Delegation Logic: A Logic-based Approach to Distributed Authorization

NINGHUI LI
Stanford University
BENJAMIN N. GROSOF
Massachusetts Institute of Technology
and
JOAN FEIGENBAUM
Yale University

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, privacy protection, and many other applications. 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, called *Delegation Logic* (DL), to represent policies, credentials, and requests in distributed authorization. In this paper, we describe D1LP, the monotonic version of DL. D1LP extends the logic-programming (LP) language Datalog 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). Our approach to defining and implementing D1LP is based on tractably compiling D1LP programs into ordinary logic programs (OLP's). This compilation approach enables D1LP to be implemented modularly on top of existing technologies for OLP, *e.g.*, Prolog.

As a trust-management language, D1LP provides a concept of proof-of-compliance that is founded on well-understood principles of logic programming and knowledge representation. D1LP also provides a logical framework for studying delegation.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection (D.4.6,K.4.2)

General Terms: Security, Languages

Additional Key Words and Phrases: access control, trust management, Delegation Logic, distributed system security, logic programs

---

Parts of this paper were published in preliminary form in the following two papers: "A Logic-Based Knowledge Representation for Authorization with Delegation (Extended Abstract)" in *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, June 1999. "A Practically Implementable and Tractable Delegation Logic" in *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000. Most of this work was performed while the first author was a PhD student at the Department of Computer Science, New York University in New York, NY 10012, the second author was a member of the IBM T.J. Watson Research Center in Hawthorne, NY 10532, and the third author was a member of the Information Sciences Research Center of AT&T Labs in Florham Park, NJ 07932.
Authors' addresses: Ninghui Li, Department of Computer Science, Gates 4B, Stanford, CA 94305-9045, USA; email: ninghui@stanford.edu; home page: http://crypto.stanford.edu/~ninghui. Benjamin N. Grosof, MIT Sloan School of Management, 50 Memorial Drive, Cambridge, MA 02142, USA; email: bgrosof@mit.edu; home page: http://www.mit.edu/~bgrosof. Joan Feigenbaum, Department of Computer Science, Yale University, PO Box 208285, New Haven, CT 06520-8285, USA; email: jf@cs.yale.edu; home page: http://www.cs.yale.edu/~jf

1. INTRODUCTION

In today's Internet, there are a large and growing number of scenarios that require authorization decisions. Such scenarios include but are not limited to electronic commerce, resource sharing within coalitions, execution of downloadable code (*e.g.*, Java applets and ActiveX controls), and privacy protection [13; 27]. Authorization in these scenarios is significantly different from that in centralized systems or even in distributed systems that are closed or relatively small. In Internet authorization scenarios, there are more entities in the system, many of which are unknown to each other, and often there is no central authority that everyone trusts.

Traditional access control mechanisms make authorization decisions based on the identity of the resource requester. Unfortunately, when the resource owner and the requester are unknown to one another, access control based on identity may be ineffective. 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 Internet authorization different from traditional access control. The goal of a growing body of work on *trust management* is to find a more expressive and "distributed" approach to authorization in these scenarios. Several trust-management (TM) systems have been proposed in recent years, *e.g.*, PolicyMaker [8; 9], KeyNote [6; 7], REFEREE [11], and SPKI/SDSI [12; 15]. A key feature of these systems is the support of delegation. There has also been a lot of work on analysis of these TM systems [1; 3; 19; 21; 30; 42].

In the "trust-management" approach to 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?"

Credentials may be facts (*e.g.*, Joe is a student at Stanford University), or, more generally, non-local policy statements that are more complicated than facts. Because credentials are not always under the control of the authorizer, they need to be protected against alteration; thus, credentials are often signed by public keys. The TM 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 "permissions" [6; 15], with various attributes of key-holders, or with fully programmable "capabilities" [8; 9; 11]. Identity information is just one kind of credential, and it may be necessary and sufficient for some applications but not for others.

Some TM systems, such as KeyNote [6] and the original SPKI 1.0, use credentials only to delegate permissions. Each credential delegates certain permissions from its issuer to its subject. A chain of one or more credentials acts as a capability,

granting certain permissions to the subject of the last credential in the chain.[1] However, in these permission-based systems, one cannot express the fact that the issuer grants permissions to all entities that have a certain property. To better understand this limitation, consider a simple example: A book store wants to give a 15% discount to students of a nearby university. Ideally, the book store would express this in one policy statement, and a student could get the discount by showing her student ID, issued by the university. However, one cannot follow this simple approach in permission-based TM systems. For instance, in KeyNote, one cannot use an assertion to express that anyone who is a student is entitled to a discount. There are several ways to work around this limitation. One approach is to have the book store delegate the discount permission to the university's key. Then the university's key explicitly delegates this permission to each student's key. However, when there are many organizations giving different permissions to students, the university has to issue a separate delegation to each student for every permission; this places too heavy an administrative burden on the university. In another approach, the university creates a new key pair representing the group of all students and issues a complete delegation from this group key to each student's public key. This would allow the book store to establish the student discount policy by granting the discount to the group key. However, the university would then have a different group key for each meaningful group, *e.g.*, faculties, graduate students, *etc.* And the book store needs to know which key corresponds to the group of students; this would require another TM system, because it cannot be done using KeyNote. There are several other work-arounds that use KeyNote's action-environment feature. Like the above two approaches, they can handle this simple example but do not scale to more complicated cases in which there are many organizations that want to give different kinds of permissions to students of the university or in which the book store wants to give discounts to different groups of different organizations, *e.g.*, students of the university, graduate students of the university, faculties of the university, employees of a nearby company and members of a professional organization, *etc.*

To simplify authorization in decentralized environments, we need a system in which access-control decisions are based on authenticated attributes of the subjects, and attribute authority is decentralized. We argue that an expressive TM language should be able to express the following:

(1) Decentralized attributes: An entity asserts that another entity has a certain attribute. For example, a university asserts that an entity is a student. A permission can be viewed as an attribute as well; granting a permission p to an entity can be viewed as asserting that the entity has the attribute p.

(2) Delegation of attribute authority: An entity delegates the authority over an attribute to another entity, *i.e.*, the entity trusts another entity's judgment about the attribute. For example, a university delegates the authority to identify students to the registrar of one campus of the university.

(3) Inference of attributes: An entity uses one attribute to make inferences about

[1] Because KeyNote and SPKI have thresholds, a capability could be a directed graph of credentials. This does not affect our discussion of their limitations below.

another attribute. For example, a book store gives a discount to any entity who is a student of a particular university.

(4) Attribute-based delegation of attribute authority: A key to scalability is the ability to delegate to strangers whose trustworthiness is determined based on their own authenticated attributes. For example, when an online book store wants to give discounts to all university students, instead of having to issue a delegation to each university, the book store should be able to issue one statement in which it delegates the authority to identify students to entities that are certified universities. The book store can then delegate the authority to identify universities to an accrediting board, avoiding having to know all the universities.

(5) Conjunction of attributes: An entity uses the conjunction of several attributes to make inferences about another attribute. For example, a book store gives a special discount to any entity that is both a student and a member of the ACM.

(6) Attribute with fields: It is often useful to have attribute credentials carry field values, such as age and credit limit. Permissions may also have fields describing resources and access modes. It is also useful to infer additional attributes based on these field values and to delegate attribute authority to a certain entity only for certain specific field values, *e.g.*, only when spending level is below a certain limit.

We believe that a desirable trust-management language should satisfy requirements in the following three areas. For each area, we list a basic requirement and a stronger requirement.

expressive power The basic requirement is that a TM language should support the six features listed above.
The stronger requirement is that it should also support other desirable features such as thresholds and some form of re-delegation control.

declarative semantics The basic requirement is that a TM language should have a declarative, clearly specified notion of proof of compliance. For example, a TM language that allows credentials to contain programs written in procedural programming languages does not satisfy this requirement.
The stronger requirement is that the notion of proof-of-compliance should be based on a well-understood, formal foundation. Furthermore, there should be a "meaning" for every set of policies and credentials, so that one can compute this meaning and inspect whether it is the same as the policy author's intention.

tractability The basic requirement is that compliance checking should be tractable, *i.e.*, polynomial in the size of policies, credentials, and requests.
The stronger requirement is that computing the meaning of a set of policies and credentials should also be tractable.

As we will discuss in Section 2, none of the previous TM languages satisfies the three basic requirements. The goal of this paper is to provide a "trust-management language" for representing authorization policies and credentials that, firstly, satisfies the three basic requirements, and secondly, also satisfies the three stronger requirements.

We view the problem of designing a TM language for representing authorization policies and credentials as a knowledge-representation problem. We adopt an approach that has proven useful in knowledge representation: the logic-programming approach. Logic-programming-based languages for representing security policies have been studied before (*e.g.*, [5; 22; 23]), but previous work focused on centralized environments and did not address the delegation aspect of distributed authorization.

