Delegation Logic: A Logic-based Approach to Distributed Authorization

by

Ninghui Li

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September 2000

Joan Feigenbaum

Alan Siegel

To my parents and my wife

Acknowledgment

First and foremost, I must thank Joan Feigenbaum, my advisor, for her guidance, patience, and encouragement over these years. If I became a better writer or speaker because of my time as a Ph.D. student, it is largely because of her.

I owe a lot to Benjamin Grosof, whose help was very important to this thesis. I have benefited a lot from the last two years' collaboration with him.

I would also like to express my gratitude to Prof. Alan Siegel, my co-advisor, on whose recommendation I started working with Joan. He is always willing to help, both on academic development and on personal issues.

I am thankful to the Computer Science Department. Special thanks to Prof. Richard Cole, Prof. Ernest Davis, Prof. Edmond Schonberg, Prof. Robert Dewar, Anina Karmen, and Rosemary Amico.

I would also like to thank my friends and fellow students Gediminas Adomavicius, Fangzhe Chang, Hseu-Ming Chen, Chen Li, Archisman Rudra, Zhe Yang, and Tao Zhao for their help and friendship.

I owe much gratitude to my parents and my sister for their unconditional support and love. I am especially grateful to my dear wife, Yu Qiu, for her love at all times. She is always emotionally supportive, and she helped improve the presentation of this thesis.

Abstract

We address the problem of authorization in large-scale, open, distributed systems. Authorization decisions are needed in electronic commerce, mobile-code execution, remote resource sharing, content advising, privacy protection, *etc*. We adopt the trust-management approach, in which "authorization" is viewed as a "*proof-of-compliance*" problem: Does a set of credentials prove that a request complies with a policy?

We develop a logic-based language Delegation Logic (DL) to represent policies, credentials, and requests in distributed authorization. Delegation Logic extends logic programming (LP) languages with expressive delegation constructs that feature delegation depth and a wide variety of complex principals (including, but not limited to, k-out-of-n thresholds).

D1LP, the monotonic version of DL, extends the LP language Datalog with delegation constructs. D2LP, the nonmonotonic version of DL, also features classical negation, negation-as-failure, and prioritized conflict handling. Our approach to defining and implementing DL is based on tractably compiling DL programs into ordinary logic programs (OLP's). This compilation approach enables DL to be implemented modularly on top of existing technologies for OLP, *e.g.*, Prolog.

As a trust-management language, Delegation Logic provides a concept of proof-of-compliance that is founded on well-understood principles of logic programming and knowledge representation. DL also provides a logical framework for studying delegation, negation of authority, conflicts between authorities, and their interplay.

Contents

Chapter 1

Introduction

In today's Internet, there are a large and growing number of scenarios that require authorization decisions. Such scenarios include electronic commerce, execution of downloadable code (*e.g.*, Java applets [3] and ActiveX controls [17]), content advising [64], privacy protection [75, 76], remote resource sharing, *etc*.

Authorization in these Internet scenarios is significantly different from that in centralized systems or even in distributed systems that are closed or relatively small. In these older settings, authorization of a request is divided into *authentication* ("who made the request?") and *access control* ("is the requester authorized to perform the action?").

In Internet authorization scenarios, often there is no relationship between a requester and an authorizer prior to a request. Because the authorizer does not know the requester directly, it has to use information from third parties who know the requester better; normally, the authorizer trusts these third parties only for certain things and only to certain degrees. This trust and delegation aspect makes distributed authorization different from traditional access control. The goal of a growing body of work on *trust manage-*

ment [9, 11, 12, 13, 20, 24, 25, 49, 50] is to find a more expressive and "distributed" approach to authorization in these scenarios.

In the "trust-management" view of distributed authorization, a "*requester*" submits a request, possibly supported by a set of "*credentials*" issued by other parties, to an "*authorizer*," who controls the requested resources. The authorizer then decides whether to authorize this request by answering the "*proof-of-compliance*" question: "Do these credentials prove that a request complies with my local policy?"

The trust-management approach adopts a key-centric view of authorization, *i.e.*, it views public keys as entities to be authorized. Moreover, it supports credentials that endow public keys with more than just identities or "distinguished names" of key holders, *e.g.*, with agreed-upon "authorizations" [24], with various attributes of key-holders, or with fully programmable "capabilities" [9, 11, 20]. Identity information is just one kind of credentials, and it may be necessary and sufficient for some applications but not for others.

A major challenge in authorization and trust management is to provide a language that facilitates specification of authorization policies, especially one can be understood and modified by people who are expert in a business domain rather than in programming. Expressive convenience and separation of the policy specification semantics from the choice and details of implementation technique are thus important goals.

The goal of this dissertation is to provide such a "trust-management language " for representing authorization policies and credentials. Our design goals for this language include:

- It should have a declarative semantics and provide a notion of proof-of-compliance that is based on well-understood logical foundations.

- It should allow the specification of complex trust and delegation relationships in distributed authorization.

- It should strike a right balance among expressive power, computational complexity, and ease of understanding.

We view the problem of designing a language for representing authorization policies and credentials as a knowledge-representation problem. We further adopt an approach that has been proved very successful in knowledge representation: the logic-programming approach. Logic-programming-based languages for representing security policies have been studied before (*e.g.*, [6, 40, 41]), but previous work focused on centralized environments and did not address the delegation aspect of distributed authorization.

In this dissertation, we propose the logic-programming-based language Delegation Logic (DL) as a trust-management language. Our approach in designing Delegation Logic is to extend well-understood logic-programming languages with features needed in distributed authorization. Specifically, DL extends Definite Ordinary Logic Programs[1] along two dimensions: delegation and nonmonotonic reasoning. DL's delegation features include explicit linguistic support for delegation depth and for a wide variety of complex principals (*e.g.*, k-out-of-n thresholds). DL's nonmonotonic expressive features include classical negation, negation-as-failure, and prioritized conflict handling.

DL differs from other proposed trust-management languages [9, 11, 12, 24] in providing a notion of "proof-of-compliance" that is not *ad hoc*; rather, it is based on model-

---

[1]See page 26 for a brief introduction to terminology in logic programming (LP) and further references on LP.

theoretic semantics of logic programs (thus abstracted away from choice and details of implementation). Moreover, DL is a tractable and practically implementable approach.

In the next chapter, we give background information on authentication, access control, and trust management. In chapter 3, we describe the monotonic version of DL, called D1LP. D2LP, the version of DL that has nonmonotonic features, is described in chapter 4. We conclude in chapter 5.

Chapter 2

Authentication, Access Control, and Trust Management

Computer security aims at ensuring that access to resources is restricted to and available to those parties with legitimate access permissions. Three mutually supportive mechanisms together provide the foundation for achieving this goal: *authentication*, *access control*, and *audit*.

We abstract a system into *entities* that are inter-connected and *resources* that are controlled by entities. Entities may include users, operating systems, processes, threads, objects, *etc*. Resources may include information, files, network connections, methods of objects, *etc*. When an entity wants to access a resource controlled by another entity, it sends a *request* to that entity. The entity that wants to access the resource is called the *requester* and the entity that controls the resource is called the *authorizer*. Traditionally, when an authorizer receives a request, it first "identifies" the requester. This task of determining a requester's identity in a rigorous manner is called *authentication*. In other words, authentication answers the question "who made this request" with an identity. Knowing the identity of the requester, the authorizer then decides whether this identity is allowed to access the requested resource. This step is called *access control*. The *audit* process gathers data about activities in the system and analyzes it to discover security violations and diagnose their cause. Analysis can occur off-line after the fact or it can occur on-line more or less in real time.

We use the term *authorization* to denote this process of "authentication + access control." This dissertation focuses on authorization in emerging applications in large-scale, open, distributed systems (*e.g.*, Internet). Examples of such applications include execution of downloadable code (*e.g.*, Java applets [3] and ActiveX controls [17]), content advising [64], privacy protection [75, 76], remote resource sharing, *etc*.

Authorization in these applications is significantly different from traditional authorization. The goal of a growing body of work on *trust management* [9, 11, 12, 13, 20, 24, 25, 49, 50] is to find a more expressive and "distributed" approach to authorization in these scenarios.

In the rest of this chapter, we first give some background information on authentication and access control, then motivate and describe the trust-management approach. See [57] for a comprehensive treatment of modern cryptography and [29, 42, 69] for introductions to communication and network security.

2.1 Authentication

In this section, we briefly review authentication techniques. Here, we only consider *entity authentication*, also known as *identification* or *identity verification*. It allows one entity (the *verifier*) to gain assurance that the identity of another entity (the *claimant*) is as declared, thereby preventing impersonation.

User-to-local-host authentication

In a user-to-local-host authentication scenario, the claimant is a human user and the verifier is a process on a trusted local host. This is the typical authentication scenario in a single-host environment. It is also the first step of a user-to-remote-host authentication scenario, which can be conceptually separated into two steps: the user first authenticates to a process on a local computer, then this process authenticates to the remote host on behalf of the user.

User-to-local-host authentication is based on some secret only the claimant can generate. This secret may be a password that the claimant knows, a cryptographic key stored in some physical equipment that the claimant has (*e.g.*, a smart card), or some biometric of the claimant (*e.g.*, fingerprint, voice-print, *etc.*) One can also combine two or more of the above approaches, *e.g.*, a smart card combined with a PIN that is required to access it.

Direct remote authentication

By direct remote authentication, we mean the authentication between two processes running on two different hosts connected by an insecure network without the help of a trusted third party. It may be the case that one process is running on behalf of a user, and, if so, this is actually a user-to-remote-host authentication scenario.

In this kind of authentication, the claimant demonstrates knowledge of some secret, *e.g.*, a password, a cryptographic key, *etc*. This secret may be shared between the claimant and the verifier. In this case, the simplest (but insecure) way is to send the secret in clear text. More secure techniques include one-time passwords, challenge-response, *etc*. Alternatively, the secret may be known only to the claimant, and the claimant can convince the verifier without revealing the secret, *e.g.*, using public-key cryptography or some "zero-knowledge proof system."

Secret-key-based authentication with a trusted third party

Secure remote authentication typically requires the use of some cryptographic algorithms. Authentication schemes that are based on symmetric cryptography require two parties to share a secret key. In a distributed system that has N entities, each pair of entities needs a different key, and so there need to be $N(N-1)/2$ keys. Obviously, this doesn't scale well. Many protocols have been developed to address this problem. They all use the notion of a *key distributed center* (KDC), also known as an *authentication server*. KDC was first introduced by Needham and Schroeder in [59]. With every entity in the system, the KDC shares a different key. These keys are called *master keys*. The KDC acts as a trusted intermediary in the authentication of two entities. During an authentication process, a session key is established to allow the two parties to communicate securely. Using KDC reduces the number of required keys from $N(N-1)/2$ to N. The most widely known system based on this approach is Kerberos [45, 60].

Public-key-based authentication with a trusted third party

Authentication schemes based on secret-key cryptography and a KDC require every entity in the system to share a secret key with the KDC. This requires significant amount of initial setup for every new entity. Also, the KDC must always be on-line and highly available, because it is needed for every authentication. This may serve well for a local-area network environment; however, it does not scale to the size of Internet.

Public-key cryptography systems make key distribution easier. Every entity has a public/private key pair and the public key can be distributed freely. The verifier can easily authenticate a claimant if it knows the public key of the claimant. However, to be sure that a public key indeed belongs to a particular entity, trusted third parties, often known as *certification authorities (CA's)*, are still needed. CA's issue digital certificates. A digital certificate is a digitally signed data record containing a public key and some information about the key holder. Unlike KDC, CA's can stay off-line.

2.2 Identity-based Public-Key Infrastructures (PKI)

To enable the authentication of entities in a global distributed system, one CA is not enough; a global public-key infrastructure (PKI) is needed. Elements of a PKI include a naming scheme of the information that is bound to public keys, data structures of certificates, methods to distribute and revoke these certificates, and the intended meaning of certificates. A certificate can bind a public key to a piece of information directly; it can also specify (implicitly or explicitly) other trust relationships that can be used to deduce other bindings.

An important design question for any PKI is "what information is bound to public keys?" Earlier PKI proposals are designed for authentication and bind a globally unique identity to each public key. In this section, we give an overview of several existing identity-based PKI's.

X.509

The *Directory Authentication Framework*, known as X.509 [39], defines a format for digital certificates. In a certificate, a public key is bound to the distinguished name (DN) of the key holder. For a large community of users, multiple CA's are needed. These

CA's sign certificates for each other. When two users certified by different CA's want to authenticate each other, a chain of certificates is used. X.509 suggests that CA's be arranged in a hierarchy fashion, so that navigation among CA's is straightforward. When a chain of certificates is used to establish a DN-to-public-key binding, the user has to trust all the CA's along the chain in that they are all honest and competent to make correct bindings. Because the user may not know the CA's, the trust can only come from the certificate chain. Therefore, a CA-to-CA certificate vouches for more than just the DN-to-public-key binding; it also implicitly suggests the trustworthiness of the CA.

Internet Privacy Enhanced Mail (PEM)

Privacy Enhanced Mail [43] is a draft Internet standard for securing the Internet e-mail system. PEM uses X.509 certificates and implements a hierarchy of certification authorities. It defines three levels of CA's: Internet Policy Registration Authority (IPRA), Policy Certification Authorities (PCA's), and Certification Authorities (CA's). The root IPRA establishes a common policy that applies to all certificates issued under this hierarchy. Each PCA must register with the IPRA and file a document describing its own policies; the IPRA then signs these policies. A PCA ensures that CA's certified by itself must conform to its policies. These policies can reveal the degree of assurance when a CA vouches for name-to-public-key bindings. A CA may be certified by multiple PCA's, meaning that it may issue certificates under multiple policies.

Pretty Good Privacy (PGP)

Pretty Good Privacy (PGP) is a software package originally developed by Phil Zimmerman [77]. It provides cryptographic routines for e-mail and file storage. In PGP, a user is identified by a *User ID*, which is an e-mail address plus a name. A PGP certificate binds a public key to a User ID. This makes sense, because PGP is designed primarily for e-mail security.

A user can get the public key of another user through some secure channels (which may be outside cyberspace) or verify the public key through such channels. When there is no such channel, *introducers* (PGP's name for CA's) are needed. An introducer vouches for a user-ID-to-public-key binding by signing it. Contrary to the strict hierarchy model in PEM, every user can serve as an introducer in PGP. And every user can choose which introducers he trusts. This is often called the "Web of Trust" model.

In PGP, all keys and certificates are stored in key rings. Each user has two key rings. The private key ring stores the public/private key pairs owned by the user. The public-key ring stores public keys the user knows. In the public-key ring, each public-key entry has two fields governing trust of this public key. The *key legitimacy field* indicates the extent to which PGP trusts the binding is valid. This field is computed by PGP. The *owner trust field* indicates the degree to which the public key is trusted to serve as an introducer. This field is specified by the user who owns the key ring. Each public-key entry has zero or more signatures, and each signature has a *signature trust field* that is the same as the owner trust field of the signing key. The key legitimacy field of a key entry is derived from the collection of the signature trust fields of all signatures. A user can configure parameters to determine how many signatures are needed to trust a key blinding as valid.

A PGP certificate means only that the signer vouches for the binding between the User ID and the public key. There is nothing in a certificate to indicate to what extent the signer trusts the key-holder's ability to sign other keys. This trust information is stored in everyone's local key ring and is not revealed to other people. Therefore, a user can only use the public keys of people he knows directly to establish key bindings. Thus the "web of trust" is a very shallow (only two levels) web.

2.3 Access Control

The basic way to model access control is a four-tuple: (S, O, A, M), where S is the set of subjects, O is the set of objects, A is the set of actions (access rights), and M is a function that maps a tuple $(s, o, a) \in S \times O \times A$ to a decision $\in \{T, F\}$. The mapping M can be stored in an access matrix, with rows corresponding to subjects, columns corresponding to objects, and matrix entries indicating allowed access rights. In practice, a typical access matrix is large and sparse, and it is difficult to store, manage, and understand such a matrix directly. Therefore, various access-control policies have been developed.

Discretionary Access Control(DAC)

In the Trusted Computer System Evaluation Criteria (TCSEC) [44], two types of access-control policies are specified: discretionary access controls (DAC) and mandatory access controls (MAC). As defined in the TCSEC, DAC is

> A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the

> sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control).

DAC permits the granting and revoking of access privileges to be left to the discretion of individual users. This is based on the notion that individual users are "owners" of objects. Ownership is usually acquired as a consequence of creating the object.

Mandatory Access Control (MAC)

MAC is defined in TCSEC as

> A means of restricting access to objects based on sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (*i.e.*, clearance) of subjects to access information of such sensitivity.

The different security levels in a system form a lattice. MAC is typically used to enforce one-directional information flow in such a lattice. The rule for read access requires that a user with a given clearance level can only read information with the same or lower classification level. The rule for write access requires that a user with a given clearance level can only write information to a target with the same or higher classification level. This prevents a user from declassifying information without authorization.

Role-based Access Control (RBAC)

Recently, role-based access control (RBAC) [67] has emerged as a promising alternative to the two traditional classes of access-control policies.

The notion of role is central to RBAC. Permissions are associated with roles, and users are granted membership in appropriate roles, thereby acquiring the roles' permissions. More advanced RBAC models include role hierarchy and constraints. Roles are created for the various functions in an organization and are assigned to and revoked from users based on users' responsibilities and qualifications. The power of RBAC comes from the fact that the notion of "role" captures the way most organizations operate.

RBAC by itself is policy neutral. It can be used to implement MAC. The notion of roles makes the creation and modification of security policies easier so that security policies can more easily evolve incrementally over the system life cycle. Roles add another layer between principals and objects, thus helping to model the many-to-many relationships between principals and objects.

In RBAC, authorization is decided on the basis of which role the principal is associated with and which access rights the role has. Thus the full identity information of a requester may not be needed in access control. As long as the requester can provide proof that it is associated with the correct role, the request can be allowed.

The Clark and Wilson model

The Clark and Wilson model [21] recommends the enforcement of two principles: the principle of *well-formed transactions* and the principle of *separation of duty*. The concept of well-formed transactions is that a user should not manipulate data arbitrarily, but only in constrained ways that preserve or ensure the integrity of data. Separation of duty means that some transactions should not be completed by a single subject. This implies that any single subject should not have all the access rights required for such a transaction. When this principle is enforced, collusion among subjects is needed for fraud to take place.

Chinese Wall policy

The Chinese Wall policy [15] can be viewed as a kind of dynamic separation of duty. In the Chinese Wall policy, data objects are grouped into *datasets*, and datasets are grouped into *conflict-of-interest classes*. If a user accesses an object in one dataset, he cannot access any object in other datasets in the same conflict-of-interest class. For example, competing companies' datasets should be grouped into one conflict of interest class, and an analyst can only advise one of the companies in that class.

Access Control mechanisms

Techniques to implement the above access-control policies include the following:

- *Access Control Lists* (ACLs): An access-control list is an attribute of a target object, stating which users can invoke which actions on it. An access-control list specifies the contents of the column related to the target object in the access-control matrix.

- *Capabilities*: A capability is effectively a ticket, possessed by a requester, that authorizes the holder to access a specified object in specified ways. Some capabilities can only be used by a specified principal, while others may be transferred to other principals.

- *Security Labels*: A security label is a set of security-attribute information bound to a user, a target, or a piece of information in transmission. The label indicates the sensitivity level of the data. This mechanism is used to implement MAC.

Language-based approaches to access control

Recently, there has been considerable research interest in language-based approaches to access control [6, 19, 40, 41, 66, 72, 73]. The goal is to provide a language that can support multiple access-control policies and achieve separation of policies from mechanisms. Most work uses logic-programming (LP) languages [6, 40, 41]; some other work [66, 73] uses languages that can be easily translated to LP languages. One important issue is whether to allow negative authorizations. If they are allowed, then there are potential conflicts among security policies. These systems differ significantly in how to deal with conflicts.

2.4 SRC logic for authentication and access control

In distributed systems, authentication is complicated by the fact that a request may originate from a distant host and traverse multiple machines and network channels that are not trusted. Abadi, Burrows, Lampson, Plotkin, Wobber, *et al.* developed a logic for authentication and access control in distributed systems [1, 48]. They also designed and implemented a security system based on this logic.

The basic concepts of SRC logic are principals and statements. Principals make statements. A **says** S means that principal A makes the statement S (an assertion or a request). There is a "speak for" relation among principals. We write $A \Rightarrow B$ when A speaks for B; this means that, if principal A makes a statement, then we can believe that principal B makes it, too. Principals include:

- *Simple principals*. Users, machines, *etc*.

- *Channels*. Network addresses, encryption keys, *etc*. If S appears on channel

C then C **says** S. In particular, for a key K, K **says** S represents a message containing S and encrypted by K. A channel is the only kind of principal that can directly make a statement. The theory cannot reason about whether a channel has made a statement; it is taken into the theory as a fact and used to deduce other results.

- *Groups*. Set of principals. A group is treated as a disjunction of its members. Any member can speak for the group.
- *Conjunction of principals*. $A \wedge B$ stands for the conjunction of A and B. $(A \wedge B)$ **says** S means both A **says** S and B **says** S.
- *Principals quoting principals*. $B \mid A$ stands for B quoting A. If B **says** A **says** S then $(B \mid A)$ **says** S.
- *Principals in roles*. A **as** R stands for A in role R. A principal can adopt a role to reduce its rights.
- *Principals acting on behalf of others*. B **for** A stands for B acting on behalf of A.

