Logic Programming Based Formal Representations For Authorization and Security Protocols

Shujing Wang

A thesis submitted for the degree of
Doctor of Philosophy at
University of Western Sydney

February 2008

Except where otherwise indicated, this thesis is my own original work. I certify that this thesis contains no material that has been submitted previously, in whole or in part, for the award of any other academic degree.

Shujing Wang
16 February 2008

To My Family

Acknowledgements

First and foremost, I would like to thank my principal supervisor, Professor Yan Zhang for his invaluable guidance and support, without which, this work would not have been possible. Yan offered me hours of his precious time each week to discuss with me the details of this work during my doctoral research endeavor in the past four years. His inspirational guidance and constructive advice taught me a lot about doing research. Moreover, his enlightenment and encouragement will benefit my whole life. I would also like to thank my co-supervisor, Dr. Yun Bai, who is always ready to help, both with academic work and personal issues. In particular, Yun has taken the time to read the thesis and provide me with technical corrections and countless suggestions.

This research has been conducted with the support of the International Postgraduate Research Scholarship provided by the Australian government and the University of Western Sydney. I would like to express my gratitude to them for the finance and research facility supports, without which, I would not have been able to complete my doctorial work.

Special thanks go to Dr. Tran Cao Son from the New Mexico State University and Dr. Kim-Kwang Raymond Choo from the Australian Institute of Criminology. Tran has provided me with constructive suggestions and technical comments about protocol specifications using logic programming. Raymond gave me invaluable advice on analyzing protocols.

I am grateful to the academic staff and research colleagues in the Intelligent Systems Laboratory research group, who provided an excellent research environment during my doctoral studies.

I am greatly indebted to my parents for their continual support and love, not only during the period I was working on this thesis, but also throughout many years of my life. My parents unconditionally help me to take care of my newborn son, which made it possible for me to complete this thesis. I am especially grateful to my husband Shengjie for his love at all times. He provided steadfast support, encouragement, and understanding to my doctorial education.

Last, but by no means the least, I thank the birth of my son, Ximing, which teaches me to appreciate what I have in life.

Abstract

Logic programming with answer set semantics has been considered appealing rule-based formalism language and applied in information security areas. In this thesis, we investigate the problems of authorization in distributed environments and security protocol verification and update.

Authorization decisions are required in large-scale distributed environments, such as electronic commerce, remote resource sharing, *etc.* We adopt the trust management approach, in which authorization is viewed as a "proof of compliance" problem. We develop an authorization language $\mathcal{AL}$ with nonmonotonic feature as the policy and credential specification language, which can express delegation with depth control, complex subject structures, both positive and negative authorizations, and separation of duty concepts. The theoretical foundation for language $\mathcal{AL}$ is the answer set semantics of logic programming. We transform $\mathcal{AL}$ to logic programs and the authorization decisions are based on answer sets of the programs. We also explore the tractable subclasses of language $\mathcal{AL}$.

We implement a fine grained access control prototype system for XML resources, in which the language $\mathcal{AL}^*$ simplified from $\mathcal{AL}$ is the policy and credential specification language. We define *XPolicy*, the XML format of $\mathcal{AL}^*$, which is a DTD for the XML policy documents. The semantics of the policy is based on the semantics of language $\mathcal{AL}$. The system is implemented using Java programming.

We investigate the security protocol verification problem in provable security approach. Based on logic programming with answer set semantics, we develop a unified framework for security protocol verification and update, which integrates protocol specification, verification and update. The update model is defined using forgetting techniques in logic programming. Through a case study protocol, we demonstrate an application of our approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

For a long time, logic programming and rule-based formalisms have been considered appealing policy specification languages, which have been demonstrated by a large body of literatures [4, 6, 58, 70, 82]. Since Answer Set Programming (ASP, logic programming with answer set semantics) emerged as a new logic programming paradigm in the late 1990s, having its roots in nonmonotonic reasoning, deductive database and logic programming with negation as failure, it has been regarded as a primary candidate for an effective knowledge representation tool. This view has been boosted by the emergence of highly efficient solvers for ASP, such as *Smodels*[74], and *Dlv*[50, 51]. The basic idea of ASP is to represent a given problem by a logic program each of whose answer sets correspond to one of solutions, and then use an answer set solver for computing answer sets of the program.

The most common type of policies are security policies, which are used to specify the principal's behaviors in the system (such as permitting or prohibiting some particular actions). In a broad sense, policies can represent the interplay between different principals. Therefore, ASP has been widely applied in information security research areas, such as cryptanalysis [53], access control and authorization [17, 58, 69, 70], and security protocol verification [4]. In this thesis, we investigate the access control and authorization and security protocol verification problems using the ASP approach.

Access Control and Authorization:

Comparing with traditional access control techniques, the ASP approach, formalizing authorization specifications and evaluations using ASP, can separate policies from implementation mechanisms and provide simple semantics for policies by ASP solvers. Jajodia *et al.* [57, 58] presented a logic programming based approach to integrate multiple access control policies into one unified framework. Their approach can specify authorization, conflict resolution, and integrity constraint checking. Bertino *et al.* [17] also proposed a logic framework in which they considered hierarchically structured domain of subjects, objects and access rights for authorization, and supported both negation as failure and classical negation.

However, the previous approaches mainly deal with authorization issues in centralized environments. 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 does not exist a 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 aspects 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 [22], has received a great attention by many researchers [67, 70]. 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 requests which should have a rich expressiveness and be understandable.

2. Finding a well theoretical foundation for checking proof of compliance.

Several trust-management systems such as PolicyMaker [22], Keynote [25], SPKI /SDSI [36, 43, 44, 45, 78], D1LP [70], and RT [69] have been developed. Here, we only give a quick review about these trust management systems and provide detailed descriptions in Section 1.2.2.

PolicyMaker [22] was the first trust management system. Its access policies and credentials are called *assertion*s 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 [25] 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. Both systems do not provide the negative authorization and re-delegation control. SPKI/SDSI has two kinds of certificates, name-definition certificates and authorization certificates. SPKI/SDSI [43, 44, 45, 78] can deal with the $k\text{-}out\text{-}of\text{-}n$ structures and handle certain types of nonmonotonic policies based on validity field of authorization certificates. It controls whether the authorization should be delegated further or not, but there is no delegation depth control. D1LP [70] is a more expressive formalization using Datalog as the sematic foundation. It supports delegation with depth control and static and dynamic threshold structures. Although D1LP is able to delegate an authorization to a conjunctive-subject structure, it can not deal with the request from conjunctive subjects which is related to *separation of duty*, an important issue in computer security literature. Moreover, it is not suitable to specify the authorization for structured resources. RT framework [69] is a role-based trust management framework which includes languages RT_0, RT_1, RT_2, RT^D, and RT^T, where RT^D, and RT^T can be used separately or together, with RT_0, RT_1 or RT_2. The semantic foundation of RT is Datalog with constraints which enable RT to express the authorization regarding structured resources and separation of duty policies.

Although the existing trust management systems may express rich delegation and authorization policies, we observe that one important issue they do not consider is to express nonmonotonic policy and its related problems. Let us consider the following scenarios:

> **Scenario 1:** In a large commercial organization, the system administrator trusts department managers and delegates them the privilege of accessing the file server with depth 1. Then managers give the privilege to the staff in their departments who are not on holiday and make them access the file server.
>
> **Scenario 2:** In a hospital database, there is a table which includes detailed information for its doctors, such as name, education background, specialized area, salaries and so on. The database administrator delegates patients to read all information about doctors except their salaries.
>
> **Scenario 3:** A bank requires two cashiers to approve a transaction requested by customers if they do not have a bad credit history.

It is easy to observe that all the above scenarios not only involve complex delegation and authorization controls, but also have nonmonotonic reasoning features. Nonmonotonic reasoning, in its broadest sense, is reasoning to conclusions on the basis of incomplete information. In Scenario 1, the managers permit the staff in their departments to access the file server. However, when a staff is on holiday, his/her manager will deny his/her access request. In Scenario 2, the patients have been allowed to read the detailed information about their doctors. When a patient try to get the salary of a doctor, the request is denied. In Scenario 3, two cashiers can approve a transaction requested by a customer. However, if a customer has a bad credit history, his/her transactions will be denied. We call these nonmonotonic reasonings because the set of plausible conclusions does not grow monotonically with increasing information. The previous approaches can not specify the policies and satisfy all the requirements in the above scenarios.

Security Protocol Verification:
In recent years, security protocols are increasingly being used in many diverse secure electronic communications and electronic commerce applications. However, despite an enormous amount of research effort expended in design and analysis of such protocols, it is still notoriously hard to verify protocols. When the security protocols are designed by hand, errors may creep in by combining protocols actions in ways not foreseen by the designer [4]. Some protocols have been found erroneous or inaccurate after they have been published many years, even since they have been proven secure [72]. This situation is further complicated by the often ambiguous definition of the goals of a security protocol, which makes it difficult to assess what really counts as a flaw.

The study of cryptographic protocols has led to the dichotomization of cryptographic protocol analysis techniques between the formal method approach and the provable security approach [33], both of which have been developed in two mostly different communities (detailed introduction in Section 1.2.3).

The formal method approach uses formal methods for protocol analysis, which include model checking, theorem proving, and logic-based approaches (including belief logics). The advantage of this approach is to verify protocols automatically and to relieve humans of tedious and error prone parts of the mathematical proofs. Moreover, this approach has the benefit of providing unambiguous specification of system requirement and precise mathematical proofs of system properties, and is able to prove insecurity by finding both known and unknown flaws in protocols [33]. The Dolev and Yao adversary model [41] is one of the first contributions to formal protocol analysis. Aiello and Massacci [4] presented an executable specification language under logic programming with answer set semantics to verify security protocols using the Dolev and Yao model. BAN logic [11] uses an approach very different from the Dolev and Yao adversary model. It is a logic of knowledge and belief, which consists of a set of possible beliefs that can be held by principles, and a set of inference rules for deriving new beliefs from old ones. The BAN logic consists of a very simple, intuitive set of rules, which made it easy to use. Even so, it is possible to point serious flaws in protocols. As a result, the logic gained wide attention and let to a host of other logics, either extending BAN logic [52] or applying the same concept to different types of problems in cryptographic protocols [90]. The main obstacle of this automatic approach is that cryptographic primitives are considered as ideal blackboxes, which is under the strong assumptions.

The emphasis of provable security approach is put on showing a reduction proof from the problem of breaking the protocol to another problem believed to be hard [49]. In this approach, adversaries are probabilistic polynomial-time Turing machines which try to win a game. Began in 1980s [49], the approach was made popular for key establishment protocols by Bellare and Rogaway [14], where they formally defined an adversary model with an associated definition of security and provided a proof of security for two-party entity authentication and key exchange protocols. Since then, there have been some extensions to Bellare-Rogaway model [1, 15, 16, 28]. Moreover, the related works resulted in two new proof models, Canetti-Krawczyk modular model [29] and Shoup key exchange model [87]. The main drawback of this approach is the difficulty of obtaining correct computational proofs of security which is dramatically illustrated by the well-known problem with the OAEP mode for public key encryption [88]. Choo *et al.* [32] pointed out the undetected flaws in proof of protocols which have been proven secure.

Recently, some research works have been done to bridge the gap between the above two approaches, which achieve automatic provability under classical computational models. Following Abadi and Rogaway [2], several recent independent projects developed by Backes *et al.* [8], Canetti *et al.* [30], Blanchet *et al.* [20, 21], and Choo *et al.* [33] are along this line. However, their works mainly focus on checking the protocols using some automatic tools. To our best knowledge, few work has been done using logic programming as a specification tool to provide automatic provability under classical computational models. Moreover, the previous approaches do not consider the update of insecure protocols.

In this thesis, we explore the distributed authorization and security protocol verification problems using Answer Set Programming. We adopt the trust management approach to investigate the distributed authorization issues and develop a unified framework for security protocol verification and update.

1.2 Background Knowledge

1.2.1 Answer Set Programming

Answer Set Programming denotes logic programming with the answer set semantics. In this section, we introduce the basic knowledge of logic programs and present the definition of the answer set semantics [9, 47]. In addition, we also introduce a widely used answer set programming solver, *Smodels* [74, 89].

A logic program consists of a finite set of rules. A rule is of the form:

$$L_0 \leftarrow L_1, \ldots, L_m,\ not\ L_{m+1}, \ldots,\ not\ L_n.$$

where m, $n \geq 0$, $n \geq m$, and each L_i is a literal. We call *not* L_i a negative literal. A program only consisting of rules which do not include negative literals is a *definite logic program*. Otherwise, it is a *normal logic program*.

A definite logic program has a unique *answer set* which is the minimal Herbrand model of this program.

Gelfond and Lifschitz [47] proposed answer sets of grounded normal logic programs that are based on the definition of reduct.

Definition 1.1 *The* reduct P^S *of a ground logic program* P *with respect to a set of atoms* S *is the definite program obtained from* P *by deleting*

(1) each rule that has a negative literal not L_i *in its body with* $L_i \in S$*;*

(2) each negative literal in the bodies of the remaining rules.

The reduct P^S is a definite logic program.

Definition 1.2 *Let* $\mathcal{M}(P^S)$ *be the answer set of the definite logic program* P^S*. A set of atoms* S *is an answer set of a normal logic program* P *iff* $S = \mathcal{M}(P^S)$*.*

A normal logic program may have one, more than one, or no answer set at all.

Example 1.1 Consider the following program Π:

$$p \leftarrow r,\ not\ q.$$
$$q \leftarrow not\ p.$$

The program has a unique answer set $\{q\}$, which is the answer set of $\Pi^{\{q\}} = \{q \leftarrow .\}$. $\square$

Example 1.2 Consider the following program Π' modified from Π:

$$p \leftarrow not\ q.$$
$$q \leftarrow not\ p.$$

The program has two answer sets $\{p\}$ and $\{q\}$. $\square$

Smodels [74] is a system for answer set computation. It consists of *smodels*, an efficient implementation of the answer set semantics for ground normal logic programs, and *lparse*, a front-end that transforms logic programs in the language of *Smodels* into ground logic programs.

The language in *Smodels* [89] includes basic terms in logic programs and some extended substance that consists of the special features of *Smodels.*

In *Smodels*, there are four different types of terms: *constants, variables, functions*, and *ranges.* A *constant* is either a symbolic constant or numeric constant starting with a lower case letter. A *variable* is a string of letters and numbers starting with an upper case letter. A *function* is either a function symbol followed by a parenthesized argument list or a built-in arithmetical expression. A *range* is of the form:

$$start..end$$

where *start* and *end* are constant valued arithmetic expressions. A *range* is a notational shortcut that is mainly used to define numerical domains in a compact way.

An *atom* is of the form $p\,(a_1, \ldots, a_n)$ where p is a *n-ary* predicate symbol and $a_1, \ldots, a_n$ $(n \geq 0)$ are terms. Generally, a *literal* is either an *atom a* or its negation *not a.* We call them *basic literals.* In *Smodels*, there are three extended literals, *constraint literals, weight literals*, and *conditional literals.* In this thesis, we do not consider the weight situation and just give the descriptions for conditional literals and constraint literals.

A *conditional literal* is of the form:

$$p\,(X)\ :\ q\,(X)$$

where $p\,(X)$ is any *basic literal* and $q(X)$ is a *domain predicate.* Formally, a predicate q of program Π is a *domain predicate* iff in the predicate dependency graph of Π, every path starting from q is free of cycles that pass through a negative edge.

A *constraint literal* is of the form:

$$lower\ \{\ l_1,\ l_2, \ldots, l_n\ \}\ upper$$

where *lower* and *upper* are arithmetic expressions and $l_1, \ldots, l_n$ are *basic* or *conditional literals.* A *constraint literal* is satisfied if the number of satisfied literals in the body of the constraint is between *lower* and *upper* (inclusive).

Smodels supports not only basic rules in logic programs, but also choice rules. We introduce them in the following.

- Basic rule:

  A basic rule is of the form:

  $$h \leftarrow a_1, \ldots, a_n,\ not\ b_1, \ldots,\ not\ b_m.$$

  If $a_1, \ldots, a_n$ are in an answer set and $b_1, \ldots, b_m$ are not, the head atom h is also put in the answer set.

  If a rule has no head, all candidate models that satisfy the rule body are discarded.

- Choice rule

  A choice rule has the following form:

  $$lower\ \{h_1, \ldots, h_n\}\ upper \leftarrow\ body.$$

  If the body of a choice rule is satisfied, the number of $h_1, \ldots, h_n$ that are true in the answer set will be between *lower* and *upper*, inclusive.

  For example, the program

  $$1\ \{a,\ b\}.$$

  has three answer sets, $\{a\}$, $\{b\}$, and $\{a,\ b\}$.

Example 1.3 Consider the program:

$q(1..2).$

$a \leftarrow 1\{p(X) : q(X)\}.$

The two rules will be grounded to give

$$q(1).\ \ q(2).$$
$$a \leftarrow 1\{\ p(1), p(2)\ \}.$$

□

Semantically the expansion of conditions takes place after the variables that also occur in another part of the rule are instantiated, which is illustrated in the following example.

Example 1.4 Consider the program:

$$d(1..2).$$
$$a(X) \leftarrow\ 1\{\ p(X,Y) : d(Y)\ \},\ d(X).$$

the variable X will be first instantiated to give the program

$$d(1).\ \ d(2).$$
$$a(1) \leftarrow\ 1\{\ p(1,Y) : d(Y)\ \},\ d(1).$$
$$a(2) \leftarrow\ 1\{\ p(2,Y) : d(Y)\ \},\ d(2).$$

In the next step the conditions are expanded to give

$$d(1).\ \ d(2).$$
$$a(1) \leftarrow\ 1\{\ p(1,1),\ p(1,2)\ \},\ d(1).$$
$$a(2) \leftarrow\ 1\{\ p(2,1),\ p(2,2)\ \},\ d(2).$$

□

1.2.2 Overview of Access Control

Access control is an important security mechanism controlling which principals (persons, processes, machines, and so on) have access to which resources. It provides confidentiality, integrity, and availability services for information systems. Nowadays, access control is pervasively used in application software, middleware, operating systems, and hardware systems. The process of developing an access control system is usually considered from different levels of abstraction [84], policies, models, and mechanisms. The concepts can be defined in the access control context as:

1. Policy: high-level guidelines or rules, which describe how to control accesses and determine the access decisions.

2. Model: formal representation of the access control policy and its functionality.

3. Mechanism: low-level functions implementing the controls determined by the model.

Based on how the policy controls accesses, access control policies are generally classified as *Mandatory Access Control (MAC)*, *Discretionary Access Control (DAC)*, and *Role-based Access Control (RBAC)*. Currently the widespread use of Internet and global internet-worked infrastructures addresses the new access control requirements for distributed environments, delegation of authorization and support for distributed information sources. The traditional solutions for distributed access control use certificates and public key cryptography to identify principals in the system and make decisions with respect to access control policies. *Trust management* approach [22, 23, 24, 25] proposed an authorization-based access control, in which the identity of the user accessing a resource is unknown and authorizations or permissions are bound directly to public keys instead of identities. In the following, we introduce MAC, DAC, $RBAC$, and $Trust\ Management$ in more details.

Mandatory Access Control

MAC is based on predefined regulations determined by a central authority. The users in the system cannot change any kind of rules. The mandatory policies are known as *multilevel security policies*, which were first introduced by Bell and La-Padula [12], and were first implemented in the Multics operating system [13]. MAC policies have been used specially in military environments where there is a clear and rigid hierarchy of authority [35]. The Bell-LaPadula model [12] and Biba model [19] are two popular models for MAC policies.

- The Bell-LaPadula Model

  In the Bell-LaPadula Model, the system is composed of subjects, objects and actions. Objects and subjects are classified into *access classes*. Each access class has a security level and a set of categories. A security level is an element from an ordered set, which determines the sensitivity of the object and subject. For example, a typical security level set is:

$$TopSecret > Secret > Confidential > Unclassified$$

A set of categories is a subset from an unordered set, which corresponds to functional or competence areas. For example:

$$Army,\ Navy,\ Nuclear,\ Administration, ...$$

Between two access classes there is a dominance relationship, denoted as '$\geq$'. A class c_1 dominates a class c_2 if and only if: (1) the security level of c_1 is greater than or equal to that of c_2, and (2) the categories of c_1 include those of c_2.

Formally, the Bell-LaPadula model presents a set of subjects S, objects O, and actions A. The set of access classes is $\mathcal{AC} = \mathcal{L} \times p(\mathcal{C})$, where $\mathcal{L}$ denotes the set of security levels, $\mathcal{C}$ denotes a set of categories, $p(\mathcal{C})$ denotes the powerset of $\mathcal{C}$ and $\times$ denotes the cartesian product. The dominance relationship($\geq$) is defined as:

$$\forall x = (L_1, C_1),\ y = (L_2, C_2),\ x \geq y\ \leftrightarrow L_1 \geq L_2 \wedge C_1 \supseteq C_2.$$

Where $C_1, C_2 \subseteq \mathcal{C}$, $L_1, L_2 \in \mathcal{L}$, and $x, y \in \mathcal{AC}$.

The access classes form a lattice $\mathcal{L}attice = (\mathcal{AC}, \geq)$. There is a function, which applies to subjects and objects and returns its classification: $\lambda : S \cup O \rightarrow \mathcal{AC}$. A set of states V is a triple $(b, \mathcal{M}, \lambda)$, where $b \in (S, O, A)$ and determines the current access requests (s, o, a), where $s \in S$, $o \in O$, and $a \in A$, and $\mathcal{M}$ is a matrix determining the relationship of subjects, objects, and actions for current state. The model states the following principals:

1. Simple security property (no-read-up): A state v satisfies the simple security property if and only if:

   $$\forall s \in S,\ o \in O : (s, o, read) \in b \Rightarrow \lambda(s) \geq \lambda(o).$$

2. security *-property (no-write-down): A state v satisfies the *-property if and only if:

$$\forall s \in S,\ o \in O : (s, o, write) \in b \Rightarrow \lambda(o) \geq \lambda(b).$$

A state is secure if it satisfies both principals. A system is defined to be secure if, from a secure state by executing a finite number of requests the system ends in another secure state.

- The Biba Model

  The Bell-LaPadula model protects the *confidentiality* of the information. It does not provide any control over its *integrity*. Biba proposed the Biba model based on Bell-LaPadula that provides integrity control [19].

  The classification of subjects and objects follows the same idea of the Bell-LaPadula model. There is also a dominance relationship between the classes, which together with the classes defines a lattice.

  The main principals can be seen analogous to the Bell-LaPadula principals:

  1. Simple integrity property (no-read-down): A state v satisfies the simple security property if and only if:

     $$\forall s \in S,\ o \in O : (s, o, read) \in b \Rightarrow \lambda(o) \geq \lambda(s).$$

  2. integrity *-property (no-write-up): A state v satisfies the *-property if and only if:

     $$\forall s \in S,\ o \in O : (s, o, write) \in b \Rightarrow \lambda(s) \geq \lambda(o).$$

  If the two principals are satisfied, the system ensures the integrity of the objects.

Due to the need for strong centralized control, the MAC policies are not very suitable in corporate environments. MAC has become very popular in military-like organizations, where there is a clear hierarchy and central authority.

Another important problem of MAC polices is known as *covert channels*, which leak information about the system unintentionally. Solutions to avoid covert channels have been the focus of a lot of research efforts [7, 48]. One of the main problems of covert channels is that they are very dependent to the implementation layer and it is difficult to prevent them in the design of the model [73].

Discretionary Access Control

In *DAC*, users are given some management capabilities to control and set access rules over some objects. Normally, there are explicit rules to state who can do what over which resources. A user can have the ability to pass its own privileges to other users and grant or revoke authorization following the administration policy of the system. The *DAC* policies and models are based on the access matrix first proposed by Lampson [65]. Harrison, Ruzzo and Ullmann presented one of the first formal models of access matrix, known as *HRU* model [55].

- Access matrix

  The main goal of the access matrix is to describe the protection state of a system. The access matrix considers *subjects* (S) that have *privileges* (also called actions or rights) over protected *objects* (O). It is important to note that in most environments, subjects are considered as a subset of objects, i.e. $S \subseteq O$. The state of a system is defined by the triplet (S, O, A), where A is the access matrix. Figure 1.1 shows the outline of the access matrix.

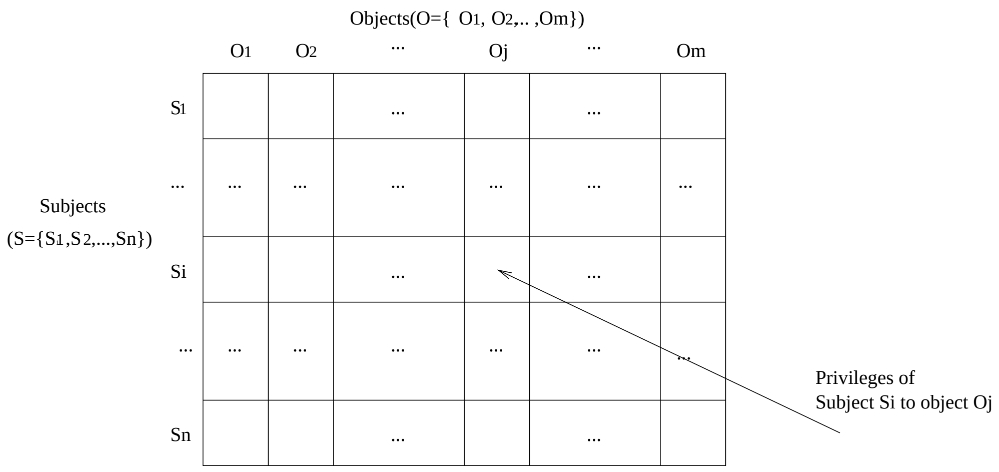

Figure 1.1: Access matrix

In the access matrix, columns are objects, and rows are subjects. Each entry of the matrix determines the privileges for a give subject over a given object. The entry $A[s_i, o_j]$ contains the actions that subject s_i can perform over object o_i. For a simple example of an access matrix showed in Figure 1.2, we can obtain that UserB can read and write FileA, which is also owned by him, that is $A[UserB, FileA] = \{read, write, own\}$.

	FileA	FileB	ProgramA	ProgramB
UserA	read	read	execute	read execute
UserB	own read write		read execute	execute
UserC	read	read write		execute

Figure 1.2: An access matrix example

A straightforward implementation of the access matrix using a two-dimensional array may be very inefficient due to the memory consumption. There are three main approaches for a reasonable implementation of the access matrix, *Authorization Tables, Access Control Lists (ACL)*, and *Capabilities.* The authorization table is composed of a set of triplets of the form $\langle user, privilege, object\rangle$ and each triplet is considered an authorization. Figure 1.3 illustrates the authorization table for the access matrix of the example in Figure 1.2.

ACL is one of the most used implementation of the access matrix. In an ACL, each object of the system has a list with the privileges for each subject. The ACL stores the access matrix by columns. On the other hand, *Capability* stores the access matrix by rows, in which each subject has a list with privileges for each object. Figure 1.4 and 1.5 illustrate the ACL and Capabilities for the access matrix examples of Figure 1.2, respectively.

- HRU model

In HRU model, changes to the state of a system were described as six primitive operations: enter an action into the access matrix, delete an action from the access matrix, create a subject, delete a subject, create an object, and delete an object. Formally, HRU defines a *command*(α) as follows:

```
$command\ \alpha(X_1,\ X_2, \ldots,\ X_k)$
    $if\ r_1 \in A[X_{s_1}, X_{o_1}] \wedge\ r_2 \in A[X_{s_2}, X_{o_2}] \wedge \ldots \wedge\ r_m \in A[X_{s_m}, X_{o_m}]$
    $then\ op_1;\ op_2; \ldots;\ op_n$
$end$
```

User	Access mode	Object
UserA	read	FileA
UserA	read	FileB
UserA	read	ProgramB
UserA	execute	ProgramA
UserA	execute	ProgramB
UserB	read	FileA
UserB	read	ProgramA
UserB	own	FileA
UserB	write	FileA
UserB	execute	ProgramA
UserB	execute	ProgramB
UserC	read	FileA
UserC	read	FileB
UserC	write	FileB
UserC	execute	ProgramB

Figure 1.3: Authorization table example

Where $n > 0$, $m > 0$, each r_i is an action, each op_i is a primitive operation, s_i and o_i are integers between 1 and k, X_i for $0 < i < k$ are formal parameters, X_{s_i} refers to a subject, and X_{o_i} refers to an object.

In [55] it is demonstrated that the safety problem of *HRU* model is undecidable. The only case where the problem is decidable is for the case of finite sets of subjects and objects in a mono-operational system. However, the commands in a mono-operational system can only have at most one primitive operation, which turns out that mono-operational systems are not practical. To improve the decidability of the safety problem, Sandhu proposed the Typed Access Matrix (*TAM*) [85] which is based on *HRU* model but introduces the notion of strong type in it. Safety problem becomes decidable in polynomial time cases where the system is monotonic, commands are limited to the parameters and there is no cyclic creation.

Role-based Access Control

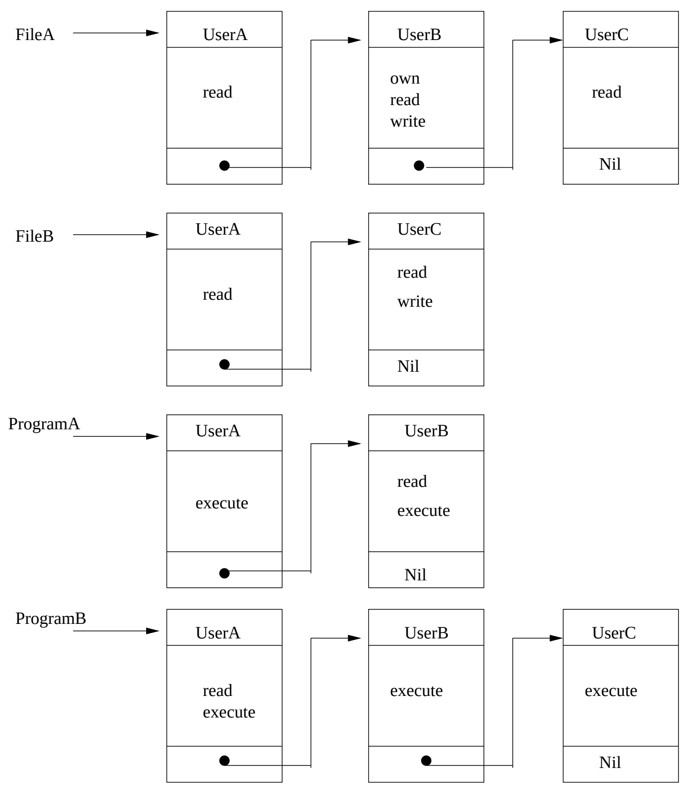

Figure 1.4: Access control list example

Role-based Access Control (RBAC) is to group privileges or authorizations that can be applied to a group of users. *RBAC* was first proposed in [46], and has received an increasing attention both from academic and commercial environments. In 2001, NIST proposed a consensus model for RBAC, based on the Ferraiolo-Kuhn model [46], in the framework developed by Sandhu [86]. The model was further refined within the RBAC community and has been adopted by the American National Standards Institute, International Committee for Information Technology Standards (ANSI/INCITS) as ANSI INCITS 359-2004 [75].

The RBAC model was defined in terms of four *model components*: core RBAC, Hierarchical RBAC, Static Separation of Duty Relations, and Dynamic Separation of Duty Relations as shown in Figure 1.6. Core RBAC is required in any RBAC system, but the other components are independent of each other and may be implemented separately. The formal definition of RBAC model is referred to [75, 86].

- **Core RBAC** defines a minimum collection of RBAC elements, element sets and relations for the RBAC model and components. The basic elements of

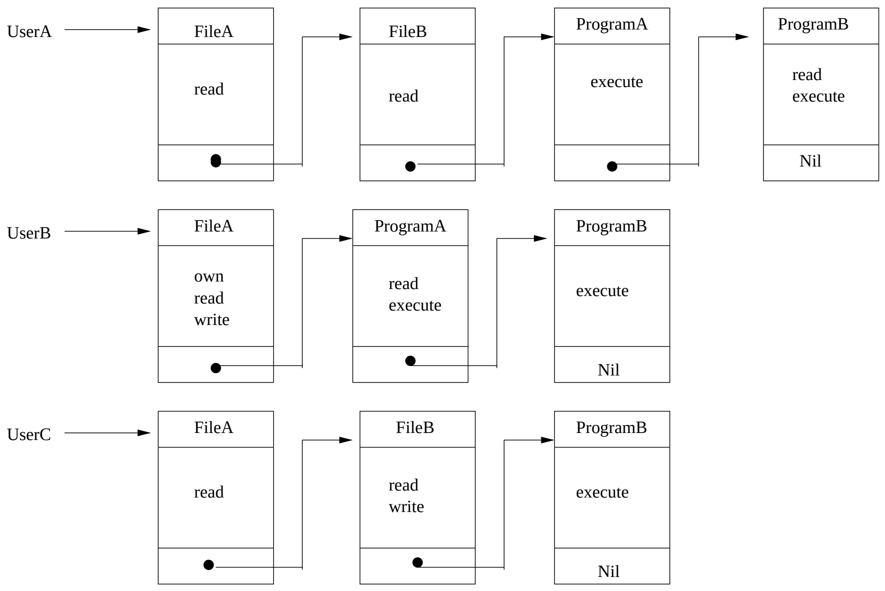

Figure 1.5: Capability example

core RBAC are *users* (USERS), *roles* (RULES), *objects* (OBS), *operations* (OPS), and *permissions* (PRMS), where a permission is defined as an approval of a particular operation performed over one or more objects in the system. Thus permissions establish a relation between operations and objects and can be expressed as: $PRMS = 2^{(OPS \times OBS)}$. The core RBAC also defines two relations, *user assignment* (UA), and permission assignment (PA), both of which are many-to-many mappings and denoted as $UA \subseteq USERS \times ROLES$, and $PA \subseteq PRMS \times ROLES$ respectively. In addition, core RBAC includes a set of *session*s (SESSIONS) where each session is a mapping between a user and an activated subset of roles that are assigned to the user. The mapping is established by two functions, *user_sessions* which gives us the set of sessions associated with a user and *session_roles* which gives us the roles activated by the session.

- **Role hierarchy RBAC** adds role hierarchy (RH) to the core RBAC, which allows to structure roles to reflect an organization's lines of authorization and responsibility. Intuitively, a role hierarchy defines a permission inheritance relation between roles. That is, role r_1 inheriting role r_2 means that all privileges of r_2 are also privileges of r_1, denoted as $r_1 \succeq r_2$. RH includes two kinds of

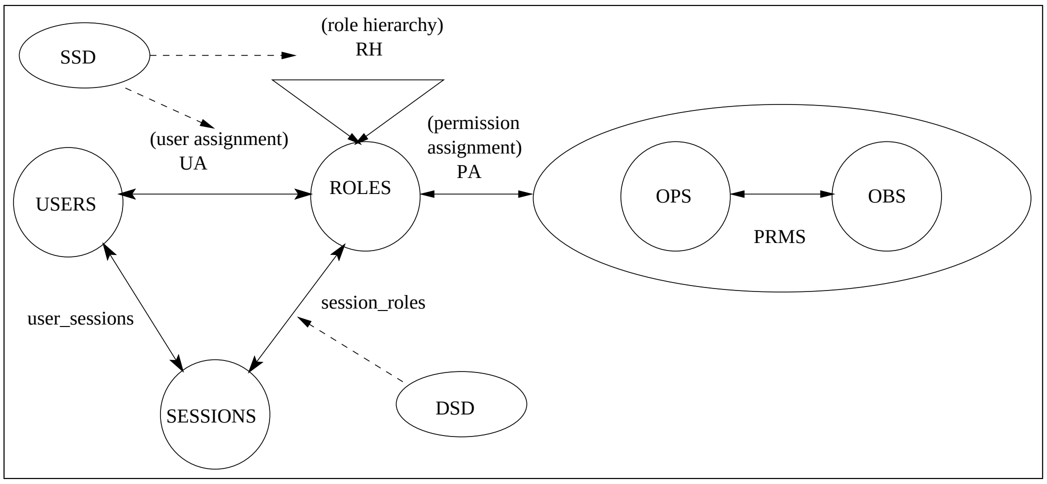

Figure 1.6: The RBAC Model

role hierarchy: *general role hierarchy*, and *limited role hierarchy*. General role hierarchy supports *multiple inheritance*, which means that a role can inherit permission from more than one role. In some scenarios, multiple inheritance can introduce conflicts and problems. Limited role hierarchy restricts the general role hierarchy by not allowing multiple inheritance.