We propose the logic-programming-based language *Delegation Logic* (DL) as a trust-management language. Our approach in designing DL is to extend well-understood logic-programming languages with features needed for distributed authorization. Specifically, DL extends Definite Ordinary Logic Programs[2] 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. In this paper, we focus on the delegation aspect and 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.

The rest of this paper is organized as follows. In Section 2, we give background information on trust management and analyze several previous TM systems. In Section 3, we introduce the language D1LP. In Section 4, 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 5. In Section 6, we discuss the tractability motivation behind a design decision of D1LP. In Section 7, we describe implementation and other issues involved in using D1LP. We conclude in Section 8.

2. BACKGROUND

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

[2] "Ordinary" logic programs (OLP's) correspond essentially to pure Prolog 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 [4; 35].

requester, the authorizer then decides whether this requester is allowed to access the requested resource. This step is called *access control*.

We use the term *authorization* to denote this process of "authentication + access control." This paper focuses on authorization in emerging applications in large-scale, open, decentralized, distributed systems (*e.g.*, Internet). Authorization in these decentralized environments is significantly different from traditional authorization in single-host systems or in centrally controlled distributed systems. Basic differences include:

—**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, decentralized 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 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 meant 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. The real property one needs for identity is that one can verify that

a request or a credential is issued by a particular identity and that one can link the particular identity to its credentials. One can argue that, in a global system, the only real "identity" to which anything can later be related is the public key. Thus, the "trust-management approach" adopts a key-centric view of authorization: Public keys are treated as principals and authorized directly.

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.

2.1 Sample Policies for Testing Expressive Features

Before we review previous TM systems, we first give a list of sample policies that can be used to test their expressive power:

Decentralized attribute A hospital HA asserts that an entity PA is a physician.

Delegation of attribute authority A hospital HM trusts another hospital HA to identify physicians.

Inference of attributes A hospital HM allows an entity to access a document if it is a physician.

Attribute-based delegation of authority A hospital HM trusts any entity that is a hospital to identify physicians.

Conjunction of attributes A hospital HM gives special permissions to anyone who is both a physician and a manager.

Attribute with fields A hospital HM allows an entity to access the records of a patient if the entity is the physician of the patient.

These six types of policies were discussed in Section 1. Here, we give two additional policies that use threshold structures.

Static threshold structures A bank requires two out of an explicitly given list of entities to cooperate in order to complete a certain transaction.

Dynamic threshold structures A bank requires two cashiers to cooperate in order to complete a certain transaction; whether an entity is a cashier is not given explicitly in this policy, but rather is determined by inferencing of that entity's attributes from credentials.

Static threshold structures are often known as "k-out-of-n thresholds." They are common in existing TM systems, *e.g.*, PolicyMaker [8; 9], KeyNote [6], and SPKI/SDSI [12; 15]. However, these systems do not have dynamic threshold structures. A static threshold structure becomes inconvenient when its threshold pool is very large, changes very often, or both. In the above cashier policy, dynamic threshold structures allow a simple and clear policy and enable the bank to change the set of cashiers without changing its policy.

2.2 Review of Previous Trust-management Systems

We now briefly review several systems for authentication and/or access control in distributed systems: PolicyMaker [8; 9], REFEREE [11], KeyNote [6], SPKI/SDSI [15; 12], and SRC logic [2; 26]. We choose these systems because

they all (more or less) satisfy the following three conditions, which we consider to be the core of the trust-management approach to distributed authorization.

—The system supports decentralized storage and management of authorization information. As a result, in a typical authorization scenario, a requester submits a request with some supporting credentials, an authorizer adds local policy and maybe other credentials, and then uses a "proof-of-compliance" procedure to determine whether the request should be authorized.

—The language for representing credentials and policies supports decentralized attributes and delegation of attribute authority.

—The language has an application-independent semantics.

2.2.1 *PolicyMaker and REFEREE.* PolicyMaker was introduced by Blaze, Feigenbaum, and Lacy, in the original paper in which the notion of Trust Management was introduced [8]. PolicyMaker's compliance-checking algorithm was later fleshed out in [9]. For more details about PolicyMaker, see [7; 8; 9].

In PolicyMaker, policies and credentials together are referred to as "*assertions*." An assertion is 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. The program f 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 [8]).

Because the assertions in PolicyMaker may include arbitrary programs in some Turing-complete programming language, it is quite expressive in the sense that one can code up complex policies and delegation relationships in PolicyMaker. However, it does not satisfy the basic requirement for declarative semantics. Moreover, the PolicyMaker framework has built-in support only for decentralized attributes; the other features need to be explicitly coded in programs by assertion authors.

REFEREE [11] is similar to PolicyMaker; it also allows arbitrary programs to be used in credentials and policies.

2.2.2 *KeyNote.* KeyNote [6] is a second-generation TM system that is based on PolicyMaker. Instead of allowing programs written in a general-purpose procedural language, KeyNote adopts a specific expression language. A KeyNote assertion is a delegation from its issuer to a licensees formula. A primitive licensees formula is simply a principal. More complicated licensees formulas are built up from principals using conjunction, disjunction, and thresholds. An assertion also has conditions written in the expression language. The intuitive meaning of an assertion is that, if the licensees support a request, and the request satisfies the conditions, then the issuer supports the request as well.

When using the KeyNote system, a calling application passes to a KeyNote evaluation engine a list of credentials, policies, requesters' public keys, and an "action environment," which consists of a list of attribute/value pairs. An action environment essentially specifies a request. The evaluation engine uses the credentials and policies to determine whether the local authority supports this request given that all the requesters support the request.

In a KeyNote assertion, the conditions only filter what requests (in the form of

action environments) are delegated in this delegation; they do not apply to the licensees, and so the licensees have to be explicitly listed. As a result, KeyNote does not support inference of attributes, attribute-based delegations, conjunction of attributes, attribute with fields, or dynamic threshold structures. Also, KeyNote has no re-delegation control; any permission can be freely re-delegated.

KeyNote satisfies the basic requirement for declarative semantics by giving a procedure to answer whether a specific request should be authorized given a set of credentials. KeyNote defines a mechanism to compute whether a particular request is authorized given a set of policies and credentials; however, it does not define a mechanism to compute the meaning of a set of policies and credentials.

2.2.3 *SPKI/SDSI.* SDSI (Simple Distributed Security Infrastructure) was originally designed by Rivest and Lampson [41]. SPKI (Simple Public Key Infrastructure) was originally designed by Ellison. Both of these systems were motivated by the inadequacy of public-key infrastructures based on global name hierarchies, such as X.509 [40] and Privacy Enhanced Mail (PEM) [25]. Later, SPKI and SDSI merged into a collaborative effort, SPKI/SDSI 2.0, about which the most up-to-date documents are [12; 15; 16]. In our discussion, we use SPKI 2.0 to denote the part of SPKI/SDSI 2.0 originally from SPKI, *i.e.*, authorization certificates (5-tuples), and SDSI 2.0 to denote the part originally from SDSI, *i.e.*, name certificates (or 4-tuples as in [15]).

SPKI 2.0 is quite similar to KeyNote. There are two main differences. The first lies in the encoding of permissions being delegated in one credential. SPKI 2.0 represent the permissions using tags in a special form of s-expressions; these s-expressions can be viewed as a particular kind of constraints over strings. For example, an s-expression "(ftp ftp.clark.net (* prefix /pub/abc/) (* set read write))" can encode the permission of ftp access to the host 'ftp.clark.net' and read/write all files and directories under '/pub/abc/'. KeyNote uses expressions written in an expression language to filter requests that can pass through a delegation. To encode the above permission in KeyNote, one can use "(protocol=='ftp' && host=='ftp.clark.net' && dir~='^/pub/abc/.*' && (access=='read' || access=='write'))," in which ~= is regular expression matching. The second difference is that SPKI 2.0 has boolean re-delegation control, while KeyNote has no re-delegation control. Otherwise, SPKI 2.0 has the same limitations as KeyNote.

The document defining SPKI [15] describes how one can chain two 5-tuples to get a new one. This operation uses tag intersection; however, only an incomplete prose description and several examples of tag intersections are given in [15]. Most work attempting to formalize SPKI avoids the complexity of tags by using simpler models and does not faithfully capture tags. A notable exception is [21], in which Howell pointed out that intersection between some kinds of tags may not be finitely representable using tags; he gave a precise specification for tag intersection, in which intersections in potentially problematic cases are artificially defined to be empty.