Statements are defined inductively as follows:

- Primitive statements are statements.
- If s and s' are statements, then $s \wedge s'$ (s and s'), $s \supset s'$ (s implies s'), and $s \equiv s'$ (s is equivalent to s') are statements.
- If A is a principal and s is a statement, then A **says** s is a statement.
- If A and B are principals, then $A \Rightarrow B$ (A speaks for B) is a statement.

SRC logic has a *hand-off axiom*:

$$(A \textbf{ says } (B \Rightarrow A)) \supset (B \Rightarrow A)$$

This means that B speaks for A when A says so. Thus a principal A can delegate all its authority to another principal B by saying that B speaks for A. To limit the authority being delegated, a principal can adopt a role before delegating.

The notion of "speak for" is very useful in distributed authentication and access control. However, as a language for authentication and access control, SRC logic is rather limited in several aspects. First, primitive statements are propositions and are not interpreted by the logic; thus, they cannot state properties of principals for use in the logic. Second, SRC logic does not have variables. Because of these two limitations, one cannot express a delegation to any principal that has a certain property, without explicitly listing all the principals. Therefore, SRC logic is too limited for expressing attribute-based access-control policies. Third, there is no direct support for thresholds. Without this support, a delegation to a threshold of principals can only be implemented by multiple delegations to conjunctions of principals; however, doing this requires an exponential number of delegations in the worst case. Fourth, there is no re-delegation control mechanism; every delegation can be freely re-delegated.

Even with the above limitations, validity in SRC logic is undecidable in general. In [1], Abadi *et al*. defined two simplified classes of access-control problems that are decidable. One class is worst-case exponential-time solvable. The other class is the result of even further simplification and is computationally tractable.

2.5 Trust-Management

Authorization in Internet services is significantly different from traditional authorization in centralized systems or in closed distributed systems. Differences exist in the following aspects:

- **Who needs protection?**: In a traditional client/server computing environment, valuable resources usually belong to servers, and it is when a client requests access to a valuable resource that the server uses an authorization procedure to protect its resources. In a large-scale, open distributed system, users access many servers and have valuable resources of their own (*e.g.*, personal information, electronic cash); indeed "client" is no longer the right metaphor. Such a user cannot trust all of the servers it interacts with, and authorization mechanisms have to protect the users' resources as well as those of the servers.

- **Whom to protect against?**: In a large, far-flung network, there are many more potential requesters than there are in a smaller, more homogeneous (albeit distributed) system. Some services, *e.g.*, Internet merchants, cannot know in advance who the potential requesters are. Similarly, users cannot know in advance which services they will want to use and which requests they will make. Thus, the authorization mechanisms must rely on delegation and on third-party credential-issuers more than ever before.

- **Who stores authorization information?**: Traditionally, authorization information, *e.g.*, an access-control list, is stored and managed by the service. Internet services evolve rapidly, and thus the set of potential actions and the users who may request them are not known in advance; this implies that authorization information will be created, stored, and managed in a dynamic, distributed fashion. Users are often expected to gather all credentials needed to authorize an action and present them along with the request. Because these credentials are not always under the control of the service that makes the authorization decision, there is a danger that they could be altered or stolen. Thus, public-key signatures (or, more generally, mechanisms for verifying the provenance of credentials) must be part of the authorization framework.

In traditional authentication and access control, the notion of *identity* plays an important role. In a traditional system, an identity often means an existing user account. User accounts are established with the system prior to the issue of any request. Earlier PKI proposals try to establish a similar global "user-account" system that gives a unique name to every entity in the system and then binds each public key to a globally unique "identity."

In Internet applications, the very notion of *identity* becomes problematic. The term *identity* originally means sameness or oneness. When we meet a previously unknown person for the first time, we cannot really identify that person with anything. In a scenario in which an authorizer and a requester have no prior relationship, knowing the requester's name or identity may not help the authorizer make a decision. One can argue that, in a global system, the only real "identity," with which anything can later be related, is the public key. Thus, the "trust-management approach" adopts a key-centric approach to authorization: Public keys are treated as principals and authorized directly.

Moreover, the trust-management approach supports the use of more expressive credentials that endow public keys with more than just identities. Credentials can bind public keys to agreed-upon "authorizations" [24], to various attributes of key-holders, or to fully programmable "capabilities" [9, 11, 20]. Authorization is framed as a *proof-of-compliance* problem: "Does the set C of *credentials* prove that the *request* r *complies* with the local security *policy* P?" Furthermore, the "trust-management approach" demands that the notion of proof-of-compliance be application-independent.

The "trust-management approach" also adopts a "peer model" of authorization. Every entity can be an authorizer, a third-party credential issuer, or a requester. An entity can act as a requester in one authorization scenario and as an authorizer (or as a third-party credential issuer) in another.

In the following, we briefly review several trust-management systems (see [13] for a more detailed survey).

2.5.1 PolicyMaker

PolicyMaker was the first "trust-management system." It was introduced in the original trust-management paper by Blaze, Feigenbaum, and Lacy [9], and its compliance-checking algorithm was later fleshed out in [11]. In this section, we give a high-level overview of the PolicyMaker system. Our description is mostly based on [11, 13]. For more details about PolicyMaker, see [9, 11, 13]. Experience using PolicyMaker in several applications is reported in [10, 46, 47].

In PolicyMaker, there are policies and credentials. They differ only in their issuers. Policies are issued by a special principal, **POLICY**, which represents local authority and is the "trust root" for granting rights. All other principals are public keys. Public keys can issue credentials as well as request for rights. Policies and credentials together are referred to as "*assertions*." An assertion is represented as a pair (f, s), where s is the source of authority (*i.e.*, the issuer of this assertion) and f is a program describing the nature of the authority being granted as well as the party or parties to whom the authority is being granted. Credentials must be signed by their issuers, and these signatures must be verified before the credentials can be used. PolicyMaker assertions can be written in any programming language that can be "safely" interpreted by a local environment. A safe version of AWK was developed for early experimental work on PolicyMaker (see [9]).

In the PolicyMaker framework, applications are responsible for collecting certificates and translating them to PolicyMaker assertions. After collection and translation is done, the application calls the PolicyMaker inference engine with a query and a set of policies and credentials. The PolicyMaker inference engine provides an environment in which the policies and credential assertions can cooperate to produce a proof that the request complies with the policies (or can fail to produce such a proof). This environment provides a method of inter-assertion communication and a method for determining that assertions have collectively succeeded or failed to produce a proof.

Inter-assertion communication in PolicyMaker is done via a simple, append-only data structure on which all assertions write intermediate results. PolicyMaker initializes the proof process by creating a "blackboard" containing only the request string r. Then PolicyMaker runs the various assertions, possibly multiple times each. When an assertion (f_i, s_i) is run, it reads the contents of the blackboard and then adds to the blackboard zero or more *acceptance records* (i, s_i, R_{ij}). Here, R_{ij} is an application-specific action that source s_i approves, based on the acceptance records previously written on the blackboard.

A proof of compliance is achieved if, after PolicyMaker has finished running assertions, the blackboard contains an acceptance record indicating that a policy assertion approves the request r. The paper [11] provides: (1) a mathematically precise formulation of the PolicyMaker compliance-checking program, (2) proof that the problem is undecidable in general and is NP-hard even in certain natural special cases, and (3) one special case of the problem that is polynomial-time solvable, is useful in a wide variety of applications, and is implemented in the current version of PolicyMaker.

2.5.2 KeyNote

KeyNote [12] was designed according to the same principles as PolicyMaker. KeyNote requires that credentials and policies be written in a specific assertion language.

A calling application passes to a KeyNote evaluator a list of credentials, policies, requester public keys, and an "Action Environment," which consists of a list of attribute/value pairs, similar in some ways to the $Unix^{TM}$ shell environment. The action environment is constructed by the calling application and contains all information that is relevant to the request and necessary for the trust decision. The action-environment attributes and the assignment of their values must reflect the security requirements of the application accurately. Identifying the attributes to be included in the action environment is perhaps the most important task in integrating KeyNote into new applications. The result of the evaluation is an application-defined string (perhaps with some additional information). This string is passed to the application. In the simplest case, it is something like "authorized."

The KeyNote assertion format resembles that of e-mail headers. As in PolicyMaker, policies and credentials (collectively called assertions) have the same format. A KeyNote assertion has a *Licensees* field. It specifies explicitly the principal or principals to which the authority is delegated. Syntactically, the Licensees field is a formula in which the arguments are public keys and the operations are conjunction, disjunction, and threshold. The semantics of these expressions is specified in [12].

2.5.3 SPKI/SDSI

SPKI (Simple Public Key Infrastructure) and SDSI (Simple Distributed Security Infrastructure) were started independently. Both of them were motivated by the inadequacy of public-key infrastructures based on global name hierarchies, such as X.509 [39] and Privacy Enhanced Mail (PEM) [43]. Later, SPKI and SDSI merged into a collaborative effort, SPKI/SDSI 2.0, about which the most up-to-date documents are [23, 25, 26].

The SPKI/SDSI approach shares many views with the trust-management approach. For example, it aims at developing a standard form for digital certificates whose main purpose is authorization rather than authentication. However, it does not define an application-independent trust-management engine. In [25], Ellison *et al.* state: "The processing of certificates and related objects to yield an authorization result is the province of the developer of the application or system." Therefore, strictly speaking, SPKI/SDSI is not a trust-management system.

SPKI/SDSI adopts the *localized naming scheme* of SDSI. In SDSI, there are *principals* and *local identifiers*. Principals are public keys and are therefore unique. Principals make statements. Each principal has its own name space. *Names* are formed by linking principals and local identifiers. For example, "keyAlice's Bob's friend" is a name. A *local name* is a principal followed by a local identifier. A *subject* is either a name or an object of the form: "`(k-of-n` K N sub_1 sub_2 `...` sub_N`)`," where K and N are both positive integers and $K \leq N$; each sub_i is a subject.

SPKI/SDSI 2.0 has two kinds of certificates. *Name-definition certificates* (*name* *certs* for short) came originally from SDSI; a name cert binds a local name to a principal or a more complex name. Name certs are used to resolve names to principals.

Authorization certificates (*auth certs*) came originally from SPKI; an auth cert delegates a certain permission from a principal (the cert's issuer) to the cert's subject. An auth certificate has the following five fields: "Issuer" (the principal that signs this cert), "Subject" (the entity being authorized in this cert), "Authority" (the specific permission being delegated in this certificate), "Delegation" (a boolean value to indicate whether the subject can further delegate the authority it gets from this certificate), "Validity" (validity period and/or checking methods). An auth cert represents a 5-tuple. Informally, a 5-tuple means "The issuer says that the subject has the stated authority during the validity period." The trust reasoning in SPKI consists of finding a delegation chain that delegates the authority from the original issuer to the final subject. For the delegation chain to work, the authority being delegated must be the intersection of all the authorities in the 5-tuples along the chain, and the delegation fields of all 5-tuples except the last one on the chain have to be true.

Chapter 3

A Monotonic Delegation Logic

In this chapter, we present a monotonic Delegation Logic that we call D1LP. It stands for version 1 of Delegation Logic Programs. We use the term D1LP to denote both the formalism and a program in this formalism. D1LP extends Datalog Definite Ordinary Logic Programs by adding an issuer to every atom and adding delegation constructs that have pre-defined meanings. "Ordinary" logic programs (OLP's) correspond essentially to pure Prolog, but without the limitation to Prolog's particular inferencing procedure. These are also known as "general" LP's (a misleading name, because there are many further generalizations of them) and as "normal" LP's. "Definite" means without negation. "Datalog" means without function symbols of non-zero arity. The "arity" of a function symbol is the number of parameters it takes. For reviews of standard concepts and results in logic programming, see [5, 14, 53].

We define two versions of D1LP; they have different expressive power and computational complexities.

- *Standard D1LP* (or just D1LP), first defined in [50], satisfies the *conjunctive-delegatee-queries* restriction: A delegation statement appearing in a query or a rule-body (because it is implicitly a query) must have a delegatee that is a principal, a principal variable, or a conjunction of principals. That is, such a delegatee is not permitted to contain a disjunction or a threshold structure (which is disjunctive in nature). This restriction is imposed to ensure computational tractability of standard D1LP. This rationale is discussed in detail in section 3.4.

- The *delegation-query-free D1LP* (D1LP^{DQF}) is further restricted from standard D1LP by completely forbidding any delegation statement to appear in a query or a rule-body. We introduce D1LP^{DQF} because it has simpler semantics and lower computational complexity than standard D1LP and still has sufficient expressive power for many applications. D1LP^{DQF} is also the foundation upon which we define the version of D2LP in this dissertation.

This chapter is organized as follows. In section 3.1, we describe the syntax, some concepts, and some examples of D1LP. In section 3.2, we define the semantics of D1LP via a transformation from D1LP into OLP. D1LP inferencing is accomplished by the combination of this transformation plus OLP inferencing. Tractability results are given in section 3.3. In section 3.4, we discuss the tractability motivation behind the conjunctive-delegatee-queries restriction for standard D1LP. In section 3.5, we describe two implementations of D1LP. We present D1LP^{DQF} in section 3.6.

3.1 Syntax, Concepts, and Examples

In this section, we define the syntax of D1LP, explain some important concepts in D1LP, and give examples of D1LP programs.

3.1.1 Syntax

1. The *alphabet* of D1LP consists of three disjoint sets, the *constants*, the *variables*, and the *predicate symbols*. Variables start with '?'. The set of *principals* is a subset of the constants. The set of principals should be distinguishable from other constants. There is a reserved principal symbol "`Local`"; it represents the trust root, *i.e.*, the authorizer of an authorization decision. When a variable appears in certain positions, *e.g.*, as an issuer, it is called a *principal variable* and can only be instantiated to a principal. A *term* is either a constant or a variable. A *principal term* is either a principal or a principal variable.

   Note that we prohibit function symbols with non-zero arities: This is the *Datalog* restriction. The Datalog restriction helps to ensure finiteness of D1LP semantics and tractability of D1LP inferencing. It can be weakened, however. See section 3.3.2 for further discussion of the relationship between the Datalog restriction and tractability of D1LP inferencing.

2. A *base atom* takes the form:

   $$pred(t_1, ..., t_n)$$

   where $pred$ is a predicate and each t_i is a term.

   A base atom encodes a belief. For example, "$\mathrm{isBusinessKey(keyBob,Bob)}$" and "$\mathrm{goodCredit(?X)}$" are base atoms that encode beliefs. When a belief talks about a security action, *e.g.*, an action to access a resource, it means a belief that this action should happen. For example, the base atom "$\mathrm{remove(file1)}$" encodes the belief that file1 should be removed.

3. A *direct statement* takes the form:

$$X \texttt{ says } ba$$

where X is a principal term, "$\texttt{says}$" is a keyword, and ba is a base atom. X is called the *issuer* of this statement.

A direct statement "$X \texttt{ says } ba$" intuitively means that X "*supports*" the belief encoded in ba. For example, "$\mathrm{keyBob\ says\ goodCredit(Carl)}$" means that keyBob supports (believes) that Carl has good credit. When ba encodes a belief about a security action, the direct statement "$X \texttt{ says } ba$" means that X believes that the action encoded in ba should happen, *i.e.*, X requests the action; therefore, this direct statement encodes a request by its issuer X. For example, one can use the direct statement "$\mathrm{keyTom\ says\ purchase(computer,\ 1500)}$" to encode that the principal keyTom requests to purchase a computer priced at $\$1500$.

4. A *static unweighted threshold structure* takes the form:

$$\texttt{threshold}\,(k,\,[A_1, \dots, A_n])$$

where "$\texttt{threshold}$" is a keyword, k is a positive integer, A_i's are principals, and $A_i \neq A_j$ for $1 \leq i \neq j \leq n$. We call k the *threshold value* and "$[A_1, \dots, A_n]$" the *threshold pool*.

For example, "$\mathrm{threshold(2,\ [cardA,\ cardB,\ cardC])}$" is a static unweighted threshold structure. It supports a base atom ba if at least two principals among the threshold pool "$\mathrm{[cardA,\ cardB,\ cardC]}$" support ba.

Static unweighted threshold structures are often known as "k-out-of-n thresholds." They are common in many existing authorization systems, *e.g.*, PolicyMaker [9, 11], KeyNote [12], and SPKI/SDSI [24, 25, 65].

5. A *static weighted threshold structure* takes the form:

$$\texttt{threshold}(k, [(A_1, w_1), \ldots, (A_n, w_n)])$$

where k and the A_i's are the same as in item 4, and the w_i's are positive integers. We call these w_i's *weights* and "$[(A_1, w_1), \ldots, (A_n, w_n)]$" a *principal-weight pair set*. This set is the threshold pool of this threshold structure.

Weighted threshold structures enable the assignment of different weights to different principals in the threshold pool. Such a threshold structure supports a base atom if the sum of the weights of all those principals that support the base atom is greater than or equal to the threshold value k.

6. A *dynamic unweighted threshold structure* takes the form:

$$\texttt{threshold}(k, ?X, Prin\ \texttt{says}\ pred(\ldots ?X \ldots))$$

where the threshold value k is an integer, $?X$ is a principal variable, and "$Prin$ $\texttt{says}$ $pred(\ldots ?X \ldots)$" is a direct statement in which the variable $?X$ occurs (one or more times).

Such a threshold structure has a *dynamic threshold pool* that is the set of all principals A such that the direct statement "$Prin$ $\texttt{says}$ $pred(\ldots A \ldots)$" is true, *i.e.*, the expression "$Prin$ $\texttt{says}$ $pred(\ldots ?X \ldots)$" becomes true when A is substituted for $?X$ for each appearance throughout the direct statement.

The reason for introducing dynamic threshold structures is that a static threshold structure becomes inconvenient when its threshold pool is very large, changes very often, or both. For example, suppose that a bank requires that two cashiers cooperate to do a certain transaction. This can be implemented by delegating the right to do this transaction to a threshold structure that represents two out of all the cashiers. Use of a dynamic unweighted threshold structure yields a simple and clear policy and enables the company to change the set of cashiers without changing its policy.

7. In D1LP, a *threshold structure* is one of the above three kinds of threshold structures. Threshold structures introduce fault tolerance and aid flexibility in joint authorization.

8. A *principal structure* is constructed from principals and threshold structures using "," (conjunction), ";" (disjunction), and parentheses.

   For example, the principal structure

   $$\begin{array}{l}(\mathrm{threshold}(1,\ ?\mathrm{X},\ \mathrm{companyA\ says\ accountant}(?\mathrm{X})),\\ \ \mathrm{threshold}(1,\ ?\mathrm{Y},\ \mathrm{companyA\ says\ manager}(?\mathrm{Y})))\end{array}$$

   represents any conjunction of an accountant in $\mathrm{companyA}$ and a manager in $\mathrm{companyA}$. Such principal structures are useful in common separation-of-duty policies.

9. A *delegation statement* takes the form:

   $$X\ \texttt{delegates}\ ba\hat{}d\ \texttt{to}\ PS$$

   where X is either a principal or a principal variable, "`delegates`" and "`to`" are keywords, ba is a base atom, d is either a positive integer or the asterisk symbol "$*$", and PS is a principal structure or a principal variable.

   X is called the *issuer* of this statement; d is called the *delegation depth* of this delegation ("$*$" means unlimited depth); and PS is called the *delegatee* of this delegation. In DL, the basic meaning of a delegation is *transferability of support*. For example, the delegation statement

$$\text{Bob delegates goodCredit(?X)\^{}1 to Carl}$$

means that Bob trusts Carl about whether someone has good credit. The meaning of delegation depths is discussed in section 3.1.2.

10. A *speaks_for statement* takes the form:

$$Y \texttt{ speaks\_for } X \texttt{ on } ba$$

where X and Y are either principals or principal variables, and ba is a base atom. The issuer of a speaks_for statement is defined to be the special principal `Local`.

A speaks_for statement "Y `speaks_for` X `on` ba" intuitively means that Y has all power that X has with respect to the base atom ba, *i.e.*, whoever trusts X about ba should trust Y equally about ba. It is similar to the delegation statement "X `delegates` baˆ$*$ `to` Y," but there are important differences. The rationale for speak_for statements is discussed in section 3.1.4.

11. A *head statement* is a direct statement, a delegation statement, or a speaks_for statement.

12. A *body statement* is a body-direct statement, a body-delegation statement, or a speaks_for statement.

13. A *body-direct statement* is more general than a direct statement in that it permits the issuer to be a principal structure.

For example, the following is a body-direct statement:

$$\text{threshold(2, [cardA, cardB, cardC]) says accountGood(?X).}$$

14. A *body-delegation statement* is more general than a delegation statement in that it allows the issuer to be a principal structure but less general in that it obeys the conjunctive-delegatee-queries restriction: Its delegatee must be a principal, a principal variable, or a conjunction of principals.

15. A *body formula* is constructed from body statements using "," (conjunction), ";" (disjunction), and parentheses.

16. A *rule*, also known as a *clause*, takes the form:

$$H \texttt{ if } F.$$

where H is a head statement and F is a body formula. H is called the *head* of the rule, and F is called the *body* of the rule. The body may be empty; if it is, the keyword "`if`" is omitted. A rule with an empty body is also called a *fact*.

If H has a principal as its issuer, then this principal is also the *issuer* of this rule. Otherwise H has a principal variable as its issuer, then the *issuer* of this rule is the principal symbol `Local`.

For example, the issuer of the rule "$\mathrm{A\ says\ p\ if\ B\ says\ q}$" and the rule "$\mathrm{A\ delegates\ p\hat{\ }1\ to\ B}$" is A. The issuer of the rule "$\mathrm{A\ speaks\_for\ B\ on\ p}$ and the rule "$\mathrm{?X\ says\ p(a)\ if\ Local\ says\ c(?X,a)}$" is `Local`. Intuitively, the issuer of a rule is the principal who has the power to issue that rule. This is further discussed in sections 3.1.3 and 3.1.4.