- **The third RBAC component**, Static Separation of Duty (SSD) Relations, enforce constraints in user assignments, which can be used in both of presence and absence of role hierarchies. This component prevents conflict of interest which may arise when a user obtains authorization for permissions associated with conflicting roles.

- **The fourth model component**, Dynamic Separation of Duty (DSD) Relations, like SSD relations, are intended to limit the user's permissions. However, DSD Relations place constraints on the activation of roles. In addition, DSD Relations support the principal of least privilege in that each user has different levels of permission at different times, depending on the role being activated.

Trust Management

The important role of trust management in the area of open distributed and decentralized systems has been recognized by many researchers. Matt Blaze and colleagues [22] have coined the term *trust management* as "a unified approach to specify and interpret security policies, credentials, and relationships which allow direct authorization of security-critical actions". Under this approach public keys are bound to authorization actions directly, and the authorization can be delegated to third parties by credentials or certificates.

Several trust-management systems such as PolicyMaker [22, 23], Keynote [24, 25], SPKI/SDSI [36, 43, 44, 45, 78], Referee [34], SD3 [59], D1LP [70], and RT framework [69] have been developed in recent years. In the following section, we review some trust management systems.

PolicyMaker [22, 23] was the first trust management system. The basic function of the system is to process queries based on local policy and credentials that contain the authorization information from trusted third parties.

The PolicyMaker language includes *queries* and *assertions*. Each query is a request for performing a particular action from a particular public key or a sequence of public keys with the form:

$$key_1, key_2, \ldots, key_n \ REQUESTS \ ActionString$$

*ActionString*s are application dependent messages that specify a trusted action requested.

Assertions represent local policy and credentials from trusted third parties with the form:

$$Source \ ASSERTS \ AuthorityStruct \ WHERE \ Filter.$$

It can be read as *Source* trusts the public keys enumerated in *AuthorityStruct* to be associated with action strings that satisfy *Filter*. A *Source* denotes the source of the assertion. There are two types of assertions, policy assertions which are local policies, and certificates in which the *Source* is the public key of a third party. *AuthorityStruct* specifies the public key to whom the assertion applies. *Filter* is the predicate that action strings must satisfy for the assertion to hold.

PolicyMaker system 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 [24, 25] 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. In KeyNote, policies and credentials are called *assertions* and both of them are specified using the same format. The main difference between them is that policies are locally trusted and they do not need any signature. An example of a KeyNote assertion extracted from [24] is showed in Figure 1.7.

```
KeyNote-Version:1
Authorizer: rsa-pkcs1-hex:“1023abcd”
Licensees: dsa-hex:“86512a1” ||
           rsa-pkcs-hex:“19abcd02”
Comment: Authorizer delegates read
           access to either of the
           Licensees
Conditions: ($file == “/etc/passwd” &&
           $access == “read”) − >
                  {return “ok”}
Signature: rsa-md5-pkcs1-hex:“f00f5673”
```

Figure 1.7: Sample KeyNote assertion

KeyNote uses a depth-first search (DFS) algorithm that attempts to satisfy at least one policy assertion. Satisfying an assertion entails satisfying both the *Conditions* field and the *Licensees* key expression.

SPKI/SDSI was merged by SPKI (Simple Public Key Infrastructure) [44] and SDSI (Simple Distributed Security Infrastructure) [78], both of which were motivated by the inadequacy of public-key infrastructures based on global name hierarchies, such as X.509 [56] and Privacy Enhanced Mail (PEM) [61]. SPKI/SDSI shares many views with trust management approach. For instance, the main purpose of its certificates is authorization instead of authentication. However, it does not define an application independent trust management engine. Therefore, strictly speaking, SPKI/SDSI is not a complete trust management system.

SPKI/SDSI has two kinds of certificates, *name certificates* which came from SDSI and *authorization certificates* which came from SPKI. A name certificate binds a local name to a principal or a more complex name. An authorization certificate delegates a certain permission from a principal (the certificate's issuer) to the certificate's subject. An authorization certificate includes the following five fields: “Issuer”, the principal who signs this certificate; “Subject”, the principal being authorized; “Authority”, the specific permission being delegated; “Delegation”, a boolean value to specify whether the subject can further delegate the authority received in this certificate; “Validity”, validity period or/and checking methods. SPKI/SDSI can deal with the *k-out-of-n* structures and handle certain types of nonmonotonic policies based on validity field of authorization certificates. It controls whether the authorization should be delegated further or not, but there is no delegation depth control.

REFEREE [34] (Rule-controlled Environment For Evaluation of Rules, and Everything Else) is a trust management system that provides a language for specifying trust policies and policy-evaluation mechanisms for Web clients and servers. REFEREE uses PICS labels [77] as credentials. A PICS label states some properties of a resource in the Internet. In this context, policies specify which credentials must be disclosed in order to grant an action.

In REFEREE credentials are executed and their statements can examine statements made by other credentials and even fetch credentials from the Internet. Therefore, policies are needed to control which credentials are executed and which are not trusted. The policies determine which statements must be made about a credential before it is safe to run it. REFEREE improves PolicyMaker [22, 23] in the sense that PolicyMaker assumes that credential-fetching and signature verification are done by the calling application.

In REFEREE there are three kinds of data types: (1) Tri-values: is one of true, false or unknown; (2) Statement lists: is a collection of assertions. A statement is formed by some content and a context for the content. The interpretation of the context depends on the agreement between REFEREE and the calling application. A statement list is an unordered list of statements; (3) Programs: A program can be a policy or a credential. A program takes a statement list defining the current evaluation context and required/optional extra arguments as an input. It returns a tri-value (the result of the program) and a statement list (a justification). The program returns true if it is possible to infer compliance with a policy (credentials were sufficient to grant the requested action). It returns false if it is not possible to infer compliance (credentials were sufficient not to grant the action) or unknown if

no inference could be made at all (credentials are not sufficient to take a decision: neither for approving nor denying the action).
SD3 [59] (Secure Dynamically Distributed Datalog) is a trust management system consisting of a high-level policy language, a local policy evaluator and a certificate retrieval system. It provides three main features:

1. Certified evaluation: the certified evaluator computes not only an answer, but also a proof that the answer is correct.

2. High-level language: SD3 policy abstracts from signature verification and certificate distribution. It makes policies easy to write and understand.

3. SD3 is programmable: Policies can be easily written and implemented.

SD3 language is an extension of Datalog. The language is extended with SDSI global names [36]. A rule in SD3 is of the form:

$$T(x, y) : -K\$E(x, y).$$

Where $T(x, y)$ holds if a digital credential asserting $E(x, y)$ and signed with the private key of K was given. Whenever a global name is used, an authentication step is needed. In addition, SD3 can refer to assertions in remote computers. Given the rule

$$T(x, y) : -(K@A)\$E(x, y).$$

The query evaluator must query a remote SD3 evaluator at an IP address A. This gives SD3 the possibility to create “chains of trust”. The evaluator consists of three elements: an optimizer, a cache, and a core evaluator. More novel implementation techniques of the evaluator as well as the theoretical foundations of SD3 are described in [91].
D1LP [70] is a logic based trust management language with features needed for distributed authorization. The alphabet of D1LP consists of three disjoint sets: the *predicate symbols*, the *variables*, and the *constants*. Variables start with “?”. The security policy is specified by a program consisting of a finite set of rules written using D1LP. The rule has the form: $H\ if\ F$, where H and F are statements issued

by principals in the system. In D1LP, there are direct statements, delegation statements, and representation statements. D1LP also supports delegation with depth control and static and dynamic threshold structures. We extract an example from [70] to demonstrate the syntax of D1LP as follows.

A merchant *ShopA* will approve a customer's order if it can determine that the customer has a good credit rating. *ShopA* trusts *BankB* and whomever *BankB* trusts in determining credit ratings. *ShopA* also has a credential issued by *BankB* saying that *BankB* believes that a principal has good credit if two out of three particular credit-card companies certify that this principal has an account in good standing. These policies and credentials are represented as follows:

$$\begin{aligned}
&ShopA\ says\ approveOrder(?U)\ if\ ShopA\ says\ creditRating(?U, good).\\
&ShopA\ delegates\ creditRating(?U, ?R)^2\ to\ BankB.\\
&BankB\ says\ creditRating(?U, good)\\
&\qquad if\ threshold(2, [cardW, cardX, cardY])\ says\ accountGood(?U).
\end{aligned}$$

D1LP transforms the policy program into an ordinary logic program (OLP), and answers access control queries using some backward OLP inference engine.

RT Framework [67, 68, 69] is a set of languages for representing policies and credentials. It is specially suited for decentralized collaborative systems and for attribute-based access control (ABAC). RT uses roles to represent attributes. An entity has an attribute if it is a member of the corresponding role. The RT framework consists of several parts which are described as follows.

- $\mathbf{RT_0}$ RT_0 [67] is the most basic language of the RT set. In RT_0 policy statements take the form of role definitions. Role definitions have a head of the form $K_A.R$ and a body. K_A represents a principal while R is a role term. The following describes the different kinds of constructions allowed in RT_0:

  1. Simple member ($K_A.R \leftarrow K_D$)
     The principal K_D is a member of the role $K_A.R$.
  2. Simple containment ($K_A.R \leftarrow K_B.R_1$)
     The role $K_A.R$ contains any principal that is a member of the role $K_B.R_1$.

3. Linking containment ($K_A.R \leftarrow K_A.R_1.R_2$)

   The role $K_A.R$ contains every role of the form $K_B.R_2$ for each K_B which is a member of the role $K_A.R_1$.

4. Intersection containment ($K_A.R \leftarrow K_{B_1}.R_1 \cap \ldots \cap K_{B_i}.R_i$).

   The role $K_A.R$ contains the intersection of the members of the roles, $K_{B_1}.R_1 \cap \ldots \cap K_{B_i}.R_i$.

5. Simple delegation ($K_A.R \Leftarrow K_B : K_C.R_2$)

   K_A delegates its control over R to K_B. If $K_C.R_2$ is present, K_A restricts its delegation in such a way that K_B can only assigned members of $K_C.R_2$ to be members of $K_A.R$.

6. Linking delegation ($K_A.R \Leftarrow K_A.R_1 : K_C.R_2$)

   K_A delegates control over R to all the members of $K_A.R_1$ and the delegation is controlled so only members of $K_C.R_2$ can be assigned as members of $K_A.R$.

- $\mathbf{RT_1}$ In RT_0, roles do not take any parameters. RT_1 role definitions have the same form as the one in RT_0 but they may contain parameterized roles. In RT_1 a role is of the form $r(p_1, \ldots, p_n)$. r is the role name and p_i can be $name = c$, $name = ?X[\in S]$ ($\in S$ is optional) or $name \in S$ where $name$ represents a name of a parameter, c represents a constant, $?X$ is a variable and S is a value set.

- $\mathbf{RT_2}$ RT_2 adds to RT_1 logical objects (also called o-set) in order to group permissions between objects. A credential in RT_2 is either an o-set-definition or a role-definition. An o-set-definition is formed by an entity followed by an o-set identifier $K.o(h_1, \ldots, h_n)$ and allows constraint variables with dynamic value sets.

- $\mathbf{RT^T}$ RT^T introduces the notion of *manifold roles* to achieve agreement of multiple principals from both one set and disjoint sets for supporting separation of duty security principle. A manifold role defines a set of principals. Each of these sets is a set of principals whose collaboration satisfies the manifold role. Manifold roles are constructed as follows:

1. Product containment ($K_A.R \leftarrow K_{B_1}.R_1 \odot \ldots \odot K_{B_k}.R_k$)
   The role $K_A.R$ contains every principal set p such that $p = p_1 \cup \ldots \cup p_k$ and p_i is a member of $K_{B_i}.R_i$ for each $0 < i \leq k$.

2. Exclusive product containment ($K_A.R \leftarrow K_{B_1}.R_1 \otimes \ldots \otimes K_{B_k}.R_k$)
   The role $K_A.R$ contains every principal set p such that $p = p_1 \cup \ldots \cup p_k$, $p_i \cap p_j = \phi \ for \ 1 < i \neq j < k$, and p_i is a member of $K_{B_i}.R_i$.

- **RT$^{\mathbf{D}}$** RT^D provides delegation of role activations which express selective use of capacities and delegation of these capacities. A delegation credential presented by a principal D takes the form of $D \xrightarrow{D\ as\ A.R} B_0$. With it a principal D activates the role $A.R$ to use in a session B_0. In addition, B_0 can further delegate this role activation to B_1 with $B_0 \xrightarrow{D\ as\ A.R} B1$.

1.2.3 Security Protocol Verification

There exist two main approaches for analyzing the security of protocols: formal method approach and provable security approach.

In formal method approach, the Dolev and Yao adversary model [41] is often the de-facto standard used in formal specification. For formal methods to verify security protocols, the formalization process itself is a serious bottleneck. It is not a trivial task for protocol designers and analysts who are not experts in formal methods to specify the protocols using complicated systems. Logic programming has a readable syntax and can express properties of messages, keys, principal actions (such as send, get and intercept messages) naturally. Moreover, logic programming with answer set semantics has some efficient solvers, which make the formalization language executable and output models (if any) to show the attacks. Aiello and Massacci [4] presented an executable specification language $\mathcal{AL}_{SP}$ under Answer Set Programming to verify security protocols using the Dolev and Yao model.

In the provable security approach, Bellare and Rogaway [14] provided the first formal definition for an adversary model with associated definition of security, which is Bellare-Rogaway model. In independent yet related work, Canetti and Krawczyk [29] developed Canetti-Krawczyk modular model.

Recently, some comprehensive efforts have been made for unifying the two approaches. Choo *et al.* [33] is along this line, which provided a formal specification and machine analysis of Canetti-Krawczyk modular model using Asynchronous Product Automata (APA), a universal state-based formal method. In this section, we introduce the ASP based security protocol verification proposed by Aiello and Massacci [4] and the integrated approach developed by Choo *et al.* [33].

- Verifying security protocols as planning in ASP

  Aiello and Massacci [4] took representing the problem of verifying security protocols as a planning problem. A planning problem in the context of security protocols, where agents exchange messages and are subject to attacks by intruders, is formulated as follows:

  1. the initial state includes the keys known to agents and the messages already exchanged (typically none) in the beginning of the protocol run;
  2. the goal state is an unwanted situation where some security violation has occurred;
  3. actions are exchanges of messages among agents.

  If a solution of the planning problem exists, then it is a sequence of actions leading to an attack.

  Aiello and Massacci [4] presented $\mathcal{AL}_{SP}$ (Action Language for Security Protocols), an executable specification language for representing security protocols, and checking the possibility of attacks. *Smodels* is used for model generation of $\mathcal{AL}_{SP}$ specification to verify whether plans to attack a protocol exist.

  The language $\mathcal{AL}_{SP}$ consists of the following predicates:

  – Basic sort predicates:
    $ag(A)$, $nonce(N)$, $key(K)$.

  – Constructor for messages:
    $\{M\}_K$, $M_1||M_2$, $h(M)$, $M_1 \oplus M_2$.

- Predicates for properties of messages:
  $part(M_1, M)$, $invKey(K, K_1)$, $symKey(K)$, $shareKey(K, A, B)$, $asymKeyPair(Kpriv, Kpub)$.
- Predicates for fluents:
  $knows(A, M, T)$, $synth(A, M, T)$, $says(A, B, M, T)$, $gets(B, M, T)$, $notes(A, M, T)$.

Language $\mathcal{AL}_{SP}$ can be used to model messages, knowledge, agent actions, protocols, and attacks. We give the following rules as examples to illustrate how to specify security protocols using $\mathcal{AL}_{SP}$. Refer to [4] for detailed modelling steps.

$$knows(A, M, T) \leftarrow got(A, M, T) \tag{1.1}$$

$$knows(A, M, T) \leftarrow said(A, B, M, T) \tag{1.2}$$

The rules 1.1 and 1.2 denote A know something because A either got it or said it to somebody.

$$\{gets(B, M, T)\} \leftarrow says(A, B, M, T) \tag{1.3}$$

$$\{gets(spy, M, T)\} \leftarrow says(A, B, M, T) \tag{1.4}$$

The rule 1.3 uses the choice rule to specify that if A attempts to send a message M to B at time T, then B may receive it. The rule 1.4 models the intruder may get any exchanged messages in the protocol run with respect to the Dolev and Yao model.

- An integrated approach for verifying security protocols

  Choo *et al.* [33] presented a formal specification framework for security protocol verification in the setting of a plan problem using APA, which is supported by the Simple Homomorphism Verification Tool (SHVT). This framework analyzes the protocols with claimed security proof under the Canetti-Krawczk modular model [29]. The tripartite key exchange protocols 8 and 9 of Hitchcock, Boyd, and González Nieto [54] are case study protocols in this work, both of which carry claimed security proofs in the Canetti-Krawczk modular model.

The framework consists of protocols specification and analysis. Similar to the formulation of the planning problem, the protocol specification has three parts: description of initial state, description of goal state and description of possible actions. In the APA specification, the protocol principals are modelled as a family of elementary automata. The various state spaces of the principals are modelled as a family of state sets. The channel through which the elementary automaton communicates is modelled by the addition and removal of messages from the shared state component *Network*, which is initially empty. Each of the elementary automata only has access to the particular state components to which it is connected. In addition to the regular protocol principals, an adversary $\mathcal{A}$ is specified which can access to the shared state component *Network*, but can not access to the internal states of the principals. We extract Figure 1.8 from [33] to illustrate a two-party protocol specification in APA.

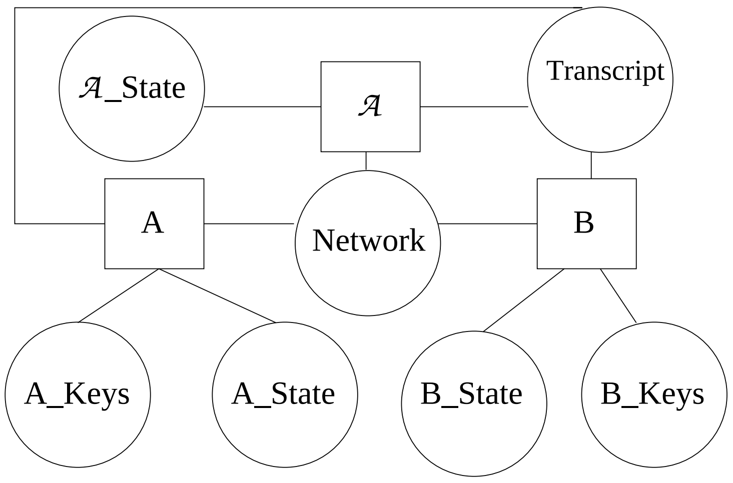

Figure 1.8: Graphical illustration of a two-party protocol in APA specification

After the specification process, a protocol can be analyzed using $SHVT$. If an agent has achieved its goal from the initial state to the goal state, a workable plan exists and the attack sequence can be found through tracing the path. We illustrate the protocol analysis process through Figure 1.9 extracted from [33], in which circles represent the various states of the protocol execution and the lines with arrows represent the state transitions between two states.

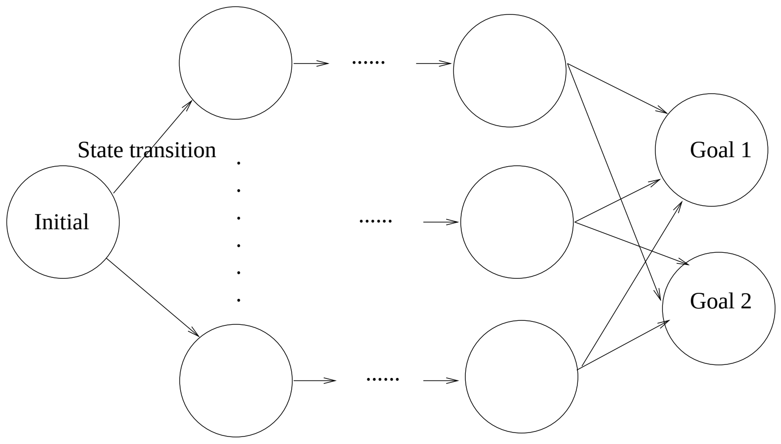

Figure 1.9: A reachability graph: protocol analysis in SHVT

1.3 Outline and Contributions of Thesis

We present a brief outline of the thesis and the main contributions.

In Chapter 2, we develop an authorization language $\mathcal{AL}$ for the policy specification in open distributed environments, present the techniques of transforming $\mathcal{AL}$ programs to executable logic programs, and investigate the computational issues in $\mathcal{AL}$. Compared with previous approaches, $\mathcal{AL}$ is characterized by the nonmonotonic feature, which makes it possible to specify nonmonotonic policies and reason to conclusions based on incomplete information. Moreover, $\mathcal{AL}$ has a rich expressive power which is able to specify delegation with the depth control, complex subject structures, and separation of duty policies. The above features make $\mathcal{AL}$ suitable for the representation of complex authorization policies, especially in distributed environments. We define the semantics of $\mathcal{AL}$ using the Answer Set Programming. Our formulation has implementation advantages due to recent development of Answer Set Programming technology in AI community, where many existing approaches do not have. We identified more general tractable classes of $\mathcal{AL}$ domains by applying some computational results in ASP. We considered that when an extended logic program is locally stratified or call-consistent, then this program must have an answer set, and such answer set can be computed in polynomial time. By examining proper conditions, we identified two subclasses of $\mathcal{AL}$ domains, for which their $\mathcal{L}_{Ans}$ translation will always be locally stratified or call-consistent. In this way, any query under those types of domains can be evaluated in polynomial time.

In Chapter 3, we implement a fine grained access control prototype system for XML documents with the delegation feature using Java, which is the implementation of the language $\mathcal{AL}$. Compared with related approaches, our system not only protects XML resources in fine grained level, but also support the complex policy specification permitted in $\mathcal{AL}$. Moreover, we specify the policies using XML documents. So in our system, both protected resources and authorization policies are XML documents stored in Xindice, a native XML databases, instead of relational databases, which makes us access the XML documents more convenient.

In Chapter 4, we propose a unified framework for integrating the protocol specification, verification, and update together based on Answer Set Programming. We define a security protocol specification language $\mathcal{L}_{sp}$, specify a security protocol through $\mathcal{L}_{sp}$, verify the protocol using the semantic of Answer Set Programming, and update the protocol by the forgetting technique in logic programs. Compared with previous approaches which usually only consist of the protocol verification, our framework integrates the protocol update. Moreover, forgetting in logic programs, an efficient technique for logic programming update has been used to repair the insecure protocols.

Finally, in Chapter 5, we present a summary of the thesis and discussions of possible research directions.

Note that parts of this thesis have already been published in journals and conference papers [93, 94, 95, 96].

Chapter 2

$\mathcal{AL}$–An Authorization Language

Authorization scenarios introduced in Chapter 1 present nonmonotonic reasoning and complex delegation and authorization features that usually can not be specified by existing trust management systems. In this chapter, we propose an authorization language $\mathcal{AL}$ which has the following features:

- Nonmonotonic reasoning through negation as failure in logic programming
- Both positive and negative authorizations
- Flexible conflict resolving policies
- Complex subject structures including subject set, and static and dynamic threshold structures
- Separation of duty policy
- Partial delegation and authorization

Using $\mathcal{AL}$, the authorization and associated delegation policies in a specific problem domain are represented as a domain description $\mathcal{D}_{\mathcal{AL}}$, which is a policy base including local policy and credentials from the trusted third parties. For each access request to the resource, the decision is made on the basis of reasoning mechanism through Answer Set Programming. It is well known that deciding whether an extended logic program has an answer set is NP-complete [9]. This implies that in general we may need an exponential algorithm to compute a program's answer sets. Consequently, computing an query $\mathcal{Q}_{\mathcal{AL}}$ under a given domain description $\mathcal{D}_{\mathcal{AL}}$ will be intractable. We investigate the computational issues related to language $\mathcal{AL}$. By examining proper conditions, we identify two subclasses of $\mathcal{AL}$ domains, for which their logic program translation will always be locally stratified and call-consistent respectively. In this way, any query under these types of domains can be evaluated in polynomial time. We also present two case studies to illustrate how to use language $\mathcal{AL}$ to specify the security policy and make authorization decisions.

2.1 Syntax, Concepts, and Examples

2.1.1 Syntax

The authorization language $\mathcal{AL}$ consists of *entities, atoms, thresholds, statements, rules* and *queries*. The formal BNF syntax of $\mathcal{AL}$ [95] is given in Figure 2.1. We explain the syntax in detail as follows.

Entities

In distributed systems, entities include *subjects* who are authorizers owning or controlling resources and requesters making 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*. Each constant entity is an element of three disjoint constant symbol sets, *SUB, OBJ*, and *PRIV*, where *SUB* is the set of subject constants, *OBJ* the set of object constants, and *PRIV* the set of privilege constants. The constant entity must start with a lower-case letter.

Correspondingly each variable entity is an element of three disjoint 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 letter.

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

$\langle rule\rangle$	$::=$	$\langle head\text{-}stmt\rangle$ [if [$\langle list\text{-}of\text{-}body\text{-}stmt\rangle$] [$with\ absence\ \langle list\text{-}of\text{-}body\text{-}stmt\rangle$]]	(1)
$\langle head\text{-}stmt\rangle$	$::=$	$\langle relation\text{-}stmt\rangle \mid \langle assert\text{-}stmt\rangle \mid \langle auth\text{-}stmt\text{-}head\rangle \mid \langle delegate\text{-}stmt\text{-}head\rangle$	(2)
$\langle list\text{-}of\text{-}body\text{-}stmt\rangle$	$::=$	$\langle body\text{-}stmt\rangle \mid \langle body\text{-}stmt\rangle, \langle list\text{-}of\text{-}body\text{-}stmt\rangle$	(3)
$\langle body\text{-}stmt\rangle$	$::=$	$\langle relation\text{-}stmt\rangle \mid \langle assert\text{-}stmt\rangle \mid \langle auth\text{-}stmt\text{-}body\rangle \mid \langle delegate\text{-}stmt\text{-}body\rangle$	(4)
$\langle relation\text{-}stmt\rangle$	$::=$	"$local$" $says\ \langle relation\text{-}atom\rangle$	(5)
$\langle assert\text{-}stmt\rangle$	$::=$	$\langle sub\rangle\ asserts\ \langle assert\text{-}atom\rangle$	(6)
$\langle auth\text{-}stmt\text{-}body\rangle$	$::=$	$\langle sub\rangle\ grants\ \langle auth\text{-}atom\rangle\ to\ \langle sub\rangle$	(7)
$\langle auth\text{-}stmt\text{-}head\rangle$	$::=$	$\langle sub\rangle\ grants\ \langle auth\text{-}atom\rangle\ to\ \langle sub\text{-}ext\text{-}struct\rangle$	(8)
$\langle delegate\text{-}stmt\text{-}body\rangle$	$::=$	$\langle sub\rangle\ delegates\ \langle auth\text{-}atom\rangle\ with\ depth\ \langle k\rangle\ to\ \langle sub\rangle$	(9)
$\langle delegate\text{-}stmt\text{-}head\rangle$	$::=$	$\langle sub\rangle\ delegates\ \langle auth\text{-}atom\rangle\ with\ depth\ \langle k\rangle\ to\ \langle sub\text{-}struct\rangle$	(10)
$\langle relation\text{-}atom\rangle$	$::=$	$below(\langle obj\rangle, \langle obj\rangle) \mid below(\langle priv\rangle, \langle priv\rangle \mid neq(\langle entity\rangle, \langle entity\rangle) \mid eq(\langle entity\rangle, \langle entity\rangle)$	(11)
$\langle assert\text{-}atom\rangle$	$::=$	$exp(\langle entity\text{-}set\rangle)$	(12)
$\langle auth\text{-}atom\rangle$	$::=$	$right(\langle sign\rangle, \langle priv\rangle, \langle obj\rangle)$	(13)
$\langle obj\rangle$	$::=$	$\langle obj\text{-}con\rangle \mid \langle obj\text{-}var\rangle$	(14)
$\langle priv\rangle$	$::=$	$\langle priv\text{-}con\rangle \mid \langle priv\text{-}var\rangle$	(15)
$\langle sub\rangle$	$::=$	$\langle sub\text{-}con\rangle \mid \langle sub\text{-}var\rangle$	(16)
$\langle sub\text{-}set\rangle$	$::=$	$\langle sub\text{-}con\rangle \mid \langle sub\text{-}con\rangle,\ \langle sub\text{-}set\rangle$	(17)
$\langle sub\text{-}struct\rangle$	$::=$	$\langle sub\rangle$ | "["$\langle sub\text{-}set\rangle$"]" | $\langle threshold\rangle$	(18)
$\langle sub\text{-}ext\text{-}set\rangle$	$::=$	$\langle dth\rangle \mid \langle dth\rangle,\ \langle sub\text{-}ext\text{-}set\rangle$	(19)
$\langle sub\text{-}ext\text{-}struct\rangle$	$::=$	$\langle sub\rangle$ | "["$\langle sub\text{-}set\rangle$"]" | $\langle threshold\rangle$ | "["$\langle sub\text{-}ext\text{-}set\rangle$"]"	(20)
$\langle entity\rangle$	$::=$	$\langle sub\rangle \mid \langle obj\rangle \mid \langle priv\rangle$	(21)
$\langle entity\text{-}set\rangle$	$::=$	$\langle entity\rangle \mid \langle entity\rangle,\ \langle entity\text{-}set\rangle$	(22)
$\langle sign\rangle$	$::=$	$+ \mid - \mid \Box$	(23)
$\langle k\rangle$	$::=$	$\langle natural\text{-}number\rangle$	(24)
$\langle threshold\rangle$	$::=$	$\langle sth\rangle \mid \langle dth\rangle$	(25)
$\langle sth\rangle$	$::=$	$sthd(\langle k\rangle$, "["$\langle sub\text{-}set\rangle$"]")	(26)
$\langle dth\rangle$	$::=$	$dthd(\langle k\rangle, \langle sub\text{-}var\rangle, \langle assert\text{-}stmt\rangle)$	(27)
$\langle query\rangle$	$::=$	$\langle sub\rangle\ requests\ (+, \langle priv\rangle, \langle obj\rangle)$ | "["$\langle sub\text{-}set\rangle$"]" $requests\ (+, \langle priv\rangle, \langle obj\rangle)$	(28)

Figure 2.1: BNF for the Authorization Language $\mathcal{AL}$

trusted subjects.

Atoms

An atom is a function symbol with n arguments − generally $n = 1$, 2, or 3 − that are 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 2-ary and expresses the relationship of two entities. We provide three relation atoms, *eq, neq*, and *below.* The atoms *eq* and *neq* denote that two entities are equal and not equal, and the atom *below* denotes the hierarchy structure for object and privilege entities. In most realistic systems, the data information is organized using hierarchy structure, such as file systems and object oriented database system. For example, $below(ftp, pub\text{-}services)$ denotes that ftp is one of $pub\text{-}services$.

2. $\langle$*assert-atom*$\rangle$. This type of atoms, denoted by *exp* $(a_1, \ldots, a_n)$, is an application dependant function symbol with n arguments, that are constant or variable entities, to state the property of entities or the relationship among them. The *assert-atom* is a kind of flexible atoms in language $\mathcal{AL}$. For example, $isaTutor(alice)$ denotes that $alice$ is a tutor.

3. $\langle$*auth-atom*$\rangle$. The *auth-atom* is of the form,

   $$right(\langle sign\rangle, \langle priv\rangle, \langle obj\rangle),$$

   in which *sign* is +, -, or $\Box$. It states $positive(+)$ privilege, $negative(\text{-})$ privilege, or both($\Box$) of them. When an auth atom is used in a delegation statement, the *sign* is $\Box$ to denote both positive and negative authorizations. For example, $right(+, update, students)$ indicates the positive *update* privilege on $students$.

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 structured resources and privileges. Language $\mathcal{AL}$ provides 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 $s_i \neq s_j\ for\ 1 \leq i, j \leq n\ (i \neq j)$. This structure states that k subjects will be chosen from the threshold pool.

The dynamic threshold structure is of the form:

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

where S is a subject variable and we require that S is an argument in the assert atom $exp(\ldots, S, \ldots)$. This structure denotes that k subjects who satisfy the assert statement will be chosen.

Rules

The rule is of the form,

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

The basic unit of a rule is a statement. Let h_0 be a head statement and b_i a body statement, then a rule is expressed as follows:

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

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.

Query

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

sub requests $right(+, p, o)$, and

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

Through a 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 a single subject. If we grant an authorization to a group subject, we permit it only when all subjects in the group request the authorization at the same time.

2.1.2 Examples 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 $below(bob,\ postgraduate)$.

Structured privileges

In a database system, there are a group of privileges *allrights* including *insert*, *delete*, and *select*.

$local$ says $below(insert,\ allrights)$.

$local$ says $below(delete,\ allrights)$.

$local$ says $below(select,\ allrights)$.

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* with two steps.

$local$ delegates $right(\square,\ access,\ X)$ with depth 2 to ipB if
  $local$ says $below(X,\ allServices)$, $local$ says $neq(X,\ ssh)$.

$local$ grants $right(+,\ access,\ X)$ to ipA 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 department, and *techC*, a technician from IT department.

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

Negative authorization

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

$$\textit{sa} \text{ grants } \textit{right}(-,\ \textit{access},\ \textit{ftp}) \text{ to } \textit{ipB}.$$

Nonmonotonic reasoning

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

$$\begin{array}{l}\textit{sa} \text{ grants } \textit{right}(+,\ \textit{access},\ \textit{mysql}) \text{ to } X \ \text{ if} \\ \quad \textit{hrM asserts isStaff}(X), \text{ with absence } \textit{hrM asserts onHoliday}(X).\end{array}$$

2.2 Semantics

2.2.1 Basic Idea

$\mathcal{AL}$ is a high level language to specify authorization policies, in which the basic units are rules as described in Figure 2.1. We use a *domain description* to specify a policy base.