SDSI 2.0 supports decentralized attributes, delegation of attribute authority, inference of attributes, and attribute-based delegation, the last of which is supported through linked local names. However, SDSI 2.0 does not support attributes with fields or conjunction of attributes. Although SPKI 2.0 can express attributes with fields using tags, SPKI/SDSI 2.0 does not really merge these features together,

because they are present in different kinds of certificates. In SPKI/SDSI 2.0, one still cannot express conjunction of attributes, attributes with fields, or dynamic threshold structures.

2.2.4 *SRC Logic for Authentication and Access Control.* Abadi, Burrows, Lampson, Plotkin, Wobber, *et al.* developed a logic for authentication and access control in distributed systems [2; 26]. They also designed and implemented a security system based on this logic. The core concept in SRC logic is a "speaks for" relation among principals; that A speaks for B means that, if principal A makes a statement, then we can believe that principal B makes it, too. The notion of "speaks for" is very useful in distributed authentication and access control, and it underlies the notion of delegation in trust management. That A speaks for B can be viewed as a delegation of all authority from B to A. The SRC logic is designed mainly for authentication; its total delegation has too coarse a granularity for access control. To limit the authority being delegated in the logic, a principal can adopt a role before delegating. By using roles, one can achieve effects roughly similar to decentralized attributes and delegation of attribute authority. However, SRC logic still lacks attribute-based delegation, attributes with fields, or threshold structures.

3. D1LP: SYNTAX, CONCEPTS, AND EXAMPLES

In this section, we first define the syntax of D1LP and show that D1LP has the expressive features listed in Section 2.1. We then explain basic concepts in D1LP and give examples of D1LP programs.

3.1 Syntax

Figure 1 gives the syntax of D1LP in BNF. The *alphabet* of D1LP consists of three disjoint sets: the *predicate symbols*, the *variables*, and the *constants*. Variables start with '?'. The set of *principals* is a subset of the constants and 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. In the following, we explain the syntax. The numbers in the text below correspond to the numbers of definitions in Figure 1.

—A *term* (3) is either a constant or a variable. When a variable appears in certain positions, it is called a *principal variable* (4) and can be instantiated only to a principal. (The exact positions in which a variable should be treated as a principal variable can be determined from the BNF in Figure 1.) A *principal term* (5) is either a principal or a principal variable.

—A base atom (6) encodes a belief. For example, "isBusinessKey(keyBob,Bob)" and "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 is a belief that this action should happen. For example, the base atom "remove(file1)" encodes the belief that file1 should be removed.

—In a direct statement (7) "X `says` ba," X is called the *issuer* of this statement. This statement intuitively means that X "*supports*" the belief encoded in ba. For example, a direct statement "Bob says remove(file1)" means that Bob supports that file1 should be removed; *i.e.*, it represents Bob's request to remove

⟨list of X⟩ ::= ⟨X⟩ | ⟨X⟩ ⟨list of X⟩ (1)

⟨formula of X⟩ ::= ⟨X⟩ | ⟨formula of X⟩ “,” ⟨formula of X⟩ |
⟨formula of X⟩ “;” ⟨formula of X⟩ | “(” ⟨formula of X⟩ “)” (2)

⟨term⟩ ::= ⟨constant⟩ | ⟨var⟩ (3)

⟨prin-var⟩ ::= ⟨var⟩ (4)

⟨prin-term⟩ ::= ⟨prin-var⟩ | ⟨prin⟩ (5)

⟨base-atom⟩ ::= ⟨pred⟩ | ⟨pred⟩ “(” ⟨list of term⟩ ”)” (6)

⟨direct-stmt⟩ ::= ⟨prin-term⟩ “says” ⟨base-atom⟩ (7)

⟨delegation-stmt⟩ ::= ⟨prin-term⟩ “delegates” ⟨base-atom⟩ “^” ⟨depth⟩
“to” ⟨prin-exp⟩ (8)

⟨depth⟩ ::= ⟨natural-number⟩ | “*” (9)

⟨representation-stmt⟩ ::= ⟨prin-term⟩ “represents” ⟨prin-term⟩ (10)

⟨prin-exp⟩ ::= ⟨prin-var⟩ | ⟨prin-struct⟩ (11)

⟨prin-struct⟩ ::= ⟨formula of prin-element⟩ (12)

⟨prin-element⟩ ::= ⟨prin⟩ | ⟨threshold⟩ (13)

⟨threshold⟩ ::= ⟨su-threshold⟩ | ⟨sw-threshold⟩ | ⟨du-threshold⟩ (14)

⟨su-threshold⟩ ::= “threshold” “(” ⟨k⟩ “,” “[” ⟨list of prin⟩ “]” “)” (15)

⟨sw-threshold⟩ ::= “threshold” “(” ⟨k⟩ “,” “[” ⟨list of prin-weight-pair⟩ “]” “)” (16)

⟨du-threshold⟩ ::= “threshold” “(” ⟨k⟩ “,” ⟨prin-var⟩ “,” ⟨prin⟩ “says” ⟨base-atom⟩ “)” (17)

⟨k⟩ ::= ⟨natural-number⟩ (18)

⟨prin-weight-pair⟩ ::= “(” ⟨prin⟩ “,” ⟨natural-number⟩ “)” (19)

⟨rule⟩ ::= ⟨head-stmt⟩ | ⟨head-stmt⟩ “if” ⟨body-formula⟩ (20)

⟨head-stmt⟩ ::= ⟨direct-stmt⟩ | ⟨delegation stmt⟩ | ⟨representation-stmt⟩ (21)

⟨body-formula⟩ ::= ⟨formula of body-stmt⟩ (22)

⟨body-stmt⟩ ::= ⟨b-direct-stmt⟩ | ⟨b-delegation-stmt⟩ | ⟨representation-stmt⟩ (23)

⟨b-direct-stmt⟩ ::= ⟨prin-exp⟩ “says” ⟨base-atom⟩ (24)

⟨b-delegation-stmt⟩ ::= ⟨prin-exp⟩ “delegates” ⟨base-atom⟩ “^” ⟨depth⟩
“to” ⟨conj-prin-exp⟩ (25)

⟨conj-prin-exp⟩ ::= ⟨prin-var⟩ | ⟨conj-prin-struct⟩ (26)

⟨conj-prin-struct⟩ ::= ⟨prin⟩ | ⟨prin⟩ “,” ⟨conj-prin-struct⟩ (27)

⟨locale-decl-stmt⟩ ::= “`Local`” “standsfor” ⟨prin⟩ (28)

⟨program⟩ ::= ⟨list of rule⟩ | ⟨locale-decl-stmt⟩ ⟨list of rule⟩ (29)

⟨query⟩ ::= ⟨body-formula⟩ “?” (30)

Fig. 1. Syntax of D1LP in BNF, in which ⟨pred⟩, ⟨var⟩, ⟨constant⟩, and ⟨prin⟩ represent a predicate symbol, a variable, a constant, and a principal respectively. The first two definitions, ⟨list of X⟩ and ⟨formula of X⟩, are macros.

file1. A statement "keyBob says goodCredit(Carl)" means that keyBob supports (believes) that Carl has good credit.

—In a *delegation statement* (8) "X `delegates` $ba\hat{}d$ `to` PE," X is called the *issuer* of this statement; d is called the *delegation depth* (9) of this delegation ("$*$" means unlimited depth); and the principal expression PE is called the *delegatee* of this delegation. In DL, the basic meaning of a delegation is *transferability of support*. For example, the delegation statement "Bob delegates goodCredit(?X)^1 to Carl" means that, if Carl supports that someone has good credit, then Bob supports it as well. The meaning of delegation depths is discussed in Section 3.3.

—In a *representation statement* (10) "Y `represents` X `on` ba," the issuer is defined to be the special principal `Local`. This representation statement intuitively means that Y has all power that X has with respect to the base atom ba, *i.e.*, if `Local` trusts X about ba, then it should also trust Y equally about ba. It is similar to the delegation statement "X `delegates` $ba\hat{}*$ `to` Y," but there are important differences. These differences and the rationale for representation statements are discussed in Section 3.5.

—A principal expression (11) is a principal variable or a principal structure (12, 13), which is a formula (2) of principals and threshold structures. Threshold structures (14) introduce fault tolerance and aid flexibility in joint authorization. D1LP has three kinds of threshold structures.

—In a *static unweighted threshold structure* (15) "`threshold` $(k, [A_1, \ldots, A_n])$," we call k the *threshold value* and "$[A_1, \ldots, A_n]$" the *threshold pool*, and require $k \leq n$ and $A_i \neq A_j$ for $1 \leq i \neq j \leq n$.
For example, "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 "[cardA, cardB, cardC]" support ba.

—In a *static weighted threshold structure* (16) "`threshold` $(k, [(A_1, w_1), \ldots, (A_n, w_n)])$," we call the *principal-weight pair set* "$[(A_1, w_1), \ldots, (A_n, w_n)]$" the threshold pool of this threshold structure, and require[3] that $k \leq n$ and $A_i \neq A_j$ for $1 \leq i \neq j \leq n$.
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.