17. A *program* is a finite set of rules. This is also known as a *logic program (LP)* or as a *rule-set*.

18. A *query* takes the form "F?" where F is a body formula.

As usual, an expression (*e.g.*, term, base atom, statement, clause, or program) is said to be *ground* if it does not contain any variables. A clause with variables stands for all its ground instantiations.

3.1.2 Discussions of delegation depth

As we defined earlier, a delegation statement has a depth, which is either a positive integer or "$*$." In this subsection, we discuss the intuition behind delegation depths.

One reason to limit delegation with respect to depth is that trust is not transitive (see [28, 56] for some interesting discussion). For example, the delegation statement

$$\mathrm{Bob\ delegates\ goodCredit(?X)\hat{}1\ to\ Carl}$$

means that Bob trusts Carl about whether someone has good credit. That is, if Carl says that someone has good credit, then Bob believes it. However, even if Bob trusts Carl about $\mathrm{goodCredit(?X)}$ and Carl trusts David about $\mathrm{goodCredit(?X)}$, Bob doesn't necessarily trust David about $\mathrm{goodCredit(?X)}$. It is imaginable that Bob trusts Carl's ability to judge whether someone has good credit, but doesn't trust Carl's ability to judge other principals' ability to judge whether someone has good credit. In this case, Bob should only delegate to Carl with depth 1. If Bob does trust the principals that Carl trusts, then Bob should delegate to Carl with depth at least 2. Whether Bob should delegate to Carl with depth 3 or higher depends on whether Bob trusts Carl's ability to judge other principals' ability to judge other principals' trustworthiness about $\mathrm{goodCredit(?X)}$. In some cases, Bob trusts Carl completely, and then Bob can delegate to Carl with depth $*$. Under this intuition, each integer depth has a distinct meaning and a larger depth conveys more trust than a smaller depth. Depth greater than 4 seems too abstract to be useful, but using depth 2 or 3 in some scenarios should be plausible.

From another point of view, delegation depth can be viewed as the number of re-delegation steps that are allowed. Consider the following delegation statement:

$$\text{Bob delegates read(file1)\^{}1 to Carl.}$$

It means that Bob delegates to Carl the permission to read file1. Given this delegation, when Carl makes a request, one can believe that Bob also supports this request. If Bob allows Carl to further delegate this permission one more step, Bob can delegate to Carl with depth 2. If Bob allows Carl to freely re-delegate this permission, Bob can delegate to Carl with depth $*$. Following is an example of using delegation depth to control re-delegation.

Example 3.1 (Delegation depth).

Given the following statements:

Alice delegates orgMember(?X)^2 to Bob.

Bob delegates orgMember(?X)^1 to Carl.

Carl delegates orgMember(?X)^1 to David.

Carl says orgMember(Jack).

David says orgMember(John).

one can infer in our DL semantics (section 3.2):

Alice delegates orgMember(?X)^1 to Carl.

Alice says orgMember(Jack).

Bob says orgMember(Jack).

Carl says orgMember(John).

but not:

Bob delegates orgMember(?X)^1 to David.

Bob says orgMember(John).

This is because Alice delegates to Bob with depth 2, but Bob only delegates to Carl with depth 1.

In DL, a depth-d delegation from A to B about a base atom ba implies all delegations of depth smaller than d from A to B about ba. In particular, a depth-$*$ delegation implies all integer-depth delegations. This is supported by both of the above two intuitive interpretations of depth.

DL uses both integer and unlimited depth to control re-delegation. There are three other approaches: (1) no control, in which every delegation can be freely re-delegated, (2) boolean control, in which a delegation allows either totally free re-delegation or no re-delegation at all, and (3) integer control (no $*$ option). In section 4 of [25], SPKI designers discussed these three approaches. They chose to use boolean control and gave the following two reasons for choosing boolean control over integer control (in section 4.1.4 of [25]):

> The integer control option was the original design and has appeal, but was defeated by the inability to predict the proper depth of delegation. One can always need to go one more level down, by creating a temporary signing key (*e.g.*, for use in a laptop). Therefore, the initially predicted depth could be significantly off.
>
> As for controlling the proliferation of permissions, there is no control on the width of the delegation tree, so control on its depth is not a tight control on proliferation.

When integer control is enhanced with $*$, the first reason doesn't hold anymore. One can always use depth-$*$ when the proper depth cannot be predicted. On the other hand, it is imaginable that one does have an accurate prediction of delegation depth in some scenarios. Moreover, in scenarios in which one needs to create a temporary signing key for laptop use, the boolean approach seems to be more inappropriate than the integer approach. Suppose that a user is originally given the permission to do something but not to re-delegate that permission and the user wants to empower a temporary key. To enable this, in the boolean approach, the user has to be given full power to re-delegate freely; in the integer approach, however, the user just needs a delegation depth that is one larger.

We do not agree with the second reason either. We can't see why one should give up depth control because width control is difficult. Moreover, as we will see in example 3.5 (page 44), it is possible to control the width of a delegation in DL, by using a conjunction of principals as its delegatee.

We think that it is still unclear how re-delegation should be handled in practice. However, we believe that it is important to work out the details of each approach. This will allow one to gain more insights into these approaches, to analyze the cost of each approach, and to test them in practice. We choose to use both integer depth and unlimited depth mainly because they are strictly more expressive than any of the other three approaches. Re-delegatabilities being "true" and "false" in the boolean approach can be represented in DL by depth $*$ and 1, respectively, but integer depths like 2 or 3 cannot be represented in boolean approach. We conjecture that delegation depths such as 2 and 3 might prove useful and natural in many practical policies.

3.1.3 Using DL in authorization scenarios

In this subsection, we discuss how DL is used in authorization scenarios. Entities in authorization scenarios are represented by principals in DL. These principals issue credentials and requests. Typically, a principal in distributed authorization is a public/private key pair. Such a principal issues a credential or a request by digitally signing a message that contains it.

When an authorizer gets a request and some credentials that support this request, this authorizer creates a query $\mathcal{Q}$ from this request and a DL program (rule-set) $\mathcal{P}$ from the combination of the request, the credentials, and the authorizer's local policies. Policies and credentials are translated into rules in DL. During the translation, the authorizer should verify that each rule is indeed made by its issuer. A rule with a principal as its issuer should be encoded in a credential that is signed by the rule's issuer. A rule with `Local` as its issuer should come from a local policy. Policies that are securely stored locally do not need to be signed. Having a program $\mathcal{P}$ and a query $\mathcal{Q}$, the authorizer decides whether to authorize this request by inferring whether the query $\mathcal{Q}$ is true relative to the program $\mathcal{P}$. DL's semantics gives a proof procedure to answer $\mathcal{Q}$ relative to $\mathcal{P}$. Consider the following example:

Example 3.2 (Determining credit status).

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 whoever 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:

$\mathrm{ShopA\ says\ approveOrder(?X)\ \ if\ \ ShopA\ says\ creditRating(?X,\ good).}$

$\mathrm{ShopA\ delegates\ creditRating(?X,\ ?R)\hat{\ }2\ to\ BankB.}$

BankB says creditRating(?X, good)

if threshold(2,[cardW,cardX,cardY]) says accountGood(?X).

Now a customer Carl sends an order to ShopA and provides the following two credentials: "cardX says accountGood(Carl)" and "cardY says accountGood(Carl)." ShopA then generates a new program consisting of the above rules and queries it with "ShopA says approveOrder(Carl)?" According to the DL semantics, the answer is true, and so ShopA should authorize this request.

Suppose that another customer David also sends an order and provides the following two credentials: "cardY says accountGood(David)" and "cardZ says accountGood(David)." ShopA will decline this request because only one principal in "[cardW, cardX, cardY]" supports accountGood(David).

Note that the above process of generating the program and inferencing is done from the authorizer ShopA's point of view. In DL, there is always a single, distinguished viewpoint: the viewpoint of the principal who is doing reasoning and making authorization decisions, *i.e.*, the current trust root, referred to as "`Local`." Note that `Local` is a special symbol that refers to the current trust root and, in particular, that it is not another principal. For example, `Local` in example 3.2 is ShopA.

Now let us step through ShopA's reasoning process in example 3.2 to derive "BankB says creditRating(Carl, good)." First, ShopA believes "cardX says accountGood(Carl)" and "cardY says accountGood(Carl)," because these two facts are signed by their respective issuers. To ShopA, the rule issued by BankB means that: "If, for some principal x, I (ShopA) believe that at least two principals in [cardW, cardX, cardY] support accountGood(x), then I (ShopA) should also believe that 'BankB says creditRating(x, good)'." From this rule and the two facts, ShopA concludes that

"BankB says creditRating(Carl, good)."

The above reasoning process is from ShopA's point of view; however, this viewpoint is not that different from that of any other principal. Because credentials are signed by their issuers, any principal that sees the above three credentials should believe that "BankB says creditRating(Carl, good)." When a principal X signs and distributes a rule "X says p if Y says q," X means: "To whoever sees this credential, if you believe that 'Y says q,' then you can also believe that 'X says p'."

3.1.4 Discussion of speaks_for statements

Now we discuss the differences between a speaks_for statement "Y `speaks_for` X `on` ba" and a depth-$*$ delegation statement "X `delegates` ba`^`$*$ `to` Y."

First, the speaks_for statement is strictly stronger than the delegation statement. If "Y `speaks_for` X `on` ba," then Y has all the power that X has on ba even if X is not allowed to re-delegate this power. In other words, conclusions drawn from a speaks_for statement don't consume any delegation depth.

For example, given the following rules:

> Alice delegates read(file1)^1 to Bob.
>
> keyBob speaks_for Bob on read(?File).
>
> keyBob says read(file1).

one can conclude that "Alice says read(file1)." But if one changes the second statement to:

> Bob delegates read(?File)^$*$ to keyBob.

then one can no longer conclude that "Alice says read(file1)," because Alice only delegates to Bob with depth 1.

Another difference is that the issuer of a speaks_for statement is always the principal `Local`, *i.e.*, the trust root. A principal B can issue a statement saying that B delegates to C. But B cannot say that C speaks_for B. Allowing a principal to say that another principal speaks_for it would circumvent depth restriction on delegations, and so speaks_for statements can only be made by the trust root. (Note that a rule whose head is of the form "?X says ..." or "?X delegates ..." also has `Local` as its issuer. Such a rule too should only be made by the trust root.)

One main reason for having speaks_for statements is to handle delegations to principals that cannot make (*i.e.*, sign) statements directly, *e.g.*, distinguished names in X.509 or local names in SPKI/SDSI. Because these names cannot sign statements, it makes sense to allow only `Local` to determine that a principal speaks_for another principal, although `Local` may rely on information from other principals to make this decision.

In example 3.2, there is a credential issued by BankB about $\mathrm{creditRating(?X, good)}$. In many scenarios, BankB is a name and cannot issue statements; the credential is most likely signed by a key of BankB. Let us call this key $\mathrm{keyBankB}$. Assume that ShopA also knows that $\mathrm{keyBankB}$ is BankB's public key for business purposes, *i.e.*, ShopA has the following statement:

$$\mathrm{ShopA\ says\ isBusinessKey(keyBankB,\ BankB).}$$

Then, by adding the following statement, ShopA can derive the same conclusions as in example 3.2.

$$\begin{array}{l} \mathrm{?Key\ speaks\_for\ ?X\ on\ creditRating(?Y,\ ?Z)} \\ \qquad \mathrm{if\quad Local\ says\ isBusinessKey(?Key,\ ?X).} \end{array}$$

Using a speaks_for statement, one can delegate a certain permission to the name of an entity and separate this delegation from the binding of a key with this name.

DL's speaks_for notion is similar to the speaks for notion in [1, 48]. However, there are two differences. First, DL's speaks_for relation is defined on a per-base-atom basis. Principal B may speak for principal A on one thing but not on another. In [1, 48], if B speaks for A, then B speaks for A with respect to everything. Second, in DL, a speaks_for statement can only be made by the trust root, but in [1, 48], "B speaks for A" is true if A says so. This is more like DL's delegation relationship, in which A delegates to B if A says so. However, the speaks_for relation in [1, 48] is unrestrictedly transitive, *i.e.*, it has no ability to restrict re-delegation; it is thus different from the delegation relation in DL.

The speaks_for relation can model the relationship between a group and its members or between the subject field and the name field in a SPKI/SDSI 4-tuple.

3.1.5 More Examples of D1LP

In this section, we show several examples that use D1LP to represent authorization policies and credentials.

Example 3.3 (Using multiple certification systems).

Alice delegates isSiteKey(?K, ?S)^3 to (XRCA,(YRCA;ZRCA)).

Alice delegates isSiteKey(?K,?S)^∗

  to threshold(1, ?X, Alice says trustedFriend(?X)).

Alice says trustedFriend(Bob).

Bob delegates isSiteKey(?K, ?S)^1 to ZRCA

  if Bob says belongsTo(?S, orga).

Bob delegates belongsTo(?S, orga)^1 to orgaKey.

YRCA delegates isSiteKey(?K,?S)^1 to YCA1.

YCA1 says isSiteKey(LKey, LSite).

ZRCA says isSiteKey(MKey, MSite).

orgaKey says belongsTo(MSite, orga).

In this example, XRCA, YRCA, and ZRCA are root keys of three public-key certification systems. They all have at most three levels of CA's. The first rule says that, for Alice to accept a binding between a public key and a site, the binding must be certified by system X and at least one of system Y and system Z. The second rule says that Alice (unconditionally) trusts anyone who is a "trusted friend" on binding public keys with sites. The third rule says that Bob is a trusted friend of Alice. The fourth rule says that Bob thinks certification by system Z is good enough if the site belongs to a specific organization orga. The fifth rule says that Bob trusts the public key orgaKey to certify that a site belongs to the organization. The rest of the rules are facts.

From the above rules and facts, DL can conclude that "Alice says isSiteKey(MKey, MSite)," because this follows from Alice's trust of Bob; however, DL cannot conclude that "Alice says isSiteKey(LKey,LSite)," because isSiteKey(LKey,LSite) is only certified by system Y.

Example 3.4 (Accessing medical records).

This example concerns access to medical records. It is based on an example in [38]. HM is a hospital that controls the medical records of some patients; it only authorizes those principals that are physicians of a given patient to access the medical record of that patient. HM trusts any known hospital to certify that a principal is the physician of a patient. HM knows some hospitals itself; furthermore, it believes that a principal is a hospital if two known hospitals certify that it is. The following D1LP program represents these policies and includes some facts.

HM says readMedRec(?X, ?Y) if HM says isPhysician(?X,?Y).

HM delegates isPhysician(?X, ?Y)ˆ1 to ?Z

if HM says isHospital(?Z).

HM delegates isHospital(?H)ˆ1

to threshold(2,?Z, HM says isHospital(?Z)).

HM says isHospital(HC). HM says isHospital(HB).

HB says isHospital(HA). HB says isHospital(HD).

HC says isHospital(HA).

HA says isPhysician(Alice, Peter).

HD says isPhysician(David, Peter).

In this example, HM initially believes that HB and HC are hospitals. Because both HB and HC certify that HA is also a hospital, HM believes that it is. Because HA says that Alice is the physician of Peter, DL can conclude that "HM says readMedRec(Alice, Peter))."

Example 3.5 (Controlling delegation width).

Suppose that Alice wants to delegate to Bob the right to access something and to allow Bob to further delegate this right as long as the principals to which Bob delegates are members of some organization orga, where membership of orga must be certified by Carl. In other words, Alice does not want to control the depth of Bob's delegation, but she wants to restrict the delegation to be within a certain domain — the members of orga. In D1LP, Alice can represent this policy by the following two delegation statements.

Alice delegates accessˆ$*$ to (Bob,tmpKey).

tmpKey delegates accessˆ1 to threshold(1,?X,Carl says member(?X,orga)).

Here, `tmpKey` is a new principal created by Alice. Alice first generates tmpKey, a new pair public/private keys, then signs the second statement with the new private key and use the new public key in the first statement. After signing the second statement, Alice can throw the new secret key away, not worrying about keeping it in a safe place.

According to this policy, Alice will delegate to a principal if both Bob and tmpKey delegate to it. Bob can delegate freely. But tmpKey only delegates to those principals certified by Carl to be members of orga, and tmpKey does not allow re-delegation. Therefore, this achieves the intended policy.

Suppose that we further have

$\mathrm{Bob\ delegates\ access\hat{\ }2\ to\ David.}$ $\quad$ $\mathrm{Carl\ says\ member(David,orga).}$

$\mathrm{Bob\ delegates\ access\hat{\ }2\ to\ John.}$

Then the delegation "$\mathrm{Alice\ delegates\ access\hat{\ }1\ to\ David}$" is a conclusion, but "$\mathrm{Alice\ delegates\ access\hat{\ }1\ to\ John}$" is not.

3.2 Semantics

In this section, we define the semantics of D1LP. This semantics defines a minimal model for every D1LP $\mathcal{P}$ and gives an answer to every query $\mathcal{Q}$ relative to $\mathcal{P}$. This semantics is defined via two transformations: $Trans$ and $RevTrans$. $Trans$ takes a D1LP and outputs an OLP. $RevTrans$ takes a set of OLP conclusions (ground facts) and outputs a set of D1LP conclusions.

A D1LP $\mathcal{P}$ is first transformed (essentially, compiled) into a definite OLP $\mathcal{O} = Trans(\mathcal{P})$ in an OLP language $\mathcal{LO}_{\mathcal{P}}$. According to the usual minimal-model semantics of OLP, this OLP $\mathcal{O}$ has a minimal model $M_{\mathcal{O}}$ that is a set of entailed ground conclusions expressed in OLP. The minimal D1LP model of $\mathcal{P}$, denoted by $M_{\mathcal{P}}$, is obtained by reverse-transforming $M_{\mathcal{O}}$ back into D1LP syntax, *i.e.*, $M_{\mathcal{P}} = RevTrans(M_{\mathcal{O}})$. $M_{\mathcal{P}}$ is a set of entailed ground conclusions expressed in D1LP. This inferencing procedure that computes the entire model $M_{\mathcal{P}}$ is called *exhaustive* (forward) inferencing. It is useful when one wants to compute all the conclusions of a program.

As in OLP, one does not always want to perform exhaustive inferencing. To answer a particular D1LP query $\mathcal{Q}$ with respect to a D1LP $\mathcal{P}$, one can transform both $\mathcal{Q}$ and $\mathcal{P}$ into OLP and then use OLP's query-answering mechanism to answer the query. The procedure that does this is described in detail in section 3.2.5. This kind of query-answering avoids computing the entire minimal D1LP model; it is also called *backward* inferencing.

In specifying $Trans$ and calculating the size of its output $\mathcal{O} = Trans(\mathcal{P})$, we use the following notation. N is the size of $\mathcal{P}$. By "size," we mean the number of symbols, *i.e.*, variables, constants, predicate symbols, keywords, logical operators, *etc.* D is the largest integer delegation depth in $\mathcal{P}$. Because it is difficult to imagine an authorization decision that distinguishes between depth, say, 12 and 13, normally we expect D to be a very small integer, *e.g.*, less than five. We define $[0..*] = [0..D] \cup \{*\}$, $[*..*] = \{*\}$, and $d < *$ for any $d \in [0..D]$. We also define the following operation: For any $d_1, d_2 \in [0..*]$,

$$d1 \oplus d2 = \begin{cases} * & \text{if } d_1 = *, \text{ or } d_2 = *, \text{ or } d_1 + d_2 > D \\ d_1 + d_2 & \text{otherwise} \end{cases}$$

We specify $Trans$ in the next three subsections, first showing how to transform a D1LP that doesn't contain threshold structures, then showing how to handle static and dynamic threshold structures as well.

3.2.1 Transformation from D1LP to OLP without threshold structures

The transformation $Trans$ generates an OLP program that propagates direct statements through delegations when the depth constraints are not exceeded. To do so, the generated program maintains the number of depth-consuming delegation steps that a conclusion has gone through. Because D1LP allows delegation queries, the generated program also has rules that chain delegations together to derive new delegations and rules to generate weak delegations from strong ones, so that, if a stronger delegation is proved to be true, then the answer to a query that is a weaker delegation is also true.

Before doing the transformation, we replace each occurrence of `Local` in $\mathcal{P}$ with the principal that is the current trust root.

There are two predicates in $Trans(\mathcal{P})$'s output language $\mathcal{LO}_{\mathcal{P}}$: $holds$ and $delegates$. The predicate $holds$, used to represent direct statements that are made in $\mathcal{P}$ or derived in the inference process, takes three parameters:

$$holds(issuer, ba, len)$$

The domain of $issuer$ is $Principals$, a set that contains all principals in $\mathcal{P}$ plus some dummy principals introduced during the body transformation, which we will define soon. The domain of ba is the set of all ground base atoms in $\mathcal{P}^{Inst}$ (ground instantiation of $\mathcal{P}$). The domain of len is $[1..*]$. Note that base atoms in $\mathcal{P}$ are used as terms here; for each predicate symbol in $\mathcal{P}$, we add to $\mathcal{LO}_{\mathcal{P}}$ a new function symbol that has the same name as that predicate symbol. The field len stores the number of delegation steps this conclusion has gone through. A "$*$" in the field len means that it has gone through more steps than we need to keep track of, *i.e.*, the number of steps is greater than the maximum integer delegation depth D.