Definition 2.1 *A* domain description $\mathcal{D}_{\mathcal{AL}}$ *of language* $\mathcal{AL}$ *is a finite set of rules.*

Definition 2.2 *The* size *of a* domain description $\mathcal{D}_{\mathcal{AL}}$*, denoted as* $|\mathcal{D}_{\mathcal{AL}}|$*, is the number of rules in* $\mathcal{D}_{\mathcal{AL}}$*.*

We present an ASP based language $\mathcal{L}_{Ans}$ for the semantics of $\mathcal{AL}$. The semantics of language $\mathcal{AL}$ is defined by translating the domain description $\mathcal{D}_{\mathcal{AL}}$ of $\mathcal{AL}$ into a logic program $\mathcal{P}$ of $\mathcal{L}_{Ans}$. We define function *TransRules*$(\mathcal{D}_{\mathcal{AL}})$ to translate a domain description $\mathcal{D}_{\mathcal{AL}}$ into a program $\mathcal{P}$, and function *TransRules*$(\mathcal{Q}_{\mathcal{AL}})$ to translate query $\mathcal{Q}_{\mathcal{AL}}$ into program Π and ground literals $\varphi(+)$ and $\varphi(-)$. $\varphi(+)$ is used for the positive privilege and $\varphi(-)$ for the negative privilege. A query is solved based on $\mathcal{P}$, Π and φ via *Smodels*. We give the formal semantics for language $\mathcal{AL}$ as follows.

Definition 2.3 *Given a domain description $\mathcal{D}_{\mathcal{AL}}$ and a query $\mathcal{Q}_{\mathcal{AL}}$ of language $\mathcal{AL}$, we define functions $TransRules(\mathcal{D}_{\mathcal{AL}}) = \mathcal{P}$ and $TransQuery(\mathcal{Q}_{\mathcal{AL}}) = \langle \Pi, \varphi(+), \varphi(-) \rangle$. 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.*[1]

2.2.2 The language $\mathcal{L}_{Ans}$

$\mathcal{L}_{Ans}$ is an ASP based language with answer set semantics. A program of $\mathcal{L}_{Ans}$ can be computed by *Smodels*. In this subsection, we first present the alphabet for language $\mathcal{L}_{Ans}$, and then give the propagation rules, authorization rules, and conflict resolution and decision rules in $\mathcal{L}_{Ans}$.

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 the translation to state a set of subjects. All the constant entities start with a lowercase letter.

   Accordingly, there are three disjoint 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 letter.

[1] *Functions TransRules and TransQuery will be specified in section 2.2.3.*

2. Function symbols:

   right($sign$, $priv$, obj), where $sign$ is $+$, $-$, or $\Box$, $priv$ is of privilege sort, and obj is of object sort.

   exp($a_1, \ldots,$ a_n), where a_i is an entity sort, and exp is an application dependant assertion atom name. For example, $isDoctorOf(alice, tom)$ states that *alice* is the doctor for *tom*.

   In *Smodels*, both functions are symbolic functions, each of which just defines a new constant as an argument for the predicates in a program. We define functions 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 variables in both kinds of functions and they are just ordinary constant arguments for the related predicates.

   max($t_1, \ldots, t_n$), where each t_i $(i = 1, 2, \ldots, n)$ is an integer. The function returns the biggest integer among $t_1, t_2, \ldots, t_n$.

   min($t_1, \ldots, t_n$), where each t_i $(i = 1, 2, \ldots, n)$ is an integer. The function returns the smallest integer among $t_1, t_2, \ldots, t_n$[2].

3. Predicate symbols:

   below($arg1$, $arg2$), where $arg1$ and $arg2$ are of the same *entity sort* to state partial order relationship in a hierarchy structure. For example, below(read, write) means the privilege *read* is dominated by *write*.

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

   auth($issuer$, $grantee$, $right(sign, priv, obj), step$), where *issuer* and *grantee* are both of *subject entity* sort, and *step* 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*

[2]Because *Smodels* does not provide *max* and *min* functions, we have extended *Smodels* by adding them in it.

and *delegatee* are of *subject entity* sorts, and *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.

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

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*.

For the group subject query, we present predicate *ggrant* and *match*.

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.

match$(sub, right(sign, priv, obj))$, where *sub* is one of *subject group entities* introduced during the translation process. It states that subjects requesting the *right* are exactly those who are authorized.

We also introduce some predicates for authorization and conflict resolving rules.

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

pos_far$(sub, right(+, priv, obj), step)$, where *sub* is of subject entity sort. It states that there is at least one negative authorization $right(-, priv, obj)$ for *sub* which has less steps than the positive authorization $right(+, priv, obj)$ with *step* for *sub*. For example, if both of $auth(local, s, right(+, read, file), 4)$ and $auth(local, s, right(-, read, file), 3)$ exist, we can get $pos\_far(s, right(+, read, file), 4)$.

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

neg_far$(sub, right(-, priv, obj), step)$, where *sub* is of subject entity sort. It is similar with $pos\_far(sub, right(+, priv, obj), step)$.

Propagation rules

We need propagation rules because in most real world situations, the work to assign the authorization to all resources is burdensome and not necessary. The security officer prefers to assign them partially and propagate them to all resources based on propagation policy. In $\mathcal{L}_{Ans}$, we have propagation rules based on the relationships between objects or privileges as follows.

$$below(A_1, A_3) \leftarrow\ below(A_1, A_2),\ below(A_2, A_3) \tag{2.1}$$

Rule (2.1) is for the structured data propagation.

Authorization rules

Using negation as failure, if there is only positive authorization and no negative authorization, we will conclude the positive authorization; if there is no positive authorization, we will conclude the negative authorization. The following is our authorization rules.

$$exist\_pos(X, right(+, P, O)) \leftarrow auth(local, X, right(+, P, O), T). \tag{2.2}$$

$$exist\_neg(X, right(-, P, O)) \leftarrow auth(local, X, right(-, P, O), T). \tag{2.3}$$

Rules (2.2) and (2.3) state that there are the positive or negative authorizations in the system respectively.

$$\begin{aligned} &grant(X, right(+, access, O)) \leftarrow \\ &\quad auth(local, X, right(+, P, O), T), \\ &\quad not\ exist\_neg(X, right(-, P, O)). \end{aligned} \tag{2.4}$$

Rule (2.4) makes the positive authorization decision for the single subject request if there is at least a positive authorization and not any negative authorization in the system.

$$grant(X, right(-, P, O)) \leftarrow not\ exist\_pos(X, right(+, P, O)). \tag{2.5}$$

Rule (2.5) makes the negative authorization decision for the single subject request if there is not any positive authorization, no matter whether there is a negative authorization or not.

$$\begin{aligned}
&ggrant(L, right(+, P, O)) \leftarrow \\
&\quad auth(local, L, right(+, P, O), T), \\
&\quad match(L, right(+, P, O)), \\
&\quad not\ exist\_neg(L, right(-, P, O)).
\end{aligned} \tag{2.6}$$

Rule (2.6) makes the positive authorization decision for the group subject request if there is at lease a positive authorization and not any negative authorization in the system, and the requesters satisfy the group subject requirement.

$$ggrant(L, right(-, P, O)) \leftarrow not\ exist\_pos(L, right(+, P, O)). \tag{2.7}$$

Rule (2.7) makes the negative authorization decision for the group subject request if no positive authorization exists.

Conflict resolution and decision rules

In an access control system, when both positive and negative authorizations are permitted, conflicts might occur. Most existing approaches deal with conflicts in the following ways: (1) No conflict policy. It relies on the security administrator to write the consistent authorization rules. If there are conflicts, errors happen [97]. (2) A fixed-conflict resolving policy based on relative authorization or specification. As pointed out in [80], this kind of policies include negative (positive)-take-precedence, strong and weak authorization, specific-take-precedence, and time-take-precedence. Moreover, Agudo et. al. [3] and Ruan et. al. [80, 81], have proposed graph-based schemes to deal with distributed authorization, in which they presented predecessor-take-precedence and strict-predecessor-take-precedence, respectively. (3) Flexible scheme to support multiple conflict resolving policies [58]. Logic-based approaches for distributed authorization can easily specify different policies that coexist in the same framework.

Since our work is logic-based approach for distributed authorization, it is feasible to integrate different conflict resolving policies into our approach. Comparing with the weighted-graph based approach [3, 81], we should mention that, it is easy to extend our language to handle weighted authorization because *Smodels* already provided weight literal representation in logic programming. In this chapter, we choose trust-take-precedence policy, similar to the work in [80], to deal with conflicts. We consider *delegation* as an action and assign the step for each authorization which is decided by the delegation step. All the authorizations arise from *local* originally. The subject *local* has the highest priority for the authorizations, while the subjects who directly receive the delegable authorizations from *local* have the second highest priority, and so on. The step number in the predicate $auth(\ldots)$ states how far the authorization is away from *local* which reflects the trust extent. The smaller the authorization step, the more trustable on this authorization. For this reason, the authorization with the smallest step overrides other ones. If the conflict occurs with the same priority, we deny the request. To find the smallest step authorization, we introduce a predicate $pos\_far(X, right(+, P, O), T)$ which is true in the program if there exists a negative authorization which has a smaller step than the positive authorization, $auth(local, right(+, P, O), T)$. In other words, the positive authorization is far from *local* at least than one corresponding negative authorization. Symmetrically, we have the predicate $neg\_far(X, right(-, P, O), T)$. They are showed as follows:

$$\begin{aligned}
&pos\_far(X, right(+, P, O), T_1) \leftarrow \\
&\quad auth(local, X, right(+, P, O), T_1), \\
&\quad auth(local, X, right(-, P, O), T_2), \\
&\quad T_1 > T_2.
\end{aligned} \tag{2.8}$$

$$\begin{aligned}
&neg\_far(X, right(-, P, O), T_1) \leftarrow \\
&\quad auth(local, X, right(-, P, O), T_1), \\
&\quad auth(local, X, right(+, P, O), T_2),
\end{aligned}$$

$$T_1 > T_2. \tag{2.9}$$

For a positive authorization $\mathcal{A}$, $auth(local, S, right(+, P, O), T)$, if a corresponding predicate $pos\_far(\ldots)$ exists, we can say that there is at least one corresponding negative authorization which has higher priority than $\mathcal{A}$ based on Rule (2.8). If there does not exist a corresponding predicate $pos\_far(\ldots)$, the priority of $\mathcal{A}$ is higher or at lease not lower than that of any corresponding negative authorization. We have the similar explanation for Rule (2.9). Therefore we have the following conflict resolution rules:

$$\begin{aligned}
&grant(X, right(+, P, O)) \leftarrow \\
&\quad auth(local, X, right(-, P, O), T_2), \\
&\quad neg\_far(X, right(-, P, O), T_2), \\
&\quad auth(local, X, right(+, P, O), T_1), \\
&\quad not\ pos\_far(X, right(+, P, O), T_1).
\end{aligned} \tag{2.10}$$

$$\begin{aligned}
&grant(X, right(-, P, O)) \leftarrow \\
&\quad auth(local, X, right(+, P, O), T_1), \\
&\quad auth(local, X, right(-, P, O), T), \\
&\quad not\ neg\_far(X, right(-, P, O), T).
\end{aligned} \tag{2.11}$$

Rule (2.10) states we make the positive decision if for the positive authorization there does not exist a predicate $pos\_far(\ldots)$ and for any corresponding negative authorization, there exists a predicate $neg\_far(\ldots)$. Rule (2.11) states we make the negative decision if the priority of the negative authorization is higher or at least not lower than that of any corresponding positive authorization.

Rules (2.10) and (2.11) are for the *single subject request*. Using the same approach, we provide the conflict resolution rules for the *group subject request*. In Rules (2.12) and (2.13), the predicate *match*$(\ldots)$ states that the group of the requesters satisfies the authorization's requirement.

$$
\begin{aligned}
&ggrant(L, right(+, P, O)) \leftarrow \\
&\quad auth(local, L, right(-, P, O), T_2), \\
&\quad neg\_far(L, right(-, P, O), T_2), \\
&\quad match(L, right(+, P, O)), \\
&\quad auth(local, L, right(+, P, O), T_1), \\
&\quad not\ pos\_far(L, right(+, P, O), T_1).
\end{aligned} \tag{2.12}
$$

$$
\begin{aligned}
&ggrant(L, right(-, P, O)) \leftarrow \\
&\quad auth(local, L, right(+, P, O), T_1), \\
&\quad auth(local, L, right(-, P, O), T_2), \\
&\quad match(L, right(-, P, O)), \\
&\quad not\ neg\_far(L, right(-, P, O), T_2).
\end{aligned} \tag{2.13}
$$

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

As shown earlier, a rule $r_{\mathcal{D}}$ in the domain description $\mathcal{D}_{\mathcal{AL}}$ is of the following form

$$h_0\ if\ b_1,\ b_2, \ldots, b_m,\ with\ absence\ b_{m+1}, \ldots,\ b_n. \tag{2.14}$$

where h_0 is the *head statement* denoted by $head(r_{\mathcal{D}})$ and b_i's 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 the context, we use *positive statements* and *negative statements* to express them respectively. In (2.14), if $m = 0$ and $n = 0$, the rule simply becomes h_0 and is 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 a logic program $\mathcal{P}$. We divide the process into three phases, body translation(see subsection 2.2.3), head translation(see subsection 2.2.3), and adding related rules (see subsections 2.2.2). 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 $\varphi(-)$.

Function symbols, *assert-atom* and *auth-atom* in language $\mathcal{AL}$, correspond to 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.

Body transformation

In language $\mathcal{AL}$, there are four types of body statements, *relation statement, assert statement, delegation statement*, and *auth statement.* As *delegation statement* and *auth statement* have similar structures, we present their transformations 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 $neq(arg_1,\ arg_2)$
   local says $eq(arg_1,\ arg_2)$

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

$$below(arg_1,\ arg_2), \tag{2.15}$$

$$neq(arg_1,\ arg_2), \tag{2.16}$$

$$eq(arg_1,\ arg_2), \tag{2.17}$$

   where arg_1 and arg_2 in $below(arg_1, arg_2)$ are of *object or privilege entity sort*; arg_1 and arg_2 in $neg(arg_1, arg_2)$ and $eq(arg_1, arg_2)$ are of the same type entity sort to specify whether they are equal or not. In *Smodels*, *neq* and *eq* are internal functions and work as constraints for the variables in the rules.

2. Assert statement:

   issuer asserts *exp*.

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

$$assert(issuer, exp), \tag{2.18}$$

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

3. Delegation body statement or auth body statement:

   issuer delegates *right* with depth k to *delegatee*,
   issuer grants *right* to *grantee.*

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

$$delegate(issuer, delegatee, right, k, T), and \tag{2.19}$$

$$auth(issuer, grantee, right, T), \tag{2.20}$$

   where k is the delegation depth and T is a step variable that indicates how many steps the delegation/right has gone through from *issuer* to *delegatee/grantee*.

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

Head transformation

In language $\mathcal{AL}$, there are four types of head statements, *relation statement, assert statement, delegation statement*, and *auth statement*. If the head statement h_0 is a *relation statement* or an *assert statement*, the translations are the same as the body statements. We adopt rules (2.15) and (2.18) to translate them respectively. Note that the relation statements for atoms *neq* and *eq* are used as variable constraints in body statements and we do not use them as head statements. Here we present the translation for *auth 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

$$auth(issuer, grantee, right(Sn, P, O), 1), \tag{2.21}$$

where 1 is the authorization step to state that the *right* is granted from *issuer* to *grantee* directly.

We further add the following propagation rules into program $\mathcal{P}$:

$$\begin{aligned} &auth(issuer, grantee, right(Sn, P_1, O), 1) \leftarrow \\ &\quad auth(issuer, grantee, right(Sn, P, O), 1),\ below(P_1, P). \end{aligned} \tag{2.22}$$

$$\begin{aligned} &auth(issuer, grantee, right(Sn, P, O_1), 1) \leftarrow \\ &\quad auth(issuer, grantee, right(Sn, P, O), 1),\ below(O_1, O). \end{aligned} \tag{2.23}$$

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 the complex subject structures, and replace its head in program $\mathcal{P}$ as follows

$$auth(issuer, l_{new}, right(Sn, P, O), 1). \tag{2.24}$$

We also need to add the propagation rules similar to rules (2.22) and (2.23) and the following rules for the complex structures.

case 1: l_{new} is $[s_1, \ldots, s_n]$

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

case 2: l_{new} is $sthd(k, [s_1, s_2, \ldots, s_n])$

$match(l_{new}, right) \leftarrow$

$$\begin{aligned}&auth(issuer, l_{new}, right, 1),\\ &k\{req(s_1, right), \ldots,\ req(s_n, right)\}k.\end{aligned}$$

case 3: l_{new} is $dthd\ (k, S, sub\ asserts\ exp(S)\ )$

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

case 4: l_{new} is $[dthd\ (k_1, S_1, s_1\ asserts\ exp_1(S_1)\ ), \ldots,$ $dthd\ (k_n, S_n, s_n\ asserts\ exp_n(S_n)\ )].$

$$\begin{aligned}&match(l_{new}, right) \leftarrow\\ &\quad auth(issuer, l_{new}, right, 1),\\ &\quad k_1\{req(S_1, right) : assert(s_1, exp_1(S_1))\}k_1,\\ &\qquad \vdots\\ &\quad k_n\{req(S_n, right) : assert(s_n, exp_n(S_n))\}k_n.\end{aligned}$$

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 $\mathcal{P}$ using:

$$delegate(issuer, delegatee, right, k, 1), \tag{2.25}$$

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

   Moreover, we need to add the following implicit rules in program $\mathcal{P}$:

   Prop-delegation rules: Based on the structured resources, the delegation can be propagated as following rules.

$$\begin{aligned}&delegate(issuer, delegatee, right(\Box, P_1, O), k, 1) \leftarrow\\ &\quad delegate(issuer, grantee, right(\Box, P, O), k, 1),\\ &\quad below(P_1, P).\end{aligned} \tag{2.26}$$

$$\begin{aligned}&\mathit{delegate}(\mathit{issuer}, \mathit{delegatee}, \mathit{right}(\square, P, O_1), k, 1) \leftarrow\\ &\quad \mathit{delegate}(\mathit{issuer}, \mathit{delegatee}, \mathit{right}(\square, P, O), k, 1),\\ &\quad \mathit{below}(O_1, O).\end{aligned} \tag{2.27}$$

Auth-delegation rule: When the *issuer* delegates a *right* to the *delegatee*, the *issuer* will agree with the *delegatee* to grant the *right* to other subjects within the delegation depth. The authorization step increases by 1.

$$\begin{aligned}&\mathit{auth}(\mathit{issuer}, S, \mathit{right}(Sn, P, O), T+1) \leftarrow\\ &\quad \mathit{delegate}(\mathit{issuer}, \mathit{delegatee}, \mathit{right}(\square, P, O), k, 1),\\ &\quad \mathit{auth}(\mathit{delegatee}, S, \mathit{right}(Sn, P, O), T).\end{aligned} \tag{2.28}$$

Delegation-chain rule: The delegation can be further delegated within the delegation depth.

$$\begin{aligned}&\mathit{delegate}(\mathit{issuer}, S, \mathit{right}(\square, P, O), \mathit{min}(k\text{-}\mathit{Step}, \mathit{Dep}), 1+T) \leftarrow\\ &\quad \mathit{delegate}(\mathit{issuer}, \mathit{delegatee}, \mathit{right}(\square, P, O), k, 1),\\ &\quad \mathit{delegate}(\mathit{delegatee}, S, \mathit{right}(\square, P, O), \mathit{Dep}, T),\ T\ <\ k.\end{aligned} \tag{2.29}$$

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

$$\begin{aligned}&\mathit{delegate}(\mathit{delegatee}, \mathit{delegatee}, \mathit{right}, \mathit{Dep}, 1) \leftarrow\\ &\quad \mathit{delegate}(\mathit{issuer}, \mathit{delegatee}, \mathit{right}, k, 1), \mathit{Dep} \le k.\end{aligned} \tag{2.30}$$

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

$$\begin{aligned}&\mathit{delegate}(\mathit{issuer}, \mathit{delegatee}, \mathit{right}, \mathit{Dep}, 1) \leftarrow\\ &\quad \mathit{delegate}(\mathit{issuer}, \mathit{delegatee}, \mathit{right}, k, 1), \mathit{Dep} < k.\end{aligned} \tag{2.31}$$

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

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

We also need to add additional rules. Because there are similar rules for different complex *delegatee* structure, here we just present the rules for the *subject set* structure as follows:

Prop-delegation rules: Based on the structured resources, the delegation can be propagated similar to those for a single *delegatee*.

$$\begin{aligned}
&delegate(issuer, l_{new}, right(\square, P_1, O), k, 1) \leftarrow \\
&\qquad delegate(issuer, l_{new}, right(\square, P, O), k, 1),\ below(P_1, P). \\
&delegate(issuer, l_{new}, right(\square, P, O_1), k, 1) \leftarrow \\
&\qquad auth(issuer, l_{new}, right(\square, P, O), k, 1),\ below(O_1, O).
\end{aligned}$$

Auth-delegation rule: If an *issuer* delegates a *right* to a group subject, and all the members in the group authorize this *right* to a subject, the *issuer* agrees with this authorization. The new authorization step is 1 plus the biggest one among the group authorizations because the trust for the new authorization is less than anyone among the group authorizations.

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

Delegation-chain rule: If an *issuer* delegates a *right* to a group subject, and all the members in the group further delegate this *right* to a subject, the *issuer* agrees with this re-delegation. The new delegation depth is the smallest one among k minus $step_i$ and Dep_i and the new delegation step is 1 plus the biggest one among the group delegations.

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

Query Transformation

In language $\mathcal{AL}$, there are two kinds of queries, single subject queries and group subject queries. We present the function *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 state the set of subjects, $[s_1, \ldots, s_n]$.

2.3 Computational Analysis

In this section, we study basic computational properties of language $\mathcal{AL}$. From Definition 2.3, we can see that given a domain description $\mathcal{D}_{\mathcal{AL}}$ and a query $\mathcal{Q}_{AL}$ (see Figure 2.1 for the syntax of a query), there are three steps to answer this query: (1) transfer $\mathcal{D}_{\mathcal{AL}}$ and $\mathcal{Q}_{AL}$ to a logic program $\mathcal{T}$; (2) compute the answer sets of $\mathcal{T}$; (3) check whether $grant(s, right(sn, p, o))$ or $ggrant(l, right(sn, p, o))$ is in all answer sets of $\mathcal{T}$. Step 1 is achieved through two transformation functions: $\mathit{TransRules}(\mathcal{D}_{\mathcal{AL}}) = \mathcal{P}$ and $\mathit{TransQuery}(\mathcal{Q}_{\mathcal{AL}}) = \langle \Pi, \varphi(+), \varphi(-)\rangle$. That is, $\mathcal{T} = \mathcal{P} \cup \Pi$. For step 2, we provide function $\mathit{stable}(\mathcal{T})$ which returns all answer sets of program $\mathcal{T}$. Step 3 is just a simple checking that can be done in linear time. So the main computational cost for our approach is on Steps 1 and 2. The following proposition presents the complexity result for achieving Step 1.

Proposition 2.1 *Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description of language $\mathcal{AL}$ and $\mathcal{Q}_{\mathcal{AL}}$ a query. Then $TranRules(\mathcal{D}_{\mathcal{AL}})$ can be computed in time $O(|\mathcal{D}_{\mathcal{AL}}|)$ and $TransQuery(\mathcal{Q}_{AL})$ can be computed in time $O(|\mathcal{Q}_{AL}|)$(* here $|\mathcal{D}_{\mathcal{AL}}|$ is the size of $\mathcal{D}_{\mathcal{AL}}$ and $|\mathcal{Q}_{AL}|$ is the length of $\mathcal{Q}_{AL}$.*).*

Proof. From the transformation process, it is clear that $\mathit{TransQuery}(\mathcal{Q}_{\mathcal{AL}})$ entirely depends on the query formula's length. So it can be obtained in linear time in terms of $\mathcal{Q}_{\mathcal{AL}}$'s size. On the other hand, rule transformation includes body transformation and head transformation. After body transformation, the number of rules does not change. So $|\mathcal{P}_{body}| = |\mathcal{D}_{\mathcal{AL}}|$. During the head transformation, for the complex structures in the auth head statement and delegation head statement, $|\mathcal{P}_{head}| = c|\mathcal{D}_{\mathcal{AL}}|$, where c is a constant number. So we conclude that $|\mathcal{P}_{head}| = O(|\mathcal{D}_{\mathcal{AL}}|)$. □

Now we consider the computation of Step 2. For $\mathit{stable}(\mathcal{T})$, we use *Smodels* to compute the answer sets of logic programs. It is well known that deciding whether a program has an answer set is NP-complete [9]. Consequently, *Smodels* usually needs exponential time to compute a program's answer sets. Therefore, it is important to identify proper subclasses of the authorization domains where queries can be answered in polynomial time. In the following section, we will define two subclasses of language $\mathcal{AL}$ in which queries can be computed in polynomial time.

The domain description of language $\mathcal{AL}$ includes finite rules and the basic unit of a rule is a statement. We have four types of statements, *relation statement, assert statement, delegation statement*, and *auth statement.* To simplify our investigation, we consider each statement as a predicate with n terms, $p(t_1, t_2, \ldots, t_n)$ in which p states the statement type, and t_is are terms to state the variable parts in the statement. The four types of statements have the following forms,

$$RelStmt(issuer, relAtomName, atomArg_1, atomArg_2)$$
$$AssertStmt(issuer, assertAtomName, atomArg_1, \ldots, atomArg_n)$$
$$DelegationStmt(issuer, receiver, step, priv, obj)$$
$$AuthStmt(issuer, receiver, sign, priv, obj)$$

For instance, we denote a relation statement, *"local says below(alice, postgraduate)"* using predicate form, $RelStmt(local, below, alice, postgraduate)$. In such predicate presentation, each term has a type which can be subject, subject structure, object, privilege, sign, integer, relation atom name, or assert atom name. Subject structures are special terms and have four types: subject set, subject static threshold, subject dynamic threshold and subject extended dynamic threshold. In the subject set $[s_1, s_2, \ldots, s_n]$ and the subject static threshold $sthd(k, [s_1, s_2, \ldots, s_n])$, there exists a *static subject pool* $[s_1, s_2, \ldots, s_n]$. Each s_i is a *member of the static subject pool.* In a *subject dynamic threshold* or *subject extended dynamic threshold*, there is a *dynamic subject pool.* Each constant subject in the domain of the application system may be a *member of the dynamic subject pool.*

Definition 2.4 *Term t_1, and t_2 are* compatible*, denoted by $t_1 \simeq t_2$, if t_1 and t_2 are the same type terms, and one of the following conditions holds:*

1. *t_1 and t_2 are constant terms with the same name;*
2. *at least one of t_1 and t_2 is a variable term; or*
3. *t_1 is a subject constant, t_2 is a subject structure, and t_1 is a member of t_2.*

For example, if term t_1 is a subject *alice*, t_2 is a subject variable S, t_3 is a static threshold $sthd(2, [bob, carol, david])$, and t_4 is a dynamic threshold $dthd(3, S, hrM$ *asserts* $isAStaff(S))$, we say $t_1 \simeq t_2$, $t_1 \simeq t_4$, $t_2 \simeq t_3$, and $t_2 \simeq t_4$.

Definition 2.5 *Two statements st_1 and st_2 are* compatible, *denoted by $st_1 \simeq st_2$, if st_1 and st_2 have the predicate forms st'_1 and st'_2 respectively, and*

1. *st'_1 and st'_2 have the same type predicates,*
2. *all the corresponding terms of st'_1 and st'_2 are compatible.*

From the above definitions, it is easy to see that a statement is compatible to itself.

Definition 2.6 *Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description of language $\mathcal{AL}$ and r_p and r_q be two rules in $\mathcal{D}$. We define a set $\mathcal{S}(r_p)$ of statements with respect to r_p as follows:*

$$\begin{aligned}
&\mathcal{S}_0 = \{head(r_p)\};\\
&\mathcal{S}_i = \mathcal{S}_{i-1} \cup \ \{head(r) \mid head(r') \simeq s \textit{ where } s \in pos(r) \textit{ and}\\
&\qquad\qquad\qquad r' \textit{ are those rules such that } head(r') \in \mathcal{S}_{i-1}\};\\
&\mathcal{S}(r_p) = \textstyle\bigcup_{i=1}^{\infty} \mathcal{S}_i.
\end{aligned}$$

We say that r_q is *defeasible through* r_p in $\mathcal{D}_{\mathcal{AL}}$ if and only if $neg(r_q) \cap^c \mathcal{S}(r_p) \neq \emptyset$ [3].

Intuitively, if r_q is defeasible through r_p in $\mathcal{D}_{\mathcal{AL}}$, then there exists a sequence of rules $r_1, r_2, \ldots, r_l, \ldots$ such that $head(r_p)$ occurs in $pos(r_1)$, $head(r_i)$ occurs in $pos(r_{i+1})$ for all $i = 1, \ldots$, and for some k, $head(r_k)$ occurs in $neg(r_q)$. Under this condition, it is clear that by triggering rule r_p in $\mathcal{D}_{\mathcal{AL}}$, it is possible to defeat rule r_q if rules $r_1, \ldots, r_k$ are triggered as well. As a special case that $\mathcal{S}(r_p) = \{head(r_p)\}$, r_q is defeasible through r_p iff $head(r_p) \in neg(r_q)$.

Definition 2.7 *Given a domain description $\mathcal{D}_{\mathcal{AL}}$, we define its* defeasible graph *$\mathcal{DG} = \langle V, E\rangle$, where V is the set of rules r_i in $\mathcal{D}_{\mathcal{AL}}$ as the vertices and E the set of $\langle r_i, r_j\rangle$ which is a directed edge to denote r_j is defeasible through r_i.*

Consider a simple example. Suppose $a, b, c, \ldots$ are statements , and $a', b', c', \ldots$ are their corresponding compatible statements in language $\mathcal{AL}$, and we have the following domain description $\mathcal{D}_{\mathcal{AL}}$:

[3]$\cap^c$ is to get a compatible joint set of two statement sets. Formally, $A \cap^c B = \{st_i | st_i \in A \textit{ and } st_i \in B \textit{ or } \exists st_j,\ st_j \in B,\ s_i \simeq s_j\}$. See Section 2.2.3 for definitions of $head(r)$, $pos(r)$ and $neg(r)$ in language $\mathcal{AL}$.

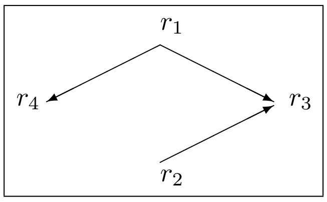

Figure 2.2: The defeasible graph of the domain description $\mathcal{D}_{\mathcal{AL}}$

r_1 : *b if a.*

r_2 : *c if b′.*

r_3 : *d if with absence c.*

r_4 : *e if with absence b.*

Based on the definitions above, we conclude that rule r_3 is defeasible through r_1 and r_2, and rule r_4 is defeasible through r_1. Then we have the defeasible graph in Figure 2.2.

The logic program $\mathcal{P}$ with answer set semantics consists of a finite set of rules. A rule r is expressed as follows:

$$L_0 \leftarrow L_1, \ldots, L_m,\ not\ L_{m+1}, \ldots,\ not\ L_n.$$

where each $L_i (0 \leq i \leq n)$ is a literal. The program $\mathcal{P}$ is *ground* if each rule in $\mathcal{P}$ is ground. Let r be a ground rule of the above form, we use $pos(r)$ to denote the set of literals in the body of r without negation as failure $\{L_1, \ldots, L_m\}$, $neg(r)$ to denote the set of literals in the body of r with negation as failure $\{L_{m+1}, \ldots, L_n\}$, and $body(r)$ for $pos(r) \cup neg(r)$. L_0 is called the head of the rule, denoted by $head(r)$. By extending these notations, we use $pos(\mathcal{P})$, $neg(\mathcal{P})$, $body(\mathcal{P})$, and $head(\mathcal{P})$ to denote the union of corresponding components of all rules in program $\mathcal{P}$, e.g. $body(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} body(r)$.

We present the concepts of local stratification and call consistence for extended logic programs [9, 99].

Definition 2.8 *Let $\mathcal{P}$ be an extended logic program and Lit be the set of all ground literals of $\mathcal{P}$.*

1. *A* local stratification *for $\mathcal{P}$ is a function* stratum *from Lit to the countable ordinals.*

2. *Given a local stratification* stratum*, we extend it to ground literals with negation as failure by setting* $stratum(not\ L) = stratum(L) + 1$*, where* L *is a ground literal.*

3. *A rule* $L_0 \leftarrow L_1, \ldots, L_m,\ not\ L_{m+1}, \ldots,\ not\ L_n$ *in* $\mathcal{P}$ *is locally stratified with respect to* stratum *if*

$$stratum(L_0) \geq stratum(L_i),\ where\ 1 \leq i \leq m,\ and$$
$$stratum(L_0) > stratum(not\ L_j),\ where\ m+1 \leq j \leq n.$$

4. $\mathcal{P}$ *is called* locally stratified *with respect to* stratum *if all of its rules are locally stratified.* $\mathcal{P}$ *is called* locally stratified *if it is locally stratified with respect to some local stratification.*

Definition 2.9 *An extended program is said to be* call-consistent *if its dependency graph does not have a cycle with an odd number of negative edges.*

Lemma 2.1 *Let* $\mathcal{P}_1$*,* $\mathcal{P}_2$ *be two locally stratified logic programs. Then the program* $\mathcal{P}_1 \cup \mathcal{P}_2$ *is locally stratified if* $head(\mathcal{P}_2) \cap body(\mathcal{P}_1) = \emptyset$*.*

Proof. Because $\mathcal{P}_1$ and $\mathcal{P}_2$ are two locally stratified propositional logic programs, their dependency graphs $\mathcal{D}_{\mathcal{P}_1}$ and $\mathcal{D}_{\mathcal{P}_2}$ both do not contain any negative cycles. Now $head(\mathcal{P}_2) \cap body(\mathcal{P}_1) = \emptyset$. We assume $\mathcal{P}_1 \cup \mathcal{P}_2$ is not locally stratified and its dependency graph $\mathcal{D}_{\mathcal{P}_1 \cup \mathcal{P}_2}$ contains a cycle with at least one negative edge, denoted by $\langle a_1, a_2, \ldots, a^-_{i-1}, a_i, \ldots, a_n, a_1 \rangle$, in which, a_is are atoms and there are edges between two conjoint atoms. We use a^-_{i-1} to denote that the edge from a_{i-1} to a_i is negative. Other edges are positive.