—A *dynamic unweighted threshold structure* (17) "`threshold` $(k, ?X, Prin$ `says` $ba)$)," in which we require that $?X$ appears in ba and define the threshold pool to be the set of all principals A such that the direct statement "$Prin$ `says` $pred(\ldots A \ldots)$" is true, *i.e.*, the expression "$Prin$ `says` $pred(\ldots ?X \ldots)$" becomes true when A is substituted for $?X$ for each appearance throughout the direct statement.

[3]It is possible to relax the restriction that $k \leq n$ for static threshold structures. For static unweighted threshold structures, if $k > n$, then the threshold structure can never be satisfied. For static weighted threshold structures, relaxing to permit $k > n$ will require a more subtle discussion about tractability. A relatively simple relaxation is to permit k to be at most αn, where α is some integer constant, *e.g.*, 10, in which case the complexity bound in Section 5 increases by a factor of α.

For example, "`threshold` (2, ?X, Bank says isCashier(?X))" is a dynamic threshold structure. In another example, the principal structure

(threshold(1, ?X, companyA says accountant(?X)),
threshold(1, ?Y, companyA says manager(?Y)))

represents any conjunction of an accountant in companyA and a manager in companyA. Note that, if a principal is both an accountant and a manager, then the principal satisfies this principal structure. If one wants to make managers and accountants disjoint, one needs to use non-monotonic features, which do not exist in D1LP.

—A *rule* (20) is also known as a *clause*. In a rule "H `if` F," H is a direct statement, a delegation statement, or a representation statement and is called the *head* (21) of the rule; F is a formula of body statements and called the *body* (22) 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 a rule's head 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, and the *issuer* of this rule is the principal symbol `Local`.
For example, A is the issuer of the rule "A says p if B says q" and the issuer of "A delegates p^1 to B." `Local` is the issuer of "A represents B on p" and the issuer of "?X says p(a) if Local says c(?X,a)." Intuitively, the issuer of a rule is the principal who has the power to issue that rule. Before using a rule, one should verify that the rule is actually issued by its issuer. This is discussed further in Sections 3.4 and 3.5.

—A *body-statement* (23) is either a body-direct statement, a body-delegation statement, or a representation statement. Body statements allow principal structures to be used as issuers, so that one can use more expressive conditions in rules. A delegation statement appearing in a query or a rule-body (because a rule-body 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 called the *conjunctive-delegatee-queries* restriction. It is imposed to ensure computational tractability of D1LP. This rationale is discussed in detail in Section 6.

—A *body-direct statement* (24) 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: "threshold(2, [cardA, cardB, cardC]) says accountGood(?X)."

—A *body-delegation statement* (25) allows the issuer to be a principal structure and requires its delegatee to be a principal, a principal variable, or a conjunction of principals.

—A *locale-declaration* statement (28) specifies that `Local` stands for a particular principal. The intended use of this statement is to specify the current trust root as this principal. Semantically, this statement is essentially a macro.

—A *program* (29) is a finite set of rules plus an optional locale-declaration statement. The locale-declaration statement is required if `Local` appears in any of

the rules. The set of rules is also known as a *logic program (LP)* or as a *rule-set*.

—A *query* (30) 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.

3.2 Expressive Power of D1LP

In this section, we show how to use D1LP to express the policies in Section 2.1.

—Decentralized attribute. HA asserts that PA is a physician:
```
HA says isPhysician(PA).
```

—Delegation of attribute authority. HM trusts HA to identify physicians.
```
HM delegates isPhysician(?X)^1 to HA.
```

—Inference of attributes. HM allows anyone who is a physician to access a document.
```
HM says access(fileA, ?X) if HM says isPhysician(?X).
```

—Attribute-based delegation of authority. HM trusts any entity that is a hospital to identify physicians.
```
HM delegates isPhysician(?X)^1 to ?Y if HM says isHospital(?Y).
```

—Conjunction of attributes. HM allows anyone who is both a physician and a manager to access a document.
```
HM says access(fileB, ?X)
   if HM says isPhysician(?X), HM says isManager(?X).
```

—Attribute with fields. HM allows an entity to access the records of a patient if the entity is the physician of the patient.
```
HM says accessDocument(?X, ?Y) if HM says isPhysicianOf(?X,?Y).
```

—Static threshold structures. A bank requires two out of four entities `A,B,C,D` to cooperate in order to approve a transaction T.
```
Bank delegates approve(T) to threshold(2, [A,B,C,D]).
```

—Dynamic threshold structures. A bank requires two cashiers cooperate to cooperate in order to approve a transaction T.
```
Bank delegates approve(T)
   to threshold(2, ?X, Bank says isCashier(?X)).
```

3.3 Discussions of Delegation Depth

As we defined earlier, a delegation statement has a depth, which is either a positive integer or "$*$." We now discuss the meaning and rationale of delegation depths.

One way to view delegation depth is the number of re-delegation steps that are allowed, where depth 1 means that no re-delegation is allowed, depth 2 means that one further step is allowed, depth 3 means that two further steps are allowed, and depth $*$ means that unlimited re-delegation is allowed. Consider the following delegation statement:

Local delegates read(file1)^2 to Bob.

It means that Local delegates to Bob the permission to read file1 and allows Bob to further delegate one more step. Bob can delegate this permission to another entity

say, Carl, but one step of depth is consumed in this re-delegation step. Carl cannot effectively delegate this permission further.

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.

DL provides both integer depth and unlimited depth as ways to control re-delegation. By contrast, there are three other more restricted approaches in the previous literature: (1) no control, in which every delegation can be re-delegated unlimitedly (*i.e.*, depth * only), (2) boolean control, in which a delegation allows either unlimited re-delegation or no re-delegation at all (*i.e.*, depth 1 or * only), and (3) integer control (*i.e.*, integer depth only). KeyNote uses the approach of no control, and SPKI uses the boolean approach. In Section 4 of [15], SPKI designers gave the following two reasons for choosing boolean control over integer control (in Section 4.1.4 of [15]):

> 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.

We disagree with these arguments for the following three reasons. First, whether delegation depths can be properly predicted depends on the application using the system. Because TM systems are designed for a broad range of applications, saying that one cannot predict adequate depth in any application seems unconvincing. Second, in scenarios in which one needs to create a temporary signing key for laptop use, DL's approach works better than SPKI's boolean approach. Suppose that one's policy is to give out permissions to users, not to let them delegate to other users, but still allow them to empower temporary keys. In the boolean approach, one has no choice but to allow infinite delegation. In DL, one can use delegation depth 2. Although this is still not perfect, as one can use the extra depth to delegate to another user, the control is much tighter than in the boolean approach. Third, we cannot see why one should give up depth control because width control is difficult. Moreover, as we will see in example 4, it is possible to control the width of a delegation in DL, by using a conjunction of principals as its delegatee.

We believe that the verdict is still out on what kind of re-delegation control will be found to be most useful in practice. However, we think that DL's approach is attractive because it is strictly more expressive than any of the other three. If one does not need integer depth in one credential, one can always use just * and 1.

A more philosophical reason for limiting delegation with respect to depth is that trust is not transitive (see [17; 36] for some interesting discussions). For example, the delegation statement

Bob delegates goodCredit(?X)^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 goodCredit(?X), and Carl trusts David about goodCredit(?X), Bob does not necessarily trust David about goodCredit(?X). It is imaginable that Bob trusts Carl's ability to judge whether someone has good credit but does not 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. 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.

The above intuition based on trust transitivity yields the same behavior as the intuition based on controlling re-delegation steps. One can use delegation depth, without knowing or understanding this more philosophical reason for delegation depths.

3.4 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 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 1 (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 whomever BankB trusts in determining credit ratings. ShopA also has a credential issued by BankB saying that BankB believes that a principal has good credit if two out of three particular credit-card companies certify that this principal has an account in good standing. These policies and credentials are represented as follows:

```
ShopA says approveOrder(?U)  if  ShopA says creditRating(?U, good).
ShopA delegates creditRating(?U, ?R)^2 to BankB.

BankB says creditRating(?U, good)
      if  threshold(2,[cardW,cardX,cardY]) says accountGood(?U).
```

Now a customer Carl sends an order to ShopA and provides the following two cre-

dentials: "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 1 is ShopA.

Now let us step through ShopA's reasoning process in example 1 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 is done 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'."

3.5 Discussion of Representation Statements

We now discuss the differences between a representation statement "Y `represents` X `on` ba" and a depth-$*$ delegation statement "X `delegates` ba`^`$*$ `to` Y."