The predicate $delegates$, used to represent delegation statements and speaks_for statements that are made in $\mathcal{P}$ or derived in the inference process, takes five parameters:

$$delegates(issuer, ba, dep, dele, len)$$

Here, dep stands for depth and $dele$ stands for delegatee. The domains of $issuer$ and ba are the same as they are in $holds$; the domain of dep is $[1..*]$; the domain of $dele$ is $Principals$, the same as that of the $issuer$ field; the domain of len is $[0..*]$. Note that, unlike the len field in a $holds$ atom, the len field in a $delegates$ atom can be 0; this will be the case for speaks_for statements. Note that the domain of $dele$ is a set of principals, and so a $delegates$ atom can only represent a delegation to a single principal. This is essential in ensuring tractability. Why this suffices will become clear later.

Function PSFormula:

We now define a function $PSFormula$. It takes two parameters: a complex principal term PS (either a principal variable or a principal structure) and an atom (of predicate $holds$ or $delegates$) without the $issuer$ field. The function $PSFormula$ is defined recursively as follows:

$PSFormula((PS1, PS2), At) = (PSFormula(PS1, At), PSFormula(PS2, At))$

$PSFormula((PS1; PS2), At) = (PSFormula(PS1, At); PSFormula(PS2, At))$

$PSFormula(X, holds(ba, l)) = holds(X, ba, l)$

$PSFormula(X, delegates(ba, dep, dele, l)) = delegates(X, ba, dep, dele, l)$

where X is either a principal variable or a single principal.

The function $PSFormula$ transforms a statement that has a complex principal structure as its issuer to an equivalent statement formula, in which each statement has a principal or a principal variable as issuer. As we will soon see, $PSFormula$ deals with delegatees that are complex principal structures and enables body statements to be more general than head statements.

For now, $PSFormula$ simply returns a formula. When we deal with threshold structures in sections 3.2.2 and 3.2.3, we will extend the definition of $PSFormula$; it will have side effects as well as returning a formula — it will generate some additional rules and introduce some new constants.

Finally, we can start defining $Trans$. It is divided into two phases: *body transformation* and *head transformation*.

Phase I: Body transformation

This phase of the transformation changes rule-bodies in $\mathcal{P}$; the result is called $\mathcal{P}_1$. It may also construct some new rules; the set of new rules is called $\mathcal{P}_1^{add}$.
This phase of transformation does the following to the *body* of each rule in $\mathcal{P}$.

1. **Holds body translation:**

   Replace each body-direct statement $\quad AS\ \texttt{says}\ ba$

   with $\quad PSFormula(AS, holds(ba, *))$.

   This step adds the length $*$ to body statements and uses $PSFormula$ to deal with complex issuers. Intuitively, a direct statement "AS says ba" in the body of a rule is true if we can prove that AS supports the base atom ba either directly or through delegation. The length $*$ means that we do not require that the conclusion be drawn within a certain number of delegation steps.

2. **Speaks_for body translation:**

   Replace each speaks_for statement $\quad B\ \texttt{speaks\_for}\ A\ \texttt{on}\ ba$

   with $\quad delegates(A, ba, *, B, 0)$.

   This means that speaks_for statements are special delegations that always have depth $*$ and length 0.

3. **Simple delegates body translation:**

   Replace each body-delegation statement AS `delegates` ba`^`d `to` B

   with $PSFormula(AS, delegates(ba, d, B, *))$,

   where B is a principal or a principal variable.

   This step is similar to step 1, but it is for delegation statements.

4. **Conjunction delegates body translation:**

   Replace each body-delegation statement

   AS `delegates` ba`^`d `to` $(B_1, \dots, B_n)$

   with $PSFormula(AS, delegates(ba, d, B_{new}, *))$,

   where $B_1, \dots, B_n$ are principals, and B_{new} is a newly created principal.

   In addition, for each B_i, $i = 1..n$, add the following fact to $\mathcal{P}_1^{add}$:

   $delegates(B_i, ba, *, B_{new}, 0)$.

   This step enables tractable inference of delegations that have conjuncts of principals as delegatees. Remember that the $dele$ field of the predicate $delegates$ is required to be a principal, rather than a conjunction of principals. Therefore, we introduce a dummy principal B_{new} to represent the principal structure $(B_1, \dots, B_n)$. That B_{new} is equivalent to $(B_1, \dots, B_n)$ is fully characterized by the relationships that B_{new} speaks for every principal in $(B_1, \dots, B_n)$. The new facts "$delegates(B_i, ba, *, B_{new}, 0)$" are introduced for this purpose. These facts are added to $\mathcal{P}_1^{add}$ instead of $\mathcal{P}_1$, because they do not need further processing; including them in the final output is sufficient.

Let $Principals$ be the set of all principals in $\mathcal{P}_1 \cup \mathcal{P}_1^{add}$.

Phase II: Head transformation

The input to this phase is $\mathcal{P}_1$. This phase of the transformation changes rule heads in $\mathcal{P}_1$; the result is called $\mathcal{P}_2$. It also constructs some new rules; the set of the new rules is called $\mathcal{P}_2^{add}$.

For each rule R in $\mathcal{P}_1$, one of the following two cases applies:

Case one: When R's head is a direct statement "A `says` ba," do the following two steps.

5. **Holds head translation:**

   Replace R's head with "$holds(A, ba, 1)$."

6. **Holds length-weakening meta-rule:**

   For each $len \in [1..D]$, add the following rule:

   $holds(A, ba, len \oplus 1)$ `if` $holds(A, ba, len)$.

   This meta-rule states that, if something can be derived with smaller length, then it can also be inferred when larger length is allowed.

Case two: When R's head is not a direct statement, *i.e.*, it is either a delegation statement or a speaks_for statement, do the following steps.

Sub-case a: If R's head is a delegation statement:

A `delegates` ba`^`d `to` BS,

i.e., a depth-d delegation from A to BS, then let ll be 1, and let B be BS if BS is a single principal or a principal variable; otherwise, let B be a newly created principal (dummy principal to represent BS).

Sub-case b: If R's head is a speaks_for statement:

B `speaks_for` A `on` ba,

then let d be $*$, ll be 0, and BS be B.

For both sub-cases, do the following steps.

7. **Delegates head translation:**

   Replace R's head with $\quad delegates(A, ba, d, B, ll)$.

   This transformation step makes sure that A delegates to B whenever A delegates to BS is true from the rule R. B is defined above. We use B in place of BS, because the delegatee field of the predicate $delegates$ can only be a principal or a principal variable.

8. **Holds propagation meta-rule:**

   For each $len \in [1..d]$, add the following rule:

   $$holds(A, ba, len \oplus ll)$$
   $$\texttt{if}\ \ delegates(A, ba, d, B, ll),\ \ PSFormula(BS, holds(ba, len)).$$

   This meta-rule propagates direct statements through a delegation as follows: If the delegation in R's head is true (by the previous step, it is true when the body of R is true), and the delegatee BS supports something within len ($\leq d$) delegation steps , then the issuer A supports the same thing within $len \oplus ll$ steps, where ll is 1 if R's head is a delegation and 0 if R's head is a speaks_for statement.

9. **Holds length-weakening meta-rule:**

   For each $len \in [d \oplus 1..D]$, add the following rule:

   $$holds(A, ba, len \oplus 1)\ \ \texttt{if}\ \ holds(A, ba, len).$$

   This meta-rule is the same as step 6. It appears again, because it is also needed for case two.

10. **Self delegation meta-rule:**

    For each $C \in Principals$, for each $dep \in [1..*]$, and for each $len \in [0..*]$, add the following fact:

    $delegates(C, ba, dep, C, len).$

    This meta-rule states that each principal delegates unconditionally to itself.

11. **Delegates length-weakening meta-rule:**

    For each $C \in Principals$, for each $dep \in [1..d]$, and for each $len \in [0..D]$, add the following rule:

    $delegates(A, ba, dep, C, len \oplus 1)$ `if` $delegates(A, ba, dep, C, len).$

    This meta-rule states that any delegation that is derived within a certain length can also be derived within a larger length.

12. **Delegates depth-weakening meta-rule:**

    For each $C \in Principals$, for each $len \in [0..*]$, and for each $dep \in [1..D]$, add the following rule:

    $delegates(A, ba, dep, C, len)$ `if` $delegates(A, ba, dep \oplus 1, C, len).$

    This meta-rule states that a smaller-depth delegation can be derived if a corresponding larger-depth delegation is derived.

13. **Delegation chaining meta-rule:**

    For each $C \in Principals$, for each $dep \in [1..d]$, and for each $len \in [0..d \ominus dep]$, add the following rule:

    $delegates(A, ba, \min(d \ominus len, dep), C, ll \oplus len)$

    `if` $delegates(A, ba, d, B, ll),$

    $PSFormula(BS, delegates(ba, dep, C, len)).$

    where, for any $d1, d2 \in [0..D]$: "$* \ominus * = *$," "$* \ominus d1 = *$," "$d1 \ominus d2 = d1 - d2$," and "$d1 \ominus * < 0$."

This is the most complex and the most expensive (in terms of the size of the new rules added) meta-rule. Intuitively, it means that, if A delegates to BS with depth d (see step 7 for the relation between B and BS), and one can also derive within $len \leq d$ steps that BS delegates to C with depth dep, then A delegates to C as well. The depth of this newly derived delegation is bounded by the depth dep, because A should not trust C more than BS trusts C. It is also bounded by the depth d minus the number of delegation steps that have already been used to derive the delegation from BS to C.

Steps 10–12 infer weak delegations from strong ones and step 13 chains delegations together.

The above meta-rules may seem unnecessarily complicated, especially in the way they deal with length and delegation depth. This complication arises because we are avoiding introducing new variables in the transformation; this is essential in proving tractability results.

The result of the transformation is: $\mathcal{O} = Trans(\mathcal{P}) = \mathcal{P}_2 \cup \mathcal{P}_1^{add} \cup \mathcal{P}_2^{add}$.

It is straightforward to show by a counting argument that the size of $\mathcal{O}$ is $O(N^3 D^2)$, where N is the size of $\mathcal{P}$, and D is the maximal integer depth used in $\mathcal{P}$. Each rule in $\mathcal{P}$ can produce $O(ND^2)$ new rules in $\mathcal{O}$, and each new rule may have a size that is $O(N)$ times the size of the original rule. A more detailed counting argument is as follows.

Our counting argument focuses on the ratio $|\mathcal{O}|/|\mathcal{P}|$, which we call the *growth factor*.

Note that $|PSFormula(BS, At)|/|At| = O(|BS|)$. Clearly, $|BS| < N$. Therefore, the growth factor of $PSFormula$ is $O(N)$.

In the body-transformation phase, a body statement is replaced by the result of a corresponding $PSFormula$ call. Therefore, $|\mathcal{P}_1|/|\mathcal{P}| = O(N)$. If a body statement has a conjunctive delegatee, the program $\mathcal{P}_1^{add}$ has one additional fact for each principal in the delegatee. Because there are at most N principals in any delegatee, and each additional fact has size linear in the size of the original body statement, $|\mathcal{P}_1^{add}|/|\mathcal{P}| = O(N)$. Note that this phase doesn't change rule-heads.

In the head-transformation phase, if a rule has a direct statement as its head, up to D new rules are added, each of which has size linear in the size of the original head. Therefore, $|\mathcal{P}_2^{add}|/|\mathcal{P}| = O(D)$. The size of $\mathcal{P}_2$ remains unchanged from $\mathcal{P}_1$, and so $|\mathcal{P}_2|/|\mathcal{P}| = O(N)$.

In the head-transformation phase, if a rule R has a delegation statement or a speaks_for statement as its head, several transformation steps apply; each adds a set of rules to $\mathcal{P}_2^{add}$, but the size of $\mathcal{P}_2$ remain unchanged from $\mathcal{P}_1$. Step 13 (the delegation chaining meta-rule) generates the largest set of rules. It adds $O(|Principals|D^2)$ transformed rules for the rule R. Recall that $Principals$ is the set of all principals in $\mathcal{P}_1 \cup \mathcal{P}_1^{add}$. Because at most one new principal is introduced per statement in $\mathcal{P}$, $|Principals| = O(N)$. Each transformed rule may use $PSFormula$ to change parts of it. Therefore, the growth factor for step 13 is $O(N^2D^2)$.

Because $\mathcal{O} = \mathcal{P}_2 \cup \mathcal{P}_1^{add} \cup \mathcal{P}_2^{add}$, $|\mathcal{O}|/|\mathcal{P}| = O(N^2D^2)$. Of this N^2D^2 growth factor, one N comes from the size of $Principals$, which is likely to be the order of $|\mathcal{P}|$. The other N comes from the bound on the size of one principal structure; this usually will be much smaller than $|\mathcal{P}|$.

3.2.2 Transformation with static threshold structures

To handle static unweighted threshold structures, we add a new function symbol "$suth$" to $\mathcal{LO}_{\mathcal{P}}$; it stands for static unweighted threshold structures. We also extend the domain of the issuer field for predicates $holds$ and $delegates$ to include terms of the form "$suth(i, [A_1, \ldots, A_n])$," where i is an integer representing the threshold value that needs to be satisfied and A_i's are principals. Then we extend the definition of $PSFormula$ to include the following:

$$PSFormula(threshold(k, [A_1, \ldots, A_n]), holds(ba, l))$$
$$= holds(suth(k, [A_1, \ldots, A_n]), ba, l)$$
$$PSFormula(threshold(k, [A_1, \ldots, A_n]), delegates(ba, dep, dele, l))$$
$$= delegates(suth(k, [A_1, \ldots, A_n]), ba, dep, dele, l)$$

The function $PSFormula$, for calls of the above forms, results in side effects besides returning a formula; it generates the following new rules. These rules reason about atoms that have issuers of the form "$suth(i, [A_1, \ldots, A_n])$."

Case one: $PSFormula$ is called with a holds atom.
"$PSFormula(threshold(k, [A_1, \ldots, A_n]), holds(ba, l))$" does the following.

- For $i = k$ to 1, for $j = 1$ to n, add the rule:

  $$holds(suth(i, [A_j, A_{j+1}, \ldots, A_n]), ba, l)$$
  $$\texttt{if}\ holds(A_j, ba, l),\ holds(suth(i-1, [A_{j+1}, \ldots, A_n]), ba, l).$$

  Here, we define $[A_{n+1}, \ldots, A_n]$ to be the empty list [].

  This meta-rule means that, if A_j supports ba and there are no fewer than $i - 1$ principals in $[A_{j+1}, \ldots, A_n]$ that support ba, then there are no fewer than i principals out of $[A_j, A_{j+1}, \ldots, A_n]$ that support ba .

- For $i = k$ to 1, for $j = 1$ to n, add the rule:

  $holds(suth(i, [A_j, A_{j+1}, \ldots, A_n]), ba, l)$

  `if` $holds(suth(i, [A_{j+1}, \ldots, A_n]), ba, l)$.

  This meta-rule means that, if there are no fewer than i principals in $[A_{j+1}, \ldots, A_n]$ that support ba, then there are no fewer than i principals in $[A_j, A_{j+1}, \ldots, A_n]$ that support ba.

- For $j = 1$ to $n + 1$, add the fact:

  $holds(suth(0, [A_j, \ldots, A_n]), ba, l)$.

  This meta-rule means that it is always true that there are no fewer than 0 principal in $[A_j, \ldots, A_n]$ that supports ba.

Case two: $PSFormula$ is called with a delegates atom.

"$PSFormula(threshold(k, [A_1, \ldots, A_n]), delegates(ba, dep, dele, l))$" does the following.

- For $i = k$ to 1, for $j = 1$ to n, add the rule:

  $delegates(suth(i, [A_j, A_{j+1}, \ldots, A_n]), ba, dep, dele, l)$

  `if` $delegates(A_j, ba, l)$,

  $delegates(suth(i - 1, [A_{j+1}, \ldots, A_n]), ba, dep, dele, l)$.

- For $i = k$ to 1, for $j = 1$ to n, add the rule:

  $delegates(suth(i, [A_j, A_{j+1}, \ldots, A_n]), ba, dep, dele, l)$

  `if` $delegates(suth(i, [A_{j+1}, \ldots, A_n]), ba, dep, dele, l)$.

- For $j = 1$ to $n + 1$, add the fact:

  $delegates(suth(0, [A_j, \ldots, A_n]), ba, dep, dele, l)$.

Each time $PSFormula$ encounters a static unweighted threshold structure,

$O(\min(k, n)n)$ new rules are generated, where k is the threshold value, and n is the size of the threshold pool. Each new rule has size linear in the size of $PSFormula$'s input. The worst-case bound for $O(\min(k, n)n)$ is $O(N^2)$. Thus, handling static unweighted threshold structures increases the growth factor of $PSFormula$ from $O(N)$ to $O(N^2)$. This increases the worst-case size of $\mathcal{O}$ from $O(N^3D^2)$ to $O(N^4D^2)$.

Static weighted threshold structures are handled similarly; a new function symbol "$swth$" is introduced. We extend the domain of the issuer field for predicates $holds$ and $delegates$ to include terms of the form "$swth(i, [(A_1, w_1), \ldots, (A_n, w_n)])$." Then we extend the definition of $PSFormula$ to include the following:

$$PSFormula(threshold(k, [(A_1, w_1), \ldots, (A_n, w_n)]), holds(ba, l))$$
$$= holds(swth(k, [(A_1, w_1), \ldots, (A_n, w_n)]), ba, l)$$
$$PSFormula(threshold(k, [(A_1, w_1), \ldots, (A_n, w_n)]), delegates(ba, dep, dele, l))$$
$$= delegates(swth(k, [(A_1, w_1), \ldots, (A_n, w_n)]), ba, dep, dele, l)$$

Case one: "$PSFormula(threshold(k, [(A_1, w_1), \ldots, (A_n, w_n)]), holds(ba, l))$" does the following.

- For $i = k$ to 1, for $j = 1$ to n, add the rule:

  $holds(swth(i, [(A_j, w_j), (A_{j+1}, w_{j+1}), \ldots, (A_n, w_n)]), ba, l)$
  `if` $holds(A_j, ba, l)$,
  $holds(swth(\max(i - w_j, 0), [(A_{j+1}, w_{j+1}), \ldots, (A_n, w_n)]), ba, l)$.

- For $i = k$ to 1, for $j = 1$ to n, add the rule:

  $holds(swth(i, [(A_j, w_j), (A_{j+1}, w_{j+1}) \ldots, (A_n, w_n)]), ba, l)$
  `if` $holds(swth(i, [(A_{j+1}, w_{j+1}), \ldots, (A_n, w_n)]), ba, l)$.

- For $j = 1$ to $n+1$, add the fact:

  $$holds(swth(0, [(A_j, w_j), \ldots, (A_n, w_n)]), ba, l).$$

Case two: "$PSFormula(threshold(k, [(A_1, w_1), \ldots, (A_n, w_n)]),$
$delegates(ba, dep, dele, l))$" does the following:

- For $i = k$ to 1, for $j = 1$ to n, add the rule:

  $$delegates(swth(i, [(A_j, w_j), (A_{j+1}, w_{j+1}), \ldots, (A_n, w_n)]), ba, dep, dele, l)$$
  $$\texttt{if} \quad delegates(A_j, ba, dep, dele, l),$$
  $$delegates(swth(\max(i - w_j, 0), [(A_{j+1}, w_{j+1}), \ldots, (A_n, w_n)]),$$
  $$ba, dep, dele, l).$$

- For $i = k$ to 1, for $j = 1$ to n, add the rule:

  $$delegates(swth(i, [(A_j, w_j), (A_{j+1}, w_{j+1}) \ldots, (A_n, w_n)]), ba, dep, dele, l)$$
  $$\texttt{if} \ delegates(swth(i, [(A_{j+1}, w_{j+1}), \ldots, (A_n, w_n)]), ba, dep, dele, l).$$

- For $j = 1$ to $n+1$, add the fact:

  $$delegates(swth(0, [(A_j, w_j), \ldots, (A_n, w_n)]), ba, dep, dele, l).$$

Handling static weighted threshold structures doesn't change the growth factor of $PSFormula$, it is still $O(N^2)$.

3.2.3 Transformation with dynamic threshold structures

To handle dynamic threshold structures, we need a listing of all principals in $\mathcal{P}$. Let M be the number of different principals in $\mathcal{P}$ and $[C_1, C_2, \ldots, C_M]$ be a list of all principals in $\mathcal{P}$.

We introduce a new function symbol "$duth$," which stands for dynamic unweighted threshold structure, and extend the domains of the issuer field for the two predicates

$holds$ and $delegates$ to include terms of the form "$duth(i, j, t)$," where i and j are integers, and t is a newly generated constant. The integer i is the threshold value that needs to be satisfied, and the integer j is an index into the list of principals "$[C_1, C_2, \ldots, C_M]$." The newly generated constant t is used to uniquely identify the overall dynamic threshold pool defined by "$?X, Prin$ says $pred(\ldots ?X \ldots)$."

We also extend the definitions of $PSFormula$ to include the following:

$$PSFormula(threshold(k, ?X, Prin \text{ says } pred(\ldots ?X \ldots)), holds(ba, l))$$
$$= holds(duth(k, 1, t), ba, l)$$
$$PSFormula(threshold(k, ?X, Prin \text{ says } pred(\ldots ?X \ldots)),$$
$$delegates(ba, dep, dele, l))$$
$$= delegates(duth(k, 1, t), ba, dep, dele, l)$$

Each time $PSFormula$ is called with a dynamic unweighted threshold structure argument, it generates a new constant t and a set of new rules, in addition to returning an atom as defined above.