From the above condition, there are the following rules in the program $\mathcal{P}_1 \cup \mathcal{P}_2$:

$$r_1 :\ a_2 \leftarrow \ldots, a_1, \ldots$$
$$r_2 :\ a_3 \leftarrow \ldots, a_2, \ldots$$
$$\ldots$$
$$r_{i-1} :\ a_i \leftarrow \ldots, not\ a_{i-1}, \ldots$$
$$\ldots$$
$$r_{n-1} :\ a_n \leftarrow \ldots, a_{n-1}, \ldots$$
$$r_n :\ a_1 \leftarrow \ldots, a_n, \ldots$$

In the above rules, at lease one is from $\mathcal{P}_1$. Without loss of generality, we assume r_1 is in $\mathcal{P}_1$, r_n should be in $\mathcal{P}_1$ also, because $a_1 = head(r_n)$, $a_1 \in body(r_1)$ and $head(\mathcal{P}_2) \cap body(\mathcal{P}_1) = \emptyset$. For the same reason, we conclude that all of $r_{n-1}, r_{n-2}, \ldots, r_2$ should be in $\mathcal{P}_1$. This implies that $\mathcal{P}_1$ is not locally stratified. The contradiction happens. So we prove that if $\mathcal{P}_1$ and $\mathcal{P}_2$ are two locally stratified logic programs and $head(\mathcal{P}_2) \cap body(\mathcal{P}_1) = \emptyset$, then $\mathcal{P}_1 \cup \mathcal{P}_2$ is locally stratified too. □

Lemma 2.2 *Let $\mathcal{P}_1$, $\mathcal{P}_2$ be two call consistent logic programs. Then the program $\mathcal{P}_1 \cup \mathcal{P}_2$ is call consistent if $head(\mathcal{P}_2) \cap body(\mathcal{P}_1) = \emptyset$.*

Proof. The dependency graphs of $\mathcal{P}_1$ and $\mathcal{P}_2$ do not have a cycle with an odd number of negative edges because $\mathcal{P}_1$ and $\mathcal{P}_2$ are call consistent. Now we assume $head(\mathcal{P}_2) \cap body(\mathcal{P}_1) = \emptyset$. Suppose $\mathcal{P}_1 \cup \mathcal{P}_2$ is not call consistent and its dependency graph has a cycle with an odd number of negative edges. We can construct an atom sequence $\langle a_1, a_2, \ldots, a_n, a_1\rangle$ to denote a cycle with an odd number of negative edges, where a_i's are atoms and the sequence denotes that there are positive or negative edges to connect the atoms one by one. We get the following rules in $\mathcal{P}_1 \cup \mathcal{P}_2$:

$$
\begin{array}{l}
r_1 : \ a_2 \leftarrow \ldots, \ [not]^4 \ a_1, \ldots \\
r_2 : \ a_3 \leftarrow \ldots, \ [not] \ a_2, \ldots \\
\vdots \\
r_{i-1} : \ a_i \leftarrow \ldots, \ [not] \ a_{i+1}, \ldots \\
\vdots \\
r_{n-1} : \ a_n \leftarrow \ldots, \ [not] \ a_{n-1}, \ldots \\
r_n : \ a_1 \leftarrow \ldots, \ [not] \ a_n, \ldots
\end{array}
$$

In the above rules, at least one is from $\mathcal{P}_1$. Without loss of generality, we assume r_1 is in P_1. Because $head(\mathcal{P}_2) \cap body(\mathcal{P}_1) = \emptyset$, r_n should be in $\mathcal{P}_1$ as well. For the same reason, we conclude that all of $r_{n-1}, r_{n-2}, \ldots, r_2$ should be in $\mathcal{P}_1$. This implies that $\mathcal{P}_1$ is not a call consistent program. The contradiction happens. This proves our result. □

[4] [not] means that "not" is an option to denote that the following atom is positive or negative.

Lemma 2.3 *Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description and $\mathcal{P}$ the translated logic program corresponding to $\mathcal{D}_{\mathcal{AL}}$ in language $\mathcal{L}_{Ans}$. If the defeasible graph $\mathcal{DG}$ of $\mathcal{D}_{\mathcal{AL}}$ does not have a cycle, then $\mathcal{P}$ is locally stratified.*

Proof. The semantics of language $\mathcal{AL}$ is to translate $\mathcal{D}_{\mathcal{AL}}$ into a logic program $\mathcal{P}$. The process includes three steps: (a) translate rules in $\mathcal{D}_{\mathcal{AL}}$ into logic program rules and obtain the logic program $\mathcal{P}_1'$; (b) get $\mathcal{P}_1 = \mathcal{P}_1' \cup \{r\}$, where r is a propagation rule with the form of Rule (2.1).

The basic unit of rules in $\mathcal{D}_{\mathcal{AL}}$ is a statement which has a corespondent predicate in logic program. From the translation process (refer to section 2.2.3), we have the following observation:

> **Observation:** If the defeasible graph of $\mathcal{D}_{\mathcal{AL}}$ does not have a cycle, then the dependency graph of program $\mathcal{P}_1'$ does not have a negative cycle. So program $\mathcal{P}_1'$ is locally stratified.

We first show the correctness of this observation. From Definition 2.7 for defeasible graph, it is clear that there is a directed edge $\langle r_i, r_j \rangle$ between two different nodes r_i, r_j if rule r_j in $\mathcal{D}_{\mathcal{AL}}$ is defeasible through r_i. This implies in the dependency graph of program $\mathcal{P}_1'$, that there will be a path with a negative label marking on some edge from an atom occurring in rule r_i's head to an atom occurring in rule r_j's negative body, and *vice versa*. This follows that if the defeasible graph of $\mathcal{D}_{\mathcal{AL}}$ does not contain a cycle, then the dependency graph of program $\mathcal{P}_1'$ does not have a negative cycle. From the well known result about dependency graph [9], we know that program $\mathcal{P}_1'$ is locally stratified. This proves our observation.

Now consider $\mathcal{P}_1$ which is $\mathcal{P}_1'$ plus the propagation rule:

$$below(A_1, A_3) \leftarrow below(A_1, A_2),\ below(A_2, A_3).$$

Because in the domain description, the relation statements related to $below(\ldots)$ are just facts about resource relationship and as conditions for authorization rules, the program $\mathcal{P}_1$ is also locally stratified.

From the observation of authorization rules and conflict resolving rules in section 2.2.2 and section 2.2.2, obviously $\mathcal{P}_2$ is locally stratified and $head(\mathcal{P}_2) \cap body(\mathcal{P}_1) = \emptyset$.

From Lemma 2.1, we conclude that the logic program $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$ is locally stratified. □

Lemma 2.4 *Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description and $\mathcal{P}$ the translated logic program in language $\mathcal{L}_{Ans}$. If the defeasible graph $\mathcal{DG}$ of $\mathcal{D}_{\mathcal{AL}}$ does not have a cycle with an odd number edges, then $\mathcal{P}$ is call consistent.*

Proof. The Lemma 2.4 can be proved in a similar way of that in Lemma 2.3. □

Theorem 2.1 *Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description. If its* defeasible graph *$\mathcal{DG}$ does not have a cycle, then $\mathcal{D}_{\mathcal{AL}}$ has a unique model that can be computed in polynomial time.*

Proof. From [9], a locally stratified logic program has the unique answer set and can be computed in a polynomial time. Based on Lemma 2.3 and the definition of semantics of domain description $\mathcal{D}_{\mathcal{AL}}$, the result is approved. □

Theorem 2.2 *Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description. If its* defeasible graph *$\mathcal{DG}$ does not have a cycle with an odd number edges, then $\mathcal{D}_{\mathcal{AL}}$ has at lease one model that can be computed in polynomial time.*

Proof. From [9], a call consistent logic program has at lease one answer set and can be computed in polynomial time. Based on Lemma 2.4 and the definition of semantics of domain description $\mathcal{D}_{\mathcal{AL}}$, the result is proved. □

2.4 Two Case Studies

In this section we present two specific authorization scenarios to demonstrate the application of language $\mathcal{AL}$.

Scenario 2.1 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 must be from different departments: a member of management, an individual from auditing, and a technician from IT department.

The system trusts the manager of Human Resource to identify the staff of the company. The domain description $\mathcal{D}_{\mathcal{AL}}$ for this scenario consists of the following rules represented by using language $\mathcal{AL}$.

$$
\begin{aligned}
&\mathit{local}\ \text{grants}\ \mathit{right}(+, \mathit{recover}, \mathit{key})\ \text{to}\\
&\quad [\ \mathit{dthd}(1, X, \mathit{hrM}\ \text{asserts}\ \mathit{isAManager}(X)),\\
&\qquad \mathit{dthd}(1, Y, \mathit{hrM}\ \text{asserts}\ \mathit{isAnAuditor}(Y)),\\
&\qquad \mathit{dthd}(1, Z, \mathit{hrM}\ \text{asserts}\ \mathit{isATech}(Z))\ ].\\
&\mathit{hrM}\ \text{asserts}\ \mathit{isAManager}(\mathit{alice}).\\
&\mathit{hrM}\ \text{asserts}\ \mathit{isAnAuditor}(\mathit{bob}).\\
&\mathit{hrM}\ \text{asserts}\ \mathit{isAnAuditor}(\mathit{carol}).\\
&\mathit{hrM}\ \text{asserts}\ \mathit{isATech}(\mathit{david}).
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{auth}(\mathit{local}, l, \mathit{right}(+, \mathit{recovery}, \mathit{key}), 1).\\
&\mathit{match}(l, \mathit{right}(+, \mathit{recovery}, \mathit{key})) \leftarrow\\
&\quad \mathit{auth}(\mathit{local}, l, \mathit{right}(+, \mathit{recovery}, \mathit{key}), 1),\\
&\quad 1\{\mathit{req}(X, \mathit{right}(+, \mathit{recovery}, \mathit{key})) :\\
&\qquad\qquad \mathit{assert}(\mathit{hrM}, \mathit{isAManager}(X))\}1,\\
&\quad 1\{\mathit{req}(Y, \mathit{right}(+, \mathit{recovery}, \mathit{key})) :\\
&\qquad\qquad \mathit{assert}(\mathit{hrM}, \mathit{isAnAuditor}(Y))\}1,\\
&\quad 1\{\mathit{req}(Z, \mathit{right}(+, \mathit{recovery}, \mathit{key})) :\\
&\qquad\qquad \mathit{assert}(\mathit{hrM}, \mathit{isATech}(Z))\}1.\\
&\mathit{assert}(\mathit{hrM}, \mathit{isAManager}(\mathit{alice})).\\
&\mathit{assert}(\mathit{hrM}, \mathit{isAnAuditor}(\mathit{bob})).\\
&\mathit{assert}(\mathit{hrM}, \mathit{isAnAuditor}(\mathit{carol})).\\
&\mathit{assert}(\mathit{hrM}, \mathit{isATech}(\mathit{david})).
\end{aligned}
$$

In this scenario, the program $\mathcal{P}$ consists of the above translated rules, and those authorization rules specified in section 2.2.2. If Alice, Bob, and David make a request to recover a key together, that is,

$$[\mathit{alice}, \mathit{bob}, \mathit{david}]\ \text{requests}\ \mathit{right}(+, \mathit{recovery}, \mathit{key}).$$

After translation, we get program Π,

$$\begin{aligned}
\{\ & req(alice, right(+, recovery, key)).\\
& req(bob, right(+, recovery, key)).\\
& req(david, right(+, recovery, key)).\ \}
\end{aligned}$$

and the ground literal $\varphi(+)$ is,

$$ggrant(l, right(+, recovery, key)).$$

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 A for complete program) has only one answer set in which $ggrant(l,\ right(+, recovery, key))$ is true. We conclude $(\mathcal{P} \cup \Pi) \models ggrant(l, right(+, recovery, key))$. Therefore the request is permitted.

Now if we consider that Alice, Bob, and Carol make the same request, the rule for $match(l, right(+, recovery,\ key))$ can not be satisfied. From the authorization rules (2.6) and (2.7) in section 2.2.2, the system denies the request from Alice, Bob, and Carol. A complete ASP program representing this scenario is given in Appendix A.

Scenario 2.2 A server provides the services including *http*, *ftp*, *mysql*, and *smtp*. It sets up a group for them, called *services*. The server delegates the right of assigning the services to the security officer *so* with depth 3. The security officer *so* grants the services to *staff*. The service *mysql* can not be accessed if the staff is on holiday. Officer *so* can get information of staff from the human resource manager *hrM*. The policies and credentials are described using language $\mathcal{AL}$ as follows.

local says below$(http, services)$.
local says below$(ftp, services)$.
local says below$(mysql, services)$.
local says below$(smtp, services)$.
local delegates $right(\square, access, services)$
  with depth 3 to *so*.
so grants $right(+, access, Y)$ to X
  if *hrM* asserts *isStaff*(X),

$local\ says\ below(Y, services),\ local\ says\ neq(Y, mysql).$

so grants $right(+, access, mysql)$ to X

if hrM asserts $isStaff(X)$,

$with\ absence\ hrM$ asserts $onHoliday(X)$.

hrM asserts $isStaff(alice)$.

hrM asserts $isStaff(bob)$.

hrM asserts $onHoliday(alice)$.

For this scenario, we give the complete $\mathcal{L}_{Ans}$ program in Appendix A. Through *Smodels*, we get one and only one answer set. In the following, we list parts of the answer set:

$auth(local, alice, right(+, access, http), 2)$

$auth(so, alice, right(+, access, http), 1)$

$delegate(local, so, right(+, access, http), 3, 1)$

$grant(alice, right(+, access, http))$

$grant(alice, right(-, access, mysql))$

The predicates $auth(\ldots)$ are helpful for us to find the authorization path. We can find that the authorization passes from *local* to *so*, and then from *so* to *alice*. In many applications, such authorization path and related delegation chain play an essential role to identify the validity of requests [36, 67].

2.5 Summary

In this chapter, we developed an authorization language $\mathcal{AL}$ to specify distributed authorization with delegation. We used Answer Set Programming as a foundational basis for its semantics and computation. As we have showed, $\mathcal{AL}$ has a rich expressive power representing not only nonmonotonic policies and positive and negative authorizations, but also structured resources and privileges, partial authorization and delegation, and separation of duty policies.

As we indicated earlier, our formulation has implementation advantages because of the recent development of Answer Set Programming technology in AI community[5], where many existing approaches do not have. Both scenarios in section 2.4 have been fully implemented through Answer Set Programming.

We also investigated the computational issue related to language $\mathcal{AL}$. It is well known that deciding whether an extended logic program has an answer set is NP-complete [9], which means that in general the answer set computation is intractable. So in practic we will need an exponential algorithm to compute a program's answer sets. As the semantics of $\mathcal{AL}$ is defined based on answer set (stable model), consequently, computing an query $\mathcal{Q}_{\mathcal{AL}}$ under a given domain description $\mathcal{D}_{\mathcal{AL}}$ will be intractable. In our formulation, we dealt with this problem in two ways. One is to employ the state of the art technology of Answer Set Programming to develop optimization strategies to improve the computation process for query evaluation. We applied *lparse* to ground and simplify the logic programs [74], which is a default front-end to *Smodels*. The other way is to identify more general tractable classes of $\mathcal{AL}$ domains by applying some computational results in logic programs. We considered that when a logic program is locally stratified or call-consistent, then this program must have an answer set, and such answer set can be computed in polynomial time. By examining proper conditions, we identified two classes of $\mathcal{AL}$ domains, for which their $\mathcal{L}_{Ans}$ translation will always be locally stratified or call-consistent. In this way, any query under those types of domains can be evaluated in polynomial time.

[5]Please refer to http://www.tcs.hut.fi/Software/smodels/index.html

Chapter 3

An Access Control System for XML documents Considering Delegation

In this chapter, we present a fine-grained access control system for XML documents considering delegation based on $\mathcal{AL}$ introduced in Chapter 2. We define an XML-based language *XPolicy* to encode policies. The policy semantics is provided through the semantics of language $\mathcal{AL}$. In our system, we use Xindice − a native XML Database system to store access control policies and XML resources. It is convenient using native database system to manage XML-based information. When we retrieve an XML document from the database, we get document content, as well as its DOM tree structure directly which has been done by Xindice. Otherwise, we have to provide the process of retrieving and updating an XML document ourselves. We first introduce the related work and present preliminary knowledge including concepts in XML, the XML DOM tree and Xindice. Then, we give detailed concepts in our system which include how to specify subjects, objects and policies, and how to define policy semantics, as well as discussions of privilege, propagation options and conflict resolutions. We also provide the execution process and some implementation issues.

3.1 Related Work

XML (Extensible Markup Language), a markup language promoted by the World Wide Web Consortium (W3C), is the de facto standard language for the representation and exchange of information on the Internet. Comparing to HTML, which was defined using only a small and basic part of SGML (Standard Generalized Markup Language: ISO 8879), XML is a sophisticated subset of SGML, designed to describe data using arbitrary tags. As its name implies, extensibility is a key feature of XML, with which users or applications are free to declare and use their own tags and attributes. Moreover, XML removes the complexity of SGML. Due to its advantages, XML is now widely accepted in the Web community, and available applications exploiting this standard include OFX (open financial exchange) [31] for describing financial transactions and OSD (Open Software Distribution) [92] for software distribution on the Net.

As more and more information is made available in XML format, both in corporate Intranets and on the global Net, concerns are being raised by developers and end-users about XML security problems. The advancement of public-key cryptograph has remedied most of the security problems in communications. In the XML area commercial products are becoming available providing security features such as digital signatures and element-wise encryption to transactions involving XML data. Recently, the problem of designing a sophisticated access control mechanism for XML information was addressed and several projects [18, 37, 38, 39, 79] have been carried out for supporting authorization-based access control to XML information in the Web.

The access control model proposed in [37, 38, 39] sets access rights to elements of XML documents and DTD using DOM trees, and controls users' access to XML data according to the information of the access rights. Figure 3.1 shows access control process using the access control model. In this access control model, the access right information is specified in XAS (XML Access Sheets). If a user requests access to an XML document, the system executes a task as shown in the lower part of Figure 3.1. First, it parses the XML document to get the DOM tree. Then it sets nodes in the DOM tree with a sign of "+" (allow) or "−" (deny) based on XAS of the XML document and its DTD. Such a task of setting access rights on the nodes of a DOM tree is called labelling. From the labelled DOM tree, finally, nodes with "−" sign are removed and those with "+" sign are shown to the user in XML format. The document that the user sees is the same as a view in that it is parts of the entire document. Parts of an XML document, whose nodes have been removed from the DOM tree, may be invalid in DTD. To solve this problem, a loosening process is necessary as shown in the upper part of Figure 3.1. The process sets all elements and attributes in DTD to be optional so that existing DTD be preserved even if nodes are removed from the DOM tree. A user only can view parts of the XML document that he/she is given a right to access. This is the way how data security (confidentiality) is guaranteed.

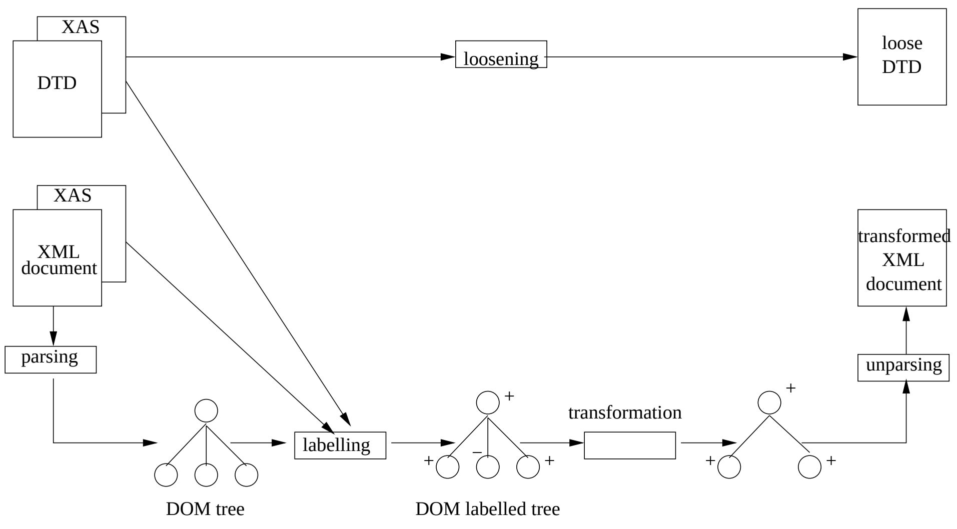

Figure 3.1: Access control process

Bertino and Ferrari [18] defined a formal access control model for XML documents, which extended the previous model proposed in Damiani *et al.* [39]. The model in [39] only provides the read access privilege. By contrast, Bertino and Ferrari [18] provided a number of specialized access modes for browsing and authoring privileges, which allow the system administrator to authorize a user to read the information in an element and/or to navigate through its links, or to modify/delete the content of an element/attribute. Moreover, the access control system in Bertino and Ferrari [18] supports a secure massive distribution of XML documents among large user communities.

Wu *et al.* [79] proposed an access control model which supports delegable authorization rules for XML documents and fine-grained access control. This model follows the basic idea in [39] and added delegation authorizations. The delegable authorization is represented as a 6-ary tuple of the form:

$$D = \langle grantee, object, authorization\_type, \\ access\_right, grantor, propagation\_type\rangle,$$

which reads that *grantee* is granted by *grantor* the *access_right* on *object* with *authorization_type* and *propagation_type*. Here,

$$\begin{aligned}
&grantor \in S,\\
&grantee \in S,\\
&object \in O,\\
&authorization\_type \in \{+, -, *\},\\
&access\_right \in \{browsing, updating\},\ and\\
&propagation\_type \in \{NO\_PROP, UP\_CASCADE, DOWN\_CASCADE\},
\end{aligned}$$

where S is a set of subjects; O is an XML document/DTD, or an element of them; *authorization_type* can be $+, -$, or $*$ to denote positive, negative and delegable administrative privilege respectively; *propagation_type* can be one of NO_PROP, $UP\_CASCADE$, and $DOWN\_CASCADE$ to denote no propagation, propagation to parent elements, and propagation to sub-elements respectively. The concept of the model is illustrated in Figure 3.2.

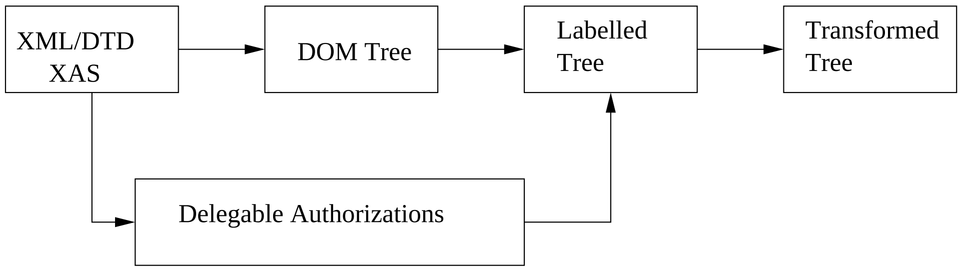

Figure 3.2: XML delegable access control process

Due to a complex structure in an ordinary file, it is difficult to restrict access to only parts of the file or to assign access rights over parts of the file. Thus access control has to be made against the entire file. However, XML has a tree-shaped hierarchical structure and an XML document is composed of several elements [37]. Each element corresponds to a node in the tree. Because each node represents a part of XML data, access control to a part of data is possible in XML by assigning access rights over elements, which is called the fine-grained access control for XML. Moreover, in distribution environments, delegable authorizations in access control system are desirable as introduced in Chapter 2. The work of Damiani *et al.*[37, 38, 39] implemented a fine-grained access control system for XML resources based on XML DOM tree, which provided both positive and negative authorizations, labelled each node in the DOM tree using "+" or "−" according to positive or negative authorization over the node, and dealt with conflicts based on conflict resolution policies in transformation phase. The work of Bertino *et al.*[18] also implemented fine-grained access control for XML supporting the formulation of high-level access control policies that have taken into account both user characteristics and document contents and structures, as well as the secure document distribution. However, the above two approaches did not have delegation features. The approach proposed by Wu *et al.* [79] encapsuled delegable authorization rules for XML documents that allow flexible data granularity. However, the subject specification in this approach only supported users and groups. Moreover, the definitions of the *General Access Rule* and the *Delegable Rule* are simple. Due to the simplicity of syntax, the previous approaches have limits in expressing authorization policies. They could not support complex subject structures and delegation with depth control. More importantly, they only have simple authorizations and do not support conditional policy rules. Let us consider the following scenario: in hospital, the patients personal information is saved as XML documents in the databases; the nurses are forbidden to read and update the patients personal information; the nurse in the department of surgery can be delegated the permission of browsing by the doctor to access the information on the patient if the doctor is in charge of the patient. The scenario includes the complex delegation and conditional policy rules that can not be specified by the previous approaches.

3.2 Preliminary

3.2.1 Basic Concepts of XML

XML [10] is a markup language for describing semi-structured information. An XML document is composed of a sequence of nested elements, each delimited either by a pair of start and end tags (e.g., ⟨*recipe*⟩ and ⟨/*recipe*⟩) or by an empty tag. An XML document is *well-formed* if it obeys the syntax of XML (e.g., nonempty tags must be properly nested; each nonempty start tag must correspond to an end tag). A well-formed document is *valid* if it conforms to a proper document type definition (DTD). A DTD is a file (external, included directly in the XML document, or both) that contains a formal definition of a particular type of XML documents.

A DTD may include declarations for elements, attributes, entities, and notations. Elements are the most important components of an XML document. Element declarations in the DTD specify the names of elements and their content. The content specification may coincide with *Empty*, *Any*, or with a group of one or more subelements/groups. *Empty* means that the element has no content, whereas *Any* means that the element may have any content. Groups can be sequences or a choice of subelements and/or subgroups. Sequences of elements are separated by a comma ",", and choices are separated by a vertical bar "|". Declarations of sequence and choices of subelements also describe the subelements cardinality; with a notation inspired by extended BNF grammars, "*" indicates zero or more occurrences, "+" indicates one or more occurrences, "?" indicates zero or one occurrence, and no label indicates exactly one occurrence. Attributes represent properties of elements. Attribute declarations specify the attributes of each element, indicating their name, type, and, possibly, default value. Attributes can be marked as *required, implied, or fixed*. Attributes marked as *required* must have an explicit value for each occurrence of the elements with which they are associated. Attributes marked as *implied* are optional. Attributes marked as *fixed* have a fixed value indicated at the time of their definition. Entities are used to include text and/or binary data in a document and can be internal or external. Internal entities are used to introduce special characters in the document or as an alias for some frequently used text. External entities are external files containing either text or binary data. Notation declarations specify how to manage the binary entities. Entities and notations are essential in the description of the physical structure of an XML document, but are not considered in our thesis, where we concentrate the analysis on the XML logical description.

Figure 3.3 illustrates an XML document example[1] describing the order information in a factory. XML documents that are valid with respect to a DTD obey the structure defined by the DTD. Intuitively, each DTD is a *schema* and XML documents that are valid according to that DTD are *instances* of that schema. Figure 3.4 is the DTD of the XML document in Figure 3.3. Note that since elements defined in a DTD may appear in an XML document zero, one or many times, according to their cardinality constraints, the structure defined by the DTD is not rigid; two distinct documents of the same DTD may differ in the number and structure of elements.

In the following, we present the formal model of XML based on [18, 37, 40]. A DTD can be represented as a labelled tree containing a node for each attribute and element in the DTD. There is an arc between elements and each element/attribute belonging to them, labelled with the cardinality of the relationship. Elements are represented as circles and attributes as squares. Let Σ be a set of DTD element and attribute identifiers and *Label* in $\{*, +, ?\}$. A DTD is formally defined as:

Definition 3.1 *(DTD). A* document type definition (DTD) *is a tuple* $t = (V_t, v_0, E_t, \phi_{E_t})$*, where:*

- $V_t = V_t^e \cup V_t^a$ *is a set of nodes where* V_t^e*s denote elements and* V_t^a*s denote attributes in the DTD. Each* $v \in V_t$ *has an associated identifier* $id_v \in \Sigma$*;*
- v_0 *is the node representing the whole DTD element (called DTD root);*
- $E_t \subseteq V_t \times V_t$ *is a set of edges, where* $e \in E_t$ *representing the element-subelement or element-attribute relationship;*
- $\phi_{E_t} : E_t \rightarrow Label$ *is the edge labelling function.*

[1]This example is extracted from the website, http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/ad/r0008719.htm

```
⟨?xml version = "1.0"?⟩
⟨Order key = "1"⟩
   ⟨Customer⟩
      ⟨Name⟩American Motors⟨/Name⟩
      ⟨Email⟩parts@am.com⟨/Email⟩
   ⟨/Customer⟩
   ⟨Part color = "black"⟩
      ⟨key⟩68⟨/key⟩
      ⟨Quantity⟩36⟨/Quantity⟩
      ⟨ExtendedPrice⟩34850.16⟨/ExtendedPrice⟩
      ⟨Tax⟩6.000000e-02⟨/Tax⟩
      ⟨Shipment⟩
         ⟨ShipDate⟩1998-08-19⟨/ShipDate⟩
         ⟨ShipMode⟩BOAT⟨/ShipMode⟩
      ⟨/Shipment⟩
      ⟨Shipment⟩
         ⟨ShipDate⟩1998-08-19⟨/ShipDate⟩
         ⟨ShipMode⟩AIR⟨/ShipMode⟩
      ⟨/Shipment⟩
   ⟨/Part⟩
   ⟨Part color = "red"⟩
      ⟨key⟩128⟨/key⟩
      ⟨Quantity⟩28⟨/Quantity⟩
      ⟨ExtendedPrice⟩38000.00⟨/ExtendedPrice⟩
      ⟨Tax⟩7.000000e-02⟨/Tax⟩
      ⟨Shipment⟩
         ⟨ShipDate⟩1998-12-30⟨/ShipDate⟩
         ⟨ShipMode⟩TRUCK⟨/ShipMode⟩
      ⟨/Shipment⟩
   ⟨/Part⟩
⟨/Order⟩
```

Figure 3.3: An Example of XML Documents

```
⟨!DOCTYPE Order [
⟨!ELEMENT Order (Customer, Part+)⟩
⟨!ATTLIST Order key CDATA#REQUIRED⟩
⟨!ELEMENT Customer (Name, Email)⟩
⟨!ELEMENT Name (#PCDATA)⟩
⟨!ELEMENT Email (#PCDATA)⟩
⟨!ELEMENT Part (key, Quantity, ExtendedPrice, Tax, Shipment+)⟩
⟨!ELEMENT key (#PCDATA)⟩
⟨!ELEMENT Quantity (#PCDATA)⟩
⟨!ELEMENT ExtendedPrice (#PCDATA)⟩
⟨!ELEMENT Tax (#PCDATA)⟩
⟨!ATTLIST Part color CDATA #REQUIRED⟩
⟨!ELEMENT Shipment (ShipDate, ShipMode)⟩
⟨!ELEMENT ShipDate (#PCDATA)⟩
⟨!ELEMENT ShipMode (#PCDATA)⟩
]⟩
```

Figure 3.4: DTD of the XML document in Figure 3.3

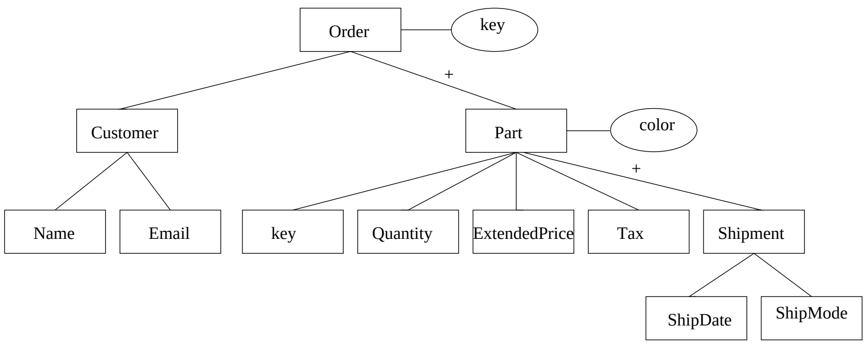

Figure 3.5: Graph representation of DTD in Figure 3.4

Figure 3.5 shows the graph representation of the DTD in Figure 3.4. Nodes representing elements are denoted by rectangles, whereas nodes representing attributes are denoted by circles.

Each XML document is modelled by a tree with nodes for each element, attribute and value in the document, and with arcs between each element and each of its subelements/attributes/values and between each attribute and each of its value(s), in which the arcs between each element/attribute and its value(s) have arrows from the elment/attribute to its value(s). Let Σ be a set of element and attribute identifiers, and $Value$ be a set of element/attribute values. An XML document can be formally defined as follows:

Definition 3.2 *(XML document). An* XML document *is a triple $d = (V_d, v_0, E_d)$, where:*

- $V_d = V_d^e \cup V_d^a \cup V_d^u$ *is a set of nodes representing elements, attributes and values respectively. Each $v \in V_d^e \cup V_d^a$ has an associated identifier in Σ, and each $v \in V_d^u$ is a value in $Value$;*
- v_0 *is the node representing the whole XML document (called document root);*
- $E_d \subseteq V_d \times V_d$ *is a set of edges, where $e \in E_d$ representing the element-subelement/attribute/value or attribute-value relationship;*

Figure 3.6 shows the graph representation of the XML document in Figure 3.3, in which nodes representing elements are denoted by rectangles, whereas nodes representing attributes are denoted by circles.

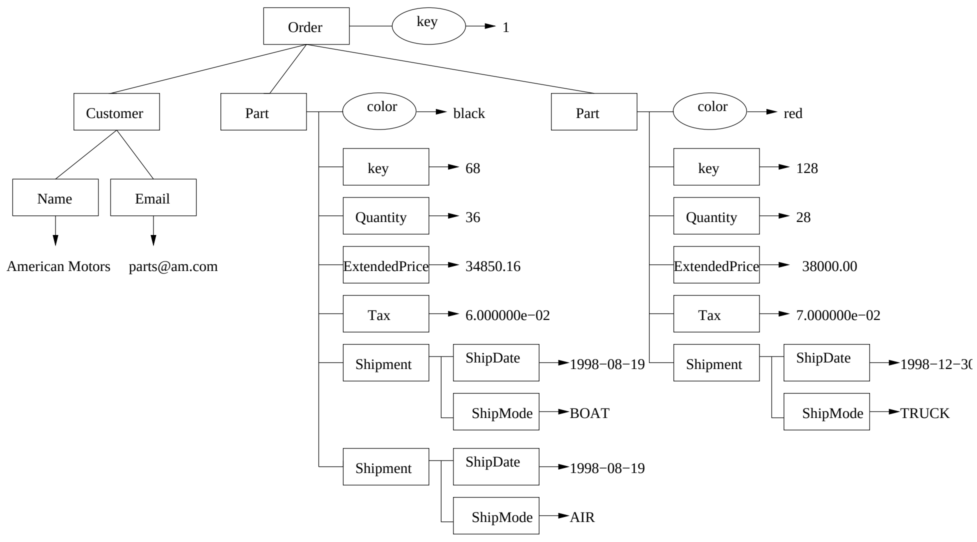

Figure 3.6: Graph representation of the XML document in Figure 3.3

3.2.2 XML DOM Tree Structure

The Document Object Model (DOM) is a recognized W3C standard for platform- and language-neutral dynamic access and update of the content, structure, and style of XML documents. It provides a standard set of objects for representing documents, a standard model of how these objects can be combined and a standard set of interfaces for accessing and manipulating them.

The basic construct of an XML document is the element and the attribute. Elements may be nested at any depth and contain other elements. An XML document can be represented by a tree structure. Nodes of the tree have different types (elements or attributes). Each element/attribute in XML is represented as a node in a DOM tree.

3.2.3 Xindice – A Native XML Database System

The Native XML Databases [26] are designed especially to store XML documents. Like other databases, they support features like transactions, security, multiple-user access, programmatic APIs, query languages, and so on. The only difference from other databases is that their internal model is based on XML and not other models, such as the relational model. The term “native XML database” first gained prominence in the marketing campaign for *Tamino*, a native XML database from Software

AG. We present its definition developed by members of the XML:DB mailing list as follows.
A native XML database:

- defines a logical model for an XML document – as opposed to the data in that document – and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order. Examples of such models are the XPath data model, the XML infoset, and the models implied by the DOM and the events in SAX 1.0.

- has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as the fundamental unit of (logical) storage.

- is not required to have any particular underlying physical storage model. For example, it can be built on a relational, hierarchical, or object-oriented databases, or use a proprietary storage format such as indexed, compressed files.

Xindice [98] is an Open Source Native XML Database System maintained by the Apache organization.