The representation statement is strictly stronger than the delegation statement. If "Y `represents` X `on` ba," then Y has all the power that X has with respect to ba, even if X is not allowed to re-delegate this power. In other words, conclusions drawn from a representation statement don't consume any delegation depth.
For example, given the following rules:

> Alice delegates read(file1)^1 to Bob.
> keyBob represents 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.

One main reason for having representation 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. Authorities are often delegated to these names instead of to keys. The trust root must determine which keys "represent" these names and issue representation statements for this.

In example 1, we have the following credential:

ShopA delegates creditRating(?U, ?R)^2 to BankB.

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 keyBankB. Assume that ShopA trusts keyCA about the bindings of keys to names, and keyCA certifies that keyBankB is BankB's public key for business purposes, *i.e.*, ShopA has the following statements:

keyCA says isBusinessKey(keyBankB, BankB).
ShopA delegates isBusinessKey(?Key, ?X)^1 to keyCA.

Then, by adding the following statement, ShopA can derive the same conclusions as in example 1.

?Key represents ?X on creditRating(?U, ?R)
if Local says isBusinessKey(?Key, ?X).

Using a representation 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.

As one can see, a representation statement essentially circumvents the delegation-depth restriction. Therefore, a rule with a representation statement in its head always has the principal `Local`, *i.e.*, the trust root, as its issuer. A principal B can issue a statement saying that B delegates to C. But B cannot say that C represents B. The trust root may rely on information from other principals to determine whether one principal represents another principal; however, only the trust root can make the decision about what information to use in determining representation statements; others cannot make that decision for the trust root. Also note that the trust root has no incentive to cheat on depth, because it can choose to authorize a request without consulting the trust-management engine. In the above example, the trust root ShopA is implicitly putting a lot of trust in keyCA (similar to the PKI being used today), but it is using delegation depth to limit trust in BankB.

The difference between representation statements and delegation statements in DL is similar to the difference between name certificates and delegation statements in SPKI/SDSI. DL's notion of representation is also similar to the notion of "speaks for" in [2; 26]; however, there are two differences. First, DL's representation relation is defined on a per-base-atom basis. Principal B may represent principal A with respect to one thing but not another. In [2; 26], if B speaks for A, then B speaks for A with respect to everything. Second, in DL, a representation statement can only be made by the trust root, but in [2; 26], "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.

3.6 More Examples of D1LP

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

Example 2 (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" for the purpose of 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 but not by system X.

Example 3 (Accessing medical records).

This example concerns access to medical records. It is based on an example in [20]. 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 isPhysicianOf(?X, ?Y)^1 to ?Z if HM says isHospital(?Z).
HM delegates isHospital(?H)^1 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 isPhysicianOf(Alice, Peter).
HD says isPhysicianOf(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 concludes that it is. Because HA says that Alice is the physician of Peter, DL can further conclude that "HM says readMedRec(Alice, Peter))."

Example 4 (Controlling delegation width).

Suppose that Alice wants to delegate to Bob the right to access a document docZ (*e.g.*, a confidential contract draft or a copyrighted report), and to allow Bob to further delegate this right as long as the principals to which Bob delegates are members of the 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 access(docZ)^* to (Bob,tmpKey).
tmpKey delegates access(docZ)^1 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 of public/private keys, then signs the second statement with the new private key and uses the new public key in the first statement. After signing the second statement, Alice can throw the new secret key away, without having to worry 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

Bob delegates access(docZ)^2 to David.
Bob delegates access(docZ)^2 to John.
Carl says member(David,orgA).

Then the delegation "Alice delegates access^1 to David" is a conclusion, but "Alice delegates access^1 to John" is not.

4. SEMANTICS OF D1LP

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* (bottom-up) 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 4.6. This kind of query-answering avoids computing the entire minimal D1LP model; it is also called *goal-directed* (top-down) inferencing.

In specifying *Trans* and calculating the size of its output $\mathcal{P} = Trans(\mathcal{P})$, 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, say, depth 7 and depth 8, 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.

4.1 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.

The first step of the transformation is to replace each occurrence (if there are any) of `Local` in rules in $\mathcal{P}$ with the principal that is specified in the locale-declaration statement. Recall that as a syntactical requirement, if `Local` occurs in rules in $\mathcal{P}$, then there must be a locale-declaration statement in $\mathcal{P}$. After the replacement, the locale-declaration statement is removed. The resulting program is $\mathcal{P}_0$.

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 $\mathcal{P}_0$ 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 $\mathcal{P}_0$ plus a set of dummy principals, one for each principal structure that appears as a delegatee

in $\mathcal{P}_0$. The domain of *ba* is the set of all ground base atoms in $\mathcal{P}_0^{Inst}$ (ground instantiation of $\mathcal{P}_0$). The domain of *len* is $[1..*]$. Note that base atoms in $\mathcal{P}_0$ are used as terms here; for each predicate symbol in $\mathcal{P}_0$, we add to $\mathcal{LO}_\mathcal{P}$ a new function symbol that has the same name as that predicate symbol. The field *len* stores one plus the number of delegation steps this conclusion has gone through; an atom "holds(Alice, goodCredit(Carl), 1)" thus represents "Alice says goodCredit(Carl)." 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* is used to represent delegation statements and representation statements that are made in $\mathcal{P}_0$ or derived in the inference process. It 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 representation statements. Note that a *delegates* atom can only represent a delegation to a *single* principal. Queries of delegations to conjunctions of principals are handled by introducing dummy principals, as will be shown soon.

We first consider how to transform a D1LP that does not contain threshold structures. We define a function *PExpand*, which will be used in the transformation.

Function *PExpand***:**

This function takes two parameters: a principal expression (a principal, a principal variable, or a principal structure) and an atom (of predicate *holds* or *delegates*) without the *issuer* field. Consider the case in which the principal expression is free of threshold structures. In this case, the function *PExpand* is defined recursively as follows:

$PExpand((PE_1, PE_2), Atom) = (PExpand(PE_1, Atom), PExpand(PE_2, Atom))$
$PExpand((PE_1; PE_2), Atom) = (PExpand(PE_1, Atom); PExpand(PE_2, Atom))$
$PExpand(X, holds(ba, l)) = holds(X, ba, l)$
$PExpand(X, delegates(ba, dep, dele, l)) = delegates(X, ba, dep, dele, l)$
where X is either a principal variable or a single principal.

The function *PExpand* transforms a statement that has a 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, *PExpand* deals with delegatees that are principal structures and enables body statements to be more general than head statements.

For now, *PExpand* simply returns a formula. When we deal with threshold structures in Sections 4.3 and 4.4, we will extend the definition of *PExpand*; then, 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 $\mathcal{P}_0$; 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 the transformation does the following to the *body* of each rule in $\mathcal{P}_0$.

(1) **Holds body translation:**
Replace each body-direct statement AE `says` ba
with $PExpand(AE, holds(ba, *))$.
This step adds the length $*$ to body statements and uses *PExpand* to deal with complex issuers. Intuitively, a direct statement "AE says ba" in the body of a rule is true if we can prove that AE 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) **Representation body translation:**
Replace each representation statement B `represents` A `on` ba
with $delegates(A, ba, *, B, 0)$.
This means that representation statements are special delegations that always have depth $*$ and length 0.

(3) **Simple delegates body translation:**
Replace each body-delegation statement AE `delegates` $ba\hat{}d$ `to` B
with $PExpand(AE, delegates(ba, d, B, *))$,
where B is a principal or a principal variable.
This step is similar to step (1), but it is for delegation statements. It adds the length $*$ to body statements, because we do not require that a conclusion be drawn within a certain number of delegation steps.

(4) **Conjunction delegates body translation:**
Replace each body-delegation statement
AE `delegates` $ba\hat{}d$ `to` $(B_1, \ldots, B_n)$
with $PExpand(AE, delegates(ba, d, B_{new}, *))$,
where $B_1, \ldots, B_n$ 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 conjunctions 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 have introduced a dummy principal B_{new} to represent the principal structure $(B_1, \ldots, B_n)$. That B_{new} represents $(B_1, \ldots, B_n)$ is characterized by the relationships that B_{new} represents for every principal in $(B_1, \ldots, B_n)$. 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.

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 is derived with smaller length, then it may 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 representation statement, do the following steps. Note that the following steps may seem unnecessarily complicated, especially in the way they deal with length and delegation depth. This complication arises from the need to avoid introducing new variables in the transformation; this is essential in proving tractability results.

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

$$A \texttt{ delegates } ba\texttt{\^{}}d \texttt{ to } BE,$$

i.e., a depth-d delegation from A to BE, then let ll be 1 (ll will be used in the transformation steps below). Let B be BE if BE is a single principal or a principal variable; otherwise, let B be the newly introduced dummy principal that represents BE.