Case one: "$PSFormula(threshold(k, ?X, Prin \text{ says } pred(\ldots ?X \ldots)), holds(ba, l))$" does the following.

- For $i = k$ to 1, for $j = 1$ to M, add the rule:

  $holds(duth(i, j, t), ba, l)$

  `if` $holds(Prin, pred(\ldots C_j \ldots), *)$,

  $holds(C_j, ba, l)$,

  $holds(duth(i - 1, j + 1, t), *)), ba, l)$.

- For $i = k$ to 1, for $j = 1$ to M, add the rule:

  $holds(duth(i, j, t), ba, l)$ `if` $holds(duth(i, j + 1, t), ba, l)$.

- For $j = 1$ to $M + 1$, add the rule:

  $holds(duth(0, j, t), ba, l).$

Case two: "$PSFormula(threshold(k, ?X, Prin$ says $pred(\ldots?X\ldots)),$

$delegates(ba, dep, dele, l))$" also does the following.

- For $i = k$ to 1, for $j = 1$ to M, add the rule:

  $delegates(duth(i, j, t), ba, dep, dele, l)$

  `if` $holds(Prin, pred(\ldots C_j \ldots), *),$

  $delegates(C_j, ba, dep, dele, l),$

  $delegates(duth(i - 1, j + 1, t), ba, dep, dele, l).$

- For $i = k$ to 1, for $j = 1$ to M, add the rule:

  $delegates(duth(i, j, t), ba, dep, dele, l)$

  `if` $delegates(duth(i, j + 1, t), ba, dep, dele, l).$

- For $j = 1$ to $M + 1$, add the rule:

  $delegates(duth(0, j, t), ba, dep, dele, l).$

For each dynamic threshold structure, $O(\min(k, M)M)$ rules are added, where k is the threshold value. Recall that M is the number of different principals in $\mathcal{P}$, and so $M = O(N)$. Thus, the worst-case growth factor of $PSFormula$ with dynamic threshold structures is still $O(N^2)$, the same as it is with static threshold structures. However, dynamic threshold structures are more expensive in practice, because M is typically much larger than n. (Recall that n, used in section 3.2.2, is the size of one static threshold pool.)

3.2.4 Reverse transformation of conclusions

In the previous three subsections, we defined the transformation from a D1LP $\mathcal{P}$ to an OLP $\mathcal{O}$. We now define a simple reverse transformation that maps an OLP model of $\mathcal{O}$ to a D1LP model of $\mathcal{P}$. This reverse transformation is useful if one wants all the D1LP conclusions entailed by $\mathcal{P}$.

- For each atom of the form: $holds(A, ba, len)$,

  where A is a principal, include the D1LP-conclusion:

  A `says` ba.

- For each atom of the form: $delegates(A, ba, *, D, 0)$,

  where A and D are principals, include the D1LP-conclusion:

  D `speaks_for` A `on` ba.

- For each atom of the form:

  $delegates(A, ba, dep, D, len)$,

  where A and D are principals and $len > 0$, include the D1LP-conclusion:

  A `delegates` $ba\hat{}dep$ `to` D.

  (Note that, because of the way the semantic transformation is defined, there are no atoms with both $len = 0$ and $dep < *$.)

Note that length can be ignored after the OLP conclusions are drawn. The minimal D1LP model of $\mathcal{P}$, denoted by $M_{\mathcal{P}}$, is obtained by applying the above reverse-transformation to $M_{\mathcal{O}}$, the minimal OLP model of $\mathcal{O}$.

3.2.5 Query answering

An answer to a D1LP query $\mathcal{Q}$ is a set of variable bindings that makes $\mathcal{Q}$ true relative to $\mathcal{P}$. When $\mathcal{Q}$ is ground, the answer is just whether $\mathcal{Q}$ is true or not. Although the truth value of $\mathcal{Q}$ relative to $\mathcal{P}$ is determined by $\mathcal{P}$'s minimal D1LP model $M_{\mathcal{P}}$, one cannot simply check whether $\mathcal{Q}$ is in $M_{\mathcal{P}}$ to answer it, because the syntactic expressiveness of a D1LP query is considerably greater than that of a D1LP conclusion. A query may have a complex principal structure as issuer, and it may have a conjunction of principals as delegatee.

Next, we give an algorithm to answer the query $\mathcal{Q}$ relative to $\mathcal{P}$, without doing exhaustive inferencing:

1. Transform $\mathcal{Q}$ into an OLP query, using the same procedure as the one used to transform rule-bodies, *i.e.*, the body transformation (see section 3.2.1). This transformation changes $\mathcal{Q}$ into an OLP query $\mathcal{Q}'$ and generates a new set of OLP rules $\mathcal{Q}^{add}$ (possibly empty).

2. Form an OLP $\mathcal{O}' = \mathcal{O} \cup \mathcal{Q}^{add}$.

3. Answer the OLP query $\mathcal{Q}'$ with respect to $\mathcal{O}'$, using some backward OLP inference engine, *e.g.*, Prolog. The resulting bindings directly yield the answer to the query $\mathcal{Q}$ relative to $\mathcal{P}$.

3.3 Tractability Results

In this section, we give upper bounds on the worst-case computational complexity of $Trans$, the transformation from D1LP to OLP, and of overall D1LP inferencing using

$Trans$. We show that $Trans$ is tractable and that, under commonly-met restrictions, overall D1LP inferencing is also tractable.

3.3.1 Tractability of the transformation from D1LP to OLP

From the counting arguments in sections 3.2.1, 3.2.2, and 3.2.3, it follows straightforwardly that the growth factor of the transformation $Trans$ is $O(N^3 D^2)$, where $N = |\mathcal{P}|$, and D is the maximal delegation depth in $\mathcal{P}$. Therefore, we have the following result.

Theorem 3.1 (Tractable Transform Size).
Given a D1LP $\mathcal{P}$, the size of $\mathcal{O} = Trans(\mathcal{P})$ is $O(N^4 D^2)$, where $N = |\mathcal{P}|$, and D is the maximal delegation depth in $\mathcal{P}$.

We observe that the definition of $Trans$ corresponds straightforwardly to an algorithm to perform this transformation. We observe further that this algorithm takes time linear in the size of the output OLP. Following these observations and theorem 3.1, we have the following theorem.

Theorem 3.2 (Tractable Transform Time).
Computing $\mathcal{O}$ takes time $O(N^4 D^2)$. The transformation from D1LP to OLP is thus computationally tractable.

As discussed before (page 46), we expect that D will typically be a small constant, *e.g.*, less than five.

Next, we discuss how the complexity picture will often in practice be significantly better than the worst-case bound of $O(N^4 D^2)$. Overall, we observe that each rule grows independently and that most rules are simple ones that have small growth factors. A rule with a direct statement as its head is simpler than a rule with a delegation statement or a speaks_for statement as its head. A rule that doesn't have any threshold structure is simpler than a rule that does.

Consider a rule R that does not contain any threshold structures (neither in the head nor in the body): Let S_R be the size of the largest principal structure in R; certainly $S_R < |R| < N$. We expect that S_R will usually be a small constant. If R's head is a direct statement, R's growth factor is $\max(S_R, D)$, which is a small constant assuming that S_R and D are small constants. Thus the simplest rules typically have a constant growth factor. If R's head is a delegation statement or a speaks_for statement, R's growth factor is "$|Principals|S_R D^2$," where $Principals$ is the set of different principals in $\mathcal{P} \cup \mathcal{P}_1^{add}$. Clearly $|Principals| = O(N)$. Assuming that S_R and D are constants, this "$|Principals|S_R D^2$" factor becomes $O(N)$ with a relatively large constant factor (*e.g.*, 20–100).

Transformation of rules that contain threshold structures is more expensive. However, having threshold structures in one rule doesn't affect the growth factor of other rules. We expect that, in practice, most D1LP programs consist mostly of simple rules that do not have threshold structures.

Now we break down the $O(N^3 D^2)$ growth factor as follows.

- Having complex principals structures contributes $O(N^2)$, which is the max of the following two cases.
  - Having conjunctions and disjunctions contributes $O(S)$, where S is the size of the largest principal structure in $\mathcal{P}$. Clearly $S = O(N)$. Typically, S is a small constant.
  - Having threshold structures contributes $O(N^2)$, which is the max of the fol-

lowing two cases.

- $*$ Having static threshold structures contributes $O(\min(k, n)n)$.
- $*$ Having dynamic threshold structures contributes $O(\min(k, M)M)$.

Note that k is typically a small constant, in which case the overall growth factor when complex principal structures are involved is $O(N)$.

- Having integer delegation depth contributes $O(D^2)$, because $Trans$ loops over all lengths and depths to derive delegation conclusions. This factor is reduced to $O(D)$ if we do not answer delegation queries, because then $Trans$ only needs to loop over all lengths.
- Answering delegation queries contributes $O(N)$, as $Trans$ needs to loop over all principals in the set $Principals$.

3.3.2 Tractability of D1LP inferencing

Next, we review some previously known results about OLP inferencing [53]. We say that an LP (either OLP or D1LP) obeys the **VB** restriction when it has an upper bound v on the number of (logical) variables. To indicate that the per-rule bound on the number of variables is v, we also say that the LP is VB(v). We say that an LP is **VBD**(v) if it is VB(v) and is either Datalog or ground. Given a definite OLP $\mathcal{O}$ that is VBD(v), its inferencing (computing its minimal model or answering a query relative to it) takes time $O(|\mathcal{O}|^{1+v})$. This is because inferencing of a definite OLP takes time linear in the size of its ground instantiation and $\mathcal{O}$'s ground instantiation has size $O(|\mathcal{O}|^{1+v})$. For each variable, there are $O(|\mathcal{O}|)$ ground terms that can be used to instantiate it, and so, for each rule, there are at most $O(|\mathcal{O}|^v)$ ways to instantiate it. Thus, instantiating $\mathcal{O}$ increases its size by a factor of $O(|\mathcal{O}|^v)$.

We cannot directly use the above result for D1LP inferencing because the generated OLP $Trans(\mathcal{P})$ is not Datalog, even though $\mathcal{P}$ is. $Trans(\mathcal{P})$ introduces logical function symbols that have non-zero arities. For each predicate $pred$ in $\mathcal{P}$, $Trans(\mathcal{P})$ has one corresponding function symbol. $Trans(\mathcal{P})$ also introduces several pre-defined function symbols for threshold structures, *e.g.*, $suth$, $duth$, *etc*.

This problem can be addressed by generating a typed OLP as output. The intuitive idea of a typed logic program is that there are several sorts of variables, each ranging over a different domain. Therefore, one can use typing to limit the terms that are used to instantiate a variable, thus limiting the instantiated size of a program.

There are different typing systems for logic programs. For our purposes, the simplest form of typing — many-sorted typing — suffices. (For more advanced typing systems in logic programs, see [63].) In a many-sorted LP language, there is a finite set of types. Variables and constants have types such as τ. Predicate symbols have types of the form $\tau_1 \times \ldots \times \tau_n$, and function symbols have types of the form $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$. Variables of one type can only be instantiated to terms of the same type. The type of a given predicate specifies the type of each of its arguments; the type of a given function symbol also specifies the type of its "return value," *i.e.*, the type of any term of the form $func(...)$. There are simple techniques to translate programs from a many-sorted language to an untyped language (see pages 18–20 of [53]). Many-sorted logic programs can be executed with the same efficiency as untyped logic programs [62].

We can use many-sorted typing to ensure that, for each variable in $Trans(\mathcal{P})$, there are $O(|\mathcal{P}|)$ ground terms that can instantiate it. Because variables in $Trans(\mathcal{P})$ come from $\mathcal{P}$, these variables must be instantiated only to ground terms in $\mathcal{P}$ and not to terms constructed using the function symbols introduced during the transformation.

The simplest typing that achieves this goal has two types in $\mathcal{LO}_{\mathcal{P}}$. All the variables and constants coming from $\mathcal{P}$ have one type. All the terms introduced during the transformation have the other type. Because all the variables in $Trans(\mathcal{P})$ actually come from $\mathcal{P}$, all the variables in $\mathcal{LO}_{\mathcal{P}}$ have the first type.

It has been argued that logic programs often make implicit assumptions about types and that a logic program only satisfies its intended meaning if type information is added to it [58]. We think that typing is potentially useful in LP-based authorization languages. In authorization, there are different types of entities, *e.g.*, subjects, objects, groups, roles, *etc*. Most predicates should only take arguments of certain types. One can add typing to DL. Actually, typing is implicitly present in DL. DL has principals and principal variables; thus, there is an implicit "principal" type. Typing can also be used to relax the Datalog restriction. As long as variables are only allowed to be instantiated to a limited number of ground terms, the program does not need to be Datalog. Although we think that adding types to DL is very useful, we do not pursue this topic further in this dissertation, because it is not our main topic.

By generating a many-sorted OLP $\mathcal{O}$, we can ensure that only ground terms from $\mathcal{P}$ are used to instantiate variables in $\mathcal{O}$. Then we have the following theorem.

Theorem 3.3 (Tractable D1LP Inferencing).
Given a D1LP $\mathcal{P}$ that is VB(v), computing the minimal D1LP model of $\mathcal{P}$ has time complexity $O(N^{v+4}D^2)$, where $N = |\mathcal{P}|$, and D is the maximal delegation depth in $\mathcal{P}$.

Proof. If $\mathcal{P}$ is VB(v), then $\mathcal{O} = Trans(\mathcal{P})$ is also VB(v), because $Trans$ doesn't introduce any new variables. All variables in $\mathcal{O}$ are from $\mathcal{P}$, and there are at most N ground terms to instantiate each variable, because all the function symbols in $\mathcal{P}$ are constants. Therefore, the instantiated size of $\mathcal{O}$ is $O(|\mathcal{O}|N^v) = O(N^{v+4}D^2)$. Then, computing the minimal OLP model of $\mathcal{O}$ takes time $O(N^{v+4}D^2)$, and the size of this model is $O(N^{v+4}D^2)$. The reverse transformation takes time linear in the size of this model. So computing the minimal D1LP model of $\mathcal{P}$ has time complexity $O(N^{v+4}D^2)$.

$\square$

3.4 Discussion of the conjunctive-delegatee-queries restriction

In the beginning of this chapter, we defined standard D1LP to obey the conjunctive-delegatee-queries restriction and said that the purpose of this restriction is to ensure tractable D1LP inferencing. In this section, we discuss in detail the tractability motivation behind the conjunctive-delegatee-queries restriction.

3.4.1 Understanding D1LP's inferencing of delegation

D1LP allows queries to contain delegations to a single principal or to a conjunction of principals. To answer these queries correctly, D1LP's semantics infers weaker delegations from stronger ones. Delegations can be compared on several bases: the base atoms that they are about, their delegation depths, and their delegatees. As we have discussed earlier, all other things being equal, a higher-depth delegation is stronger than a lower-depth one. This stronger-than relation is transitive and reflexive; therefore, it is a partial order. Not all pairs of delegations are comparable; for example, "$\mathrm{A\ delegates\ p\hat{}1\ to\ B}$" and "$\mathrm{A\ delegates\ q\hat{}2\ to\ B}$" are not, because they are about two different base atoms.

We now consider the stronger-than relation between two delegations that differ only in their delegatees. For simplicity of presentation, we omit the implicit base atom and the depth in the following discussion. We compare the relative strength of delegations on the basis of the intuitive interpretation of delegation: "A delegates to B" means that "if B says something, then A agrees." Following this interpretation, "$\mathrm{A\ delegates\ to\ (B;C)}$" is logically equivalent to the conjunction of "$\mathrm{A\ delegates\ to\ B}$" and "$\mathrm{A\ delegates\ to\ C}$." By contrast, "$\mathrm{A\ delegates\ to\ (B,C)}$" is weaker than either "$\mathrm{A\ delegates\ to\ B}$" or "$\mathrm{A\ delegates\ to\ C}$," because either of the last two delegations implies the first, but not *vice versa*.

One can view principals as propositions and principal structures as negation-free formulas in propositional logic. A principal or a principal structure is true if it "says" something (the implicit base atom we are talking about) and is false if it doesn't. Given two principal structures PS_1 and PS_2, "$\mathrm{A\ delegates\ to}\ PS_1$" is stronger than "$\mathrm{A\ delegates\ to}\ PS_2$" if and only if $PS_2 \Rightarrow PS_1$ is a tautology. The reason is as follows. Given that "$\mathrm{A\ delegates\ to}\ PS_1$," if PS_1 says something, then A agrees. If further given $PS_2 \Rightarrow PS_1$, then whenever PS_2 says something (thus PS_2 is true), PS_1 is also true, and so A also agrees. Therefore, "$\mathrm{A\ delegates\ to}\ PS_2$" should also be true.

A conjunction of principals can be equivalently represented by the set of all principals in it, and so we often call a conjunction of principals a *principal set* and use set operators directly on it. For any two different principal sets PE_1 and PE_2, "$\mathrm{A\ delegates\ to}\ PE_1$" is different from "$\mathrm{A\ delegates\ to}\ PE_2$"; furthermore, they are incomparable when neither $PE_1 \subseteq PE_2$ nor $PE_2 \subseteq PE_1$.

Therefore, "$\mathrm{A\ delegates\ to}\ ((B_{11}; \ldots; B_{1n}), (B_{21}; \ldots; B_{2n}), \ldots, (B_{m1}; \ldots; B_{mn}))$" would imply "$\mathrm{A\ delegates\ to}\ (B_{1i_1}, B_{2i_2}, \ldots, B_{mi_m})$," where $1 \leq i_j \leq n$ for $j \in [1..m]$. There are n^m such delegation conclusions; this number is worst-case exponential in the size of the original delegation.

To be tractable, D1LP's semantics only generates delegations to a single principal as conclusions. Therefore, it can only directly answer those delegation queries that have a single principal as delegatee. However, it is often useful to have delegations to a conjunction of principals as queries. For example, when a request is signed by multiple principals, one may need to determine whether there is a delegation from `Local` to the conjunction of all the signers. Therefore, standard D1LP permits delegations to a *conjunction* of principals as queries. Such a query is handled by first introducing a new "dummy" principal to represent the conjunction and then transforming the query into a delegation to the new dummy principal. (See page 50 for details.) D1LP's semantics doesn't automatically generate conclusions that are delegations to conjunctions of principals; it only generates such a delegation when someone asks about it by including it in a query or a rule-body. This is a form of lazy evaluation.

Example 3.6 (Answering delegation queries).

r1: A delegates p^2 to (B1, B2).

r2: B1 delegates p^1 to (C1,C2).

r3: B2 delegates p^1 to (C3, C4).

r4: A says qq if A delegates p^1 to (C1, C2, C3, C4, C5).

Given these rules, one should conclude that "A says qq." We now give a high-level description of how this is done in D1LP inferencing. $Trans$, the transformation from D1LP to OLP, would generate a new principal T1, replace the last rule with "A says qq if A delegates p^1 to T1," and add facts "T1 speaks_for C1 on p," "T1 speaks_for C2 on p," ..., and "T1 speaks_for C5 on p." (Note that $Trans$ generates rules in OLP syntax, but here we use D1LP syntax for ease of discussion.)

From rule r2 and the two facts "T1 speaks_for C1 on p" and "T1 speaks_for C2 on p," one concludes that "B1 delegates p^1 to T1 on p." Similarly, from rule r3 and the two facts "$\mathrm{T1\ speaks\_for\ C3\ on\ p}$" and "$\mathrm{T1\ speaks\_for\ C4\ on\ p}$," one concludes that "$\mathrm{B2\ delegates\ p\hat{\ }1\ to\ T1}$." Then, from these two new conclusions and rule $\mathrm{r1}$, one concludes that "$\mathrm{A\ delegates\ p\hat{\ }1\ to\ T1}$." (Note that all depth constraints are satisfied.)

3.4.2 Handling delegation queries with disjunctive delegatees

Having explained how standard D1LP deals with conjunctive delegation queries, we now generalize D1LP to allow delegation queries that contain disjunctions or static threshold structures. We give an approach that is based on transforming principal structures into "normal forms" and then replace a disjunctive delegation query with an equivalent query that is a conjunction of delegation statements. This approach has exponential computational complexity.

We now define the *normal form of a principal structure*. A principal structure PS is in normal form when it is of the form: $(PE_1; PE_2; ...; PE_r)$, where each PE_i is a principal set and, for any $i \neq j$, $PE_i \not\subseteq PE_j$. If one views PS as a propositional formula; then PS's normal form $PE_1; PE_2; ...; PE_r$ is the result of converting the propositional-logic formula into its reduced disjunctive normal form (DNF). "Reduced" means that there is no subsumption, neither within a disjunct PE_i (*i.e.*, no repetitions of principals) nor between any two disjuncts (*i.e.*, no disjunct is a subset of another disjunct).

The normal form of "$\mathrm{threshold}(k,\ [(A_1, w_1), (A_2, w_2), \ldots, (A_n, w_n)])$" is the disjunction of all minimal subsets of $\{A_1, \ldots, A_n\}$ whose corresponding weights sum to be greater than or equal to k. For example, the normal form of "$\mathrm{threshold(3,\ [(A,2),\ (B,1),\ (C,1),\ (D,1)]}$" is "$\mathrm{((A,B);\ (A,C);\ (A,D);\ (B,C,D))}$." After two principal structures have been transformed, their conjunction and disjunction can be transformed to normal forms using methods from propositional logic.

Knowing a principal structure PS's normal form: $(PE_1; \ldots ; PE_r)$, a delegation "PS_0 delegates to PS" in a query or a rule-body is transformed into "(PS_0 delegates to PE_1, PS_0 delegates to PE_2, $\ldots$, PS_0 delegates to PE_r)." Now consider an example that has a delegation to a threshold structure in a rule-body.