The Xindice server is designed to store collections of XML documents. Collections can be arranged in a hierarchy similar to that of a typical UNIX or Windows file system. In Xindice the data store is rooted in a database instance that can also be used as a document collection. This database instance can then contain any number of child collections. In a default install of Xindice the database instance is called "*db*" and all collection paths will begin with "*/db*". It is possible to rename the database instance if desired though it is not necessary to do so. Collections are referenced in a similar manner to how you would work with an ordinary hierarchical file system.

Xindice currently supports XPath as a query language. In many applications XPath is only applied at the document level but in Xindice XPath queries can be executed at either the document level or the collection level. This means that a query can be run against multiple documents and the result set will contain all matching nodes from all documents in the collection. The Xindice server also supports the creation of indexes on XML documents to speed up commonly used XPath queries.

We present Xindice's feature summary as follows:

- Document Collections: Documents are stored in collections that can be queried as a whole. We can create collections that contain just documents of the same type or we can create a collection to store all our documents together.

- XPath Query Engine: To query the Document Collections we use XPath as defined by the W3C. This provides a reasonably flexible mechanism for querying documents by navigating and restricting the result tree that is returned.

- XML Indexing: In order to improve the performance of queries over large numbers of documents we can define indexes on element and attribute values. This can dramatically speed up query response time.

- XML:DB XUpdate Implementation: When we store XML in the database we are able to change that data without retrieving the entire document. XUpdate is the mechanism to do server side updates of the data. It is an XML based language for specifying XML modifications and allows those modifications to be applied to entire document collections as well as single documents.

- Java XML:DB API Implementation: For Java programmers Xindice provides an implementation of the XML:DB API. This API is intended to bring portability to XML database applications just as JDBC has done for relational databases. Most applications developed for Xindice will use the XML:DB API.

- Command Line Management Tools: To aid the administrator, Xindice provides a full suite of command line driven management tools. Just about everything you can do through the XML:DB API can also be done from the command line.

- Modular Architecture: The Xindice server is constructed in a very modular manner. This makes it easy to add and remove components to tailor the server to a particular environment or to embed it into another application.

3.3 Concepts and Process in Our Access Control System

XML is widely used in Web communities. Access control policies for XML resources must cope with a dynamic subject population and support a wide spectrum of protection granularity, ranging from a set of documents to specific elements within a document.

In our system, access control policies include local authorization rules and credentials issued by third parties which will be specified by the language $\mathcal{AL}$. Language $\mathcal{AL}$ supports both *single subject requests* and *group subject requests*. Because the *group subject request* is only suitable for some specific situations which need special hardware architectures and software systems, for the simplicity and practicality of the system implementation, we only consider *single subject requests* in our system. Consequently, we need authoring to a single subject, instead of a complex subject structure. Although we do not authorize a privilege to a group of subjects, we still support delegating a privilege to a group of subjects , from which if all the subjects in the group authorize the privilege to the same subject, we can conclude the subject holds the privilege.

Based on the above restrictions, we obtain a fragment of language $\mathcal{AL}$, denoted as $\mathcal{AL}^*$. The complete syntax of $\mathcal{AL}^*$ is presented in Appendix B. Our system defines an XML-based language *XPolicy*, the XML format of language $\mathcal{AL}^*$ to specify access control policies. Actually, *XPolicy* is a DTD. We can encode the policy using an XML document valid with respect to *XPolicy*. In this section, we illustrate how to specify subjects, objects, and policies using *XPolicy*. We also discuss privilege, propagation option and conflict resolution issues in our system and give the definition of policy semantics. Finally, we present the execution process of our system.

3.3.1 The XML-based Policy Language *XPolicy*

We use a *policy base* to specify a set of authorization policies including local authorization policies and credentials from third parties, which is equivalent to a *domain description* of $\mathcal{AL}^*$ but in XML format. Figure 3.7 shows *XPolicy*, a DTD which is the XML format of language $\mathcal{AL}^*$ with the same expressive power as $\mathcal{AL}^*$. We write a policy base using an XML document according to *XPolicy*.

The root element in an XML policy file is *policybase* which includes one or more than one subelements, *rule*s. A rule consists of a head statement, positive statements and negative statements which have the following form in language $\mathcal{AL}^*$:

$$h\ if\ a_1, a_2, \ldots, a_m\ with\ absence\ b_1, b_2, \ldots, b_n.$$

where if $n = 0$ and $m = 0$, the rule is a fact. A rule has a head statement and has no body statement or one or more than one body statements. In the XML policy file, the element *rule* has subelements, *head*, *posbody* and *negbody*. The number of *posbody* and *negbody* can be zero, one or more than one. We also present the element and related attributes definitions for statements, subject structures, and object in language *XPolicy* which are described in detail in the following sections.

3.3.2 Subject Specification

In our system, subjects can be authorizers or requesters. Public keys are viewed as subjects to be authorized and the authorization can be delegated to third parties. Our system supports four subject structures: *subject*, *subject set*, *static threshold*, and *dynamic threshold*. We demonstrate them using the following examples:

Example 3.1 subject structures:
subject: $a, b, c, d, e, \ldots$
subject set: $[a, b, c, d, e]$
static threshold: $sthd(2, [a, b, c, d, e])$
dynamic threshold: $dthd(2, X, a\ asserts\ isACashier(X))$ □

In previous approaches [18, 39, 79], their systems described subjects using fixed hierarchy structures. However, our system not only has the complex subject structures, but also can specify subjects and relationships among them more flexibly using *assert statements*. For instance, *assert statements* can specify that a subject is a member of a group, has certain features, or has a relationship with other subjects.

Generally, *assert statements* are used as conditions to restrict authorizations as showed in the following example.

```
⟨!DOCTYPE policybase [
⟨!ELEMENT policybase (rule+) ⟩
⟨!ELEMENT rule (head, posbody∗, negbody∗)⟩
⟨!ELEMENT head (stmt-type, (rel-stmt | assert-stmt | del-stmt-head | auth-stmt))⟩
⟨!ELEMENT posbody (stmt-type,(rel-stmt | assert-stmt | del-stmt-body | auth-stmt))⟩
⟨!ELEMENT negbody (stmt-type,(rel-stmt | assert-stmt | del-stmt-body | auth-stmt))⟩
⟨!ELEMENT stmt-type (#PCDATA)⟩
⟨!ELEMENT rel-stmt (issuer, relTitle, arg+)⟩
⟨!ELEMENT assert-stmt (issuer, assertTitle, arg+)⟩
⟨!ELEMENT del-stmt-head (issuer, priv, obj, step, (sub | sub-set | sthd | dthd))⟩
⟨!ELEMENT del-stmt-body (issuer, priv, obj, step, sub⟩
⟨!ELEMENT auth-stmt (issuer, sign, priv, obj, sub⟩
⟨!ELEMENT issuer (#PCDATA)⟩
⟨!ELEMENT relTitle (#PCDATA)⟩
⟨!ELEMENT arg (argTitle, arg-type)⟩
⟨!ELEMENT argTitle (#PCDATA) ⟩
⟨!ELEMENT arg-type (#PCDATA) ⟩
⟨!ELEMENT assertTitle (#PCDATA)⟩
⟨!ELEMENT priv (#PCDATA)⟩
⟨!ELEMENT obj (#PCDATA)⟩
⟨!ELEMENT step (#PCDATA)⟩
⟨!ELEMENT sign (+ | − | □)⟩
⟨!ELEMENT sub (#PCDATA)⟩
⟨!ELEMENT sub-set (sub+)⟩
⟨!ELEMENT sthd (k, sub-set)⟩
⟨!ELEMENT k (#PCDATA)⟩
⟨!ELEMENT dthd (k, sub, assert-stmt)⟩
⟨!ATTLIST rule
          hnum CDATA#REQUIRED
          posbnum CDATA#REQUIRED
          negbnum CDATA#REQUIRED⟩
⟨!ATTLIST relTitle argNum CDATA#REQUIRED)⟩
⟨!ATTLIST assertTitle argNum CDATA#REQUIRED)⟩
⟨!ATTLIST posbody id CDATA#REQUIRED⟩
⟨!ATTLIST negbody id CDATA#REQUIRED⟩
⟨!ATTLIST arg id CDATA#REQUIRED⟩
⟨!ATTLIST subset subNum CDATA#REQUIRED⟩
⟨!ATTLIST sub id CDATA⟩
⟨!ATTLIST dthd id CDATA#REQUIRED⟩
]⟩
```

Figure 3.7: XPolicy: DTD of a policy base

Example 3.2 A doctor can read a patient's document if he is in charge of the patient in hospital hM.

In this authorization rule, we can use the assert statement, *The hospital hM asserts doctor A is in charge of the patient B*, as a condition (as a body statement in the rule). □

In *XPolicy*, the assert statement is defined as follows:

$\langle$ *!ELEMENT assert-stmt(issuer, assertTitle, arg+)* $\rangle$

$\langle$ *!ELEMENT issuer(#PCDATA)* $\rangle$

$\langle$ *!ELEMENT assertTitle(#PCDATA)* $\rangle$

$\langle$ *!ELEMENT arg(argTitle, arg-type)* $\rangle$

$\langle$ *!ELEMENT argTitle(#PCDATA)* $\rangle$

$\langle$ *!ELEMENT arg-type(#PCDATA)* $\rangle$

$\langle$ *!ATTLIST assertTitle argNum CDATA#REQUIRED* $\rangle$

$\langle$ *!ATTLIST arg id CDATA#REQUIRED* $\rangle$

In the XML policy file, we provide the argument type information *arg-type* which can be *subject* or *object* for assert statements for the simplicity and efficiency of the system implementation.

We use the above definition to encode the assert statement in Example 3.2:

```
⟨assert-stmt⟩
  ⟨issuer⟩hA⟨/issuer⟩
  ⟨assertTitle argNum = 2 ⟩DoctorOf⟨/assertTitle⟩
  ⟨arg id = 1⟩
    ⟨argTitle⟩A⟨/argTitle⟩
    ⟨arg-type⟩subject⟨/arg-type⟩
  ⟨/arg⟩
  ⟨arg id = 2⟩
    ⟨argTitle⟩B⟨/argTitle⟩
    ⟨arg-type⟩subject⟨/arg-type⟩
```

$\langle /arg\rangle$
$\langle /assert\text{-}stmt\rangle$

The definitions for subject structures in *XPolicy* are presented as follows.

$\langle !ELEMENT\ sub(\#PCDATA)\rangle$
$\langle !ELEMENT\ sub\text{-}set(sub+)\rangle$
$\langle !ELEMENT\ sthd(k,\ sub\text{-}set)\rangle$
$\langle !ELEMENT\ dthd(k,\ sub,\ assert\text{-}stmt)\rangle$
$\langle !ELEMENT\ k(\#PCDATA)\rangle$
$\langle !ATTLIST\ subset\ subNum\ CDATA\#REQUIRED\rangle$
$\langle !ATTLIST\ sub\ id\ CDATA\rangle$
$\langle !ATTLIST\ dthd\ id\ CDATA\#REQUIRED\rangle$

We encode *subject*, *subject set*, *static threshold*, and *dynamic threshold* examples in Example 3.1 using *XPolicy* respectively as follows.

- subjects
  $\langle sub\rangle a\langle /sub\rangle$
  $\langle sub\rangle b\langle /sub\rangle$
  $\langle sub\rangle c\langle /sub\rangle$
  $\langle sub\rangle d\langle /sub\rangle$
  $\langle sub\rangle e\langle /sub\rangle$

- subject set
  $\langle sub\text{-}set\ subNum = 5\rangle$
    $\langle sub\ id = 1\rangle a\langle /sub\rangle$
    $\langle sub\ id = 2\rangle b\langle /sub\rangle$
    $\langle sub\ id = 3\rangle c\langle /sub\rangle$
    $\langle sub\ id = 4\rangle d\langle /sub\rangle$
    $\langle sub\ id = 5\rangle e\langle /sub\rangle$
  $\langle /sub\text{-}set\rangle$

- static threshold
  $\langle sthd\rangle$

```
   ⟨k⟩2⟨/k⟩
   ⟨sub-set subNum = 5⟩
      ⟨sub id = 1⟩a⟨/sub⟩
      ⟨sub id = 2⟩b⟨/sub⟩
      ⟨sub id = 3⟩c⟨/sub⟩
      ⟨sub id = 4⟩d⟨/sub⟩
      ⟨sub id = 5⟩e⟨/sub⟩
   ⟨/sub-set⟩
⟨/sthd⟩
```

- dynamic threshold

```
⟨dthd id = 0⟩
   ⟨k⟩2⟨/k⟩
   ⟨sub⟩X⟨/sub⟩
   ⟨assert-stmt⟩
      ⟨issuer⟩a⟨/issuer⟩
      ⟨assertTitle argNum = 1⟩isACashier⟨/assertTitle⟩
      ⟨arg id = 1⟩
         ⟨argTitle⟩X⟨/argTitle⟩
         ⟨arg-type⟩subject⟨/arg-type⟩
      ⟨/arg⟩
   ⟨/assert-stmt⟩
⟨/dthd⟩
```

3.3.3 Object Specification

In our system, access control policies can be specified for a whole DTD or an XML document. Moreover, policies specified at the DTD level can be applied to XML documents valid to the DTD.

In the following, we use the term *object* to denote the DTD/documents or portions of those documents to which a policy applies. The object specification includes three parts: (1) a document/DTD; (2) the element(s) within the document/DTD; (3) the attribute(s) within the specified element(s). The first part is required by all object specifications. The last two parts are optional. Elements can be denoted by a set of their identifiers, or by a path from the document/DTD root element to the nodes representing the element(s). Based on Definitions 3.1 and 3.2, the object specification is defined as follows:

Definition 3.3 *(Object Specification). Let $t = (V_t, v_0, E_t, \phi_{E_t})$ be a DTD and $d = (V_d, v_0, E_d)$ be an instance of t. An* object specification *is of the form:*

doc-spec.Root-spec[.element-spec][.attrs]

where:

- *doc-spec is a document name. There is a special doc-spec, 0 which means this object specification is for the DTD, instead of an XML document.*
- *Root-spec is a root identifier, id_{v_0} with v_0 in t or d.*
- *element-spec is an element specification for the DTDs or documents which has the following two forms:*
  - *a set of element identifiers in t or d, that is* element-spec=$(id_1, \ldots, id_n)$, *with $id_i \in \{id_v \mid v \in V_t^e \backslash \{v_0\}$ or $v \in V_d^e \backslash \{v_0\}\}$.*
  - *a path expression, that is,* element-spec = path_expr, *where* path_expr *is specified based on the following grammar:*

    $$\begin{aligned} \text{path\_expr} ::= * \mid\ & tag\ [\ (\ [text\ =\ n_0\ [\ ,attr_1\ =\ n_1, \ldots\ ]\ ]\ )\ ] \\ \mid\ & path\_expr.path\_expr \end{aligned}$$

    *where * denotes all elements, tag is the identifier of an element in t or d, "text $=$ n_0" denotes the value of the element is n_0, and "attr$_1$ $=$ n_1" denotes that the value of $attr_1$ is n_1.*
- *attrs is a set of attributes of the elements in t or d.*

Example 3.3 Consider the DTD in Figure 3.4 and the XML document in Figure 3.3. Suppose the XML document' name is *aOrder*. The object specification examples are presented as follows:

- 0.Order.Part.key: it denotes the key numbers of Parts in all instances of the DTD *Order*.
- 0.Order.Part(color = red).Shipment: it denotes the *Shipment* elements for all *Part*s that have the color *red*.
- aOrder.Order.{Quantity, ExtendedPrice, ShipMode, ShipDate}: it denotes a set of objects including quantity, price, and shipment information in the XML document *aOrder*.

□

In language *XPolicy*, the object is of the form:

$\langle !ELEMENT\ obj(\#PCDATA)\rangle$

The object specifications in Example 3.3 can be encoded as:

- $\langle obj\rangle 0.Order.Part.key\langle /obj\rangle$
- $\langle obj\rangle 0.Order.Part(color = red).Shipment\langle /obj\rangle$
- $\langle obj\rangle aOrder.Order.\{Quantity, ExtendedPrice, ShipMode, ShipDate\}\langle /obj\rangle$

3.3.4 Discussions on Privileges, Propagation and Conflict Resolution

Privileges involving write operations on XML documents

Currently, our system has been implemented only to support read privilege, since in practice, most XML applications are read-only. In fact, we can extend our system to support write privileges including insert, delete and update as follows.

In our system, we label a DOM tree of an XML document with respect to authorizations and make the access control decision based on the pruned DOM tree. Insert privileges are evaluated by executing the labelling process on the document with the new node inserted. If the labelling process produces a positive label on the new node, the insert operation completes successfully, otherwise this operation is denied. The delete operation of a node is allowed only if the labelling of the document produces a final positive label for the node to be deleted. Update operations are evaluated by executing the labelling process on both the existing document and on the new updated document. If the final label associated with the node being updated is positive in both versions, the update operation is permitted; otherwise it is rejected. When the XML document is modified, the system must also check the correctness of the document with respect to the DTD and if the document is not valid, the write operations must not be allowed. Consider write requests refer to a set of nodes. A transactional mechanism can be introduced based on the "deferral" of controls, analogous to the SQL command *set constraints deferred* that relational systems offer for the management of constraints. The mechanism is that write operations are collected in an atomic sequence, and all the checks on the correctness of the updates are deferred at the end of the sequence. Each single write operation is checked for permissions and correctness. If a single one is not allowed, the sequence is invalid and the XML data are rolled back to their original state.

Propagation options

Our system supports authorizations at all levels of granularity, from the DTD to single elements/attributes within individual documents. Propagation options state how policies specified at a given level of a DTD/document hierarchy propagate to other levels. In our system, we add special propagation rules in policy base to choose different propagation options. There are two options: one is no propagation of the policy; the other is that the authorizations are propagated to all the direct and indirect subelements of the elements specified in the policy specification. For the second option, the authorizations specified on a DTD are propagated to all XML documents that are instances of the DTD; the authorizations specified on an element in an XML document are propagated to its attributes and subelements (including their attributes), which makes authorizations more simple and flexible.

Conflict resolution

Since we support both positive and negative authorizations, conflict authorization may arise when a subject is granted two contradicted authorizations over the same object. A proper conflict resolution policy is needed to handle this problem. In our system, we adopt the conflict resolution policy described in Chapter 2 that is a *trust-take-precedence* policy. As discussed earlier, the authorization must be first delegated from the local administrator *local* and therefore the authorization granted from *local* can never be overridden by those of other grantors. We can say that *local* has the highest priority, while the subjects who directly receive the delegable authorizations from *local* have the second highest priorities, and so on. The priorities of grantors decrease along the delegation path. If the authorizations with the same priorities have conflicts, we use the *negative-take-precedence* policy to solve it.

3.3.5 Policy Specification and Semantics

We specify authorization policies as a policy base which is an XML document valid with respect to *XPolicy* presented in Section 3.3.1. Based on components introduced previously, we use an example to demonstrate the specification of a policy base using *XPolicy*.

Example 3.4 A factory *fA* stores the order information for each customer using an XML document with respect to the DTD presented in Figure 3.4. For the order information, *fA* has the following local authorization policies and credentials from third parties:

1. *fA* permits requests to access the customer information in all the orders from all people except its competitors.
2. *fA* prohibits its competitors to read any order information.
3. *fA* delegates the read privilege over orders to customers who made the order with delegation depth 2.
4. *fA* asserts that a is a customer.
5. *fA* asserts that d is a competitor.
6. a grants the read privilege over its *Part* information to b.
7. a grants the read privilege over its orders to d.

$\Box$

Using language $\mathcal{AL}^*$, we encode the local policies and credentials into a domain description $\mathcal{D}_{\mathcal{AL}^*}$, in which each rule specifies the corresponding policy. For instance, r_1 is for policy 1, and so on. In the object specifications, there may be variables, sets of elements, and/or attribute restricted elements that are not supported by the semantics of language $\mathcal{AL}^*$. In the next section, we will provide the object specification process in detail. Here we only focus on the syntax expression.

r_1: *local* grants $right(+, read,$ "0.*Order.Customer*") to X if with absence *local* asserts isACompetitor(X).

r_2: *local* grants $right(-, read,$ "0.*Order*") to X if *local* asserts isACompetitor(X).

r_3: *local* delegates $right(\Box, read,$ "*X.Order.Part*")
with depth 2 to "*X.Order.Customer*".

r_4: *local* asserts $isACustomer(a)$.

r_5: *local* asserts $isACompetitor(d)$.

r_6: a grants $right(+, read,$ "*aOrder.Order.Part*") to b.

r_7: a grants $right(+, read,$ "*aOrder.Order*") to d.

From the definition of *XPolicy*, we obtain the policy base $\mathcal{PB}$, an XML document valid with respect to *XPolicy* as follows. Here we use r_1 and r_7 to illustrate the structure of the policy base. The complete XML document for Example 3.4 is presented in Appendix C.

```
⟨policybase⟩
  ⟨rule hnum = 1 posbnum = 0 negbnum = 1⟩
    ⟨head⟩
      ⟨stmt-type⟩auth-stmt⟨/stmt-type⟩
      ⟨auth-stmt⟩
        ⟨issuer⟩local⟨/issuer⟩
        ⟨sign⟩p⟨/sign⟩
        ⟨priv⟩read⟨/priv⟩
        ⟨obj⟩0.Order.Customer⟨/obj⟩
        ⟨sub⟩X⟨/sub⟩
      ⟨/auth-stmt⟩
    ⟨/head⟩
    ⟨negbody id = 1⟩
      ⟨stmt-type⟩assert-stmt⟨/stmt-type⟩
      ⟨assert-stmt⟩
        ⟨issuer⟩local⟨/issuer⟩
        ⟨assertTitle argNum = 1⟩isACompetitor⟨/assertTitle⟩
        ⟨arg id = 1⟩
          ⟨argTitle⟩X⟨/argTitle⟩
          ⟨arg-type⟩subject⟨/arg-type⟩
        ⟨/arg⟩
      ⟨/assert-stmt⟩
    ⟨/negbody⟩
  ⟨/rule⟩
    ⋮
  ⟨rule hnum = 1 posbnum = 0 negbnum = 0⟩
    ⟨head⟩
      ⟨stmt-type⟩auth-stmt⟨/stmt-type⟩
      ⟨auth-stmt⟩
        ⟨issuer⟩a⟨/issuer⟩
        ⟨sign⟩p⟨/sign⟩
        ⟨priv⟩read⟨/priv⟩
        ⟨obj⟩aOrder⟨/obj⟩
        ⟨sub⟩d⟨/sub⟩
      ⟨/auth-stmt⟩
    ⟨/head⟩
  ⟨/rule⟩
⟨/policybase⟩
```

In our system, we define *XPolicy* as the language for specifying access control policies and provide its semantics through language $\mathcal{AL}^*$. We write a policy base $\mathcal{PB}$, which is an XML document according to *XPolicy* for specifying a set of DTD-level and doc-level authorization policies over the XML documents. When a subject requests to access an XML document, our system returns a view that depends on authorization policies.[2]

The semantics of language *XPolicy* is defined by three steps: (1) transform a policy base in *XPolicy* into a logic program through the domain description format in language $\mathcal{AL}^*$; (2) compute the answer sets of the logic program and generate authorization results; (3) present the requester's view based on the authorizations.

To provide the requester's view, we first parse the XML document and obtain the DOM Tree of the XML document. At the same time, we get the parent and child relationships between the elements in the document, $parent(n_1, n_2)$, where n_1 and n_2 are elements of the document which are specified according to Definition 3.3.

In *XPolicy*, we have the object specifications that includes variables, a set of elements and attribute restricted elements. Before transforming the policy base into a domain description, we have to process those situations in the object specifications which are described in detail in Section 3.3.6.

After processing the object specifications, we obtain a domain description $\mathcal{D}_{\mathcal{AL}^*}$ from a policy base $\mathcal{PB}$. Similar to the process in Chapter 2, we use function $TransRules(\mathcal{D}_{\mathcal{AL}^*})$ to get a logic program and then add propagation option and conflict resolution rules into this program, which is finally a logic program $\mathcal{P}$. Through $SModels$, we compute the answer sets of program $\mathcal{P}$, $Ans(\mathcal{P})$, from which we extract a set of authorizations, $\{(s, sn, p, n)\}$ based on predicate $grant(s, right(sn, p, n))$, where s is a subject, sn is "+" or "-" to denote positive or negative authorizations, p is the privilege, and n is a node of the document specified using the format of object specifications. We define the process using the function, $FindAuth(Ans(\mathcal{P}))$, which checks the set of answer sets of logic program $\mathcal{P}$ and returns a set of 4-ary

[2]We do not mention the query to an element in an XML document. After we have computed the view of a requester over an XML document, the query to one of its elements can be responded with the element and its subelements in this view or will be denied if the element is not in this view.

tuples $\{(s, sn, p, n)\}$. The view of requesters is generated through labelling the XML document's DOM tree using the authorizations. This view is formed by nodes in the DOM tree that are permitted for the requesters to access. In our system, the query is that a subject s requests a privilege p over an XML document o, denoted as $\mathcal{Q} = (s, p, o)$. The definition for the semantics of *XPolicy* is as follows.

Definition 3.4 *Given a policy base $\mathcal{PB}$ of language XPolicy, and a query $\mathcal{Q} = (s, p, o)$. Let $\mathcal{D}_{\mathcal{AL}^*}(\mathcal{PB})$ be the equivalent domain description in language $\mathcal{AL}^*$, $\mathcal{P} = TransRules(\mathcal{D}_{\mathcal{AL}^*}(\mathcal{PB}))$, and $AUTH = FindAuth(Ans(\mathcal{P}))$. We define the result of a query $\mathcal{Q}$ to $\mathcal{PB}$ to be $Result(\mathcal{PB}, \mathcal{Q}) = \{\ n \mid (s, +, p, n) \in AUTH\ \&\ parent(o, n)\ \}$.*

3.3.6 Execution Process

In this section, we present the mechanism of the system execution. The valid XML documents and their corresponding policy base which consists of a set of authorization rules setting for the XML documents, are stored in Xindice, a native XML database. A user requests a whole XML document and the system responses with a valid XML document only including the information the user is permitted to access according to the policy base in the requested document.

Our system computes the requester's view over a valid XML document online. Its execution phases, showed in Figure 3.8, mainly include five steps: (1) Parsing. Process the XML documents into a DOM tree and generate the tag and parent-children relationship library for the elements in the requested XML document. (2) Policy reasoning. Transform the policy base into a logic program, compute its answer sets by *SModels* and generate the authorization results. (3) Labelling. Mark the elements using the authorization results. (4) Pruning. Remove the nodes with negative authorizations in the DOM tree. (5)Unparsing. Restore the pruned DOM tree into an XML document. We present the steps in detail as follows.

1. Parsing

   In this step, we transform the XML documents into the DOM tree structure and obtain tags of the elements and the parent-children information between elements in them. The XML resources are stored in Xindice database in our

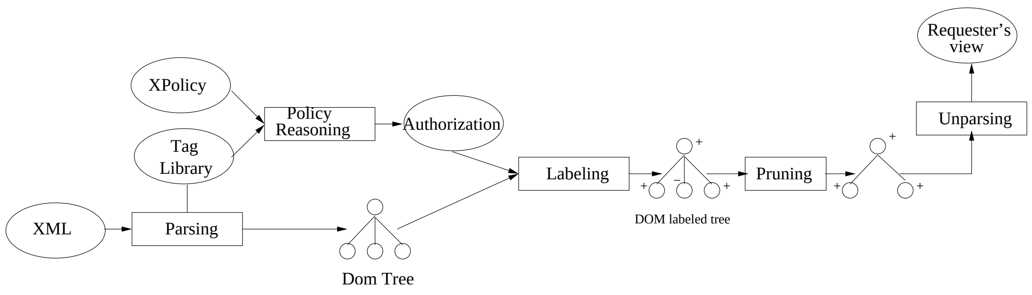

Figure 3.8: System execution process

system. XML:DB API of Xindice provides *XMLResource* class to access to the underlying XML data as either text or a DOM Node. Then we can call $getContentAsDom()$ to get the XML document or the element as a DOM Node. After that, we go through the DOM tree, tag the Node and its subNodes and generate a library for tags and parent-children relationships between them.

In the generated library we store tags and parent-children relationships. Each element/attribute in the XML document has a tag denoted as (ne, tok), where *ne* is a node expression following the definition of object specification in Definition 3.3 and *tok* is a token for the element/attribute with the form *doc-spec.seqNo* in which *doc-spec* is the XML document's name and *seqNo* is an increasing number to denote the sequence we access the nodes in the DOM tree. For instance, we present the tags of the elements from the XML document illustrated in Figure 3.3.

- The tag of *Order*: (*aOrder.Order*, *aOrder*.0)
- The tag of *Customer*: $(aOrder.Order.Customer, aOrder.1)$
- The tag of *Part* with black color:
  (*aOrder.Order.Part*(*color* = *black*), *aOrder*.2).

The parent-children relationships are a set of binary relations with the form, $parent(tok_1, tok_2)$ to denote that tok_1 is the parent node of tok_2, in which tok_1 and tok_2 are tokens. Note that in authorization policies, the text or attribute value restrictions for an element are optional. However, in a tag, we will add the text value and all existed attributes' values for an element. An object specification in the tag determines one and only one element in the

XML document, equivalent to a node in its DOM tree. However, an object specification in authorization policies may define more than one elements of the XML document.

2. Policy reasoning.

   In this step, we first transform a policy base into a domain description, then transform the domain description into a logic program, and finally compute the answer sets of the logic program using $SModels$. The authorizations are extracted from the answer sets. In this section, we use $\mathcal{PB}$ to denote a policy base and $\mathcal{D}_{\mathcal{AL}^*}$ to denote the corresponding domain description.

   (a) Transforming a policy base $\mathcal{PB}$ into a $\mathcal{D}_{\mathcal{AL}^*}$

   We divide the rules in $\mathcal{PB}$ into two sets: $\mathcal{PB}^{O^+}$ and $\mathcal{PB}^{O^-}$. The rules that have object specifications fall in $\mathcal{PB}^{O^+}$. Other rules are included in $\mathcal{PB}^{O^-}$. We process them separately. The rules in $\mathcal{PB}$ and $\mathcal{D}_{\mathcal{AL}^*}$ are denoted as $r_{\mathcal{PB}}$ and $r_{\mathcal{AL}^*}$ respectively.

   - For rules in $\mathcal{PB}^{O^-}$:
     Each rule $r_{\mathcal{PB}}$ in $\mathcal{PB}^{O^-}$ is equivalent with a rule $r_{\mathcal{AL}^*}$ and can be transformed into $r_{\mathcal{AL}^*}$ directly. Then we define a transforming function, $tran\_pb(\mathcal{PB}^{O^-}) = {\mathcal{D}_{\mathcal{AL}^*}}^{O^-}$.
   - For rules in $\mathcal{PB}^{O^+}$:
     The object specification is of the form:

     doc-spec.Root-spec[.element-spec][.attrs].

     In an object specification, there may be variables and the *element-spec* may be a set of elements. A rule $r_{\mathcal{PB}}$ in $\mathcal{PB}^{O^+}$ is called a *ground policy rule* if the object specifications in it do not include variables and a set of elements. The non-ground policy rules in $\mathcal{PB}^{O^+}$ need to be instantiated first.
     In the object specification, we restrict the type of variables only to *doc-spec*. Then we ground the variables in it using the values of *doc-spec* of the DTD. In a rule $r_{\mathcal{PB}}$, if the *element-spec* in an object

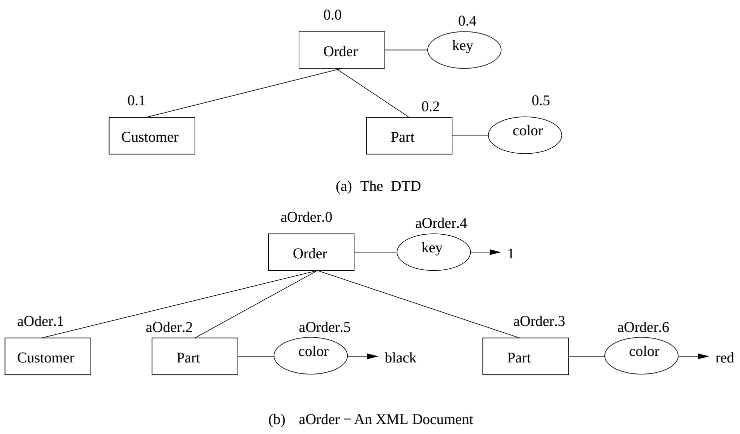

Figure 3.9: An example for tags

specification is a set of elements, for each element in the set, we add a rule in which the *element-spec* is replaced by the element.

After the instantiation process, we may get a set of rules for each non-ground rule. So far, all rules in $\mathcal{PB}^{O^+}$ are grounded including DTD-level and doc-level object specifications. According to tags in the tag library, we replace the doc-level object specifications with corresponding *tok*s. For the DTD level object specifications, we generate tags in which the *doc-spec*s are 0 and *seqNo*s for them are obtained from corresponding element nodes in the doc-level object specifications. At the same time, we add into the tag library a binary relation $instance(a, b)$ to denote that the element a in the XML document is an instance of element b in its DTD.

After the transformation of rules in $\mathcal{PB}^{O^+}$ and $\mathcal{PB}^{O^-}$, we obtain $tranToD(\mathcal{PB}^{O^+})$ and $tranToD(\mathcal{PB}^{O^-})$ respectively. Then we have $\mathcal{D}_{\mathcal{AL}^*} = tranToD(\mathcal{PB}^{O^+}) \cup tranToD(\mathcal{PB}^{O^-})$.

We use the following example to illustrate operations in the tag library.

Example 3.5 The Figure 3.9 (a) shows the simplified DTD in Figure 3.4; The Figure 3.9 (b) shows the XML document in Figure 3.3.

The tag of the root element in (b) is $(aOrder.Order, aOrder.0)$. Then the tag of the root element in (a)is $(0.Order, 0.0)$.

Note that for each element in a DTD, there may be a few elements in an instance of the DTD that are corresponding to it. For instance, the element *Part*: in the XML document, there are two *Part* elements, $(aOrder.Order.Part(color = black), aOrder.2)$ and $(aOrder.Order.Part(color = red), aOrder.3)$, in which one has the black color and the other has the red color. The *Part* element in DTD uses the *seqNo* of the first one and therefore we generate the tag $(0.Order.Part, 0.2)$ for it. We add the tags into the library and replace the DTD object specifications in the rules using the new generated *tok*s. At the same time, we generate the binary relation $instance(a, b)$ where a and b are *tok*s from the DTD and XML document to denote that element a in XML document is the instance of element b in DTD. In this example, we can obtain $instance(aOrder.0, 0.0)$, $instance(aOrder.2, 0.2)$, $instance(aOrder.3, 0.2)$, and so on. □

(b) Transforming a $\mathcal{D}_{\mathcal{AC}^*}$ into a logic program $\mathcal{P}$

A domain description $\mathcal{D}_{\mathcal{AC}^*}$ is the subset of $\mathcal{D}_{\mathcal{AC}}$. In Chapter 2, we use a predicate $below(\ldots)$ to denote the relationships between objects. In our access control system for XML resources, we use $parent(a, b)$ and $instance(a, b)$ to denote the object relationships instead of *below*. Correspondingly, we record the *parent* and *instance* information in the tag library. We use them to encode the propagation option rules and replace the corresponding propagation rules in Chapter 2.