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

$$B \texttt{ represents } A \texttt{ on } ba,$$

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

For both sub-cases, do the following steps.

(7) **Delegates head translation:**
Replace R's head with $delegates(A, ba, d, B, ll)$.
We use B in place of BE, because the delegatee field of the predicate *delegates* is restricted to be a *single* principal or a principal variable.

(8) **Holds propagation meta-rule:**
For each $len \in [1..d]$, add the following rule:
$holds(A, ba, len \oplus ll)$
`if` $delegates(A, ba, d, B, ll)$, $PExpand(BE, holds(ba, len))$.
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 BE 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 representation statement.

(9) **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:
$delegates(A, ba, \min(d \ominus len, dep), C, len \oplus ll)$

$\texttt{if}\ \ delegates(A, ba, d, B, ll),$
$PExpand(BE, delegates(ba, dep, C, len)).$

where, for any $d1, d2$ such that $0 \leq d2 \leq d1 \leq D$:

$$\begin{aligned} d1 \ominus d2 &= d1 - d2 \\ * \ominus d1 &= * \\ * \ominus * &= * \end{aligned}$$

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 BE with depth d (see step 7 for the relation between B and BE), and one can also derive within $len \leq d$ steps that BE 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 BE 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 BE to C.

Steps 8 and 9 provide a key part of the relationship between dummy principals and the principal structures they represent.

(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 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:
$delegates(A, ba, dep, C, len)\ \ \texttt{if}\ \ delegates(A, ba, dep \oplus 1, C, len).$
This meta-rule states that a smaller-depth delegation may be derived if a corresponding larger-depth delegation is derived.

(12) **Holds length-weakening meta-rule:**
For each $len \in [1..D]$, 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.

(13) **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:
$delegates(A, ba, dep, C, len \oplus 1)\ \ \texttt{if}\ \ delegates(A, ba, dep, C, len).$
This meta-rule states that any delegation that is derived within a certain length may also be derived within a larger length.

Example 5 (Holds propagation).
r1: B_1 delegates p^2 to B_2.
r2: B_2 delegates p^1 to B_3.
r3: B_3 says p.
From r3, applying the holds head translation (5), one has "$holds(B_3, p, 1)$."

From r2, applying the delegates head translation (7), one has "$delegates(B_2, p, 1, B_3, 1)$."

From r2, applying the holds propagation meta-rule (8) with $len = 1$, one has "$holds(B_2, p, 2)$ `if` $delegates(B_2, p, 1, B_3, 1)$, $holds(B_3, p, 1)$."

From the above rules, one concludes "$holds(B_2, p, 2)$."

From r1, applying (7), one has "$delegates(B_1, p, 2, B_2, 1)$." Applying (8) with $len = 2$, one has "$holds(B_1, p, 3)$ `if` $delegates(B_1, p, 2, B_2, 1)$, $holds(B_2, p, 2)$."

From the above rules, one concludes "$holds(B_1, p, 3)$."

Example 6 (Delegation chaining).

r1: B_1 delegates p^3 to B_2.

r2: B_2 delegates p^$*$ to B_3.

r3: B_3 delegates p^4 to B_4.

From r3, applying (7), one has "$delegates(B_3, p, 4, B_4, 1)$."

From r2, applying (7), one has "$delegates(B_2, p, *, B_3, 1)$."

From r2, applying delegation-chaining meta-rule (9) with $C = B_4$, $dep = 4$, and $len = 1$, one has "$delegates(B_2, p, 4, B_4, 2)$ `if` $delegates(B_2, p, *, B_3, 1)$, $delegates(B_3, p, 4, B_4, 1)$."

From the above rules, one concludes "$delegates(B_2, p, 4, B_4, 2)$."

From r1, applying (7), one has "$delegates(B_1, p, 3, B_2, 1)$." Applying (9) with $C = B_4$, $dep = 4$, and $len = 2$, one has "$delegates(B_1, p, 1, B_4, 3)$ `if` $delegates(B_1, p, 3, B_2, 1)$, $delegates(B_2, p, 4, B_4, 2)$."

From the above rules, one concludes "$delegates(B_1, p, 1, B_4, 3)$." The delegation depth of this conclusion is 1, because B_1 delegates to B_2 with depth 3 and two steps are consumed while deriving the delegation from B_2 to B_4.

Example 7 (Self Delegation).

r1: A delegates p^3 to (B_1,B_2).

r2: B_1 delegates p^3 to B_2.

From r2, applying (7), one has "$delegates(B_1, p, 3, B_2, 1)$."

From the self delegation meta-rule (10), with $C = B_2$, $dep = 3$, $len = 1$, one has "$delegates(B_2, p, 3, B_2, 1)$."

For r1, a dummy principal T is introduced to represent (B_1, B_2). Applying (7), one has $delegates(A, p, 3, T, 1)$.

From r1, applying (9) with $C = B_2$, $dep = 3$, $len = 1$, one has "$delegates(A, p, 2, B_2, 2)$ `if` $delegates(A, p, 3, T, 1)$, $delegates(B_1, p, 3, B_2, 1)$, $delegates(B_2, p, 3, B_2, 1)$."

From the above, one concludes that "$delegates(A, p, 2, B_2, 2)$."

Example 8 (Conjunctive delegation in body).

r1: A delegates p^1 to (B_1, B_2).

r2: A says qq if A delegates p^1 to (B_1, B_2, B_3).

Given rules r1 and r2, one should conclude $holds(A, qq, 1)$.

Two dummy principals are introduced: T_1 for (B_1, B_2) and T_2 for (B_1, B_2, B_3).

From r2, applying the conjunctive delegates body translation (4) and the head translation (5), one has "$holds(A, qq, 1)$ if $delegates(A, p, 1, T_2, *)$" as well as "$delegates(B_1, p, *, T_2, 0)$," "$delegates(B_2, p, *, T_2, 0)$," and "$delegates(B_3, p, *, T_2, 0)$."

From r1, applying (7), one has "$delegates(A, p, 1, T_1, 1)$"; applying (9) with $C = T_2$, $dep = 1$, and $len = 0$, one has "$delegates(A, p, 1, T_2, 1)$ if $delegates(A, p, 1, T_1, 1)$, $delegates(B_1, 1, p, T_2, 0)$, $delegates(B_2, 1, p, T_2, 0)$."

From the rules above, one can then infer "$delegates(A, p, 1, T_2, 1)$."

From r1, applying the delegates length-weakening meta-rule (13), one has "$delegates(A, p, 1, T_2, *)$ if $delegates(A, p, 1, T_2, 1)$." Therefore, one has "$delegates(A, p, 1, T_2, *)$."

Therefore, one concludes that "$holds(A, qq, 1)$."

The result of the transformation *Trans* is:

$$\mathcal{O} = Trans(\mathcal{P}) = \mathcal{P}_2 \cup \mathcal{P}_1^{add} \cup \mathcal{P}_2^{add}.$$

The transformation process introduces dummy principals to replace principal structures that occur as delegatees. The transformation process also outputs the list $\mathcal{DUM}$ of all dummy principals and which principal expressions they replace. This list is useful when converting conclusions drawn from $\mathcal{O}$ back to D1LP.

Lemma 1. *Given a D1LP $\mathcal{P}$ that does not use any threshold structures, let N be the size of $\mathcal{P}$, D be the maximal integer depth in $\mathcal{P}$, and $\mathcal{O} = Trans(\mathcal{P})$, then $|\mathcal{O}| = O(N^3 D^2)$.*

Proof. Our counting argument focuses on the ratio $|\mathcal{O}|/|\mathcal{P}|$, which we call the *growth factor*. We show that the growth factor is $O(N^2 D^2)$.

Note that $|PExpand(BE, Atom)|/|Atom| = O(|BE|)$. Clearly, $|BE| < N$. Therefore, the growth factor of *PExpand* is $O(N)$.

In the body-transformation phase, a body statement is replaced by the result of a corresponding *PExpand* 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 does not 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 representation statement as its head, several transformation steps apply; each adds a set of rules to $\mathcal{P}_2^{add}$, but the size of $\mathcal{P}_2$ remains unchanged from $\mathcal{P}_1$. Step 9 (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 *PExpand* to change parts of it. Therefore, the growth factor for step 9 is $O(ND^2)$ times the growth factor of *PExpand*.

Because $\mathcal{O} = \mathcal{P}_2 \cup \mathcal{P}_1^{add} \cup \mathcal{P}_2^{add}$, $|\mathcal{O}|/|\mathcal{P}| = O(N^2 D^2)$, and so this claim holds. ∎

Of this $N^2 D^2$ 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 $|\mathcal{P}|$.