Example 3.7 (Query of delegation to static threshold structures).

A delegates p^1 to (B,(C;D)).

A delegates p^1 to (C,D).

A says do_something if A delegates p^1 to threshold(2,[B,C,D]).

The last rule would be translated into the following rule:

A says do_something

if A delegates to (B,C), A delegates to (B,D), A delegates to (C,D).

D1LP's semantics will answer true to each of "A delegates to (B,C)," "A delegates to (B,D)," and "A delegates to (C,D)," using the dummy-principal approach we described earlier. Therefore, "A says do_something" is true (concluded).

When a principal structure is converted to its normal form, its size may grow exponentially. The normal form of the threshold structure "threshold(k, $[A_1, \ldots , A_n]$)" has size $\Theta(\binom{n}{k})$, which is exponential in k. The principal structure "$((A_{11}; \ldots ; A_{1n})$, $(A_{21}; \ldots ; A_{2n}), \ldots , (A_{m1}; \ldots ; A_{mn}))$" contains mn principals and has size $\Theta(mn)$, but its normal form has size $\Theta(mn^m)$.

Besides being expensive, such inferences about delegation to disjunctions may be too tricky for normal users to understand. In fact, we have not yet encountered a realistic example that requires delegation queries that contain disjunctions. Therefore, we believe that the conjunctive-delegatee-queries restriction leaves D1LP with all of the expressive power needed in practice.

3.4.3 Queries of delegations to dynamic threshold structures

A delegation query that has a dynamic threshold structure as delegatee is nonmonotonic (in the sense used in knowledge representation) in that it may goes from true to false when more information is known. Therefore, such a query should not be allowed in D1LP, because D1LP is monotonic. Consider the following example.

Example 3.8 (Nonmonotonicity of delegation to dynamic threshold structures).

A delegates to (B,C).	A says friend(B).
A delegates to (B,D).	A says friend(C).
A delegates to (C,D).	A says friend(D).

From the above program, one should conclude that "A delegates to threshold(2, ?X, A says friend(?X))," because the three delegation facts in the program combined are equivalent to "A delegates to threshold(2, [B,C,D])"; however, if we further add the fact "A says friend(E)" to the program, the delegation "A delegates to threshold(2, ?X, A says friend(?X))" is no longer true, because the delegation "A delegates to threshold(2, [B,C,D,E])" also implies that "A delegates to (B,E)," which is not a conclusion of the program.

3.5 Implementation

In this section, we describe two implementations of D1LP.

3.5.1 A Java implementation

We have implemented, in Java, a compiler that does the transformation from D1LP to OLP and the reverse transformation from OLP to D1LP (section 3.2) and does exhaustive D1LP inferencing by combining that compiler with an existing OLP inferencing engine. The implemented compiler can generate OLP in the syntactic formats of a variety of OLP inferencing engines, both forward reasoning ones (*e.g.*, Smodels [61]) and backward reasoning ones (*e.g.*, XSB [70]).

The compiler and D1LP inferencing engine are integrated as an extension to the IBM CommonRules system [37], a Java library that includes (among its capabilities) a rule translation format ("interlingua", encoded in XML) and sample translators to talk to multiple OLP inferencing engines. The D1LP compiler reuses classes and code from the CommonRules core, especially for specification and inferencing.

This implementation is expressively restricted, and slightly different syntactically, from the version of D1LP given in this paper. (It was based on an earlier design.)

3.5.2 An XSB implementation

We have also implemented D1LP in XSB [70], a Prolog-variant logic-programming system developed by Warren *et al.* at SUNY Stony Brook. This implementation is called XD1LP and is available [51]. XD1LP includes a compiler that compiles a D1LP into OLP rules in an internal format and a meta-interpreter that can answer queries using these rules. XD1LP turns the XSB engine into a D1LP engine.

XSB has several nice features that most Prolog systems do not have. It uses SLG resolution [18], which has tabling ability. SLG resolution enables XSB to evaluate correctly many recursive logic programs that would make SLD-resolution-based Prolog systems get into an infinite loop. This is crucial to our work, because delegation relationships can be circular. XSB also supports HiLog (higher-order logic programming). We can use this ability of XSB to extend D1LP to allow base atoms to be HiLog terms. Even without this HiLog ability, we can still extend D1LP to allow a variable to appear in place of a base atom. For example "A $\mathrm{speaks\_for}$ B on $?\mathrm{P}$" means that A speaks_for B on everything.

To use XD1LP in real applications, one may need to use interfaces between XSB and other languages, *e.g.*, Java [16], C, Oracle, and ODBC (see XSB manual, available from [70]).

XD1LP uses an alternative transformation that is different from (but similar to) the one we gave in section 3.2. This alternative transformation generates an output program that has size linear in the size of the input program, but it does introduce new variables. We call this an "ungrounded transformation." XD1LP can handle everything in the syntax described in section 3.1. With ungrounded transformation, it can also handle a fourth kind of threshold structures: *dynamic weighted* threshold structures, which take the form:

$$\texttt{threshold}(k,\, ?X,\, ?W,\, Prin\ \texttt{says}\ \mathit{pred}(\ldots ?X \ldots ?W \ldots))$$

where $?X$ and $?W$ are variables. The threshold pool of such a threshold structure is a principal-weight pair set that consists of every principal-weight pair (A, w) such that w is the largest positive integer that makes "$Prin$ says $pred(\ldots A \ldots w \ldots)$" true. We leave dynamic threshold structures out of standard D1LP because handling them is computationally expensive and because we haven't found practical examples requiring them.

3.5.3 Other issues in using DL

There are several additional infrastructural issues, beyond what we discussed here, that are practically important for developing real-world systems based on DL, and which are the subject of current and future work. For example, which data structures and communication protocols should one use for exchanging DL rules between distributed applications/principals/Internet-sites? An approach we are currently exploring is encoding DL in XML syntax, building upon the XML Business Rules Markup Language for OLP's that is supported by IBM CommonRules [37].

However, there are work-arounds for using DL in the absence of such a communication infrastructure. One way is first to translate certificates from multiple public-key infrastructure systems into DL "facts" and then write local policies to control the use of these certificates. For example, these local policies may specify trust of different PKI systems for various purposes and to varying degrees and/or how certification from multiple systems is required to gain sufficient confidence for critical applications.

Another unresolved infrastructural question is how an authorizer obtains all the credentials needed to make the decision. There are several possible scenarios for how such credentials should flow to the authorizer. One is that the requester submits credentials together with its request. Another is that the authorizer asks the requester for additional credentials during the evaluation of the request. Yet another is that the authorizer asks other entities for relevant credentials during the evaluation of the request. Mixes of the above are also interesting. How to obtain relevant credentials dynamically during DL inference is a topic we are exploring.

3.6 D1LP^{DQF}

In this section, we define a restricted version of D1LP that we call D1LP^{DQF}. It is a restriction of D1LP obtained by imposing the "*delegation-query-free (DQF) restriction*": Delegation statements are not allowed to appear in queries or rule-bodies (because rule-bodies are queries by nature). The DQF restriction is stronger than the conjunctive-delegatee-query restriction that ensures that D1LP is tractable.

The reason that we introduce D1LP^{DQF} is because it is much simpler and has lower computational complexity than standard D1LP but still has sufficient expressive power for many applications.

Under the DQF restriction, only "who says what?" can be answered, not "who delegates to whom?" Despite this restriction, D1LP^{DQF} still has significant expressive power. In many applications, ultimately one wants to know "who says what?"

One way to use D1LP^{DQF} is to use direct statements to represent attributes of principals; for example, groups, roles, and authorizations can all be viewed as attributes. Using D1LP^{DQF}, a principal can bind attributes to principals, delegate this binding authority to other principals, and reason about attributes of principals.

3.6.1 Transformation from D1LP^{DQF} to OLP

Because a D1LP^{DQF} is also a D1LP, the algorithm that computes the semantics of D1LP can also be used for any D1LP^{DQF}. However, because D1LP^{DQF} satisfies the DQF restriction, the transformation from D1LP^{DQF} to OLP can be simplified substantially.

First, because there are no delegation queries, the predicate $delegates$ is not needed anymore. Only the predicate $holds$ is needed. Moreover, only five of the thirteen steps in $Trans$ are needed. In the following, we briefly list these five steps.

Phase I: Body transformation

Do the following to the *body* of each rule in $\mathcal{P}$.

1. **Holds body translation:**

   Replace each body-direct statement AS `says` ba.

   with $PSFormula(AS, holds(ba, *))$.

Phase II: Head transformation

For each rule R in $\mathcal{P}_1$, exactly one of the following two cases applies:

Case one: When R's head is a direct statement "A `says` ba," do the following two steps.

2. **Holds head translation:**

   Replace R's head with "$holds(A, ba, 1)$."

3. **Holds length-weakening meta-rule:**

   For each $len \in [1..D]$, add the following rule:

   $holds(A, ba, len \oplus 1)$ `if` $holds(A, ba, len)$.

Case two: When R's head is not a direct statement, *i.e.*, it is either a delegation statement or a speaks_for statement, let $body_R$ be R's body, and do the following steps.

Sub-case a: If R's head is a delegation statement:

A `delegates` ba`^`d `to` BS,

i.e., a depth-d delegation from A to BS, then let ll be 1.

Sub-case b: If R's head is a speaks_for statement:

B `speaks_for` A `on` ba,

then let d be $*$, ll be 0, and BS be B.

For both sub-cases, remove the rule R and do the following.

4. **Holds propagation meta-rule:**

   For each $len \in [1..d]$, add the following rule:

   $holds(A, ba, len \oplus ll)$

   `if` $body_R$, $PSFormula(BS, holds(ba, len))$.

   Note that this meta-rule is different from its corresponding one in $Trans$ in that it uses $body_R$ instead of a $delegates$ atom. This is because the output language for D1LP^{DQF} doesn't have the predicate $delegates$.

5. **Holds length-weakening meta-rule:**

   For each $len \in [d \oplus 1..D]$, add the following rule:

   $holds(A, ba, len \oplus 1)$ `if` $holds(A, ba, len)$.

Threshold structures are handled similarly to that in $Trans$, see subsections 3.2.2 and 3.2.3.

It is straightforward to show, by a counting argument, that the size of $\mathcal{O}$ is $O(N^3 D)$, where N is the size of $\mathcal{P}$ and D is the maximal integer depth used in $\mathcal{P}$. Each rule in $\mathcal{P}$ can produce $O(D)$ new rules in $\mathcal{O}$, and each new rule may have a size that is $O(N^2)$ times the size of the original rule, caused by threshold structures. We get rid of a factor of N, because we do not need to loop over all principals to generate all delegation results, and a factor of D, because we do not need to loop over all possible delegation depths to generate all delegations.

Chapter 4

A Nonmonotonic Delegation Logic

In this chapter, we expressively generalize D1LP to have nonmonotonic features, including negation-as-failure, classical negation, and prioritized conflict handling. The resulting formalism is called D2LP, which stands for version 2 of Delegation Logic Programs. We use the term D2LP to denote both the formalism and a program in this formalism.

There is a large amount of literature about nonmonotonic extensions to OLP (see [5, 14] for surveys). We chose the approach of Generalized Courteous Logic Programs (GCLP) [34, 35, 36, 37]. GCLP is based on Courteous LP (CLP) [32, 33]. CLP features negation-as-failure, classical negation, and prioritized conflict handling. In CLP, each rule can optionally have a label; conflicts between rules are resolved by priority relationships among labels, defined through a reserved predicate $overrides$. GCLP extends CLP to have mutual exclusion constraints (mutex's) and (prioritized) reasoning about the priority relationships. D2LP combines these nonmonotonic features with the delegation constructs in D1LP.

Many security policies are (logically) nonmonotonic or at least are more easily or more naturally specified in a nonmonotonic formalism. In many applications, a common policy is to make a decision in one direction, *e.g.*, in favor of authorizing a request, if there is no information/evidence to the contrary, *e.g.*, no known revocation. Using *negation-as-failure* (a.k.a. *default negation* or *weak negation*) is often an easy and intuitive way to do this. Also useful in representation of many policies is *classical negation* (a.k.a. *explicit negation* or *strong negation*), which allows policies that explicitly forbid something. As argued in [40, 41], this allows more flexible security policies. Introducing classical negation leads to the potential for *conflict*. Conflict-handling mechanisms are thus needed.

Existing work on logic-based languages for authorization often uses formalism and results from nonmonotonic reasoning, *e.g.*, [6, 40, 41, 72]. The main differences between D2LP and previous work are as follows. First, D2LP has delegation constructs that deal with the multi-agent aspect of distributed authorization. In defining D2LP, we have to deal with the interaction between delegation constructs and nonmonotonic expressive features. This interaction results in some subtleties. Because of these subtleties, we require D2LP not to have queries about delegation statements. Second, D2LP is also different from the languages in [6, 40, 41, 72] in that it has *prioritized* conflict handling. This is especially useful in resolving conflicting advice from different, but apparently both trustworthy, sources.

Our technical approach to defining D2LP is based on tractably compiling a D2LP into a Generalized Courteous LP (GCLP), which is in turn tractably compiled into an Ordinary LP (OLP). We show that D2LP is thus tractable and practically implementable on top of existing technologies for OLP, *e.g.*, Prolog, SQL databases, and other rule-based systems.

The rest of this chapter is organized as follows. In section 4.1, we discuss the motivations and usefulness of prioritized conflict handling and give the syntax and semantics of GCLP. In section 4.2, we discuss some subtleties in extending DL to have prioritized conflict handling, define the syntax and semantics of D2LP, and show that D2LP is tractable.

4.1 GCLP

Recently, there has been a lot of interest in language-based approaches to security policies, *e.g.*, [6, 40, 41, 54, 72]. The goal is to provide a unified framework that can support multiple access-control policies and achieve separation of policies from mechanisms. Most approaches use logic-programming languages; other approaches [66, 73] use languages that can easily be translated to LP languages.

Many of these languages have negative authorizations, *i.e.*, policies that explicitly forbid something. When both positive and negative authorizations can be specified, conflicts may arise. Existing languages deal with conflict handling in one or more of the following ways: (1) Do not resolve conflicts; only define semantics for conflict-free policy programs [72]. (2) Use totally ordered rules to resolve conflicts [73]. (3) Define a fixed conflict-resolution policy based on relative authority and/or specificity [6, 7]. (4) Add a paraconsistent layer, and use negation-as-failure to resolve conflicts [40, 41]. For example, in [40, 41], one can write

$$\mathrm{do(file2,\ s,\ +a)} \leftarrow \mathrm{dercando(file2,\ s,\ +a)}\ \&\ \neg\mathrm{dercando(file2,\ s,\ \text{-}a)}.$$

This specifies that the negative authorization "dercando(file2, s, -a)" wins over the corresponding positive one. However, one can specify relative priorities only between pairs of mutually-conflicting conclusions, not between rules used to derive these conclusions.

We argue that these approaches are undesirably limited. In many cases, there isn't a meaningful total ordering. Relative authority and specificity are important sources of conflict-resolution information, but they are not the only sources. Other sources may also be useful, *e.g.*, recency and relative importance of rules even from a single authority. GCLP explicitly supports flexible conflict-resolution policies. Therefore, GCLP itself, even before adding delegation constructs, is a useful language for expressing authorization policies (albeit in a more centralized environment).

We now review the preexisting GCLP formalism, presenting it in a somewhat different notation. We use a version of GCLP that has been further generalized from the version in [35] to allow a rule label to be any logical term, rather than simply a constant. This version of GCLP is implemented in IBM CommonRules v 1.1 [37].

4.1.1 Syntax of GCLP

In GCLP, a *classical literal* takes the form at or $\neg\, at$, where at is an atom. A *literal* takes the form L or $\sim L$, where $\sim$ stands for negation-as-failure, and L is a classical literal. A GCLP rule takes the form:

$$\langle lab \rangle \;\; L_0 \leftarrow BodyFormula.$$

Here, lab is a term, L_0 is a classical literal, and $BodyFormula$ is a formula built from literals using "," (conjunction) and ";" (disjunction). We call lab, L_0, and $BodyFormula$ the *label*, the *head*, and the *body*, respectively, of this rule. The label of a rule can be empty, as can the body. A variable starts with a question mark. All variables in a rule's label must also appear either in the rule's head or in its body. We say that a rule is *label-range-restricted* if all variables in a rule's label also appear in the rule's head.

A special binary predicate $overrides$ is used to specify prioritization among mutually conflicting rules. The atom "$overrides(lab_1, lab_2)$" means that a rule that has label lab_1 should take precedence if it conflicts with a rule that has label lab_2. Except for this pre-defined semantics, the predicate $overrides$ is no different from any other predicate.

A *mutual exclusion constraint* (*mutex*) takes the form:

$$\perp \leftarrow L_1, L_2 \mid BodyFormula.$$

"L_1, L_2" is called the *focus* of the mutex and $BodyFormula$ is called the *body* of the mutex.[1] When $BodyFormula$ is empty, the symbol "|" is omitted. Intuitively, this mutex specifies that L_1 and L_2 conflict with each other if $BodyFormula$ is true. An atom at always conflicts with $\neg\, at$; this conflict does not need to be specified explicitly. Note that mutex's do not have labels, because labels are only used to resolve conflicts and nothing conflicts with a mutex. We introduce mutex's because some conflicts cannot be conveniently represented using classical negations. Examples include classification of users into three or more mutually disjoint groups and choices among some mutually exclusive actions, *etc*. For example, a constraint that no user is allowed to be associated with both the engineer role and the tester role can be represented as follows:

$$\perp \leftarrow hasRole(?U, Engineer), hasRole(?U, tester).$$

A GCLP consists of a set of rules and mutex's. A rule or a mutex with variables stands for all of its ground instantiations.

---

[1]In [35], a mutex is syntactically restricted such that the variables appearing in its body must also appear in its focus. Here, we relax this restriction.

4.1.2 An example of GCLP

Policies represented in GCLP can also be represented in OLP. Indeed, any GCLP can be compiled into an OLP. However, GCLP's mutex and prioritized-conflict-handling features offer expressive convenience and clarity.

Consider the database authorization model in [7]. An authorization may be specified for a single user or for a group. A group may contain users and other groups as members but may not contain itself as a member, directly or indirectly. A user u has the authorizations specified for u and for all the groups that u belongs to, directly or indirectly. Authorizations can be either positive or negative and either strong or weak. Conflicts between positive and negative authorizations are resolved as follows:

- Strong authorizations always win over conflicting weak authorizations.
- A conflict between two strong authorizations cannot be resolved; policies having such conflicts are inconsistent.
- In a conflict between two weak authorizations, the more specific authorization wins; if neither is more specific, the conflict cannot be resolved. For example, if Alice is a member of the Scientist group, which is a member of the Researcher group, which is in turn a member of the Employee group, then authorization for the Researcher group should take precedence over a conflicting authorization for the Employee group.

Example 4.1. The above database authorization policy can easily be represented in GCLP. Giving the group $Researcher$ a weak authorization to perform the "select" operation on the table $T5$ can be represented by the following rule:

$\langle$auth(weak,Researcher)$\rangle$ authorizes(?A, select, T5) $\leftarrow$

user(?A),member(?A, Researcher).

The conflict-resolution policies can be represented by the following two rules:

overrides(auth(strong,?G1), auth(weak,?G2)).

overrides(auth(weak,?G1), auth(weak,?G2)) $\leftarrow$ member(?G1, ?G2).

4.1.3 Semantics of GCLP

The intuitive meaning of GCLP's conflict handling mechanism is as follows. When a rule's body is true, we say that this rule is *ready* and that it generates a *candidate* for its head. A candidate R for a classical literal L_1 is refuted if there is a candidate Q for another classical literal L_2 such that L_2 conflicts with L_1 and $overrides(Q's\ label, R's\ label)$ is true. A literal L is true if and only if there is an unrefuted candidate for it and there is no unrefuted candidate for any classical literal that conflicts with L. This semantics never concludes both L and something that conflicts with L. For example, when there are unrefuted candidates for both p and $\neg p$, neither p nor $\neg p$ is concluded; they "skeptically defeat" each other. This semantics is "skeptical" (in the sense used in the nonmonotonic reasoning literature) in that it doesn't conclude a conclusion when "in doubt" about it.[2]

GCLP's semantics is formally defined by the following transformation from a

[2]We can choose a paraconsistent semantics if so desired, by dropping the requirement that L is true only when there doesn't exist any unrefuted candidate for any classical literal that conflicts with L. A paraconsistent semantics may conclude L and $\neg L$ at the same time. This may be desirable for some applications. See [22] for a survey on paraconsistent semantics for logic programs.

GCLP to an OLP. We follow the transformation that is given in [35] and implemented in IBM CommonRules V1.1, but we change the notation slightly. A GCLP $\mathcal{G}$ is compiled into an OLP $\mathcal{O}$ through the following steps.

1. **a.** For each predicate $pred/z$ in $\mathcal{G}$ (z is the arity of the predicate $pred$, *i.e.*, the number of arguments it takes), introduce a new predicate n_pred/z to represent $pred$'s classical negation and add the following mutex to $\mathcal{G}$.

   $$\perp \leftarrow pred(?x_1, \ldots, ?x_z), n\_pred(?x_1, \ldots, ?x_z).$$

   b. Then, in $\mathcal{G}$, replace each literal $\neg\, pred(t_1, \ldots, t_z)$ with $n\_pred(t_1, \ldots, t_z)$.

   c. Denote by $\mathcal{G}_0$ the result of the above transformation. Let $\mathcal{G}_1$ be the set of rules in $\mathcal{G}_0$ and $\mathcal{G}'$ be the set of all mutex's in $\mathcal{G}_0$. Let $\mathcal{O}$ be an empty set.