In our system, we have two propagation options: one is *NO_PROP* and the other is *DOWN_PROP*. For the *NO_PROP*, we do not need to add propagation rules. For the *DOWN_PROP*, the authorizations are propagated from DTD level to its instances and from an element to its direct and indirect subelements and attributes. We have the following rules for the *DOWN_PROP* option.

$$parent(A, C) \leftarrow parent(A, B),\ parent(B, C).$$

This rule creates the parent relationships between indirect parent and child nodes. Then the following four rules are for the authorization and delegation propagation. They denote that the authorization/delegation is propagated from DTD-level to doc-level and from parent nodes to child nodes respectively.

$$\begin{aligned}
&auth(S_1, S_2, right(Sn, P, O_1), Step) \leftarrow \\
&\qquad auth(S_1, S_2, right(Sn, P, O_2), Step),\ instance(O_1, O_2). \\
&auth(S_1, S_2, right(Sn, P, O_1), Step) \leftarrow \\
&\qquad auth(S_1, S_2, right(Sn, P, O_2), Step),\ parent(O_2, O_1). \\
&delegate(S_1, S_2, right(\Box, P, O_1), Dep, Step) \leftarrow \\
&\qquad delegate(S_1, S_2, right(\Box, P, O_2), Dep, Step),\ instance(O_1, O_2). \\
&delegate(S_1, S_2, right(\Box, P, O_1), Dep, Step) \leftarrow \\
&\qquad delegate(S_1, S_2, right(\Box, P, O_2), Dep, Step),\ parent(O_2, O_1).
\end{aligned}$$

In our system, the conflict resolution policy following Chapter 2 takes trust precedent policy. If conflicts happen in the authorizations with the same priorities, our system takes negative authorization. Then we use the conflict resolution rules for single subject request described in Chapter 2[3].

After transforming a $\mathcal{D}_{\mathcal{AL}^*}$ into a logic program and adding propagation and conflict resolution rules, we obtain a logic program $\mathcal{P}$.

(c) Obtain the authorization results

Using $SModels$, we compute the answer sets of program $\mathcal{P}$ in which we extract the authorization results from the predicate $grant(s, right(sn, p, o))$. The authorizations are organized as $AUTH = \{(s, sn, p, o)\}$.

3. Labelling.

The goal of this step is to label the DOM Tree according to the authorizations obtained from the policy reasoning process. Because we have processed the propagation and conflict resolution in logic programs, the propagation and conflict issues have been removed from the answer sets. The we do not need to consider them in labelling process.

[3]There are conflict resolution rules for both *single subject request* and *group subject request* in Chapter 2. We only choose rules for the *single subject request*.

4. Pruning.

   After labelling process, we get a tree in which the nodes are labelled using positive or negative authorizations. For the final authorization decisions, we should remove the nodes with negative authorizations the accesses to which are denied. So we need to prune the nodes with negative authorizations from the DOM Tree.

5. Unparsing.

   The final step is to generate a valid XML document in text format. The XML:DB API provides functions to add a DOM structure document into the database and retrieve a document in both text and DOM format. We list the functions in the following:

   setContentAsDOM() // for storing a DOM format document.
   getContent() // for retrieving a text format document.
   getContentAsDom() // for getting a DOM tree format document.

   After obtaining the pruned DOM Tree document, we save the DOM tree format document into the database and then we can get it using text format.

3.4 Implementation Issues

We design the prototype system with the fine-grained access control for XML resources based on language $\mathcal{AL}^*$, which considers the delegation issue in distributed environments. Our system accepts requests for XML documents. After checking the authorizations over the XML document and its DTD, the system presents the contents that are permitted for requesters. In this section, we present the system architecture, the system components and implementation issues.

3.4.1 System Architecture

As a server-side web application, our system is designed using Java Server Pages (JSP) and servlet technologies based on object oriented programming concepts.

JSP is mainly used to implement the system interfaces that are web pages showed to users. A servlet is a Java program that runs in a Web server. Typically, the servlet takes an HTTP request from a browser, generates dynamic content (e.g., by querying a database), and provides an HTTP response back to the browser. Alternatively, it can be accessed directly from another application component or send its output to another component. Most servlets generate HTML text, but a servlet might instead generate XML to encapsulate data. For a Web application, we need to implement a servlet by extending the *javax.servlet.http.HttpServlet* abstract class. The *HttpServlet* class mainly includes the following functions:

- *init*(...): Initialize the servlet.
- *destroy*(...): Terminate the servlet.
- *doGet*(...): Execute an HTTP GET request.
- *doPost*(...): Execute an HTTP POST request.
- *doPut*(...): Execute an HTTP PUT request.
- *doDelete*(...): Execute an HTTP DELETE request.
- *service*(...): Receive HTTP requests and, by default, dispatch them to the appropriate *doXXX*() functions.
- *getServletInfo*(...): Retrieve information about the servlet.

A servlet class that extends *HttpServlet* implements some or all of these functions, which will override the original implementations as necessary to process the request and return the response as desired.

Each function takes as input an *HttpServletRequest* instance (an instance of a class that implements the *javax.servlet.http.HttpServletRequest* interface) and an *HttpServletResponse* instance (an instance of a class that implements the *javax.serv let.http.HttpServletResponse* interface).

The *HttpServletRequest* instance provides information to the servlet regarding the HTTP request, such as request parameter names and values, the name of the remote host that made the request, and the name of the server that received the request. The *HttpServletResponse* instance provides HTTP-specific functionality in sending the response, such as specifying the content length and MIME type and providing the output stream.

We implement a servlet, the class *Task* inherited from *HttpServlet*, which is the process center to receive requests, communicate with other classes which actually do the operations, and respond to the requests using the results obtained from the related classes. As a subclass of *HttpServlet*, *Task* overrides the *doPost*(...) function. In *Task*, *doGet*(...) does the same work as *doPost*(...), then the *doGet*(...) function is implemented by calling *doGet*(...) function directly. In the system, we also design other classes for data structures of answer sets and policy rules, and policy management and decision engines.

Figure 3.10 shows the system architecture. We implement **RI** (Request Interface) through which the requests from the global network ask to access XML documents and **PMI** (Policy Management Interface) through which the system administrator manages the policy bases for the XML documents using JSP. For the policy management security, we create a bean for the valid authentication of the system administrator. If the system administrator has been authenticated correctly, we set the related property in the bean valid. For each page that is for policy operations, we check whether the property in the bean is valid and decide whether the page can be accessed.

DE (Decision Engine) is the access control decision engine which receives the access request and makes the decision based on the XML document requested and its policy base. **DE** obtains the authorizations through computing the answer sets of the logic program generated from the policy base by *SModels*. **PME** (Policy Management Engine) is the policy management engine which processes the policy operations requested from the system administrator.

In our system, the protected XML documents and their corresponding policy bases are stored in Xindice database that are denoted as **XML** and **XPolicy** respectively in Figure 3.10.

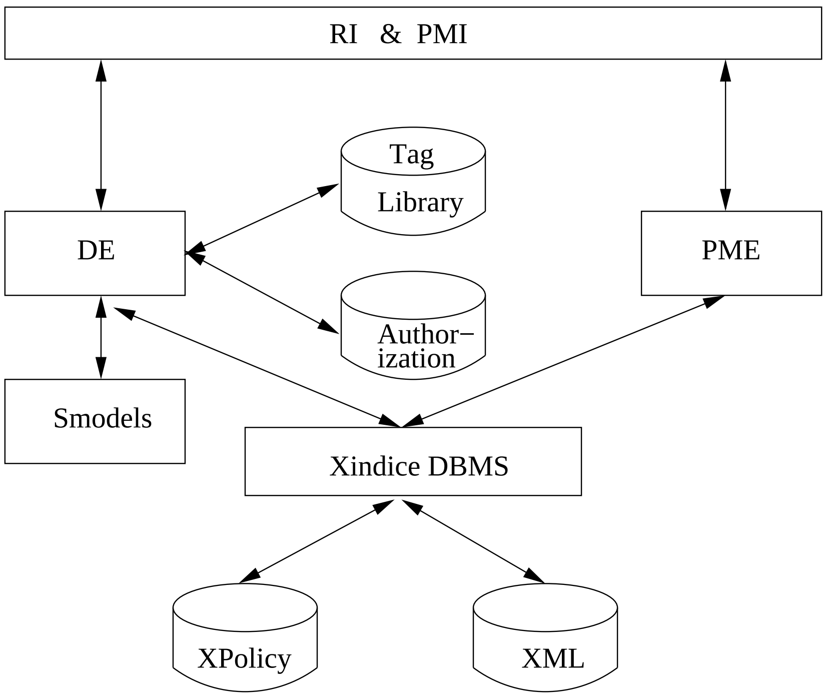

Figure 3.10: System architecture

For the efficiency of the system, we cache the tag library and authorization information for the most requested XML documents. If the XML document and its corresponding policy base have not been modified, **DE** does not need to compute the authorization again, from which, the requester's view can be directly generated.

3.4.2 The Classes of Data Structures for Policy Base and Answer Sets

Our system has basic data structures that are used to store the policy base and answer set information. Java is an object oriented programming language, in which everything is considered as an object and each object is implemented as a class. Our system consists of classes and the relationships among them. We design a set of classes for the policy base and answer sets. Figure 3.11 shows the class diagram for the set of classes to encode the policy base.

In the class diagram of UML, each rectangle denotes a class which can be subdivided into three components. The top component is for the name of the class, the second is for the attributes of the class, and the third is for the functions of the class.

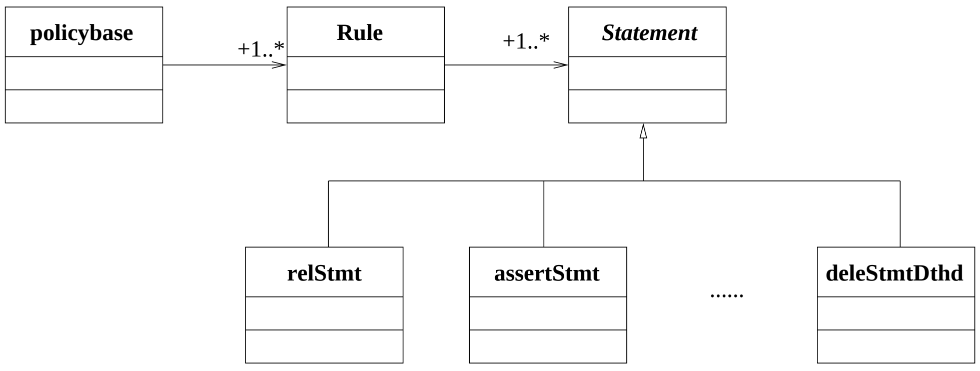

Figure 3.11: Class diagram for policy base classes

The name in italics denotes that the class is an abstract class. Here we omit the last two components and only use the simplified class diagram to show the relationships between classes. The line with an arrow denotes the reference association between two classes. The line with a blank triangle denotes that the class is the subclass of the other one pointed at by the triangle. As shown in Figure 3.11, we define the class *policybase* with the reference to the class *Rule* which holds the reference to the class *Statement*. The cardinality $1..*$ in the reference relationship denotes that an instance of the class *policybase* references to one or more than one instances of the class *Rule* and the same explanation for the classes *Rule* and *Statement*. *Statement* is an abstract class which is inherited by a few subclasses, *relStmt*, *assertStmt*, *authStmt*, *deleStmtSub*, *deleStmtSubSet*, *deleStmtSthd* and *deleStmtDthd*. They extend the class *Statement* with special attributes and functions and implement the abstract functions defined in class *Statement*. In the following, we present some important attributes and functions.

In the class *Statement*, there are some static attributes to define the statement types:

public static final int $RELATION\_STMT = 0$;
public static final int $ASSERT\_STMT = 1$;
public static final int $AUTH\_STMT = 2$;
public static final int $DELE\_STMT\_SUB = 3$;
public static final int $DELE\_STMT\_SUBSET = 4$;
public static final int $DELE\_STMT\_STHD = 5$;

public static final int $DELE\_STMT\_DTHD = 6$;

The class *Statement* is an abstract class in which the common attributes and functions for the specific statements are defined. Since all kinds of statements have an $issuer$, we define the attribute $issuer$ and related functions *getIssuer*() and *setIssuer*(*String issuer*) in the class *statement* as follows:

String $issuer$ = NULL;
public void *setIssuer*(String $person$) { $issuer = person$; };
public String *getIssuer*() { return $issuer$; }

We also define the following abstract functions in class *Statement*:

public abstract int *getType*();
public abstract String $tranToAL$();
public abstract String $tranToLpH$();
public abstract String $tranToLpB$();
public abstract void $setFromDb$(Node *originalnode*, String $prefix$).

The above functions are abstract that should be implemented in the subclasses. For instance, the function *getType*() is to get the statement type. In the class *assertStmt*, we have the following implementation:

public abstract int *getType*() { return ASSERT_STMT };

The specific statements implement the function $tranToAL$() to transform the statement to the format in language $\mathcal{AL}^*$. The functions $tranToLpH$() and $tranToLpB$() are implemented to transform the statement to the predicate in the logic program as a head statement and body statement respectively. The function $setFromDb$() is to get the statement information from the policy base database. In other words, it is to read the detailed statement information from the database and set them to the corresponding attributes of the specific statement classes. For instance, in the assert statement, we can get the $issuer$, $assertTitle$, and $arguments$ of the assert statement from the policy base stored in the XML database and assign them to corresponding attributes of class $assertStmt$.

In class *Rule*, there are an attribute for a *head* statement, an attribute for a list of *positive body* statements, and an attribute for a list of *negtive body* statements.

The class *policybase* includes a list of rules. We implement the functions in class *policybase* to load the rules into the rule list from the policy base database and to add, delete, and retrieve a rule from the rule list.

To generate the authorizations from answer sets, we implement the class $answerSets$ which is for a list of answer sets.

In this section, we mainly focus on the introduction of classes for the data structures and the relationships between them. In Section 3.4.3, we give more attributes and functions of them related with the policy management and decision engines.

3.4.3 Policy Management and Decision Engines

The focus of this section is the internal implementation mechanisms of the policy management and access control decision engines which are the core components of our system. Here we present the detailed internal implementation issues. In the following sections we use pseudocodes to explain our design instead of the source codes in Java.

Policy Management Engine

Policy management functions include loading and unloading XML documents and their corresponding policy bases into the system which are stored in Xindice, a native XML database system, as well as adding, deleting, and updating rules in the policy bases. The policy management engine consists of a set of classes to implement the above functions based on $XML : DB\ API$ as shown in Figure 3.12.

Apparently, all the policy management functions invoke the database operations. The abstract class *Action* mainly creates a database connection and gets the database entrance which is required by other classes in the policy management engine.

During the system process, the database is accessed continually. To control the system resources efficiently and safely, it is better for the system to keep one global database collection instance during its life cycle. All the database operations

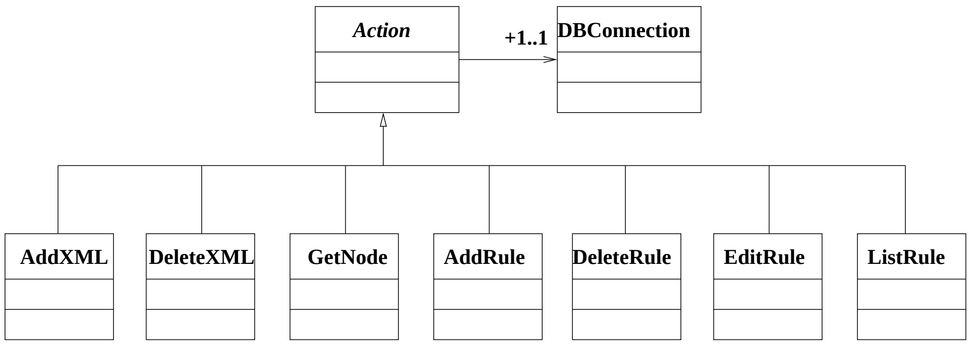

Figure 3.12: Class diagram for policy management

communicate with the same global instance. We implement it using the singleton pattern in Java. We define a static database connection in the class *Action* as follows:

protected static DBConnection $connection = new\ DBConnection()$.

The class *Action* provides the function:

public Collection $getCollection(\ldots)$,

through which the subclasses of *Action* can obtain the global instance of the database collection.

We implement the following subclasses of *Action*:

- AddXML: add an XML document into the database.
- DeleteXML: delete an XML document from the database.
- GetNode: retrieve a node which can be the root node or an ordinary node in an XML document DOM tree.
- AddRule: it is to add a policy rule into the policy base which has the format defined in *XPolicy*.
- DeleteRule: delete a policy rule in a policy base.
- EditRule: edit a policy rule in a policy base.
- ListRule: list the rules in a policy base stored in the database.

Attribute	**Type**	**Description**
docRoot	XNode	The root node of *doc*
requester	String	The requester
reqPriv	String	The requested privilege
reqObj	String	The requested XML document
tab_lib	ArrayList	The tags of nodes
parent	ArrayList	The parent and child relationships
instance	ArrayList	The instance information
auth_lib	ArrayList	The authorizations related to the request

Table 3.1: Attributes in Class *policybase*

Decision Engine

The decision engine receives a request from the **RI** (Request Interface), loads the requested XML document and its corresponding policy base, transforms the policy base into a logic program, computes the answer sets of this program using *SModels*, extracts authorizations over the requester, and finally generates the requester's view. During the decision process, the engine needs data structures to store the different components. The data structures and functions in the decision engine are defined as attributes and functions in the class *policybase*. Table 3.1 summaries the important attributes required by the decision engine in the class *policybase*.

The *docRoot* is used to record the root node of the requested XML document which is an instance of the class *XNode*. We define the class *XNode* which implements the interface $Node$ in the package $org.w3c.dom$. The following attributes and functions are added into *XNode*:

```
{ private byte auth;
  public string tag;
  public void label();
  public void prune();  }.
```

We use *tag* to store the "*tok*" of the node and *auth* to represent whether the node has been granted privileges to the requester, each bit of which represents one privilege. The *auth* has 8 bits that can express the states of 8 privileges. Here we only use one bit and other 7 bits are reserved to extend the system later. Table 3.2 shows the bits assignments.

Bit Value	**Privilege**	**Value**
0	Read	1
1	Reserved	2
2	Reserved	4
3	Reserved	8
4	Reserved	16
5	Reserved	32
6	Reserved	64
7	Reserved	128

Table 3.2: Bits assignment for privileges in $XNode$

We denote the value of a privilege as V_p. We set and remove this privilege in the byte *auth* using the following expressions respectively:

setting operation: $auth \ = \ auth \mid V_p$;
removing operation: $auth \ = \ auth \ \& \ (255 - V_p)$

Initially, we set the *auth* zero to denote that there is no any privilege permitted. The function *label*() sets the *auth* of the node and its child nodes based on the *auth_lib*. The function *prune*() removes the node and its child nodes with negative authorizations.

The attributes *tag_lib*, *parent*, *instance*, and *auth_lib* are defined as the type *ArrayList*, which is an implemented *List* with an array. The *ArrayList* allows rapid random access to its elements, but is slow when inserting and removing elements from the middle of a list. The reasons we define the above four attributes as the type *ArrayList* are that they will be created and accessed in sequence and we do not have inserting and removing operations over them. The *ArrayList* is a storage cabinet to hold the objects. Each time elements in an *ArrayList* are retrieved, their types should be expressed explicitly. All the above four attributes hold a list of *String*[2]'s.

For each element *tag_lib*[*i*] in *tag_lib*, the *tag_lib*[*i*][0] and *tag_lib*[*i*][1] denote the *ne* and *tok* of a node respectively as defined in Section 3.3.3. The *parent*[*i*] represents *tok*s of two nodes extracted from *tag_lib* to denote that *parent*[*i*][0] is the parent node of *parent*[*i*][1]. The *instance*[*i*] also holds *tok*s of two nodes to denote that *instance*[*i*][0] is an instance of *instance*[*i*][0]. For an *auth_lib*[*i*], *auth_lib*[*i*][0] represents the sign of the authorization, in which " − " means the negative authorization and "+" means the positive authorization. The $auth\_lib[i][1]$ represents the *tok* of the object in the authorization. Then the $auth\_lib[i]$ denotes that the user *requester* is granted the $auth\_lib[i][0]$ privilege *reqPriv* over the object $auth\_lib[i][1]$.

In the decision engine, the main function is $computeView()$ defined in the class *policybase*. When our system receives a request $req(s, p, o)$, it creates an instance of *policybase* as follows:

```
policybase pb = $new\ policybase(s,\ p,\ o)$;
```

The construct function of *policybase* does the following initial process:

```
{  $requester = s$;
   $reqPriv = p$;
   $reqObj = o$;
   $doc = new\ secDocument(o)$;
   $docRoot = doc.root$;
}
```

We present the function $computeView()$ and other related functions as follows:

```
void $computeView()$ {
  $docRoot.parse(0)$;
  $pb.tranToLp()$;
  $pb.obtainAuth()$;
  $docRoot.label()$;
  $docRoot.prune()$;
}
void $parse(int\ seq)$ {
  $this.tag = seq$;
  If $this \neq docRoot$
     $String[2]\ pt = \{this.parent().tag, this.tag\}$;
     $parent.add(pt)$;
  For $n$ in $this.getChildNodes()$
     $i = seq + 1$;
```

```
    do $n.parse(i)$;
}
void $tranToLp()$ {
  $lp = pb.getDomainRules()$;
  For each $rule$ in $pb$
    $lp+ = rule.tranToLp()$;
  EndFor
  $lp+ = pb.getPropRules()$;
  $lp+ = pb.getConflictResRules()$;
}
void $obtainAuth()$ {
  Create a new process $smprocess$ to run Smodels
  $ansSets = smprocess.output()$;
  For each predicate $pred == grant(s, right(sn, p, o))$ do
    If $pred$ exists in all answer sets and
          $s == pb.requester$ and $p == pb.reqPriv$
      $String[2]\ auth = \{sn, o\}$;
      $auth\_lib.add(auth)$;
    EndIf
  EndFor
}
void $label()$ {
  If $this.tag == auth\_lib[i][1]$ and $auth\_lib[i][0] ==$ "+"
    $this.auth = this.auth \mid 1$;
  Else
    $this.auth = this.auth\ \&\ 254$;
  EndIf
  For $n$ in $this.getChildNodes()$
    do $n.label()$;
  EndFor
```

```
}
void $prune()$ {
  For $n$ in $this.getChildNodes()$
     do $n.prune()$;
  EndFor
  If $this.getChildNodes() == \emptyset$ and $this.auth \neq$ “ + ”
     remove the current node;
  EndIf
}
```

Note that in the function *obtainAuth*(), we create a new process to run *Smodels* using *Runtime.getRuntime*(). In the function $tranToLp()$, the attribute *instance* of *policybase* is built in the function $rule.tranToLp()$.

3.4.4 Security Consideration for Policy Management

As a web application, the users in the global network can access the pages in the system freely. However, the policy management pages apparently should only permit the authorized users to access.

We implement a java bean, *validAdminBean* to denote the identity verification state, which includes one property “*private boolean valid*”. For this property, we have the following two public functions:

public boolean $getValid()$ { $return\ valid$; };
public void $setValid(boolean\ val)$ { $this.valid = val$; }.

After a user requests the *login* page and fills in his/her password, the system runs the $verification$ process, in which the system checks whether the password is valid or not, and if the user passes the verification, the system creates the java bean *validAdmin* and sets the property by the function $setValid(true)$; otherwise, the system will redirect the user back to the *login* page. In the beginning of each page related to the policy management, we add codes to check whether the bean *validAdmin* exists. The source codes for the above functions are implemented using the standard tag library in JSP.

3.5 Summary

So far, the fine grained access control prototype system for XML documents considering delegation has been implemented, which is a web application to protect XML resources in distributed environments. The users in Internet can make requests to access the XML documents stored in the system. Based on the policy bases for the resources, the system responses with the permitted content.

In our system, the policy specification language $\mathcal{AL}^*$ was simplified from $\mathcal{AL}$ and its XML format, *XPolicy* was developed to specify the policy bases for the protected XML documents. We discussed the privilege, propagation option and conflict resolution problems in the system and defined the policy semantics through *Smodels*. The java implementation of the system was also presented including the system architecture, details of designing data structures and algorithms, and the security considerations of the policy management.

Chapter 4

A Unified Framework for Security Protocol Verification and Update

In this chapter, we propose a unified framework that analyzes security protocols, which were proven secure under models in provable security approach, and repairs protocols that we find insecure against certain types of attacks. Our framework is based on Answer Set Programming to verify a security protocol. More importantly, the protocol update is also integrated into our framework.

Although only Bellare-Rogaway model [14, 15, 16] is provided in this chapter, we should state that our approach is general enough to be used for other models. At this point, our research is close to Choo *et al.* [33], which analyzes the provably secure protocols under Canetti-Krawczyk model.

The framework includes the following three components:

1. Protocol specification: We use ASP to specify the protocol, adversary actions, and attacks based on the adversary model. Instead of the definition of security in Bellare-Rogaway adversary model, we give the definition of insecurity on which we specify the attacks. In this chapter, we choose the Boyd-González Nieto conference key agreement protocol as our case study protocol, which carries a claim of provable security using Bellare-Rogaway adversary model.

2. Protocol verification: After the specification phase, we generate a logic program, which can be computed by *Smodels*. Attacks exist if there are answer sets from which our framework can pick up the attack traces.

3. Protocol update: In our framework, we set up a set of generic update methods for certain attacks, which have been revealed previously, such as unknown key share attack, reflection attack, and the key-replicating attack. If attacks are found in a protocol, its specification will be rewritten with respect to the set of update methods and updated through the forgetting algorithm, which is developed based on a logic programming update technique [100]. Then the updated program will be re-analyzed and checked whether flaws in the protocol have been repaired.

In the following, we first introduce the adversary model modified from Bellare-Rogaway model and our case study protocol - Boyd-Gonzá́lez Nieto conference key agreement protocol which has been proven secure under Bellare-Rogaway model. After defining the security protocol specification language $\mathcal{L}_{sp}$, we present the security protocol specification using $\mathcal{L}_{sp}$, the protocol verification, and the protocol update based on ASP. Through our case study protocol, we demonstrate the application of our approach.

4.1 Concepts and Notations

4.1.1 Adversary Model

In the setting of the provable security approach for protocols, the adversary model comprises principals taking part in the protocol and a powerful, probabilistic, and polynomial time adversary. The adversary $\mathcal{A}$ controls all communications of all principals in the model by interacting with a set of *oracles*, each of which represents an instance of a principal in a specific protocol run. Each principal has an identifier U_i from a finite set $\{U_1, U_2, \ldots, U_n\}$ and they can run the multiple sessions concurrently. Oracle $\Pi^s_{U_i}$ represents the sth instance of principal U_i in a specific protocol run. $\mathcal{A}$ interacts with the protocol session through *queries*, which are described as follows.

$Send(U_i, s, m)$: Send message m to oracle $\Pi^s_{U_i}$. The oracle will return to the adversary next message according to the protocol specification (this includes the possibility that m is not of the expected format in which case $\Pi^s_{U_i}$ may simply halt). If $\Pi^s_{U_i}$ accepts the session key or halts, this is included in the response.

Reveal(U_i, s): Reveal the session key (if any) accepted by $\Pi^s_{U_i}$. This query models the adversary's ability to find the session key.

Corrupt(U_i): This query allows the adversary to corrupt the principal U_i at will, and thereby learn the complete internal state (e.g., password) of the corrupted principal.

The definition of partnership is used to restrict the adversary's *Reveal* and *Corrupt* queries to oracles that are not partners of the oracle whose key the adversary is trying to guess. The way of defining partner oracles has varied in different approaches. In the more recent research [60, 62, 64], partners have been defined by having the same session identifier (SID) which consists of a concatenation of the messages exchanged between the two principals. In our case study protocol, since all messages are broadcasted, we expect all oracles in the same session to derive the same session identifiers. Therefore we define $SID(\pi^s_U)$ as the concatenation of all messages that oracle Π^s_U has sent and received.

Definition 4.1 *A set of oracles are* partnered *if they have accepted with the same session identifier (SID) and have agreed on the same set of principals and on the initiator of the protocol.*

The notion of *freshness* is defined based on the notion of partnership in definition 4.1.

Definition 4.2 *An oracle $\Pi^s_{U_i}$ is* fresh *at the end of its execution if $\Pi^s_{U_i}$ has accepted with set of partners Π^*, $\Pi^s_{U_i}$ and all oracles in Π^* are unopened (have not been sent a Reveal query), and all principals in Π^* (including U_i) have not been corrupted.*

Instead of the definition of security in Bellare-Rogaway model, we present the definition of *insecurity* in our model. Our approach analyzes the protocol and check whether there are attacks in it. If an attack is found, we consider the protocol is insecure. However, if there is no attack, we can not claim that the protocol is secure. Definition of insecurity for the protocol depends on notions of partnership and freshness of oracles.

Definition 4.3 *A protocol is* insecure *in our framework if one of the following conditions is satisfied*:

1. *Two fresh non-partner oracles accept the same session key,*
2. *Some fresh oracle accepts certain session key, which has been exposed,* and
3. *Some fresh oracle accepts and terminates with no partner.*

4.1.2 Boyd-Gonzǻlez Nieto Conference Key Agreement Protocol

The Boyd-Gonzǻlez Nieto protocol for conference key agreement [27] carries a claimed proof of security in Bellare-Rogaway model [14, 15, 16]. This protocol involves a set of n principals, $\mathcal{U} = \{U_1, \ldots, U_n\}$. Figure 4.1 shows message flows in a protocol run without any disruption from the adversary. In this protocol, the notation (e_U, d_U) denotes encryption and signature keys of principal U respectively; $\{.\}_{e_U}$ denotes the encryption of any messages under the encryption key e_U; $S_{d_U}\{.\}$ denotes the signature of any messages under the signature key d_U; N_i denotes a *nonce*; H denotes some one-way function and SK_{U_i} denotes the session key accepted by principal U_i in the end of the protocol run, and * denotes a broadcast message.

In the protocol run, there is a special principal called *initiator* and other principals called *responder*s. The initiator U_1 chooses a nonce N_1, and sends it to *responders* in an authenticated and confidential way. The responders broadcast their nonces to other principals in $\mathcal{U}$. By the end of the protocol run, all principals compute their session keys $SK_{U_i} = H(N_1||N_2||N_3 \ldots ||N_n)$ respectively.

1. $U_1 \rightarrow * : \mathcal{U} = \{U_1, U_2, \ldots, U_n\}, S_{d_{U_1}}(U, \{N_1\}_{e_{U_2}}, \ldots, \{N_1\}_{e_{U_n}})$
2. $U_1 \rightarrow * : \{N_1\}_{e_{U_i}}$ for $1 < i \leq n$
3. $U_i \rightarrow * : U_i, N_i$

$$SK_{U_i} = H(N_1||N_2||N_3 \ldots ||N_n)$$

Figure 4.1: Boyd-Gonzǻlez Nieto Conference Key Agreement Protocol

4.2 A Security Protocol Specification Language $\mathcal{L}_{sp}$

The language $\mathcal{L}_{sp}$ includes numbers, constants, variables, function symbols and predicate symbols.

1. Numbers
   We use numbers to denote message id, message type, and time in a protocol run.

2. Constants
   Constants are arbitrary strings of English characters and numbers that begin with a lowercase letter to denote principal identifiers who take part in the protocol run. We have a special constant, *all* to denote all principals in the protocol run. Generally, we use a to denote the adversary, u_i to denote a regular participant in the protocol run.

3. Variables
   Variables are arbitrary strings of English characters and numbers that begin with a uppercase letter to denote principals, messages, keys, and times. Generally, we use $A,\ B,\ C, \ldots$ to denote principals, $M,\ M_i$ to denote message *id*s, $P,\ P_i$ to denote message types, $K,\ K_i$ to denote keys, $T,\ T_i$ to denote times, and $S,\ S_i$ to denote basic elements of messages such as principal identifiers, random nonces, encryptions, and signatures.

4. Function symbols
   Function symbols are strings that start with a lowercase letter. There are two kinds of functions in $\mathcal{L}_{sp}$, arithmetic function and symbolic function. Both functions are supported by *Smodels*. In *Smodels*, arithmetic functions are internal functions that can be computed and symbolic functions just define new constants as an argument for predicates after the grounding process in the application. For instance, $5 * 3 + 2$ equals 17, and $pKey(A)$ defines a constant to denote principal A's public key. We present the following symbolic functions.

- $pKey(A)$, $sKey(A)$, $sig\_sKey(A)$, and $sig\_vKey(A)$

  $pKey(A)$ and $sKey(A)$ denote public and secret keys of the principal A for encryption respectively; $sig\_sKey(A)$, and $sig\_vKey(A)$ denote signing and verifying keys of the principal A for the signature purpose.

- $n(N)$, $nonce(A, n(N))$

  $n(N)$ denotes a unique random nonce identified by the sequence number N in a protocol run. $nonce(A, n(N))$ denotes that $n(N)$ is the nonce selected by principal A which may be not true. We explain this point in predicate *holds* later.

- $msg(S_1, S_2, \ldots, S_n)$

  msg is a function with variable parameters. It denotes one message which is the concatenation of a few basic elements, S_1, $S_2, \ldots,$ S_n, where S_i can be a principal identity, a random nonce, an encryption, or a signature. In the remainder of the paper, for readability, we use the mathematical form $S_1||S_2\ \ldots||S_n$ to represent this function.

- $sign(K, msg(.))$

  It denotes a signature of $msg(.)$ using key K.

- $enc(K, msg(.))$

  It denotes an encryption of $msg(.)$ using key K.

We should point out that from definitions, it is clear that *msg*, *enc* and *sign* are defined in a recursive way. However, in most protocols, even complex ones, the format and number of valid messages are usually fixed, which can be expressed by finite basic elements, except some special protocols [76, 83].

5. Predicate symbols

   Predicate symbols are strings that start with a lowercase letter. We present the predicate symbols that are common for most security protocols.

   We start with some basic sort predicates to model the basic components in the language.

   - $player(A)$ denotes that A is a regular participant.

- $adversary(A)$ denotes that A is the adversary.
- $agent(A)$ denotes that A is a principal who may be a regular participant or the adversary. We have the following implicit rules for principals.

  $agent(A) \leftarrow player(A).$

  $agent(A) \leftarrow adversary(A).$
- $ag\_id(A, N)$ denotes that A's id number is N.
- $key(K)$ denotes that K is a key for encryption or signature which can be $pKey(A)$, $sKey(A)$, $sig\_sKey(A)$, or $sig\_vKey(A)$.
- $asymKeyPair(K_1, K_2)$ denotes that K_1 and K_2 are a pair of keys in asymmetric encryption system. For instance, we have $asymKeyPair$ $(sKey(A), pKey(A))$ and $asymKeyPair(pKey(A), sKey(A))$ for the encryption keys.

Next, we present predicates for modelling actions in the protocol run. Note that all the following predicates have the time feature because we consider a protocol run is a sequence of actions.

- $sends(A, B, M, P, T)$ denotes that A sends B the message M with type P at time T.
- $gets(A, M, P, T)$ denotes that A gets the message M with type P at time T.
- $intercept(A, M, P, T)$ denotes that A intercepts the message M with type P at time T.
- $holds(A, X, T)$ denotes that A knows X at time T, where X can be a message id or a basic element of messages.
- $contains(M, P, msg(S_1, S_2, \dots, S_n))$ denotes that the content of the message M with type P is $msg(S_1, S_2, \dots, S_n)$. We introduce this predicate to connect a message id with its real message content.

In this section, we only present common functions and predicates for most security protocols. It is necessary and feasible to provide extra functions and predicates for particular protocols.