4.2 Generating Typed OLP

To ensure tractability, we would like $Trans(\mathcal{P})$ to be Datalog when $\mathcal{P}$ is. However, $Trans(\mathcal{P})$ introduces logical function symbols that have non-zero arities. For each predicate *pred* in $\mathcal{P}$, $Trans(\mathcal{P})$ has one corresponding function symbol. As we will see in Sections 4.3 and 4.4, $Trans(\mathcal{P})$ also introduces several pre-defined function symbols for threshold structures.

This problem is 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 [39].) 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 *func* 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 [35]). Many-sorted logic programs can be executed with the same efficiency as untyped logic programs [38].

We 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. This fact will be crucial to the tractability result of D1LP inferencing in Section 5.2.

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 [37]. We observe that, in authorization, there are conceptually different types of entities, *e.g.*, subjects, objects, groups, roles, *etc.* Most predicates conceptually should only take arguments of certain types. One might thus add typing directly and explicitly to DL and its syntax. Actually, a degree of typing is implicitly present in DL. DL has principals and principal variables; thus, there is an implicit "principal" type. Typing might also be used to relax the Datalog restriction on DL's syntax. 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 potentially very useful, we do not pursue this topic further in this paper, because it is not our main topic.

4.3 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 *PExpand* to include the following:

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

The function *PExpand*, 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: *PExpand* is called with a *holds* atom.
"$PExpand(threshold(k, [A_1, \ldots, A_n]), holds(ba, l))$" does the following.

—For $i = k$ to 1, for $j = 1$ to n, add the rule:
$holds(suth(i, [A_j, A_{j+1}, \ldots, A_n]), ba, l)$
`if` $holds(A_j, ba, l)$, $holds(suth(i-1, [A_{j+1}, \ldots, A_n]), ba, l)$.
When $j = n$, the added rule contains $[A_{n+1}, \ldots, A_n]$, which we define 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 principals in $[A_j, \ldots, A_n]$ that support ba.

Case two: *PExpand* is called with a *delegates* atom.
"$PExpand(threshold(k, [A_1, \ldots, A_n]), delegates(ba, dep, dele, l))$" does the following. This case is similar to case one, but with *delegates* taking the place of *holds*.

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

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

Next we consider how the size of $\mathcal{O}$ is affected. A call of *PExpand* with a static unweighted threshold structure returns an atom of the same size as its input and generates a number of new rules. We define the growth factor of *PExpand* with threshold structures to be the total size of all the generated rules plus its output, divided by the size of its input.

Lemma 2. *The growth factor of PExpand with a static unweighted threshold is* $O(N^2)$.

Proof. Each time *PExpand* encounters a static unweighted threshold structure, $O(k, n)$ new rules are generated, where $k \leq n$ is the threshold value, and n is the size of the threshold pool. Each new rule has size linear in the size of *PExpand*'s input. The worst-case bound for $O(kn)$ is $O(N^2)$.

Static *weighted* threshold structures are handled similarly to static *unweighted* threshold structures; 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 *PExpand* to include the following:

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

Case one: "$PExpand(threshold(k, [(A_1, w_1), \ldots, (A_n, w_n)]), holds(ba, l))$" 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: *PExpand* is called with a static weighted threshold structure and a *delegates* atom. This case is similar to case one, but with *delegates* taking the place of *holds*. The details are omitted.

Lemma 3. *The growth factor of PExpand with a static weighted threshold is* $O(N^2)$.

Proof. Similar to Lemma 2.

4.4 Transformation with Dynamic Threshold Structures

To handle dynamic threshold structures, we need a listing of all principals in $\mathcal{P}_0$. Let M be the number of different principals in $\mathcal{P}_0$ and $[C_1, C_2, \ldots, C_M]$ be a list of all principals in $\mathcal{P}_0$.

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 *PExpand* to include the following:

$$\begin{aligned}
&PExpand(threshold(k, ?X, Prin \text{ says } pred(\ldots ?X \ldots)), holds(ba, l)) \\
&\quad = holds(duth(k, 1, t), ba, l) \\
&PExpand(threshold(k, ?X, Prin \text{ says } pred(\ldots ?X \ldots)), \\
&\qquad\qquad delegates(ba, dep, dele, l)) \\
&\quad = delegates(duth(k, 1, t), ba, dep, dele, l)
\end{aligned}$$

Each time *PExpand* 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: "$PExpand(threshold(k, ?X, Prin \text{ says } pred(\ldots ?X \ldots)), holds(ba, l))$" does the following. If k is greater than M, do nothing — the threshold cannot be satisfied. Otherwise, do the following.

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

$$\begin{aligned}
&holds(duth(i, j, t), ba, l) \\
&\quad \texttt{if}\ \ holds(Prin, pred(\ldots C_j \ldots), *), \\
&\qquad holds(C_j, ba, l), \\
&\qquad holds(duth(i - 1, j + 1, t), *)), ba, l).
\end{aligned}$$

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

$$holds(duth(i, j, t), ba, l) \ \ \texttt{if}\ \ holds(duth(i, j + 1, t), ba, l).$$

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

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

Case two: *PExpand* is called with a dynamic unweighted threshold structure and a *delegates* atom. This case is similar to case one, but with *delegates* taking the place of *holds*. The details are omitted.

Lemma 4. *The growth factor of PExpand with a dynamic unweighted threshold is* $O(N^2)$.

Proof. 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 $\mathcal{P}_0$, and so $M = O(N)$. Thus, the worst-case growth factor of *PExpand* 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 4.3, is the size of one static threshold pool.)

Dynamic weighted threshold structures

We also considered incorporating *dynamic weighted threshold structures* [31] into DL; however, they pose difficulties for ensuring tractability.

4.5 Reverse Transformation of Conclusions

In Section 4.1, we defined the transformation from a D1LP $\mathcal{P}$ to an OLP $\mathcal{O}$. We now define a 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 $\mathcal{P}$. In the transformation from D1LP to OLP, a dummy principal is introduced when a principal expression other than a principal or a principal variable occurs as the delegatee of a delegation statement. In the reverse transformation, these principals are replaced by the principal structures to which they correspond, when they appear as delegatees of conclusions of *delegates*. The mapping $\mathcal{DUM}$ of dummy principals to principal structures, produced by the transformation, is needed here. The reverse transformation is as follows.

—For each atom of the form: $holds(A, ba, len)$,
where A is a non-dummy principal, include the D1LP-conclusion:
A `says` ba.

—For each atom of the form: $delegates(A, ba, *, D, 0)$,
where A and D are non-dummy principals, include the D1LP-conclusion:
D `represents` A `on` ba.

—For each atom of the form:
$delegates(A, ba, dep, D, len)$,
where A is a non-dummy principal, D is a principal, and $len > 0$.
When D is a non-dummy principal, include the D1LP-conclusion:
A `delegates` ba`^`dep `to` D.
Otherwise, when D is a dummy principal introduced for the principal structure BE, include the D1LP-conclusion:
A `delegates` ba`^`dep `to` BE.
(Note that, because of the way the semantic transformation is defined, there are no atoms with both $len = 0$ and $dep < *$.)

Note that we ignore conclusions that have dummy principals or threshold structures as issuers and do not convert them back to D1LP, because these are intermediate results. Also 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 $\mathcal{O}$.

4.6 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 the truth value of $\mathcal{Q}$. 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 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 4.1). 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 $\mathcal{P}$.

5. 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*. In Section 5.1, we show that *Trans* is tractable. In Section 5.2, we show that, under the restriction that each rule has a bounded number of variables, overall D1LP inferencing is also tractable. This restriction is similar to that for guaranteeing tractability of Datalog inferencing and relational databases.

5.1 Tractability of the Transformation from D1LP to OLP

From Lemmas 1, 2, 3, and 4 in Sections 4.1, 4.3, and 4.4, it follows straightforwardly that the growth factor of the transformation *Trans* is $O(N^3D^2)$, where $N = |\mathcal{P}|$, and D is the maximal delegation depth in $\mathcal{P}$. Therefore, we have the following result.

Theorem 5 (Tractable Transform Size).
Given a D1LP $\mathcal{P}$, the size of $\mathcal{O} = Trans(\mathcal{P})$ is $O(N^4D^2)$, 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 5, we have the following theorem.

Theorem 6 (Tractable Transform Time).
Computing $\mathcal{O}$ takes time $O(N^4D^2)$. The transformation from D1LP to OLP is thus computationally tractable.

As discussed before, 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 $O(N^4D^2)$. 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 representation statement as its head. A rule that does not have any threshold structure is simpler than a rule that does.

Consider a rule R that does not contain any threshold structures (either in the head or in the body): Let S_R be the size of the largest principal structure in R;

certainly S_R is bounded by the size of R, which is in turn bounded by 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 representation statement, R's growth factor is "$|Principals|S_R D^2$." 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 does not 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 following 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 $Principals$.