2. **a.** For each predicate $opred/z$ in $\mathcal{G}_1$ (including the new predicates introduced in step 1.a for classical negation), introduce two new predicates: $opred^u/z$ and $opred^s/z$.

   b. For any literal $L = opred(t_1, \ldots, t_z)$ in $\mathcal{G}_1$, define L^u to be $opred^u(t_1, \ldots, t_z)$ and L^s to be $opred^s(t_1, \ldots, t_z)$. Intuitively, L^u is true when there is an unrefuted candidate for L, and L^s is true when L is skeptically defeated, *i.e.*, when L^u is true, and there is an unrefuted candidate for some literal that conflicts with L.

3. For each mutex μ in $\mathcal{G}'$, do the following:

   Let μ be "$\perp \leftarrow L_1, L_2 \mid body_\mu$."

   a. Define $ready_\mu$ to be the atom $ready\_pred_\mu(?x_1, \ldots, ?x_w)$, in which $ready\_pred_\mu$ is a new predicate, and "$?x_1, \ldots, ?x_w$" are all the variables in "L_1, L_2."

   b. Add the following rule to $\mathcal{O}$:

   $$ready_\mu \leftarrow body_\mu.$$

The purpose of this step is to handle those mutex's whose bodies contain variables that do not appear in their heads. This step is not needed (and does not exist) in [35], because this kind of mutex is not allowed in [35]. Without this step, we should replace the $ready_\mu$ in step 5.b with $body_\mu$.

4. For each rule R in $\mathcal{G}_1$, do the following:

   Let R be "$\langle lab_R \rangle \;\; head_R \leftarrow body_R$."

   a. Define $ready_R$ to be the atom $ready\_pred_R(?x_1, \ldots, ?x_w)$ and $refuted_R$ to be the atom $refuted\_pred_R(?x_1, \ldots, ?x_w)$. Here, $ready\_pred_R$ and $refuted\_pred_R$ are new predicates and "$?x_1, \ldots, ?x_w$" are all the variables in "$\langle lab_R \rangle \;\; head_R$."

   b. Then add the following three rules to $\mathcal{O}$:

   $$ready_R \leftarrow body_R.$$

   $$head^u_R \leftarrow ready_R, \sim refuted_R.$$

   $$head_R \leftarrow head^u_R, \sim head^s_R.$$

   Intuitively, they mean the following: A rule is ready if its body is true; when a rule is ready and is not refuted, its head has an unrefuted candidate; if a literal has an unrefuted candidate and is not skeptically defeated, then the literal is true, *i.e.*, concluded.

5. For each pair of rules R, Q in $\mathcal{G}_1$ and each mutex μ in $\mathcal{G}'$:

   a. Let R be "$\langle lab_R \rangle \;\; head_R \leftarrow body_R$"; let Q be "$\langle lab_Q \rangle \;\; head_Q \leftarrow body_Q$"; and let μ be "$\perp \leftarrow L_1, L_2 \mid body_\mu$." Without loss of generality, assume that no variable appears in any two of R, Q, and μ; one can always rename variables when necessary.

   b. Let θ be the most general unifier of "$(head_R, head_Q)$" and "(L_1, L_2)." If θ

exists, add the following two rules to $\mathcal{O}$:

$$refuted_R\theta \leftarrow ready_R\theta, ready_\mu\theta,\ ready_Q\theta,\ overrides(lab_Q, lab_R)\theta.$$

$$head^s_R\theta \leftarrow head_\mu\theta, ready_\mu\theta,\ head^u_Q\theta.$$

Here, appending θ to a term (*e.g.*, $overrides(lab_Q, lab_R)\theta$) means applying the substitution θ to the term. The first rule means that R is refuted if R is ready, Q's head conflicts with R's head, Q is ready, and Q's label overrides R's label. The second rule means that R is skeptically defeated if R has an unrefuted candidate, Q's head conflicts with R's head, and Q's head has an unrefuted candidate.

The resulting program $\mathcal{O}$ is an OLP of size $O(|\mathcal{G}|^3)$. Recall that the "size" of a program is the number of symbols, *i.e.*, variables, constants, predicate symbols, logical operators, *etc*. Note that $|\mathcal{G}_0| = O(|\mathcal{G}|)$, because the new mutex's being added in step 1.a are about predicates in $\mathcal{G}$. Moreover, for each rule or mutex R in $\mathcal{G}_0$, steps 2–4 contribute to $\mathcal{O}$ a set of new rules that has total size $O(|R|)$. For each tuple Q, R and μ in $\mathcal{G}_0$, assume, without loss of generality, that R is the largest one; then step 5 may generate two new rules, each of which has size $O(|R|)$. We charge this constant growth factor to R. After step 5 is finished, each rule or mutex will be charged at most $O(\max(N_R N_\mu, N_R^2))$ times, where N_R is the number of rules in $\mathcal{G}_0$, and N_μ is the number of mutex's in $\mathcal{G}_0$. Both N_R and N_μ are $O(N)$; therefore, $|\mathcal{O}| = O(|\mathcal{G}_0|^3) = O(|\mathcal{G}|^3)$.

The semantics of a GCLP is defined by the semantics of its corresponding OLP. There are multiple choices available for OLP semantics. Among them, the two leading ones are well-founded semantics (WFS) [30] and stable model semantics [31]. Some OLP programs do not have a stable model, and some have more than one. Furthermore, even for a propositional program, determining whether it has a stable model is NPcomplete [55]. On the other hand, WFS assigns a unique three-valued model to every OLP program. For a finite ground program, the complexity of computing its well-founded model is worst-case quadratic in the its size. Mainly because of these behavior and computational complexity features of WFS, the previous definition of GCLP [34, 35] recommends to use WFS for GCLP. We like these advantages of WFS as well.

The GCLP model of $\mathcal{G}$ can be computed as follows: First transform $\mathcal{G}$ into an OLP $\mathcal{O}$; then compute the WFS model of $\mathcal{O}$, and, finally, translate the conclusions in $\mathcal{O}$'s WFS model back into GCLP by discarding the new predicates introduced in steps 2-4 and translating $n\_pred(t_1, \ldots, t_z)$ to $\neg\, pred(t_1, \ldots, t_z)$. Note that we assume that the predicates introduced during the transformation are distinguishable from those in the input program.

One can also compile GCLP queries into OLP queries and evaluate them in OLP. To compile a GCLP query into OLP, one only needs to replace $\neg\, pred$ with n_pred. In OLP inferencing, one can detect whether there is a conflict about a given literal lit by checking whether lit^s is true.

4.1.4 Complexity results

Theorem 4.1. *The transformation from a GCLP $\mathcal{G}$ to its corresponding OLP takes time $O(N^3)$, and it generates an output OLP of size $O(N^3)$, where $N = |\mathcal{G}|$.*

Proof. The $O(N^3)$ size bound follows from the definition of the transformation. The definition corresponds straightforwardly to an algorithm linear in the size of the output OLP. Note that there are linear-time algorithms for unification. (See page 26 of [53] for further references.) □

The worst-case size is reached when both the number of mutex's and the number of rules are linear in N, and almost all pairs of rules are potentially in conflict. This is highly unlikely in practice. If an input program has the property that, for any rule, there are at most a constant number of rules that potentially conflict with it, then the resulting program has size linear in the size of the input program.

The following theorem shows that GCLP inferencing is tractable under restrictions similar to those under which OLP inferencing is tractable (*e.g.*, Datalog and bounded number of logical variables per rule). We say that an LP obeys the VBL(v) restriction when it is VB(v) (page 66) and each rule is label-range-restricted (page 85). If an LP is both VBL(v) and Datalog, we say that it is VBLD(v).

Theorem 4.2. *If a GCLP $\mathcal{G}$ is VBLD(v), then inferencing of $\mathcal{G}$ takes time $O(N^{2(3+v)})$, where $N = |\mathcal{G}|$.*

Proof. First $|\mathcal{O}| = N^3$. The key observation is that, when $\mathcal{G}$ is VBLD(v), the transformation maintains the per-rule number-of-variables bound. Then the ground instantiation of $\mathcal{O}$ has size N^{3+v}, because the Datalog restriction implies that there are $O(N)$ terms that can be used to instantiate each variable. Furthermore, because WFS inferencing takes worst-case quadratic time, we have the bound $O(N^{2(3+v)})$ for GCLP inferencing.

We now show that the transformation maintains the variable bound. Each new rule added in step 3 or step 4 has at most v variables, because it only has variables from one rule in $\mathcal{G}$. Now consider the two rules added in step 5:

$$refuted_R\theta \leftarrow ready_R\theta, ready_\mu\theta,\ ready_Q\theta,\ overrides(lab_Q, lab_R)\theta.$$
$$head^s_R\theta \leftarrow head^u_R, ready_\mu\theta,\ head^u_Q\theta.$$

Because each rule in $\mathcal{G}$ is label-range-restricted, the only variables in the two new rules before applying θ are the variables in "$(head_R, head_Q)$" and "(L_1, L_2)." Note that, under the Datalog restriction, when θ unifies two terms t_1 and t_2, $nv((t_1\theta, t_2\theta)) \leq \min(nv(t_1), nv(t_2))$, where $nv(t)$ is the number of variables in t. Similarly, when θ unifies t_1, t_2, and t_3, $nv((t_1\theta, t_2\theta, t_3\theta)) \leq \min(nv(t_1), nv(t_2), nv(t_3))$.

If the mutex μ is from $\mathcal{G}$, then there are at most v variables in "(L_1, L_2)." Because θ unifies "$(head_R, head_Q)$" and "(L_1, L_2)," there are at most v variables in each of the two new rules. If μ is a mutex added in the transformation, then $(L_1, L_2) = (pred(?x_1, \ldots, ?x_z), n\_pred(?x_1, \ldots, ?x_z))$. Therefore, θ unifies the arguments of $head_R$, the arguments of $head_Q$, and $(?x_1, \ldots, ?x_z)$. Again there are at most v variables in each of the two new rules. □

Inferencing for a Datalog OLP that is VBD(v) takes time $O(N^{2(1+v)})$, and so the worst-case GCLP inferencing complexity is equivalent to adding two variables per rule.

4.2 D2LP: A Nonmonotonic Delegation Logic

In this section, we define D2LP, a nonmonotonic delegation logic, by extending D1LP to have the nonmonotonic features in GCLP.

4.2.1 Subtleties of integrating delegation and nonmonotonicity

D1LP's semantics answers both "who says what?" and "who delegates to whom?". However, answering delegation queries and simultaneously resolving conflicts is subtle. Consider the following example:

Example 4.2 (Interaction of delegation and conflict resolving).

$\langle$A1$\rangle$ Alice delegates p^2 to Bob.

$\langle$B1$\rangle$ $\mathrm{Bob\ delegates\ p\hat{\ }1\ to\ Carl.}$

$\langle$B2$\rangle$ $\mathrm{Bob\ says\ \neg\, p.}$

$\langle$B3$\rangle$ $\mathrm{Bob\ says\ overrides(B2,\ B1).}$

$\langle$C1$\rangle$ $\mathrm{Carl\ says\ p.}$

Should one conclude that "$\mathrm{Alice\ delegates\ p\hat{\ }1\ to\ Carl}$"? By chaining the delegation $\langle$A1$\rangle$ and $\langle$B1$\rangle$, one can argue that it should be concluded. Then, because of the fact $\langle$C1$\rangle$, one should also conclude that "$\mathrm{Alice\ says\ p.}$" However, this is counter-intuitive. Intuitively, the conclusion p propagates from Carl through Bob to Alice. However, by $\langle$B2$\rangle$ and $\langle$B3$\rangle$, the conclusion p is blocked at Bob; therefore it should not reach Alice.

Motivated by this subtlety, in this paper, we restrict D2LP to satisfy the "delegation-query-free (DQF) restriction." I.e., we define D2LP on top of D1LP^{DQF} (section3.6), instead of standard D1LP.

4.2.2 Syntax of D2LP

The syntax of D2LP is extended from that of D1LP^{DQF} in the following ways:

(1) classical negation ($\neg$) can be used before base atoms;

(2) negation-as-failure ($\sim$) can be used before body statements;

(3) a rule can optionally have a label;

(4) there are mutex statements;

(5) there is a predefined predicate $overrides$.

Next, we give a concise, self-contained version of D2LP's syntax.

1. A *base atom* takes the form $pred(t_1, \dots, t_n)$. A *base literal* takes the form[3] $pred(t_1, \dots, t_n)$ or $\neg\, pred(t_1, \dots, t_n)$. Like GCLP, D2LP has a reserved binary

[3]We recommend using ! in place of $\neg$ when ASCII representation is desired.

predicate $overrides$ for prioritization.

2. A *direct statement* takes the form "X `says` lit." A *delegation statement* takes the form "X `delegates` lit`^`d `to` XS." A *speaks_for statement* takes the form "Y `speaks_for` X `on` lit." Here X and Y are *principal terms*. A principal term is either a principal or a principal variable. lit is a base literal. XS is a *complex principal term*, *i.e.*, either a *principal structure* or a principal variable; it is called the *delegatee* of the delegation statement. Principal structures are constructed from principals and threshold structures using ","(conjunction) and ";"(disjunction). (See section 3.1 for definition and discussion of threshold structures, delegation statements, and speaks_for statements.)

   A *mutex statement* takes the form "X `says` $lit1$ `opposes` $lit2$." Here X is a principal term, and $lit1$ and $lit2$ are base literals. Intuitively, this statement means that, in X's view, $lit1$ and $lit2$ conflict with each other, *i.e.*, "X says $lit1$" and "X says $lit2$" are mutually exclusive.

3. A *body statement* is either a *body-direct statement*, which takes the form "XS says lit," or a *negation-as-failure statement*, which takes the form: "$\sim XS$ says lit." Here XS is a complex principal term.

4. A rule takes the form: $\langle lab \rangle\ head\ \ \texttt{if}\ \ body.$

   Here $head$ is a direct statement, a delegation statement, a speaks_for statement, or a mutex statement, and $body$ is a formula constructed from body statements using ","(conjunction) and ";"(disjunction). Note that delegation statements, speaks_for statements, and mutex statements are not allowed to appear in rule-bodies. The rule labels are used to resolve conflicts between direct statements derived from rules.

Example 4.3 (Nonmonotonic credit determining).

In this program, Alice authorizes anyone she believes to have good credit to do transactions and delegates the right to determine who has good credit to credit bureaus and allows them to further delegate one more step. Alice also delegates the ability to determine that someone has bad credit to any fraud expert, and bad-credit information overrides good-credit information. Finally, there is an expert Bob whom Alice trusts unconditionally on with respect to credit-worthiness. Bob's decision has the highest priority. These policies are represented by the following program:

$\text{Alice says authorizes(?P, transaction)} \;\; \text{if} \;\; \text{Alice says credit(?P, good).}$

$\langle\text{trusted}\rangle \;\; \text{Alice delegates credit(?P, ?Status)\^{}* to Bob.}$

$\langle\text{good}\rangle \;\; \text{Alice delegates credit(?P, good)\^{}2 to ?X if Alice says creditBureau(?X).}$

$\langle\text{bad}\rangle \;\; \text{Alice delegates credit(?P, bad)\^{}1 to ?X} \;\; \text{if Alice says fraudExpert(?X).}$

$\text{Alice says credit(?P, good) opposes credit(?P, bad).}$

$\text{Alice says overrides(bad, good).}$

$\text{Alice says overrides(trusted, good).}$

$\text{Alice says overrides(trusted, bad).}$

4.2.3 Semantics of D2LP

D2LP's semantics is defined by transforming a D2LP $\mathcal{P}$ into a GCLP $\mathcal{G}$. This transformation is, for the most part, similar to the transformation from D1LP into OLP in section 3.2.

In addition to the reserved predicate $overrides$, $\mathcal{G}$ has one more predicate, $holds$. An atom of $holds$ takes the form: $holds(X, lt, len)$ (see page 47).

For each $pred$ in $\mathcal{P}$, we introduce two function symbols $pred$ and nd_pred. The function $pred$ is used to represent a base atom, and nd_pred is used to represent a negated base atom. We use $\overline{lt}$ to denote the term that corresponds to lt's classical negation.

In the transformation, we need to use the function $PSFormula$ defined in section 3.2 to expand statements with complex principal structures as issuers.

Transformation 0: Label and Negation transformation

This transformation changes rules in $\mathcal{P}$; the result is called $\mathcal{P}_0$.

- For each rule R in $\mathcal{P}$, let R be "$\langle labc(t_1, \dots, t_n)\rangle\ \ head_R$ `if` $body_R$"; change its label to "$labc(X, t_1, \dots, t_n)$," where X is the issuer of $head_R$.

  This step makes sure that each principal has a different name space for labels.

- In $\mathcal{P}$, replace each base literal $\neg\, pred(\dots)$ with $nd\_pred(\dots)$, in which nd_pred is a newly introduced predicate.

Transformation I: Body transformation

This transformation changes rule-bodies in $\mathcal{P}_0$; the result is called $\mathcal{P}_1$.

- **Holds body translation:**

  Replace each body-direct statement "XS `says` lt"
  with "$PSFormula(XS, holds(lt, *))$."

- **Negation body translation:**

  Replace each negation-as-failure statement "$\sim XS$ `says` lt" with the "DeMorganization" of "$\sim PSFormula(XS, holds(lt, *)$," *i.e.*, the negation $\sim$ is pushed inside conjunctions and disjunctions.

For example, the statement $\sim (A, threshold(2, [B, C, D]))$ `says` $p(a)$ is replaced by $(\sim holds(A, p(a), *); \sim holds(suth(2, [B, C, D]), p(a), *))$.

Transformation II: Head transformation

This transformation changes rule heads in $\mathcal{P}_1$, removes some rules, and adds some new rules; the result is called $\mathcal{P}_2$.

For each rule R in $\mathcal{P}_1$, let R be "$\langle lab_R \rangle$ $head_R$ `if` $body_R$"; there are three cases for $head_R$:

Case one: $head_R$ is a direct statement "X `says` lt."

- Replace R's head with "$holds(X, lt, 1)$."

Case two: $head_R$ is a mutex statement "X `says` $lt1$ `opposes` $lt2$."

- Remove R and, for each $len1, len2 \in [1..*]$, add the rule:

  $\perp \leftarrow holds(X, lt1, len1), holds(X, lt2, len2) \mid body_R.$

Case three: $head_R$ is either a delegation statement or a speaks_for statement.

If $head_R$ is a delegation statement "A `delegates` lt`^`d `to` BS," then let ll be 1.

If $head_R$ is a speaks_for statement "B `speaks_for` A `on` lt," then let d be $*$, ll be 0, and let BS be B.

- **Delegation expansion:** Remove R and, for each $len \in [1..d]$, add the rule:

  $\langle lab_R \rangle \;\; holds(A, lt, len \oplus ll) \leftarrow body_R, PSFormula(BS, holds(lt, len)).$

In cases one and three, also do the following.

- For each $len \in [1..D]$, add the rule

  $holds(X, lt, len \oplus 1) \leftarrow holds(X, lt, len).$

- For each $len1, len2 \in [1..*]$, add the mutex:

  $\perp \leftarrow holds(A, lt, len1), holds(A, \overline{lt}, len2).$

- If lt is "$overrides(labc_1(t_{11}, \ldots, t_{1u}), labc_2(t_{21}, \ldots, t_{2v}))$," then add the rule:

$$overrides(labc_1(X, t_{11}, \ldots, t_{1u}), labc_2(X, t_{21}, \ldots, t_{2v}))$$
$$\texttt{if}\ \ holds(X, overrides(labc_1(t_{11}, \ldots, t_{1u}), labc_2(t_{21}, \ldots, t_{2v})), *).$$

Threshold structures are handled similarly to the way they are handled in subsections 3.2.2 and 3.2.3. Here, we omit the details.

The goal of the above transformation is to define the intended semantics of D2LP. We have found several possible further tweaks to the transformation that would "optimize" it in the sense of resulting in fewer output rules while maintaining equivalent semantics. However, these optimizations do not improve the asymptotic bound of the output size. Thus, we choose to use this computationally slightly more expensive but clearer definition.

An important property of this transformation is that it does not introduce any new variables. More precisely, for each rule in $\mathcal{G}$, all the variables in it come from a single rule in $\mathcal{P}$. As in section 3.3.2, we assume that the transformation generates a typed GCLP. The transformation from GCLP to OLP simply passes the typing onto the generated OLP.

4.2.4 Inferencing and Complexity Results

Theorem 4.3. *The transformation from D2LP to GCLP generates an output program of size $O(N^3D)$. Computing the transformation takes time $O(N^3D)$.*

Proof. By the counting argument in section 3.2, the function $PSFormula$ has an $O(N^2)$ growth factor. The delegation-expansion step generates the largest output among all steps. It generates $O(D)$ rules, and each one uses $PSFormula$. Therefore, this step has growth factor $O(N^2D)$. Thus, the output program has size $O(N^3D)$. The definition of the transformation corresponds straightforwardly to an algorithm linear in the size of the output program. □

D2LP inferencing can be done by first compiling a D2LP into a GCLP, then computing its minimal GCLP model, and finally translating the conclusions back into D2LP. Each GCLP conclusion $holds(A, pred(\ldots), len)$ is translated to "A says $pred(\ldots)$", and each conclusion $holds(A, nd\_pred(\ldots), len)$ is translated to "A says $\neg pred(\ldots)$." Another way to do inferencing is as follows. First compile a D2LP program and a D2LP query into GCLP; a query is compiled in the same way as a rule-body. Then compile the GCLP query and program into OLP. And finally use an OLP inference engine to answer these queries.