4.3 Protocol Specification

A protocol is a process to exchange a sequence of messages among principals in the protocol run. For computational efficiency and expressive concision, in our specification, we represent a message using a message id which is a unique number for each message sent by principals, together with a message type which is the order number of the message in protocol flows. For instance, in our case study protocol showed in Figure 4.1, the message order number, 1, 2, and 3 before message bodies are their message types.

In [4], Aiello *et al.* proposed a logic programming based approach for the security protocol verification. Comparing with their work, integrating both protocol verification and update into our framework distinguishes our approach from theirs in a significant way. In the protocol specification, we express messages using message id and type, instead of the message content directly as in [4]. In this way, it reduces information redundancy in answer sets and makes the specification and answer sets more concise. Moreover, it is more suitable to design an algorithm for finding attack traces. In [4], Aiello et. al. presented predicates $said(A, B, M, T)$ and $got(A, M, T)$ to denote that they are true when their corresponding actions $says$ and $gets$ happened before time T. Although it provides flexibility to explicitly specify information about past runs of the protocol, there are increasing efficiency and memory costs in their approach based on our logic programming experiments.

4.3.1 Modelling Security Protocols

Now we present how to model a security protocol through the case study protocol (refer to the appendix D for the complete specification program). For specification simplicity and efficiency, we simplify the case study protocol to a two-party protocol showed in Figure 4.2 as explained in [60]. Because in a protocol flow of Figure 4.1, messages 1 and 2 can be sent concurrently, in the simplified protocol, we merge them into one message.

1. $U_1 \rightarrow U_2 : \mathcal{U} = \{U_1, U_2\}, S_{d_{U_1}}(\mathcal{U}, \{N_1\}_{e_{U_2}}), \{N_1\}_{e_{U_2}}$
2. $U_2 \rightarrow U_1 : U_2, N_2$

$$SK_{U_1} = H(N_1||N_2) = SK_{U_2}$$

Figure 4.2: Simplified Boyd-González Nieto Conference Key Agreement Protocol

Let $\mathcal{U} = \{U_1, U_2\}$. The initiator, U_1 encrypts N_1 using the public key of U_2, signs $\mathcal{U}$ and the encrypted nonce $\{N_1\}_{e_{U_2}}$, and broadcasts $\mathcal{U}$, the signature value and the encrypted nonce in message flow 1. The principal, U_2, upon receiving the initial message, will respond with his/her identity and a random nonce in message flow 2.

The first part of protocol specification is to set up principals and their keys through predicates, $player(A)$, $agent(A)$, $ag\_id(A, N)$, and $key(K)$, where K is a key function. For instance, in our case study protocol, we have

$$player(u_1),\ player(u_2),\ adversary(a)$$

$$ag\_id(u_1, 0),\ ag\_id(u_2, 1),\ ag\_id(a, 2)$$

$$key(pKey(A)) \leftarrow agent(A).$$

$$key(sKey(A)) \leftarrow agent(A).$$

$$key(sig\_sKey(A)) \leftarrow agent(A).$$

$$key(sig\_vKey(A)) \leftarrow agent(A).$$

The second part is to model the relationship between keys of principals. In our case study protocol, there are encryption and signature keys which are specified as follows.

$$asymKeyPair(pKey(A), sKey(A)) \leftarrow agent(A).$$
$$asymKeyPair(sig\_sKey(A), sig\_vKey(A)) \leftarrow agent(A).$$
$$asymKeyPair(K_1, K_2) \leftarrow asymKeyPair(K_2, K_1).$$

The third part is about message flows in a protocol. During a protocol run, we assume that if a principal A sends a message to B and the adversary does not intercept it, B will receive it at the next time. We model the assumption using the following rule:

$$\begin{aligned} gets(B, M, P, T+1) \leftarrow\ & sends(A, B, M, P, T), \\ & neq(A, B), not\ intercept(a, M, P, T+1). \end{aligned}$$

A protocol consists of a sequence of messages. Except the first message which is sent by the initiator of the protocol run, principals will check preconditions before they send a response message. As explained in [4], a protocol is denoted as:

$$\begin{array}{ll} A \rightarrow B_{i_1} : m_{i_1}, p_{i_1} & \%\ first\ message\ A\ must\ send \\ \quad \ldots & \\ B_{j_1} \rightarrow A : m_{j_1}, p_{j_1} & \%\ first\ message\ A\ must\ receive \\ \quad \ldots & \\ A \rightarrow B_{i_s} : m_{i_s}, p_{i_s} & \%\ last\ message\ A\ must\ send\ before\ m \\ \quad \ldots & \\ B_{i_r} \rightarrow A : m_{i_r}, p_{i_r} & \%\ last\ message\ A\ must\ receive\ before\ m \\ \quad \ldots & \\ A \rightarrow B : m, p & \end{array}$$

As showed above, principal A will send message (m, p) to B, if we check that a sequence of messages have been received and sent before (m, p) in a correct run. We encode this as the following rule:

$$\begin{aligned} & sends(A, B, m, p, T+1) \leftarrow \\ & \qquad sends(A, B_{i_1}, m_{i_1}, p_{i_1}, T_{i_1}), \ldots, sends(A, B_{i_s}, m_{i_s}, p_{i_s}, T_{i_s}), \\ & \qquad gets(A, m_{j_1}, p_{j_1}, T_{j_1}), \ldots, gets(A, m_{i_r}, p_{i_r}, T_{i_r}), \\ & \qquad T_{j_1} > T_{i_1}, \ldots, T_{i_r} > T_{i_s},\ b_1, \ldots, b_n \\ & contains(m, p, msg(.)). \end{aligned}$$

In the above rule, $b_1, \ldots, b_n$ are protocol dependent literals. We consider preconditions for sending message m by principal A as actions that A has performed in previous steps according to the protocol run. Protocol dependant literals are usually to check the freshness of random nonces or timestamps and other conditions needed by particular protocols. Because we represent a message using a message id and type in predicate *sends*, we should add a fact rule, which denotes what the message is about using the predicate *contains*(.).

For instance, in our case study protocol, principal u_1 sends an initial message to start a protocol run. We model it as the following rules:

$sends(u_1, all, 0, 0, 0).$
$contains(0, 0, agset(u1, u2)).$
$contains(0, 0, sign(sig\_sKey(u_1), agset(u1, u2)||enc(pKey(u_2), n(0)))).$
$contains(0, 0, enc(pKey(u_2), n(0))).$

Finally, we model the principal's knowledge including the principal's initial knowledge base and knowledge change during the protocol run. Each principal taking part in the protocol run has an initial knowledge base such as other principals' public keys. While sending and receiving messages, principals will hold them and derive more information by breaking or decrypting all messages for which they have a key. Their knowledge will change during the protocol run. We use predicate $holds(.)$ to specify principals' knowledge.

For encryption and signature keys in the case study protocol, we code initial knowledge bases for principals using the following rules:

$$holds(A, pKey(B), 0) \leftarrow agent(A), agent(B).$$

$$holds(A, sig\_vKey(B), 0) \leftarrow agent(A), agent(B).$$

$$holds(A, sKey(A), 0) \leftarrow agent(A).$$

$$holds(A, sig\_sKey(A), 0) \leftarrow agent(A).$$

Then we write the following rules to model principals' knowledge change during the protocol run.

$$holds(A, M, T) \leftarrow gets(A, M, P, T).$$

$$holds(A, M, T) \leftarrow sends(A, B, M, P, T).$$

$$holds(A, S, T) \leftarrow holds(A, M, T), contains(M, P, S).$$

$$holds(A, S_1, T) \leftarrow holds(A, M, T), contains(M, P, S_1|| \dots ||S_n).$$

$$\dots$$

$$holds(A, S_n, T) \leftarrow holds(A, M, T), contains(M, P, S_1|| \dots ||S_n).$$

$$\begin{aligned}
&holds(A, S_1, T) \leftarrow holds(A, enc(K_1, S_1 || \ldots || S_n), T),\\
&\qquad\qquad holds(A, K_2, T_1), asymKeyPair(K_1, K_2).
\end{aligned}$$

$$\ldots$$

$$\begin{aligned}
&holds(A, S_n, T) \leftarrow holds(A, enc(K_1, S_1 || \ldots || S_n), T),\\
&\qquad\qquad holds(A, K_2, T_1), asymKeyPair(K_1, K_2).
\end{aligned}$$

4.3.2 Modelling Adversary Actions

In the adversary model described in Section 4.1.1, the adversary $\mathcal{A}$ is able to intercept messages, swap data components in the intercepted messages to form new messages, or fabricate new messages according to message forms in security protocols. $\mathcal{A}$ can send messages to oracles through *Send* query. If any oracles have accepted a session key, $\mathcal{A}$ can obtain the session key through *Reveal* query.

$\mathcal{A}$ controls all the communication among principals taking part in the protocol run. We start to model that $\mathcal{A}$ is able to get all messages. In the rules of the following sections, α denotes the adversary.

$$gets(\alpha, M, P, T+1) \leftarrow sends(A, M, P, T).$$

Next, after getting messages, $\mathcal{A}$ is able to decide intercepting them or not, which can be modelled by choice rules in *Smodels* as follows:

$$\{intercept(\alpha, M, P, T)\} \leftarrow gets(\alpha, M, P, T).$$

Although $\mathcal{A}$ is able to fabricate new messages randomly, which are of infinite forms, the protocol analyst modelling the adversary action usually creates messages according to forms in the protocol. For instance, the initial message in our case study protocol includes three sub-messages: a set of principals, $\mathcal{U} = \{U_1, U_2\}$; a signature of $\mathcal{U}$, and the encryption of the random nonce of U_1 using the public key of principal U_2, $\{N_1\}_{e_{U_2}}$; and the encryption, $\{N_1\}_{e_{U_2}}$.

$$Initial\ message\ m : \{\mathcal{U} = \{U_1, U_2\} \ || \ S_{d_{U_1}}(\mathcal{U}, \{N_1\}_{e_{U_2}}) \ || \ \{N_1\}_{e_{U_2}}\}$$

In this message, if $\mathcal{A}$ alters the value $\{N_1\}_{e_{U_2}}$, the signature verification will make the message invalid and the protocol run will halt. However, $\mathcal{A}$ can modify it to a new message as follows and send it to U_2.

$$Modified\ message\ m' : \{\mathcal{U} = \{a, U_2\} \mid\mid S_{d_a}(\mathcal{U}, \{N_1\}_{e_{U_2}}) \mid\mid \{N_1\}_{e_{U_2}}\}$$

When getting this modified message, U_2 considers that $\mathcal{A}$ initiates a protocol run.

4.3.3 Modelling Attacks

In our framework, the adversary model is closely based on Bellare-Rogaway model. If protocols with claimed security under Bellare-Rogaway model are found to be violating any of the conditions in the definition of *insecurity* in section 4.1.1, they will be insecure in Bellare-Rogaway model. Moreover, the proof of the protocol will also be invalid.

Based on the definition of *insecurity*, we should model the SIDs and session keys of principals. The SID of a principal is the concatenation of all messages he receives and sends. We use predicate *inSidList(U,M)* to record the messages that the principal U receives and sends.

$$inSidList(U, M) \leftarrow sends(U, all, M, P, T).$$
$$inSidList(U, M) \leftarrow gets(U, M, P, T).$$

The following two rules specify that two principals have the same SID, where the first one denotes that $sid\_neq\_pair(U_1, U_2)$ is true if a message is in the session id list of principal U_1 and not in the session id list of principal U_2, and the second one specifies conditions which should be satisfied for two principals to have the same SIDs.

$$\begin{aligned}&sid\_neq\_pair(U_1, U_2) \leftarrow \\ &\qquad inSidList(U_1, M),\ not\ inSidList(U_2, M),\ neq(U_1, U_2). \\ &same\_sid\_pair(U_1, U_2) \leftarrow \\ &\qquad not\ sid\_neq\_pair(U_1, U_2),\ not\ sid\_neq\_pair(U_2, U_1),\ neq(U_1, U_2).\end{aligned}$$

In our case study protocol, the session key of a principal is a one-way hash function of the concatenations of random nonces of all principals taking part in the conference protocol.

$$sk(A, h(n(M_1), n(M_2))) \leftarrow holds(A, agset(B, C), T),$$
$$holds(A, nonce(B, n(M_1)), T_1),\ holds(A, nonce(C, n(M_2)), T_2).$$

The following rule models that principal U_1 and U_2 have the same session keys:

$$same\_sk\_pair(U_1, U_2) \leftarrow$$
$$sk(U_1, h(n(M_1), n(M_2))),\ sk(U_2, h(n(M_1), n(M_2))),\ neq(U_1, U_2).$$

Consider the condition 1 in insecurity definition (Definition 4.3) for an instance, if two non-partner oracles have the same session keys, the protocol is insecure. Here the two oracles are not partners if they have different SIDs. The attack is then modelled as follows:

$$attack \leftarrow same\_sk\_pair(U_1, U2),\ not\ same\_sid\_pair(U_1, U2).$$

Note that $same\_sk\_pair(U_1, U_2)$ denotes that principals U_1 and U_2 have the same session keys and $same\_sid\_pair(U_1, U_2)$ denotes that principals U_1 and U_2 have the same SIDs.

4.4 Protocol Verification

After specifying security protocols, adversary actions, and attacks using language $\mathcal{L}_{sp}$, we merge three parts into a logic program $\mathcal{P}$ in which we add a constraint rule,

$$\leftarrow not\ attack.$$

We use *Smodels* sytem to verify security protocols in the following steps:

1. Using *lparse*, we obtain a finite ground logic program $\mathcal{P}^g$ from program $\mathcal{P}$.

2. Using *smodels*, we compute answer sets of ground program $\mathcal{P}^g$.

3. If no answer set exists, the attack does not exist for protocol runs up to time ${t_{max}}^1$.

[1] t_{max} is a max time limit setting up in the logic program.

4. If there is an answer set, we collect atoms representing actions, *sends*, *gets* and *intercept* that are true in the model, from which we can find the sequence of actions that represents an attack trace.

We have conducted a verification experiment for our case study protocol using a PC with an AMD Athlon1.73GHz CPU and 512MB RAM running Linux system, in which the grounded protocol program had 7,601 atoms and 957,709 rules. The verification result is presented in Figure 4.3 which shows the protocol run in the presence of adversary $\mathcal{A}$.

```
Attack found in time: 5.460
   The attack 0 is as follows:
      t0.  u1 ---> all: (0,0)  (agset(u1,u2)||
                                sign(sig_sKey(u1),agset(u1,u2)||enc(pKey(u2),n(0)))||
                                enc(pKey(u2),n(0)))
      t1.   a <--- (0,0)
      t1.   a intercepts (0,0) %%%%%%%%%%%%%%%%
      t1.   a --->  u2: (11,0) (agset(a,u2)||
                                sign(sig_sKey(a),agset(a,u2)||enc(pKey(u2),n(0)))||
                                enc(pKey(u2),n(0)))
      t2.  u2 <--- (11,0)
      t3.  u2 ---> all: (8,1)  (nonce(u2,n(8)))
      t4.  u1 <--- (8,1)
      t4.   a <--- (8,1)
```

Figure 4.3: A verification result of our case study protocol

The verification result in Figure 4.3 is described as follows:

1. At time t_0, initiator u_1 broadcasts an initial message which has three parts: the set of principals in the protocol run, i.e. $agset(u_1, u_2)$; the signature of the principal set and encrypted random nonce $n(0)$ under the public key of principal u_2: $sign(sig\_sKey(u_1), agset(u_1, u_2)||enc(pKey(u2), n(0)))$; and the encryption of the encrypted random nonce $n(0)$: $enc(pKey(u2), n(0))$.

2. At time t_1, $\mathcal{A}$ receives the message and intercepts it. After modifying the principal set to $agset(a, u_2)$ and making a new signature using his own signature key, $sign(sig\_sKey(a), agset(a, u_2)||enc(pKey(u_2), n(0)))$, $\mathcal{A}$ fabricates a new message, and sends it to the principal u_2. Now $\mathcal{A}$ acts as an initiator and starts a different session.

3. At time t_2, principal u_2 receives the message from $\mathcal{A}$ and believes that $\mathcal{A}$ initiates a protocol run.

4. At time t_3, principal u_2 broadcasts his identifier and random number.

5. At time t_4, both u_1 and $\mathcal{A}$ receive the random nonce of principal u_2. Principal u_1 believes that he finishes his own session with u_2. However, u_2 believes he is in a different session from $\mathcal{A}$.

In Figure 4.3, an attack was found in 5.460 seconds. We observe that principal u_1's SID is $(0,8)$ and principal u_2's SID is $(11,8)$. Then u_1 and u_2 are not partners in this protocol run since they do not have matching SIDs. Principal u_1 believes that the session key $SK_{u_1} = h(n(0)||n(8))$ is being shared with u_2, but u_2 believes that the session key $SK_{u_2} = h(n(0)||n(8)) = SK_{u_1}$ is being shared with $\mathcal{A}$. Although $\mathcal{A}$ does not know the session key as $\mathcal{A}$ does not know the value of $n(0)$, as u_2' partner he is able to get $SK_{u_2} = h(n(0)||n(8))$ from u_2 which is the same as SK_{u_1}. Our case study protocol is therefore not secure under Bellare-Rogaway model as being claimed.

In the existing proof, the security of the protocol is proved by finding a reduction to the security of the encryption and signature schemes. The number of principals allowed in the proof model is equal to the number of participants in the proof simulation which is two in the above attack. However, in order to carry out the attack, the adversary $\mathcal{A}$ corrupts a third participant and take part in the protocol run as an ordinary participant. Since the third participant does not exist in the model assumed by the proof, the proof shows the protocol is secure. However, it fails.

4.5 Protocol Update

Following the protocol verification process as described in Section 4.4, a natural question is: whether is it possible to correct the protocols that have been found insecure?

In this section, we first introduce the concept of forgetting in logic programs

[100], which is an efficient logic programming update technique. Then we present an approach to repair security protocols using the forgetting algorithm.

4.5.1 Forgetting in Logic Programs

The concept of forgetting in logic programs [100] is an extension of forgetting in classical propositional theories developed initially by Lin and Reiter [71]. The basic problem of forgetting in logic programs is that after forgetting a set of atoms from a logic program, what the original logic program will be. Intuitively, after forgetting a set of atoms, all these atoms in the set should be eliminated in some way, those atoms having certain connections to forgotten atoms through rules in the program might or might not be affected depending on whether they are positive or negative related to forgotten atoms, and all other atoms should not be affected. Since two forms of negations named classical negation and negation as failure are allowed in logic programs, forgetting atoms involved in these negations have to be treated in different ways. To formalize the above idea, there are two kinds of forgettings, named strong and weak forgettings, which are based on a program transformation called *reduction*. Here we will only introduce weak forgetting as the strong forgetting is not directly relevant in our protocol update approach.

As introduced in Chapter 1, a rule in the logic program is of the form:

$$r:\ l_0 \leftarrow l_1, \ldots, l_m,\ not\ l_{m+1}, \ldots,\ not\ l_n.$$

In the rule r, each l_i is a literal. l_0 is called the *head* of the rule and $\{l_0\}$ is denoted as $head(r)$, while the set of literals $\{l_1, \ldots, l_m\}$ is called the positive body of the rule, denoted as $pos(r)$ and the set of literals $\{l_{m+1}, \ldots, l_n\}$ is called the negative body of the rule, denoted as $neg(r)$.

Definition 4.4 [100] *(Program reduction) Let* Π *be a program and* p *an atom. We define the* reduction *of* Π *with respect to* p*, denoted as* $Reduct(\Pi, \{p\})$*, to be a program obtained from* Π *by (1) for each rule* r *with* $head(r) = p$ *and each rule* r' *with* $p \in pos(r')$*, replacing* r' *with a new rule* r''*:* $head(r') \leftarrow (pos(r') - \{p\}), pos(r), neg(r), neg(r')$*; (2) if there is such rule* r' *in* Π *and has been replaced by* r'' *in (1), then removing rule* r *from the remaining program. Let* P *be a set of* *propositional atoms. Then the reduction of* Π *with respect to* P *is inductively defined as follows:*

$$Reduct(\Pi, \phi) = \Pi,$$
$$Reduct(\Pi, P \cup \{p\}) = Reduct(Reduct(\Pi, \{p\}), P).$$

Definition 4.5 [100] *(Weak forgetting) Let* Π *be a logic program, and* p *a propositional atom. We define a program to be the result of weakly forgetting* p *in* Π*, denoted as* $WForgetLP(\Pi, \{p\})$*, if it is obtained from the following transformation:*

(1) $\Pi' = Reduct(\Pi, \{p\})$*;*
(2) $\Pi' = \Pi' - \{r | head(r) = \{p\}\}$*;*
(3) $\Pi' = \Pi' - \{r | p \in pos(r)\}$*;*
(4) $\Pi' = (\Pi' - \Pi^*) \cup \Pi^\dagger$*, where* $\Pi^* = \{r | p \in neg(r)\}$ *and* $\Pi^\dagger = \{r' | r' : head(r) \leftarrow pos(r),\ not\ (neg(r) - \{p\})$ *where* $r \in \Pi^*\}$*;*
(5)$WForgetLP(\Pi, \{p\}) = \Pi'$.

Weakly forgetting a set of atoms can be defined:

$$WForgetLP(\Pi, \phi) = \Pi,$$
$$WForgetLP(\Pi, P \cup \{p\}) = WForgetLP(WForgetLP(\Pi, \{p\}), P).$$

Here we give an example of weak forgetting in logic programs.

Example 4.1 Let $\Pi = \{b \leftarrow a,\ c.\ \ d \leftarrow\ not\ a.\ \ e \leftarrow\ not\ f.\}$. Then we have

$$WForgetLP(\Pi, \{a\}) = \{d \leftarrow .\ \ e \leftarrow\ not\ f.\}.$$

□

In [100], Zhang and Foo showed that the theory of (strong and weak) forgettings has important applications in solving information conflict under multi-agent environments. It can be served as a unified foundation for various logic program update approaches.

4.5.2 The Protocol Update Model

In this subsection, we define a protocol update model to update protocol specifications with respect to pre-defined protocol repair methods through weak forgetting in logic programming introduced previously.

We consider a set of update methods for protocol repair, denoted as $\mathcal{C} = \{c_1, \ldots, c_k\}$, where each c_i is a method for the protocol repair. We should mention that set $\mathcal{C}$ may be dynamic in the sense that new update methods can be added when new attacks are identified and their corresponding update methods are available. Moreover, for a particular protocol, there may be some specific repair methods. For instance, the following are some generic update methods for security protocols against certain type of attacks.

1. In encryption scheme, add identities of the sender and the intended receiver in encrypted messages.

2. In signature scheme, add identities of the sender and the intended receiver in generated signatures.

3. In MAC scheme, add identities of the sender and the intended receiver in generated MAC digests.

4. In key derivation function, include SID information. For instance, in our case study protocol, session key is constructed as $SK_{U_i} = \mathcal{H}(sid||N_1||N_2)$.

As showed in previous sections, after specifying a security protocol, we obtain a logic program $\mathcal{P}$ which consists of a finite set of rules of the form:

$$r: \ h \leftarrow a_1, \ldots, a_m, \ not\ b_1, \ldots, \ not\ b_n.$$

If an update method c_i is used to repair a protocol, it will alter or delete some rules in program $\mathcal{P}$. We say these rules are *affected* by method c_i, while the others are not affected. In order to specify the protocol update, we introduce new atoms, p_i and q_i which are called *marking atoms*, to denote original rules affected, and updated rules by method c_i respectively. That is, having the set of update methods $\mathcal{C} = \{c_1, \cdots, c_k\}$, we extend language $\mathcal{L}_{sp}$ by adding a set of marking atoms: $\mathcal{L}^*_{sp} = \mathcal{L}_{sp} \cup \{p_i, q_i \mid i = 1, \ldots, k\}$.

Definition 4.6 *(Protocol update model). Given a security protocol program $\mathcal{P}$ and a set of update methods $\mathcal{C} = \{c_1, \ldots, c_k\}$, the* protocol update model *for protocol program $\mathcal{P}$ on update method set $\mathcal{C}$ is $\mathcal{M} = (\mathcal{P}_u, \mathcal{F})$, where $\mathcal{P}_u$ is a logic program in language $\mathcal{L}^*_{sp}$, and $\mathcal{F}$ is a set of atom pairs, if*

1. *$\mathcal{F} = \{(p_1, q_1), \ldots, (p_k, q_k)\}$ and*

2. *$\mathcal{P}_u$ is generated from program $\mathcal{P}$ as follows:*

   Step 1: We divide program $\mathcal{P} = \mathcal{P}^c \cup \mathcal{P}^{uc}$, where $\mathcal{P}^c = \{r |\ r$ has been affected by at least one update method in $\mathcal{C}\}$ and $\mathcal{P}^{uc} = \{r |\ r$ has not been affected by any update methods in $\mathcal{C}\}$.

   Step 2: Checking each rule r in program $\mathcal{P}^c$,

   $$r : h \leftarrow\ a_1, \ldots, a_m,\ not\ b_1, \ldots,\ not\ b_n.$$

   Rule r is affected by a set of update methods $\{c_{l_1}, \ldots, c_{l_s}\} \subseteq \mathcal{C}$. We rewrite r into $r*$ as follows to denote that rule r affected by $\{c_{l_1}, \ldots, c_{l_s}\}$,

   $$r* : h \leftarrow\ a_1, \ldots, a_m,\ not\ b_1, \ldots,\ not\ b_n,\ p_{l_1}, \ldots,\ p_{l_s}.$$

   Update method c_{l_i} may generate new rules to replace the original rule r which have the following form:

   $$r^{l_i} : h' \leftarrow\ a'_1, \ldots,\ not\ b'_1, \ldots,\ q_{l_i}.$$

   where h', a'_i and b'_i are predicates that can be ones that appear in r or can be protocol dependant predicates.

   Step 3: We have marking rules for update methods $\{c_1, \ldots, c_k\}$.

   $$\mathcal{P}^m = \{p_i \leftarrow\ not\ q_i.\ q_i \leftarrow\ not\ p_i. |\ 1 \leq i \leq k\}$$

   $$\mathcal{P}_u = \mathcal{P}^{uc} \cup \{r^*, r^i |\ r \in \mathcal{P}_c,\ 1 \leq i \leq k\} \cup \mathcal{P}^m.$$

From Definition 4.6, we can observe the following changes for rewriting protocol specification program. In Step 1, rules in $\mathcal{P}$ are classified into two classes: $\mathcal{P}^c$ contains the rules affected by update methods, and $\mathcal{P}^{uc}$ contains other rules that are not affected. In Step 2, if a rule $r \in \mathcal{P}^c$ is affected by update method c_{l_1}, $\cdots$,

c_{l_n}, we then rewrite r to r^* by adding p_{l_1}, $\cdots$, p_{l_n} to r's body. On the other hand, update methods $c_{l_i} \in \{c_{l_1}, \cdots, c_{l_n}\}$ may generate a new updated rules such as r^{l_i}. We also add these rules into $\mathcal{P}_c$. Finally, we generate a set $\mathcal{P}^m$ of rule pairs of the form $p_i \leftarrow \textit{not } q_i$ and $q_i \leftarrow \textit{not } p_i$, which is for the purpose of deriving consistent updated protocol.

Now, the program $\mathcal{P}_u$ includes all update information with respect to update methods in $\mathcal{C}$. If we choose to forget all update information which has been marked by atoms q_i's, we are able to obtain the original protocol specification $\mathcal{P}^0$ using weak forgetting algorithm introduced previously. That is

$$\mathcal{P}^0 = WForgetLP(\mathcal{P}_u, \{q_1, \ldots, q_k\}). \tag{4.1}$$

We can obtain the updated protocol specification for update method c_i, if we forget update information except that related to c_i and original information that has been affected by c_i. Then we compute the following program for the updated protocol specification $\mathcal{P}^i$ with respect to c_i[2]:

$$\mathcal{P}^i = WForgetLP(\mathcal{P}_u, \{q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_k\} \cup p_i). \tag{4.2}$$

Note that each $\mathcal{P}^i$ represents one repaired protocol by using update method c_i. Following the verification approach described in section 4.4, we can verify the updated protocol specification, check if the protocol has been repaired, and identify which method is valid for the protocol repair.

Example 4.2 In section 4.3, we specified a case study protocol - Boyd-González Nieto conference key agreement protocol, under our logic programming language $\mathcal{L}_{sp}$, which is formalized as the program $\mathcal{P}$ (see Appendix D for the complete description of program $\mathcal{P}$). Now we present two update methods c_1 and c_2 to repair this protocol, where c_1 is based on our proposed generic update methods, in which we add identities of the sender and intended receiver in encrypted messages, and c_2 is from [32], which is similar to c_1. The difference between the two methods is that c_2 does not include

[2]In our approach, if we need combining two update methods to repair a protocol, we can define a new update method for them and rewrite the rules affected by both methods.

the identity of the receiver in the encrypted messages. The update model of program $\mathcal{P}$ on update methods set $\mathcal{C} = \{c_1, c_2\}$ is $\mathcal{M} = (\mathcal{P}_u, \mathcal{F})$, where $\mathcal{F} = \{(p_1, q_1), (p_2, q_2)\}$.

We present rules affected by two update methods, c_1 and c_2, and give updated rules for $\mathcal{P}_u$. Refer to Appendix D for other rules that are not affected.

In program $\mathcal{P}$, rules that include encryption messages are affected by both update methods, c_1 and c_2.

r1: $\mathit{contains}(0, 0, \mathit{enc}(\mathit{pKey}(u_2), n(0)))$.

r2: $\mathit{contains}(0, 0, \mathit{sign}(\mathit{sig\_sKey}(u_1), \mathit{agset}(u_1, u_2)||\mathit{enc}(\mathit{pKey}(u_2), n(0))))$.

r3: $\mathit{contains}(M_1, 0, \mathit{enc}(\mathit{pKey}(U), n(M))) \leftarrow$
$\qquad \mathit{intercept}(a, M, 0, T), \mathit{sends}(a, U, M_1, 0, T)$.

r4: $\mathit{contains}(M_1, 0, \mathit{sign}(\mathit{sig\_sKey}(a), \mathit{agset}(a, U), \mathit{enc}(\mathit{pKey}(U), n(M)))) \leftarrow$
$\qquad \mathit{intercept}(a, M, 0, T), \mathit{sends}(a, U, M_1, 0, T)$.

r5: $\mathit{holds}(C, \mathit{nonce}(A, n(N)), T) \leftarrow \mathit{holds}(C, M, T), \mathit{holds}(C, \mathit{sKey}(B), 0),$
$\qquad \mathit{contains}(M, 0, \mathit{agset}(A, B)), \mathit{contains}(M, 0, \mathit{enc}(\mathit{pKey}(B), n(N)))$.

Rules r_1, r_2, r_3, and r_4 denote the detailed information of a message using predicate *constains*, for which update forms are similar. We take r_3 as an instance to demonstrate what rules will be generated with respect to update methods, c_1 and c_2 as follows.

$$\begin{aligned}
r_3^* &: \mathit{contains}(M_1, 0, \mathit{enc}(\mathit{pKey}(U), n(M))) \leftarrow \\
&\qquad \mathit{intercept}(a, M, 0, T), \mathit{sends}(a, U, M_1, 0, T), p_1, p_2. \\
r_3^1 &: \mathit{contains}(M_1, 0, \mathit{enc}(\mathit{pKey}(U), A||B||n(M))) \leftarrow \mathit{intercept}(a, M, 0, T), \\
&\qquad \mathit{sends}(a, U, M_1, 0, T), \mathit{contains}(M, 0, \mathit{enc}(\mathit{pKey}(U), A||B||n(M))), q_1. \\
r_3^2 &: \mathit{contains}(M_1, 0, \mathit{enc}(\mathit{pKey}(U), A||n(M))) \leftarrow \mathit{intercept}(a, M, 0, T), \\
&\qquad \mathit{sends}(a, U, M_1, 0, T), \mathit{contains}(M_1, 0, \mathit{enc}(\mathit{pKey}(U), A||n(M))), q_2.
\end{aligned}$$

Rule r_3^* represents the original rule r_3 affected by c_1 and c_2 in $\mathcal{P}_u$, rule r_3^1 represents the updated rule with respect to update method c_1, and rule r_3^2 represents the updated rule with respect to update method c_2. For rules r_1, r_2, and r_4, we generate similar rules as r_3.

Rule r_5 models how a principal obtains nonce information and we generate the following rules for r_5.

$$\begin{aligned}
&r_5^* : holds(C, nonce(A, n(N)), T) \leftarrow holds(C, M, T), holds(C, sKey(B), 0),\\
&\quad contains(M, 0, agset(A, B)), contains(M, 0, enc(pKey(B), n(N))), p_1, p_2.\\
&r_5^1 : holds(C, nonce(A, n(N)), T) \leftarrow \ holds(C, M, T),\\
&\quad contains(M, 0, enc(pKey(B), A||B||n(N))),\ holds(C, sKey(B), 0),\ q_1.\\
&r_5^2 : holds(C, nonce(A, n(N)), T) \leftarrow \ holds(C, M, T),\\
&\quad\ contains(M, 0, enc(pKey(B), A||n(N)),\ holds(C, sKey(B), 0),\ q_2.
\end{aligned}$$

We have a set of marking rules:

$$\mathcal{P}^m = \{p_1 \leftarrow \ not\ q_1.\ p_2 \leftarrow \ not\ q_2.\ q_1 \leftarrow \ not\ p_1.\ q_2 \leftarrow \ not\ p_2.\}$$

Then the updated program for our case study protocol with respect to update methods c_1 and c_2 is as follows:

$$\mathcal{P}_u = (\mathcal{P} - \{r_i\}) \cup \{r_i^*, r_i^1, r_i^2\} \cup \mathcal{P}^m.$$

where $i = 1, 2, 3, 4, 5$.

Based on Equations (4.1) and (4.2) and the forgetting algorithm introduced previously, we obtain updated specifications for original protocol, update method c_1 and c_2 respectively.

$$\begin{aligned}
\mathcal{P}^0 &= WForgetLP\ (\mathcal{P}_u, \{q_1, q_2\})\\
&= (\mathcal{P} - \{r_1, r_2, r_3, r_4, r_5\}) \cup \{r_1^*, r_2^*, r_3^*, r_4^*, r_5^*\} \cup \{p_1 \leftarrow .\ \ p_2 \leftarrow .\}\\
\mathcal{P}^1 &= WForgetLP\ (\mathcal{P}_u, \{q_2, p_1\})\\
&= (\mathcal{P} - \{r_1, r_2, r_3, r_4, r_5\}) \cup \{r_1^1, r_2^1, r_3^1, r_4^1, r_5^1\} \cup \{q_1 \leftarrow .\ \ p_2 \leftarrow .\}\\
\mathcal{P}^2 &= WForgetLP(\mathcal{P}_u, \{q_1, p_2\})\\
&= (\mathcal{P} - \{r_1, r_2, r_3, r_4, r_5\}) \cup \{r_1^2, r_2^2, r_3^2, r_4^2, r_5^2\} \cup \{q_2 \leftarrow .\ \ p_1 \leftarrow .\}
\end{aligned}$$

We can observe that after forgetting, program $\mathcal{P}^0$ is the same as the original specification program $\mathcal{P}$ (no repair happens). Program $\mathcal{P}^1$ keeps the update information for update method c_1 and program $\mathcal{P}^2$ for update method c_2.

