **,** *depth***,** *step***)**, where *issuer* and *delegatee* are of *subject entity* sorts, *depth*, and *step* are natural numbers or variables. *depth* states how far the *right* can be delegated further. *step* states how many steps the delegation has gone through.

   grant(*sub*, $right(sign, priv, obj)$**)**, where *sub* is of *subject entity* sort. It states that the $right(sign, priv, obj)$ is granted to *sub*.

   ggrant(*sub*, $right(sign, priv, obj)$**)**, where *sub* is one of *subject group entities* introduced during the translation process. It states that the $right(sign, priv, obj)$ is granted to a set of subjects.

   req(*sub*, $right(+, priv, obj)$), where *sub* is of *subject entity* sort. It states that *sub* requests the $right(+, priv, obj)$.

   holds(*sub*, $exp(s)$), where *sub* is one of *subject group entities* introduced during the translation process, $exp(s)$ assert atom. It states that the group subject includes subject *s*.

   match($sub, right(+, priv, obj)$), where *sub* is one of *group subject entities* introduced during the translation process. It states that the subject group requests *right* together.

   exist_pos($sub, right(+, priv, obj)$), where *sub* is of subject entity sort. It states there is positive *privilege* on *obj* for *sub*.

   exist_subpos($sub, right(+, priv, obj)$), where *sub* is of subject entity sort. It states there is partial positive *privilege* on *obj* for *sub*.

   exist_neg($sub, right(-, priv, obj)$), where *sub* is of subject entity sort. It states there is negative *privilege* on *obj* for *sub*.

   exist_subneg($sub, right(-, priv, obj)$), where *sub* is of subject entity sort. It states there is partial negative *privilege* on *obj* for *sub*.

   pos_far($sub, right(+, priv, obj), time$), where *sub* is of subject entity sort. It states that there is at lease a positive authorization for *sub* that has more steps than some negative authorizations.

neg_far$(sub, right(-, priv, obj), time)$, where sub is of subject entity sort. It states that there is at lease a negative authorization for sub that has more steps than some positive authorizations.

Appendix B

The program for the scenario in section 4

```
time(1..6).
subjects(alice;bob;carol;david;list).
#domain subjects(X),subjects(Y),subjects(Z).
#domain time(T),time(T1),time(T2).

% Beginning of translation
assert(hrM,isAManager(alice)). assert(hrM,isAnAuditor(bob)).
assert(hrM,isAnAuditor(carol)). assert(hrM,isATech(david)).

% For auth transformation.
auth(local,list,right(pp,recovery,key),1).
holds(list,isAManager(X)):- auth(local,list,right(pp,recovery,key),1),
      assert(hrM,isAManager(X)), req(X,right(pp,recovery,key)).
holds(list,isAnAuditor(X)):-auth(local,list,right(pp,recovery,key),1),
      assert(hrM,isAnAuditor(X)), req(X,right(pp,recovery,key)).
holds(list,isATech(X)):- auth(local,list,right(pp,recovery,key),1),
      assert(hrM,isATech(X)), req(X,right(pp,recovery,key)).
match(list,right(pp,recovery,key)):- 1{holds(list,isAManager(X))}1,
      1{holds(list,isAnAuditor(Y))}1, 1{holds(list,isATech(Z))}1.

% Request transformation
req(alice,right(pp,recovery,key)).
req(bob,right(pp,recovery,key)).
req(david,right(pp,recovery,key)).

% Authorization rule.
% No structured resources and privileges,
% we do not need exist_subpos and exist_subneg.
exist_pos(X,right(pp,recovery,key)):-auth(local,X,right(pp,recovery,key),T).
exist_neg(X,right(mm,recovery,key)):-auth(local,X,right(mm,recovery,key),T).
ggrant(l,right(pp,recovery,key)):-auth(local,X,right(pp,recovery,key),T),
      match(X,right(pp,recovery,key)), not exist_neg(X,right(mm,recovery,key)),
      not exist_subneg(X,right(mm,recovery,key)).
ggrant(l,right(mm,recovery,key)):-auth(local,X,right(mm,recovery,key),T),
      match(X,right(pp,recovery,key)), not exist_pos(X,right(pp,recovery,key)).
ggrant(l,right(mm, recovery,key)):-
      not ggrant(l,right(pp,recovery,key)).
```