Example 4.4 (Nonmonotonic credit determining (continued)).

We now add the following facts to example 4.3:

$\text{Alice says creditBureau(cb1).}$ $\text{Alice says fraudExpert(Carl).}$

$\text{cb1 says credit(Jack, good).}$ $\text{Bob says credit(John, good).}$

$\text{Carl says credit(John, bad).}$ $\text{Carl says credit(Jack, bad).}$

Then, one can conclude "$\text{Alice says credit(John, good)}$" and "$\text{Alice says credit(Jack, bad)}$."

Recall that we say that an LP is VBLD(v) if it is Datalog, each rule in it has at most v variables, and, for each rule, the variables in its label also appear in its head.

Theorem 4.4. *Inferencing of a D2LP $\mathcal{P}$ that is VBLD(v) takes time polynomial in $(ND)^v$. When v is a constant and $D = O(N)$, inferencing of a D2LP takes time polynomial in N.*

Proof. Given a D2LP $\mathcal{P}$ that is VBLD(v), the output GCLP $\mathcal{G}$ has size $O(N^3D)$, and G is VBL(v). Typing ensures that variables in $\mathcal{G}$ will only be instantiated to constants in

$\mathcal{P}$. (Note that $\mathcal{P}$ is Datalog.) When we compile $\mathcal{G}$ into an OLP $\mathcal{O}$, $|\mathcal{O}| = O((N^3D)^3)$, and $\mathcal{O}$ has the same variable bound. Instantiating $\mathcal{O}$ increases its size by $O(N^v)$, and so the size of instantiated $\mathcal{O}$ is $O(N^{9+v}D^3)$. Inferencing of $\mathcal{O}$ takes time quadratic in its instantiated size; therefore, the inferencing time is polynomial in $O(N^{2(9+v)}D^6)$. $\square$

The complexity bound in the proof is a high-degree polynomial. We can get a tighter bound by doing a more detailed analysis. One observation is that not all pairs of rules in $\mathcal{G}$ are potentially in conflict with each other. Detailed analysis of the theoretical worst-case complexity of D2LP inferencing is not the most important task at hand, however; rather, the practicality of D2LP needs to be tested in experiments.

Chapter 5

Conclusions

In this dissertation, we presented the logic-programming-based language Delegation Logic (DL) for representing security policies and credentials for authorization in large-scale, open, distributed systems.

Our general approach to designing DL was to extend existing well-understood logic-programming (LP) languages with features that are needed in distributed authorization. More specifically, we added issuers and delegation constructs to existing LP languages. D1LP, the monotonic version of Delegation Logic, extends Definite Ordinary LP (a.k.a. Definite LP, see [53]), and D2LP, the nonmonotonic version of Delegation Logic, extends Generalized Courteous Logic Programs (GCLP) [34, 35, 36]. Our approach to defining the semantics of DL was to define a transformation from DL programs into programs in the underlying logic-programming language. We defined a transformation from D1LP into OLP and a transformation from D2LP into GCLP, for which there already exists a transformation into OLP. We showed that each of these transformation steps is computationally tractable and that D1LP inferencing and D2LP inferencing are thus tractable under a broad restriction similar to that which ensures tractability of OLP inferencing.

The transformation-based approach gives us easy access to the established results for OLP; for example, every DL program has a minimal model. Thus, DL differs from other proposed trust-management engines [9, 11, 12, 25] in providing a notion of "credentials proving that a request complies with a policy" that is not entirely *ad hoc*; rather, it is based on model-theoretic semantics and is thus abstracted away from choice and details of implementation.

The transformation-based approach also yields a natural implementation architecture for DL; it can be implemented by using a *delegation compiler* from DL to OLP. This enables DL to be implemented modularly on top of existing technologies for OLP, which include not only Prolog but also SQL relational databases and many other rule-based/knowledge-based systems. We discussed two existing implementations of D1LP.

A major challenge in designing DL is the need to strike a right balance among expressive power, computational complexity, and ease of understanding. In order to allow the specification of complex trust and delegation relationships in distributed authorization, we add to DL explicit linguistic support for delegation depth and for a wide variety of complex principals (*e.g.*, dynamic threshold structures). To achieve tractable D1LP inferencing, we impose the conjunctive-delegatee-queries restriction on D1LP. To achieve an intuitive semantics for D2LP, we restrict D2LP to be delegation-query free, because of some subtleties that arise when the delegation constructs are combined with the nonmonotonic features.

A major challenge in designing a knowledge representation (KR), especially one that is nonmonotonic or multi-agent, is to achieve usefully rich expressiveness and intuitively natural semantics, *together with* moderate computational complexity and relative ease of incorporation into existing software environments. We believe Delegation Logic represents significant progress along these lines. Other contributions of this work include:

- Delegation Logic provides a notion of "proof of compliance" that is founded on well-understood principles of logic programming and knowledge representation.
- Delegation Logic provides an expressive yet tractable and practically implementable trust-management language.
- Delegation Logic provides a logical framework for studying various delegation features, nonmonotonic security policies, and their interplay.

Bibliography

[1] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin, "A Calculus for Access Control in Distributed Systems," *ACM Transactions on Programming Languages and Systems*, 15:4, pp. 706–734, October 1993. Also available as SRC Research Report 70.
ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-070.pdf

[2] Ross J. Anderson, "A Security Policy Model for Clinical Information Systems," in *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pp. 30–43, IEEE Computer Society Press, Los Alamitos, CA, 1996.
http://www.cl.cam.ac.uk/ftp/users/rja14/oakpolicy.ps.Z

[3] Ken Arnold and James Gosling, *The Java Programming Language, Second Edition*, Addison Wesley, Reading, MA, 1998.

[4] Tuomas Aura, "Fast Access Control Decisions from Delegation Certificate Databases," in *Proceedings of 3rd Australasian Conference on Information Security and Privacy (ACISP '98)*, Lecture Notes in Computer Science, vol. 1438, pp. 284-295, Springer, Berlin, 1998.
http://www.tcs.hut.fi/Publications/papers/aura/aura-acisp98.ps

[5] Chitta Baral and Michael Gelfond, "Logic Programming and Knowledge Representation," *Journal of Logic Programming*, 19/20, pp. 73–148, May/Jun 1994. http://www.cs.utep.edu/gelfond/papers/survey.ps

[6] Elisa Bertino, Francesco Buccafurri, Elena Ferrari, and Pasquale 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, IEEE Computer Society Press, Los Alamitos, CA, 1999.

[7] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati, "A Flexible Authorization Mechanism for Relational Data Management Systems," *ACM Transactions on Information Systems*, 17:2, pp. 101-140, April 1999. This extends [8].

[8] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati, "Supporting Multiple Access Control Policies in Database Systems," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pp. 94–107, IEEE Computer Society Press, Los Alamitos, CA, 1996.
http://www.isse.gmu.edu/˜csis/publications/oklnd96-samarati.ps

[9] Matt Blaze, Joan Feigenbaum, and Jack Lacy, "Decentralized Trust Management," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pp. 164–173, IEEE Computer Society Press, Los Alamitos, CA, 1996. http://www.crypto.com/papers/policymaker.pdf

[10] Matt Blaze, Joan Feigenbaum, Paul Resnick, and Martin Strauss, "Managing Trust in an Information-Labeling System," in *European Transactions on Telecommunication*, vol 8, pp. 491–501, 1997.
http://www.si.umich.edu/˜presnick/papers/bfrs/

[11] Matt Blaze, Joan Feigenbaum, and Martin Strauss, "Compliance-Checking in the PolicyMaker Trust Management System," in *Proceedings of Financial Crypto '98*, Lecture Notes in Computer Science, vol. 1465, pp. 254–274, Springer, Berlin, 1998. http://www.crypto.com/papers/pmcomply.pdf

[12] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis, "The KeyNote Trust-Management System, Version 2," Internet Engineering Task Force RFC 2704, September 1999. http://www.ietf.org/rfc/rfc2704.txt

[13] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis, "The Role of Trust Management in Distributed Systems," in *Secure Internet Programming*, Lecture Note of Computer Science, vol. 1603, pp. 185-210, Springer, Berlin, 1999. http://www.crypto.com/papers/trustmgt.pdf

[14] Gerhard Brewka and Jürgen Dix, "Knowledge Representation with Logic Programs," in *Handbook of Philosophical Logic*, second edition, vol. 6, Oxford University Press, Oxford, England, to appear 2001. A preliminary version is available at http://www.uni-koblenz.de/~dix/Papers/01_Handbook_PhilLog.ps.gz

[15] David F. C. Brewer and Michael J. Nash, "The Chinese Wall Security Policy," in *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pp. 206–218, IEEE Computer Society Press, Los Alamitos, CA, 1989.

[16] M. Calejo, "InterProlog: A Java front-end and enhancement for Prolog," http://dev.servisoft.pt/interprolog/.

[17] David Chappell, *Understanding ActiveX and OLE*, Microsoft Press, Redmond, WA, 1996.

[18] Weidong Chen and David S. Warren, "Tabled Evaluation with Delaying for General Logic Programs," *Journal of the ACM*, 43:1, pp. 20–74, Janurary 1996. ftp://ftp.cs.sunysb.edu/pub/XSB/doc/SLG/slg-jacm.ps.gz

[19] Laurence Cholvy and Frédéric Cuppens, "Analyzing Consistency of Security Policies," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 103-112, IEEE Computer Society Press, Los Alamitos, CA, 1997. http://www.cert.fr/francais/deri/cholvy/monweb/oakland97.ps

[20] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss, "REFEREE: Trust Management for Web Applications," *World Wide Web Journal*, 2, pp. 706–734, 1997. http://www.research.att.com/~trmaster/TRs/97/97.2/97.2.1.body.ps

[21] David D. Clark and David R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," in *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pp. 184–194, IEEE Computer Society Press, Los Alamitos, CA, 1987.

[22] Carlos V. Damásio and Luís M. Pereira, "A Survey of Paraconsistent Semantics for Logic Programs," in D.M. Gabbay and Ph. Smets, editors, *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, vol. 2, pp. 241-320, Kluwer Academic Publishers, 1998.

[23] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest, "Certificate Chain Discovery in SPKI/SDSI," manuscript, Nov 1999. http://theory.lcs.mit.edu/~rivest /ClarkeElElFrMoRi-CertficateChainDiscoveryInSPKISDSI.ps

[24] Carl Ellison, “SPKI/SDSI Certificate Documentation.” http://world.std.com/˜cme/html/spki.html

[25] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen, “SPKI Certificate Theory,” Internet Engineering Task Force RFC 2693, September 1999. http://www.ietf.org/rfc/rfc2693.txt

[26] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen, “Simple Public Key Certificate,” Internet Draft (Work in Progress), July 1999. http://world.std.com/˜cme/spki.txt

[27] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi, *Reasoning About Knowledge*, MIT Press, Cambridge, MA, 1995.

[28] Hal Finney, “Transitive Trust and MLM,” post to cypherpunks mailing list, May 1996. http://www.inet-one.com/cypherpunks/dir.1996.05.02-1996.05.08/msg00415.html

[29] Warwick Ford, *Computer Communications Security – Principles, Standard Protocols and Techniques*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[30] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf, “The Well-Founded Semantics for General Logic Programs,” *Journal of the ACM*, 38:3, pp. 620-650, July 1991. http://www.cs.columbia.edu/˜kar/pubsk/wfjacm.ps

[31] Michael Gelfond and Vladimir Lifschitz, “The Stable Model Semantics for Logic Programming,” in *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 1070–1080, MIT Press, Cambridge, MA, 1988.

[32] Benjamin N. Grosof, "Courteous Logic Programs: Prioritized Conflict Handling for Rules," IBM Research Report RC20836, May 1997. This is an extended version of [33]. http://www.research.ibm.com/people/g/grosof/paps/rc20836.pdf

[33] Benjamin N. Grosof, "Prioritized Conflict Handling for Logic Programs," in *Proceedings of the International Symposium on Logic Programming (ILPS-97)*, pp. 197–211, MIT Press, Cambridge, MA, 1997.

[34] Benjamin N. Grosof, "Compiling Prioritized Default Rules Into Ordinary Logic Programs," IBM Research Report RC 21472, May 1999. http://www.research.ibm.com/people/g/grosof/paps/rc21472.pdf

[35] Benjamin N. Grosof, "A Courteous Compiler From Generalized Courteous Logic Programs To Ordinary Logic Programs," manuscript, August 1999. This extends [34]. http://www.research.ibm.com/people/g/grosof/paps/gclp_report1.pdf

[36] Benjamin N. Grosof, Yannis Labrou, and Hoi Y. Chan, "A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML," in *Proceedings of the 1st ACM Conference on Electronic Commerce (EC-99)*, ACM Press, New York, 1999. http://www.research.ibm.com/people/g/grosof/paps/ec99_proceedings.pdf

[37] Benjamin N. Grosof, Hoi Y. Chan, *et al.*, IBM CommonRules home pages, http://alphaworks.ibm.com/tech/commonrules and http://www.research.ibm.com/rules/

[38] Amir Herzberg, Yosi Mass, Joris Mihaeli, Dalit Naor, and Yiftach Ravid, "Access Control Meets Public Key Infrastructure, Or: Assigning Roles to

Strangers," in *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pp. 2–14. IEEE Computer Society Press, Los Alamitos, CA, 2000. http://www.hrl.il.ibm.com/TrustEstablishment/paper.pdf

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

[40] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian, "A Logical Language for Expressing Authorizations," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 31–42, IEEE Computer Society Press, Los Alamitos, CA, 1997.
http://www.isse.gmu.edu/˜csis/publications/oak97-jss.ps

[41] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and E. Bertino, "A Unified Framework for Enforcing Multiple Access Control Policies," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 474-485, 1997. http://www.isse.gmu.edu/˜csis/publications/sigmod97.ps

[42] Charlie Kaufman, Radia Perlman, and Mike Speciner, *Network Security: Private Communication in a Public World*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[43] Stephen T. Kent, "Internet Privacy Enhanced Mail," *Communications of the ACM*, 36:8, pp. 48–60, August 1993.

[44] National Computer Security Center, U.S. Department of Defense, *Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD, 1985. http://csrc.nist.gov/publications/history/dod85.pdf

[45] John T. Kohl, B. Clifford Neuman, and Theodore Y. Ts'o, "The Evolution of

the *Kerberos* Authentication Service," in F. Brazier and D. Johansen, editors, *Distributed Open Systems*, pp. 78-94, IEEE Computer Society Press, Los Alamitos, CA, 1994. ftp://athena-dist.mit.edu/pub/kerberos/doc/krb_evol.PS

[46] Raph Levien, Lewis McCarthy, and Matt Blaze, "Transparent Internet E-mail Security," manuscript, 1996. http://www.crypto.com/papers/email.pdf

[47] Jack Lacy, James H. Synder, and David P. Maher, "Music on the Internet and the Intellectual Property Protection Problem," in *Proceedings of the International Symposium on Industrial Electronics*, pp. 77-83, IEEE Press, 1997. http://www.a2bmusic.com/docs/musicipp.doc

[48] Butler Lampson, Martín Abadi, Mike Burrows, and Edward Wobber, "Authentication in Distributed Systems: Theory and Practice," *ACM Transactions on Computer Systems*, 10:4, pp. 265–310, November 1992. Also available as SRC Research Report 83.
ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-083.pdf

[49] Ninghui Li, Joan Feigenbaum, and Benjamin N. Grosof, "A Logic-Based Knowledge Representation for Authorization with Delegation (Extended Abstract)," in *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW-12)*, pp. 162-174, IEEE Computer Society Press, Los Alamitos, CA, 1999. Full paper available as IBM Research Report RC21492. http://cs.nyu.edu/ninghui/papers/ibmrr.ps

[50] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum, "A Practically Implementable and Tractable Delegation Logic," in *Proceedings of the 2000 IEEE Sym-*

posium on Security and Privacy, pp. 27–42, IEEE Computer Society Press, Los Alamitos, CA, 2000. http://cs.nyu.edu/ninghui/papers/s&p00.ps

[51] Ninghui Li, "XD1LP: An Implementation of D1LP in XSB." http://cs.nyu.edu/ninghui/xd1lp/

[52] Ninghui Li, "Local Names for SPKI/SDSI," in *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pp. 2–15, IEEE Computer Society Press, Los Alamitos, CA, 2000. http://cs.nyu.edu/ninghui/papers/csfw13.ps

[53] John W. Lloyd, *Foundations of Logic Programming, Second, Extended edition*, Springer, Berlin, 1987.

[54] E. Lupu and M. Sloman, "Conflict in Policy-based Distributed Systems Management," *IEEE Transaction on Software Engineering – Special Issue on Inconsistency Management*, 25/26, November/December 1999. http://www-dse.doc.ic.ac.uk/~ecl1/papers/TSE/web.pdf

[55] W. Marek and M. Trusczynski, *Nonmonotonic Logic – Context-Dependent Reasoning*, Springer, Berlin, 1993.

[56] Ueli Maurer, "Modelling a Public-Key Infrastructure," in *Proceedings of the 1996 European Symposium on Research in Computer Security*, Lecture Notes in Computer Science, vol. 1146, pp. 325–350, Springer, Berlin, 1997. ftp://ftp.inf.ethz.ch/pub/publications/papers/ti/isc/Pub_Key_Model.ps.gz

[57] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone,

Handbook of applied cryptography, CRC Press, Boca Raton, FL, 1996. http://cacr.math.uwaterloo.ca/hac/

[58] Lee Naish, "Types and the Intended Meaning of Logic Programs," in [63], pp. 189–216, 1992.

[59] Roger M. Needham and Michael D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM*, 21:12, pp. 993-999, December 1978.

[60] B. Clifford Neuman and Theodore Ts'o, "Kerberos: An Authentication Service for Computer Networks," *IEEE Communication Magzine*, 32:9, pp. 33–38, September 1994. http://nii.isi.edu/publications/kerberos-neuman-tso.html

[61] Ilkka Niemelä and Patrik Simons, *et al.*, "Smodels home page." http://www.tcs.hut.fi/Software/smodels/

[62] Peter Padawitz, *Computing in Horn Clause Theories*, EATCS monographs on Theoretical Computer Science, vol. 16, Springer, Berlin, 1988.

[63] Frank Pfenning (editor), *Types in Logic Programming*, MIT Press, Cambridge, MA, 1992.

[64] Paul Resnick and James Miller, "PICS: Internet Access Controls without Censorship," *Communications of the ACM*, 39:10, pp. 87–93, October 1996. http://www.bilkent.edu.tr/pub/WWW/PICS/iacwc.htm

[65] Ron Rivest and Bulter Lampson, "SDSI - A Simple Distributed Security Infrastructure," October 1996. http://theory.lcs.mit.edu/~cis/sdsi/sdsi11.html

[66] Tatyana Ryutov and Clifford Neuman, "Representation and Evaluation of Security Policies for Distributed System Services," in *Proceedings of the 2000 DARPA Information Survivability Conference and Exposition*, IEEE Computer Society Press, Los Alamitos, CA, January 2000.
http://www.isi.edu/~tryutov/papers/discex2000.pdf

[67] Ravi Sandhu, "Role-Based Access Control Models," *Advances in Computers*, vol. 46, Academic Press, San Diego, CA, 1998.
http://www.list.gmu.edu/articles/advcom/a98rbac.ps

[68] Ravi Sandhu and Pierangela Samarati, "Authentication, Access Control and Intrusion Detection," in Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, CRC Press, Boca Raton, FL, 1997.
http://www.list.gmu.edu/articles/crc/c96acid.ps

[69] Williams Stallings, *Cryptography and Network Security: Principles and Practice*, second edition, Prentice Hall, Englewood Cliffs, 1999.

[70] David S. Warren *et al.*, "The XSB Programming System."
http://www.cs.sunysb.edu/~sbprolog/xsb-page.html

[71] Edward Wobber, Martín Abadi, Mike Burrows, and Butler Lampson, "Authentication in the TAOS Operating System," *ACM Transactions on Computer Systems*, 12:1, pp. 3–32, Februry 1994. Also available as SRC Research Report 117.
ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-117.pdf

[72] Thomas Y.C. Woo and Simon S. Lam, "Authorization in Distributed Systems: A New Approach," *Journal of Computer Security*, 2:2/3, pp. 107–136, 1993.
ftp://ftp.cs.utexas.edu/pub/lam/final-1.ps.gz

[73] Thomas Y.C. Woo and Simon S. Lam, "Designing a Distributed Authorization Service," in *Proceedings of IEEE INFOCOM '98*, March 1998. ftp://ftp.cs.utexas.edu/pub/lam/info98b.ps.gz

[74] Thomas Y.C. Woo and Simon S. Lam, "Authentication for Distributed Systems," in Dorothy Denning and Peter Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*, ACM Press and Addison Wesley, 1998. ftp://ftp.cs.utexas.edu/pub/lam/denning.ps.Z

[75] WWW Consorortium, "Platform for Privacy Preferences (P3P) Project." http://www.w3.org/P3P/

[76] WWW Consorortium P3P Preference Interchange Language Working Group, "A P3P Preference Exchange Language (APPEL)," W3C Working Draft, April 2000, http://www.w3.org/TR/P3P-preferences

[77] Philip R. Zimmermann, *The Official PGP User's Guide*, MIT Press, Cambridge, MA, 1995.