Program $\mathcal{P}^1$ and $\mathcal{P}^2$ can be re-analyzed as described in section 4.4. Because both methods add the identification of sender in encrypted messages, they prevent the adversary from fabricating the initial message and avoid the unknown key share attack. Take rule r_5 from program $\mathcal{P}^2$ for instance: rule r_5 is removed from program $\mathcal{P}$, and rules r_5^*, r_5^1 and r_5^2 are generated into program $\mathcal{P}_u$; rule r_5^* is removed by forgetting atom p_2 and rule r_5^1 is removed by forgetting atom q_1 from program $\mathcal{P}_u$. Finally, rule r_5^2 remains in program $\mathcal{P}_u$. After forgetting process, the marking rule, $q_2 \leftarrow \; not \; p_2$, becomes $q_2 \leftarrow$, which makes r_5^2 is valid in program $\mathcal{P}^2$. In rule r_5^2, instead of from the principal set in rule r_5, the principal C obtains the nonce owner information from encrypted messages. $\Box$

4.6 Summary

We developed a unified framework for security protocol verification and update. In our framework, we defined a logic based protocol specification language $\mathcal{L}_{sp}$ and provided the methodology of protocol specification. Using *Smodels*, we do the protocol verification. Moreover, based on the forgetting algorithm of logic programs, we defined a protocol update model which can be used to update a security protocol with respect to a set of pre-defined update methods. Through the case study protocol, Boyd-González Nieto conference key agreement protocol, we illustrated how to specify, verify and update security protocols using our approach.

In our approach, we have done some implementations using $c++$ programming. After the protocol specification and verification, if attacks happen, the attack traces should be identified. We implemented the attack tracing program to pick up the attacks automatically. During the protocol update, the focus is the forgetting algorithm in logic programs. We implemented this algorithm in which the input is the grounded logic program obtained from $lparse$ and after forgetting process the output is a logic program in the format that can be accepted by $smodels$. The program has been applied in our case study protocol.

Chapter 5

Conclusion

In this final chapter, we present the summary of the thesis and discuss further research directions.

5.1 Summary of Research

In this thesis, we focused on logic programming based knowledge representations for complex authorizations and security protocols.

For the authorization in open distributed environments, we adopt the trust management approach, in which designing a policy specification language with a rich expressiveness and finding the theoretical foundation for the authorization decisions are required. We developed the policy specification language $\mathcal{AL}$, in which the nonmonotonic feature distinguishes it from languages in many other trust management systems. $\mathcal{AL}$ can be used to express nonmonotonic policies, delegation with depth control, both positive and negative authorizations, complex subject structures, and separation of duty policies. The semantics of $\mathcal{AL}$ is defined based on Answer Set Programming. We transformed an $\mathcal{AL}$ program into a logic program. The transformation is achieved in linear time. With the nonmonotonic feature, $\mathcal{AL}$ has the intractable computational property in general. We identified two tractable subclasses of $\mathcal{AL}$ based on the notions of call consistent and locally stratified logic programs respectively. Through two case studies, we demonstrated important applications of our approach.

We then applied our approach to the access control for XML documents. XML has become a standard language for the representation and exchange of information on the Internet. Due to the structures of XML, the access control to the elements in an XML document is possible. We have implemented a fine grained access control prototype system for XML documents with the consideration of delegation. In our system, we obtained a policy specification language $\mathcal{AL}^*$ simplified from $\mathcal{AL}$ and used its equivalent XML format *XPolicy* to express the authorization policies over XML documents. The requester's view in our system was computed based on the semantics of the language $\mathcal{AL}^*$. As a web application, the system was implemented using JSP and Servlet techniques. Both the protected resources and the authorization policy bases are XML documents. Then in the system design, we chose *Xindice*, a native XML database system to store those XML information. The details of the design concepts and implementation were also presented.

We developed a unified framework in which we not only use formal verification under adversary models in the provable security theory, but also integrate protocol analysis and update into the approach. As logic programming is a declarative executable approach for knowledge representation and reasoning, in our framework, we defined a security protocol specification language $\mathcal{L}_{sp}$ under Answer Set Programming to specify security protocols carrying claimed security proof under adversary models. Using *Smodels* we verified protocols we have modelled. Furthermore, through the proposed update model, we can update protocols that are found insecure. As a case study, Boyd-González Nieto conference key agreement protocol has been specified, verified and updated using our approach.

In summary, in this thesis we demonstrated significant applications of Answer Set Programming as a formal representation mechanism in the areas of the authorization specification and security protocol verification. More importantly, we believe that the techniques and methods developed from this thesis make an important step toward a better understanding of the complex distributed authorization and security protocol verification and update.

5.2 Discussions of Future Directions

Much work remains to be done in this area. Here we list a few interesting research directions on how this research can be further extended.

- For the authorization in distributed environments, delegation is an important feature that distinguishes distributed authorization from traditional centralized authorization. To answer an access request, our current approach in Chapter 2 will only compute a result to grant, deny, or be undecided to the request. However, very often, it is more useful to also explain why such request can be granted, denied or undecided which is the *Delegation Chain Discovery* problem. In a distributed environment, this could be difficult to achieve because the underlying delegation procedure may be very complex [67]. Using Answer Set Programming, it is possible to efficiently retrieve such complex delegation chains from the answer sets that we have computed.

  Another important direction in this area is related to the temporal dimension of the authorization. In many real-world applications, users must be given access authorizations to resources only for the time in which they are expected to need them. Currently, $\mathcal{AL}$ does not specify time-dependant authorizations. The temporal authorization extension of our work entails two questions, including how to model the temporal intervals and how to model the relationship between the temporal intervals in $\mathcal{AL}$. We can extend $\mathcal{AL}$ to express time by adding two extra parameters to each authorization atom for representing the starting and ending time points of the interval. There exists some research work [5, 63] on relations between time intervals, among which Allen [5] found that a total of 13 possible disjoint relations may exist between any two temporal intervals and proposed an algebra to represent a network of interval relations. We can extend $\mathcal{AL}$ to model temporal interval relations based on the existed algebra results.

- For the access control prototype system introduced in Chapter 3, two possible improvements may be made:

  - In the specification of objects, our approach only permitted the element value and attribute values as the restriction for the *element-spec*. If functions could be added as a condition to specify the elements, the system should have a richer expressive power and more flexible.

  - In our approach, the answer sets of a logic program were computed by creating a new process for $SModels$ which is a $c{+}{+}$ application program. The system would be more efficient if $SModels$ can be transplanted into Java functions.

- In Chapter 4, we have chosen a case study protocol to demonstrate our protocol verification and update framework. More protocols should be investigated to find the popular sequence of games technique employed in many cryptographic proofs.

Appendix A

Logic Programs for Authorization Scenarios

A.1 The Program for Scenario 2.1

```
time(1..5).
subjects(alice;bob;carol;david;list).
gsub(list).

#domain subjects(X;Y;Z).
#domain gsub(G).
#domain time(T).

% 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(+,recovery,key),1).

match(list,right(+,recovery,key)):-
    auth(local,list,right(+,recovery,key),1),
    1{req(X,right(+,recovery,key)): assert(hrM,isAManager(X))}1,
    1{req(Y,right(+,recovery,key)): assert(hrM,isAnAuditor(Y))}1,
    1{req(Z,right(+,recovery,key)): assert(hrM,isATech(Z))}1.
```

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

% Authorization rule.
exist_pos(X,right(+,recovery,key)):-
       auth(local,X,right(+,recovery,key),T).
exist_neg(X,right(-,recovery,key)):-
       auth(local,X,right(-,recovery,key),T).
ggrant(G,right(+,recovery,key)):-
      auth(local,G,right(+,recovery,key),T),
      match(G,right(+,recovery,key)),
      not exist_neg(G,right(-,recovery,key)).
ggrant(G,right(-, recovery,key)):-
      not exist_pos(G,right(+,recovery,key)).
```

A.2 The Program for Scenario 2.2

```
lenth(0..5).
sign(+;-).
obj(http;smtp;ftp;mysql;services).
sub(alice;bob;local;so;hrM).

#domain lenth(Step).
#domain lenth(T;T1;T2).
#domainlenth(Dep).
#domain sign(Sn).
#domain obj(O).
#domain sub(X;Y;Z).

below(http, services).
below(mysql, services).

below(smtp, services).
below(ftp, services).

assert(hrM, isStaff(alice)).
assert(hrM, isStaff(bob)).
assert(hrM, onHoliday(alice)).

% delegation rule.
delegate(local, so, right(both,access,services),3,1).
% add implied rules.
delegate(local, so, right(both,access,O),3,1):-
        delegate(local,so,right(both,access,services),3,1),
        below(O,services).

auth(local,X,right(Sn,access,services),T+1):-
        delegate(local,so,right(both,access,O),3,1),
        auth(so,X,right(Sn,access,O),T).

delegate(local,X,right(both,access,O),min(3-Step,Dep),1+Step):-
        delegate(local,so,right(both,access,O),3,1),
        delegate(so,X,right(both,access,O),Dep,Step),
        Step < 3.

delegate(so,so,right(both,access,services),Dep,1):-
        delegate(local,so,right(both,access,services),3,1),
        Dep <= 3.

delegate(local,so,right(both,access,services),Dep,1):-
        delegate(local,so,right(both,access,services),3,1),
        Dep < 3.

auth(so,X,right(+,access,O),1):- assert(hrM,isStaff(X)),
        below(O,services), neq(O,mysql).

auth(so,X,right(+,access,mysql),1):-
        assert(hrM,isStaff(X)), not assert(hrM,onHoliday(X)).

% authorization rules.
exist_pos(X,right(+,access,O)):-
        auth(local,X,right(+,access,O),T).

exist_neg(X,right(-,access,O)):-
        auth(local,X,right(-,access,O),T).

grant(X,right(+,access,O)):-
        auth(local,X,right(+,access,O),T),
        not exist_neg(X,right(-,access,O)).

grant(X,right(-,access,O)):-
        not exist_pos(X,right(+,access,O)).

%conflict rules.
pos_far(X,right(+,access,O),T1):-
        auth(local, X, right(+,access,O),T1),
        auth(local, X, right(-,access,O),T2),
        T1>T2.

neg_far(X,right(-,access,O),T1):-
        auth(local, X, right(-,access,O),T1),
        auth(local, X, right(+,access,O),T2),
        T1>T2.

grant(X,right(pp,access,O)):-
        auth(local, X, right(-,access,O),T1),
        neg_far(X,right(-,access,O),T1),
        auth(local, X, right(+,access,O),T2),
        not pos_far(X, right(+,access,O),T2).

grant(X,right(-,access,O)):-
        auth(local,X,right(+,access,O),T1),
        auth(local, X, right(-,access,O),T2),
        not neg_far(X, right(=,access,O),T2).
```

Appendix B

The Syntax of Language $\mathcal{AL}^*$

$$
\begin{array}{rcl}
\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{head-stmt} \rangle & ::= & \langle \textit{relation-stmt} \rangle \mid \langle \textit{assert-stmt} \rangle \mid \\
 & & \langle \textit{auth-stmt} \rangle \mid \langle \textit{delegate-stmt-head} \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{body-stmt} \rangle & ::= & \langle \textit{relation-stmt} \rangle \mid \langle \textit{assert-stmt} \rangle \mid \\
 & & \langle \textit{auth-stmt} \rangle \mid \langle \textit{delegate-stmt-body} \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} \rangle & ::= & \langle \textit{sub} \rangle\ \textit{grants}\ \langle \textit{auth-atom} \rangle\ \textit{to}\ \langle \textit{sub} \rangle \\
\langle \textit{delegate-stmt-body} \rangle & ::= & \langle \textit{sub} \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{relation-atom} \rangle & ::= & \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 \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} \rangle \mid \langle \textit{sub} \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{entity} \rangle & ::= & \langle \textit{sub} \rangle \mid \langle \textit{obj} \rangle \mid \langle \textit{priv} \rangle
\end{array}
$$

$$\begin{aligned}
\langle \textit{entity-set} \rangle &::= \langle \textit{entity} \rangle \mid \langle \textit{entity} \rangle,\ \langle \textit{entity-set} \rangle \\
\langle \textit{sign} \rangle &::= + \mid\ - \mid \Box \\
\langle k \rangle &::= \langle \textit{natural-number} \rangle \\
\langle \textit{threshold} \rangle &::= \langle \textit{sth} \rangle \mid \langle \textit{dth} \rangle \\
\langle \textit{sth} \rangle &::= \textit{sthd}(\langle k \rangle, \text{``[''} \langle \textit{sub-set} \rangle \text{``]''}) \\
\langle \textit{dth} \rangle &::= \textit{dthd}(\langle k \rangle, \langle \textit{sub-var} \rangle, \langle \textit{assert-stmt} \rangle) \\
\langle \textit{query} \rangle &::= \langle \textit{sub} \rangle\ \textit{requests}\ (+, \langle \textit{priv} \rangle, \langle \textit{obj} \rangle)
\end{aligned}$$

Appendix C

An Example for the Policy Base

The complete version of the policy base in Example 3.4.

```
⟨policybase⟩
  ⟨rule hnum=1 posbnum=0 negbnum=1⟩
    ⟨head⟩
      ⟨stmt-type⟩auth-stmt⟨/stmt-type⟩
      ⟨auth-stmt⟩
        ⟨issuer⟩fA⟨/issuer⟩
        ⟨sign⟩p⟨/sign⟩
        ⟨priv⟩read⟨/priv⟩
        ⟨obj⟩Order.Customer⟨/obj⟩
        ⟨sub⟩X⟨/sub⟩
      ⟨/auth-stmt⟩
    ⟨/head⟩
    ⟨negbody⟩
      ⟨stmt-type⟩assert-stmt⟨/stmt-type⟩
      ⟨assert-stmt⟩
        ⟨issuer⟩fA⟨/issuer⟩
        ⟨assertTitle argNum=1⟩isACompetitor⟨/assertTitle⟩
        ⟨arg⟩
          ⟨argTitle⟩X⟨/argTitle⟩
          ⟨arg-type⟩subject⟨/arg-type⟩
        ⟨/arg⟩
      ⟨/assert-stmt⟩
    ⟨/negbody⟩
  ⟨/rule⟩
  ⟨rule hnum=1 posbnum=1 negbnum=0⟩
```

```
    ⟨head⟩
        ⟨stmt-type⟩auth-stmt⟨/stmt-type⟩
        ⟨auth-stmt⟩
            ⟨issuer⟩fA⟨/issuer⟩
            ⟨sign⟩m⟨/sign⟩
            ⟨priv⟩read⟨/priv⟩
            ⟨obj⟩Order⟨/obj⟩
            ⟨sub⟩X⟨/sub⟩
        ⟨/auth-stmt⟩
    ⟨/head⟩
    ⟨posbody⟩
        ⟨stmt-type⟩assert-stmt⟨/stmt-type⟩
        ⟨assert-stmt⟩
            ⟨issuer⟩fA⟨/issuer⟩
            ⟨assertTitle argNum=1⟩isACompetitor⟨/assertTitle⟩
            ⟨arg⟩
                ⟨argTitle⟩X⟨/argTitle⟩
                ⟨arg-type⟩subject⟨/arg-type⟩
            ⟨/arg⟩
        ⟨/assert-stmt⟩
    ⟨/posbody⟩
⟨/rule⟩
⟨rule hnum=1 posbnum=0 negbnum=0⟩
    ⟨head⟩
        ⟨stmt-type⟩assert-stmt⟨/stmt-type⟩
        ⟨assert-stmt⟩
            ⟨issuer⟩fA⟨/issuer⟩
            ⟨assertTitle argNum=1⟩isACustomer⟨/assertTitle⟩
            ⟨arg⟩
                ⟨argTitle⟩a⟨/argTitle⟩
                ⟨arg-type⟩subject⟨/arg-type⟩
        ⟨/arg⟩
      ⟨/assert-stmt⟩
    ⟨/head⟩
  ⟨/rule⟩
  ⟨rule hnum=1 posbnum=0 negbnum=0⟩
    ⟨head⟩
      ⟨stmt-type⟩assert-stmt⟨/stmt-type⟩
      ⟨assert-stmt⟩
        ⟨issuer⟩fA⟨/issuer⟩
        ⟨assertTitle argNum=1⟩isACompetitor⟨/assertTitle⟩
        ⟨arg⟩
          ⟨argTitle⟩d⟨/argTitle⟩
          ⟨arg-type⟩subject⟨/arg-type⟩
        ⟨/arg⟩
      ⟨/assert-stmt⟩
    ⟨/head⟩
  ⟨/rule⟩
  ⟨rule hnum=1 posbnum=0 negbnum=0⟩
    ⟨head⟩
      ⟨stmt-type⟩dele-stmt-head⟨/stmt-type⟩
      ⟨dele-stmt-head⟩
        ⟨issuer⟩fA⟨/issuer⟩
        ⟨priv⟩read⟨/priv⟩
        ⟨obj⟩Order.Parts⟨/obj⟩
        ⟨step⟩2⟨/step⟩
        ⟨sub⟩X⟨/sub⟩
      ⟨/dele-stmt-head⟩
    ⟨/head⟩
  ⟨/rule⟩
  ⟨rule hnum=1 posbnum=0 negbnum=0⟩
    ⟨head⟩
```

```
        ⟨stmt-type⟩auth-stmt⟨/stmt-type⟩
        ⟨auth-stmt⟩
            ⟨issuer⟩a⟨/issuer⟩
            ⟨sign⟩p⟨/sign⟩
            ⟨priv⟩read⟨/priv⟩
            ⟨obj⟩aOrder⟨/obj⟩
            ⟨sub⟩d⟨/sub⟩
        ⟨/auth-stmt⟩
    ⟨/head⟩
  ⟨/rule⟩
⟨/policybase⟩
```

Appendix D

The Logic Program for a Security Protocol

The logic program for the Boyd-González Nieto conference key agreement protocol.

```
% The basic elements in the protocol
#const t_max = 4.
#const m = 3*(t+1).

time(0..t_max).
aid(0..2). % agent id
mid(0..m). % message id
mt(0..1). % message type

player(u1;u2).
adversary(a).
agent(u1;u2;a).

ag_id(u1,0).
ag_id(u2,1).
ag_id(a,2).

#domain time(T;T1;T2).
#domain mid(M;M1;N).
#domain aid(I).
#domain mt(P).
#domain agent(A;B;C;D).
#domain player(U1;U2;U).
```

```
% Relationships between keys
key(pKey(A)).
key(sKey(A)).
key(sig_sKey(A)).
key(sig_vKey(A)).
asymKeyPair(pKey(A),sKey(A)).
asymKeyPair(sKey(A),pKey(A)).
asymKeyPair(sig_sKey(A),sig_vKey(A)).
asymKeyPair(sig_vKey(A),sig_sKey(A)).

% The message flows in the protocol
sends(u1,all,0,0,0).

contains(0,0,msg(agset(u1,u2)).

contains(0,0,msg(sign(sig_sKey(u1),
        msg(agset(u1,u2),enc(pKey(u2),msg(n(0))))))).
contains(0,0,msg(enc(pKey(u2),msg(n(0))))).

holds(u1,nonce(u1,n(0)),0). holds(u1,agset(u1,u2),0).

gets(a,M,P,T+1):-sends(U,all,M,P,T).
gets(B,M,P,T+1):-sends(A,B,M,P,T),
        neq(A,B), not intercept(a,M,P,T+1).
gets(B,M,P,T+1):-sends(A,all,M,P,T),
        neq(A,B), not intercept(a,M,P,T+1).
sends(U,all,M1,1,T+1):-
        gets(U,M,0,T), assign(M1,I*(t+1)+T+1),
ag_id(U,I).
contains(M1,1,msg(A,n(M1))):- sends(A,all,M1,1,T).
```

```
holds(A,nonce(A,n(M1)),T):-
        sends(A,all,M1,1,T),contains(M1,1,msg(A,n(M1))).

{intercept(a,M,0,T)}:- gets(a,M,0,T).

sends(a,B,M1,0,T):- intercept(a,M,0,T),
        assign(M1,I*(t+1)+T), ag_id(a,I), holds(a,agset(A,B),T1).

contains(M1,0,msg(agset(a,U))):-
        intercept(a,M,0,T), sends(a,U,M1,0,T).
contains(M1,0,msg(sign(sig_sKey(a),
        msg(agset(a,U),enc(pKey(U),msg(n(M)))))):-
                intercept(a,M,0,T),sends(a,U,M1,0,T).

contains(M1,0,msg(enc(pKey(U),msg(n(M))))):-
        intercept(a,M,0,T), sends(a,U,M1,0,T).

% Knowledge
holds(A,pKey(B),0).
holds(A,sig_vKey(B),0).
holds(A,sKey(A),0).
holds(A,sig_sKey(A),0).

holds(A,M,T):- gets(A,M,P,T).

holds(A,nonce(B,n(N)),T):- holds(A,M,T),
contains(M,1,msg(B,n(N))).

holds(C,agset(A,B),T):-
        holds(C,M,T), contains(M,0,msg(agset(A,B))).
holds(C,nonce(A,n(N)),T):- holds(C,M,T),
        holds(C,sKey(B),0), contains(M,0,msg(agset(A,B))),
        contains(M,0,enc(pKey(B),msg(n(N)))).

% Attacks
sk(A,h(n(M),n(M1))):- holds(A,agset(B,C),T),
        holds(A,nonce(B,n(M)),T1), holds(A,nonce(C,n(M1)),T2).
same_sk_pair(A,B):-
        sk(A,h(n(M),n(M1))), sk(B,h(n(M),n(M1))), neq(A,B).
inSidList(U,M):- sends(U,all,M,P,T).

inSidList(U,M):- gets(U,M,P,T).

sid_neq_pair(U,U1):-
        inSidList(U,M), not inSidList(U1,M),neq(U,U1).
same_sid_pair(U,U1):- not sid_neq_pair(U,U1),
         not sid_neq_pair(U1,U), neq(U,U1).

attack:- same_sk_pair(U,U1), not same_sid_pair(U,U1).

:- not attack.
```

Appendix E

The Publication List During the PhD Study

The following papers have been published, and contain materials based on the content of this thesis.

1. S. Wang and Y. Zhang. Answer set programming for distributed authorization: the language, computations, and application. In *Proceedings of the AI2005: 18 th Australian Joint Conference on Artificial Intelligence*, pp. 1191-1194, 2005.

2. S. Wang and Y. Zhang. Specifying distributed authorization with delegation using logic programming. In *Proceedings of the 9th International Conference on Knowledge-Based & Intelligent Information & Engineering Systems*, pp. 761-767, 2005.

3. S. Wang and Y. Zhang. A formalization of distributed authorization with delegation. In *Proceedings of the 10th Australian Conference on Information Security and Privacy*, pp. 303-315, 2005.

4. S. Wang and Y. Zhang. Handling distributed authorization with delegation through answer set programming. *International Journal of Information Security*, 6(1): 27-46, 2007.

Bibliography

[1] M. Abdalla, O. Chevassut, P. Fouque, and D. Pointcheval. A simple threshold authenticated key exchange from short secrets. In *Advances in Cryptology - Asiacrypt 2005*, pp. 566-584.

[2] M. Abadi and P. Rogaway. Reconciling two views of cryptography (The computaiotnal soundness of formal encryption). *Journal of Cryptology*, 15(2): 103-127, 2002.

[3] I. Agudo, J. Lopez and J. A. Montenegro. A representation model of trust relationships with delegation extensions. In *Proceedings of the 3th International Conference on Trust Management*, pp. 116-130, 2005.

[4] L. C. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic*, 2(4): 542-580, 2001.

[5] J. F. Allen. Maintaining Knowledge about Temporal Intervals. Communications of the ACM, 26(11):832-843, 1983.

[6] Chr. Anger, K. Konczak, Th. Linke and T. Schaub. A glimpse of answer set programming. *Künstliche Intelligenz*, 19(1): 12-17, 2005.

[7] V. Atluri, S. Jajodia, and B. George. *Multilevel Secure Transaction Processing.* Kluwer Academic Publishers, 1999.

[8] M. Backes and C. Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *STACS 2003*, pp. 310-329, 2003.

[9] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

[10] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau and J. Cowan. Extensible markup language (XML) 1.1 (Second Edition). World Wide Web Consortium (W3C). http://www.w3.org/TR/2006/REC-xml11-20060816/.

[11] M. Burrows, M. Abadi and R. Needham. A logic of authentication. In *Proceedings of the Royal Society, Series A 426(1871)*, pp. 233C271, 1989.

[12] D. E. Bell and L. J. LaPadula. Secure computer system: mathematical foundations. Technical Report ESD-TR-278, vol.1, 1973. The Mitre Corp.

[13] D. E. Bell and L. J. LaPadula. Secure computer system: unified exposition and multics interpretation. Technical Report ESD-TR-278, vol.4, 1973. The Mitre Corp.

[14] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology - Crypto 1993*, pp. 232-249, 1993.

[15] M. Bellare and P. Rogaway. Provably secure session key distribution - the three party case. In *Proceedings of the 27th ACM Symposium on the Theory of Computing*, pp. 57-66, 1995.

[16] M. Bellare and D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *Advances in Cryptology - Eurocrypt 2000*, pp. 139-155, 2000.

[17] 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)*, pp. 175-189, 1999.

[18] E. Bertino and E. Ferrari. Secure and selective dissemination of XML documents. *ACM Transaction on Information and Systems*, 5(3): 290-331, 2002.

[19] K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, 1977. The Mitre Corp.

[20] B. Blancher. A computationally sound mechanized prover for security protocols. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, 2006.

[21] B. Blancher and D. Pointcheval. Automated security proofs with sequences of games. In *Advances in Cryptology - Crypto 2006.*

[22] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the symposium on security and privacy*, pp. 164-173, 1996.

[23] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance-checking in the Policy-Maker trust management system. In *Proceedings of Second International Conference on Financial Cryptography (FC'98)*, pp. 254-274, 1998.

[24] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems. In *Secure Internet Programming*, pp. 185-210, 1999.

[25] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The keyNote trust-management system, Version 2, Internet Engineering Task Force RFC 2704, 1999. http://www.ietf.org/rfc/rfc2704.txt.

[26] R. Bourret. XML and Databases.
http://www.rpbourret.com/xml/XMLAndDatabases.htm.

[27] C. Boyd and J. M. González Nieto. Round-optimal contributory conference key agreement. In *PKC 2003*, pp. 161-174, 2003.

[28] E. Bresson, O. Chevassut, and D. Pointcheval. Provably authenticated group Diffie-Hellman key exchange - The dynamic case. In *Advances in Cryptology - Asiacrypt 2001*, pp. 209-223.

[29] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Advances in Cryptology - Eurocrypt 2001*, pp. 453-474.

[30] R. Canetti and M. Fischlin. Universally composable commitments. In *Advances in Cryptology - Crypto 2001*, pp. 19-40, 2001.

[31] CheckFree Corp. *Open Financial Exchange Specification 1.0.2*, 1998. http://www.ofx.net/.

[32] K. R. Choo. Errors in computational complexity proofs for protocols. In *Advances in Cryptology - Asiacrypt 2005*, pp. 624-643.

[33] K. R. Choo. Refuting security proofs for tipartite key exchange with model checker in planning problem setting. In *the 19th IEEE Computer Security Foundations Workshop - CSFW 2006*, pp. 297-308.

[34] Y. H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: trust management for web applications. *World Wide Web Journal*, 2(3): 127-139, 1997.

[35] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pp. 184-195, 1987.

[36] D. Clarke, J. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI, *Journal of Computer Security*, 9(4): 285–322, 2001.

[37] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Securing XML documents. In *Proceedings of 7th International Conference on Extending Database Technology*, pp. 121-135, 2000.

[38] E. Damiani, S. Vimercati, S. Paraboschi, and P. Samarati. Design and implementation of an access processor for XML documents. In *Proceedings of the 9th international WWW conference*, pp. 59-75, 2000.

[39] E. Damiani, S. Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security*, 5(2): 169-202, 2002.

[40] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. Flexible authentication of XML documents. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pp. 136-145, 2001.

[41] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transaction of Information Technology*, 29(2): 198-208.

[42] C. Ellerman. *Channel Definition Format (CDF)*, 1997. http://www.w3.org/TR/xlink.

[43] J. Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Masters Thesis, 1998. MIT LCS. http://theory.lcs.mit.edu/ cis/theses/elien-masters.ps.

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

[45] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Simple public key certificate, Internet Draft, 1999.

[46] D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proceedings of 15th NIST-NCSC National Computer Security Conference*, pp. 554-563, 1992.

[47] M.Gelfond and V.Lifschitz (1988) The stable model semantics for logic programming. *Logic Programming: Proc. of the Fifth Int'L Conf. and Symp.*, pp. 1070-1080.

[48] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Prodeedings of IEEE Symposium on Security and Privacy*, pp. 75-86, 1984.

[49] S. Goldwasser and S. Micali. Probabilistic Encryption. *Journal of Computer and System Sciences*, 128(3): 170-199, 1984.

[50] W. Faber, N. Leone, and G. Pfeifer. Experimenting with heuristics for answer set programming. In *Proceedings of IJCAI'01*, pp. 635-640, 2001.

[51] W. Faber, N. Leone, and G. Pfeifer. Optimizing the computation of heuristics for answer set programming system. In *Proceedings of LPNMR'01*, pp. 295-308, 2001.

[52] L. Gong, R. Needham and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of 1990 IEEE Symposium on Research in Security and Privacy*, pp. 234C248, 1990.

[53] M. Hietalahti, F. Massacci, and I. Niemelä. Des: A challenge problem for nonmonotonic reasoning systems. In *Proceedings of the International Workshop on Nonmonotonic Reasoning*, 2000.

[54] Y. Hitchcock, C. Boyd, and J. M. González Nieto. Tripartite key exchange in the Canetti-Krawczyk proof model (Extended version available from http://sky.fit.qut.edu.au/ boydc/). In *INDOCRYPT 2004*, pp. 17-32, 2004.

[55] M. Harrison, W. Ruzzo, and J. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8): 461-471, 1976.

[56] ITU-T Rec. X.509 (revised), The directory - authentication framework, International Telecommunication Union.

[57] S. Jajodia, P. S amarati, and V. S. Subrahmanian. A logic language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 31–42, 1997.

[58] S. Jajodia, P. Samarati, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2): 214-260, 2001.

[59] T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pp. 106-115, 2001.

[60] I. R. Jeong, J. Katz, and D. H. Lee. One-round protocols for two-party authenticated key exchange. In *ACNS 2004*, pp. 220-232, 2004.

[61] S. T. Kent. Internet privacy enhanced mail, *Communications of the ACM*, 36(8): 48-60, 1993.

[62] H. Krawczyk. HMQV: a high-performance secure Diffie-Hellman protocol. In *CRYPTO 2005*, pp. 546-566, 2005.

[63] A. Krokhin, P. Jeavons, P. Jonsson. Reasoning about Temporal Relations: The Tractable Subalgebras of Allens Interval Algebra. Journal of the ACM, 50(5):591-640, 2003.

[64] C. Kudla and K. G. Paterson. Modular security proofs for key agreement protocols. In *ASIACRYPT 2005*, pp. 549-569, 2005.

[65] B. W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18-24, 1974.

[66] 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*, pp. 162-174, 1999.

[67] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1): 35-86, 2003.

[68] N. Li and J. Mitchell. RT: A role-based trust-management framework. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, pp. 201-212, 2003.

[69] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pp. 114-130, 2002.

[70] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: a logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1): 128-171, 2003.

[71] F. Lin and R. Reiter, Forget it! In *Working Notes of AAAI Fall Symposium on Relevance*, pp. 154-159, 1994.

[72] G. Lowe. Some new attacks upon security protocols. In *Proceedings of the 9th IEEE Computer Security Foundatons Workshop*, pp. 162-169, 1996.

[73] J. McLean. Security models. In *Encyclopedia of Software Engineering.* John Wiley & Sons, 1994.

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

[75] NIST. *American National Standard 359-2004*. http://csrc.nist.gov/rbac/rbacSTD-ACM.pdf.

[76] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security* 6(1): 85-128, 1998.

[77] P. Resnick and J. Miller. PICS: Internet access controls without censorship. *Communications of the ACM*, 39(10): 87-93, 1996.

[78] R. L. Rivest and B. Lampson. SDSI - a simple distributed security infrastructure. 1996. http://theory.lcs.mit.edu/ rivest/sdsi11.html.

[79] J. Wu, J. Seberry, Y. Mu, and C. Ruan. Delegatable access control for fine-grained XML. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems*, pp. 270-274, 2005.

[80] C. Ruan and V. Varadharajan. Resolving conlicts in authorization delegations. In *Proceedings of the 7th Australian Conference on Information Security and Privacy*, pp. 271-285, 2002.

[81] C. Ruan and V. Varadharajan. A weighted graph approach to authorization delegation and conflict resolution. In *Proceedings of the 9th Australian Conference on Information Security and Privacy*, pp. 402-413, 2004.

[82] C. Ruan, V. Varadharajan and Y. Zhang. Logic-based reasoning on delegatable authorizations. In *Proceedings of the 13th International Symposium on Foundations of Intelligent Systems,* pp. 185-193, 2002.

[83] P. Ryan and S. Schneider. An attack on a recursive authentication protocol: a cautionary tale. *Information Processing Letters*, 65(15): 7-16, 1998.

[84] P. Samarati and S. De Capitani di Vimercati. Access control: policies, models, and mechanisms. *Foundations of Security Analysis and Design*, 2001.

[85] R. Sandhu. The typed access matrix model. In *Proceedings of the IEEE symposium on Research in SEcurity and Privacy,* pp. 122-136, 1992.

[86] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38-47, 1996.

[87] V. Shoup. On formal models for secure key exchange (Version 4) (Technical Report No. RZ3120(#93166)). IBM Research, Zurich.

[88] V. Shoup. OAEP reconsidered. In *Advances in Cryptology - Crypto 2001*, pp. 239-259.

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

[90] P. Syverson and P. van Oorschot. On unifying some cryptographic protocol logics. In Proceedings of 1994 IEEE Computer Security Foundations Workshop VII, pp. 14C29, 1994.

[91] J. Trevor and D. Suciu. Dynamically distributed query evaluation. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 28 - 39, 2001.

[92] A. Van Hoff, H. Partovi, and T. Thai. *The Open Software Description Format (OSD)*, 1997. http://www.w3.org/TR/NOTE-OSD.html.

[93] S. Wang and Y. Zhang. Answer set programming for distributed authorization: the language, computations, and application. In *Proceedings of the AI2005: 18 th Australian Joint Conference on Artificial Intelligence*, pp. 1191-1194, 2005.

[94] S. Wang and Y. Zhang. Specifying distributed authorization with delegation using logic programming. In *Proceedings of the 9th International Conference on Knowledge-Based & Intelligent Information & Engineering Systems*, pp. 761-767, 2005.

[95] S. Wang and Y. Zhang. A formalization of distributed authorization with delegation. In *Proceedings of the 10th Australian Conference on Information Security and Privacy*, pp. 303-315, 2005.

[96] S. Wang and Y. Zhang. Handling distributed authorization with delegation through answer set programming. *International Journal of Information Security*, 6(1): 27-46, 2007.

[97] T. Y.C. Woo and S. S. Lam, Authorization in distributed systems: a new approach. *Journal of Computer Security*, 2(2-3): pp. 107-136, 1993.

[98] Xindice 1.1 User Guide. Avalable at: http://xml.apache.org/xindice/guide-user.html.

[99] Y. Zhang. Two results for prioritized logic programming. *Theory and Practice of Logic Programming*, 3(2): 223-242, 2003.

[100] Y. Zhang and N. Foo. Solving logic program conflict through strong and weak forgettings. *Artificial Intelligence*, 170(8-9): 739-778, 2006.