5.2 Tractability of D1LP Inferencing

Next, we review some previously known results about OLP inferencing [35]. 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. Datalog means without function symbols of non-zero arity. Function symbols with non-zero arity makes it possible to generate an infinite number of ground terms; therefore, the minimal model of a non-Datalog program may be infinite. For example, given one arity-1 function symbol f and one constant a, one can construct terms such as $f(a), f(f(a)), f(f(f(a)))$, *etc.*

Given a definite OLP $\mathcal{S}$ that is VBD(v), its inferencing (computing its minimal model or answering a query relative to it) takes time $O(|\mathcal{S}|^{v+1})$. This is because inferencing of a definite OLP takes time linear in the size of its ground instantiation, and $\mathcal{S}$'s ground instantiation has size $O(|\mathcal{S}|^{v+1})$. For each variable, there are $O(|\mathcal{S}|)$

ground terms that can be used to instantiate it, and so, for each rule, there are at most $O(|\mathcal{S}|^v)$ ways to instantiate it. Thus, instantiating $\mathcal{S}$ increases its size by a factor of $O(|\mathcal{S}|^v)$.

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

Theorem 7 (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 $\mathcal{O} = \mathit{Trans}(\mathcal{P})$ is also VB(v), because *Trans* does not introduce any new variables. Recall our discussion above of typing in $\mathcal{O}$. 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. By Theorem 5, 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. By Theorem 6, the transformation takes time $O(N^4D^2)$. So, overall, computing the minimal D1LP model of $\mathcal{P}$ has time complexity $O(N^{v+4}D^2)$. ∎

6. DISCUSSION OF THE CONJUNCTIVE-DELEGATEE-QUERIES RESTRICTION

In Section 3.1, we defined D1LP to obey the conjunctive-delegatee-queries restriction: A delegation statement appearing in a query or a rule-body 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). In this section, we discuss in detail the tractability motivation behind the conjunctive-delegatee-queries restriction.

6.1 Understanding D1LP's Inferencing of Delegation

D1LP allows delegation statements to appear in queries. 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, "A delegates p^1 to B" and "A delegates q^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, "A delegates to (B;C)" is logically equivalent to the conjunction of "A delegates to B" and "A delegates to C." By contrast, "A delegates to (B,C)" is weaker than

either "A delegates to B" or "A delegates to C," because either of the last two delegations implies the first, but not *vice versa*.

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 PT_1 and PT_2, "A delegates to PT_1" is different from "A delegates to PT_2"; furthermore, they are incomparable when neither $PT_1 \subseteq PT_2$ nor $PT_2 \subseteq PT_1$.

As one can see, given N principals, the number of different principal sets among these principals is exponential in N. In fact, one delegation fact "A delegates to $((B_{11}; \ldots; B_{1n}), (B_{21}; \ldots; B_{2n}), \ldots, (B_{m1}; \ldots; B_{mn}))$" would imply n^m delegations "A delegates to $(B_{1i_1}, B_{2i_2}, \ldots, B_{mi_m})$," where $1 \leq i_j \leq n$ for $j \in [1..m]$. Therefore, a semantics that generates all delegations to conjunctions of principals could be exponential in size of the program.

In order to keep D1LP's semantics tractable, we impose the limitation that D1LP only generates delegation to a *single* principal as conclusions. As a result, D1LP's semantics can only directly answer those delegation queries that have a single principal as delegatee. A delegation query with a delegatee that is a principal conjunction 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. D1LP's semantics only generates as a conclusion a delegation to a conjunction of principals when this delegation is asked, *i.e.*, included in a query or a rule-body. This is a form of lazy evaluation.

Conjunctive delegation queries are useful in the following scenario: 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.

Example 9 (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 represents C1 on p," "T1 represents C2 on p," ..., and "T1 represents 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 represents C1 on p" and "T1 represents C2 on p," one concludes that "B1 delegates p^1 to T1 on p." Similarly, from rule r3 and the two facts "T1 represents C3 on p" and "T1 represents C4 on p," one concludes that "B2 delegates p^1 to T1." Then, from these two new conclusions and rule r1, one concludes that "A delegates p^1 to T1." (Note that all depth constraints are satisfied.)

6.2 Handling Delegation Queries with Disjunctive Delegatees

Having explained how D1LP deals with conjunctive delegation queries, we now show how to extend that approach to answer delegation queries that contain disjunctions

or static threshold structures, if they are allowed. This approach has exponential computational complexity.

Let us 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 does not. Given two principal structures PS_1 and PS_2, "A delegates to PS_1" is stronger than "A delegates to PS_2" if and only if $PS_2 \Rightarrow PS_1$ is a tautology. The reason is as follows. Given that "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, "A delegates to PS_2" should also be true.

We give an approach for answering delegation queries that is based on transforming principal structures into "normal forms" (essentially disjunctive normal forms of formulas in propositional logic) and then replace a disjunctive delegation query with an equivalent query that is a conjunction of delegation statements. A principal structure PS is in normal form when it is of the form: $(PT_1; PT_2; ...; PT_r)$, where each PT_i is a principal set and, for any $i \neq j$, $PT_i \not\subseteq PT_j$. If one views PS as a propositional formula; then PS's normal form $PT_1; PT_2; ...; PT_r$ 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 PT_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 "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 "threshold(3, [(A,2), (B,1), (C,1), (D,1)]" is "((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: $(PT_1; \ldots; PT_r)$, a delegation "PE delegates to PS" in a query or a rule-body is transformed into "(PE delegates to PT_1, PE delegates to PT_2, ..., PE delegates to PT_r)." Now consider an example that has a delegation to a threshold structure in a rule-body.

Example 10 (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 $\Theta(mn^m)$. This motivates the conjunctive-delegatee-query restriction.

6.3 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 go from true to false when more information is known. Consider the following example.

Example 11 (Nonmonotonicity of Delegation to dynamic threshold).

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

7. DISCUSSION

7.1 Implementation

We have implemented D1LP in XSB [18], a Prolog-variant logic-programming system developed by Warren *et al.* at SUNY Stony Brook. This implementation is called XD1LP and is available [28]. 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 [10], 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.

XD1LP implements a slightly less recent version of D1LP, which omits locale-declaration statements.

XD1LP uses an alternative transformation that is different from (but similar to) the one we gave in Section 4. 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."

7.2 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, how can different entities agree on meanings of predicates? Which data structures and com-

munication protocols should one use for exchanging DL rules between distributed applications/principals/Internet-sites?

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 ways in which such credentials could 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.

7.3 Extending D1LP with Nonmonotonic Features

In Section 1, we mentioned that DL also extends Definite Ordinary Logic Programs with nonmonotonic reasoning. The version of DL with nonmonotonic features is called D2LP and is described in [29]. D2LP has negation-as-failure (a.k.a. default negation), classical negation, and prioritized conflict handling. As discussed in [29], integrating delegation and nonmonotonicity results in some difficulties. Because of these difficulties, D2LP as defined in [29] prohibits delegation statements from appearing in queries or rule-bodies. Also note that, when using nonmonotonic policies, one often needs *complete* information about certain things in order to derive correct conclusions, and complete information is inherently hard to obtain in distributed environments. These difficulties need to be resolved before one can use nonmonotonic policies in trust management.

7.4 Subsequent Related Work

Several related works have appeared since the first paper on Delegation Logic appeared in 1999. SD3 (Secure Dynamically Distributed Datalog) [24] and Binder [14] both use the approach of extending Datalog with explicit issuers. The RT (Role-based Trust-management) framework [34; 33] is also based on Datalog. RT addresses name agreement and distributed storage and discovery of credentials.

8. CONCLUSIONS

In this paper, we presented D1LP, the monotonic version of a logic-programming-based trust-management language Delegation Logic (DL) for representing security policies and credentials for authorization in large-scale, open, distributed systems. We gave a list of sample policies for testing expressive powers of TM systems and showed that previous TM systems lack important expressive features. (Exceptions are those systems, *e.g.*, PolicyMaker, that allow programs in general-purpose programming languages to be used in credentials and policies, and thus can express almost any policy. However, declarativeness and usability suffer as a result.) We

also showed that D1LP can express these policies.

Our general approach to designing DL was to extend existing well-understood logic-programming (LP) languages with features that are needed in distributed authorization. D1LP extends Definite Ordinary LP (a.k.a. Definite LP, see [35]) with issuers and delegation constructs. Our approach to defining the semantics of DL was to define a transformation from DL programs into programs in the underlying logic-programming language. The transformation-based approach gives us easy access to the established results for OLP; for example, every DL program has a minimal model. Therefore, D1LP has a clearly specified notion of proof of compliance that is based on model-theoretic semantics and is thus abstracted away from choice and details of implementation. We showed that each of these transformation steps is computationally tractable and that D1LP inferencing is thus tractable under a broad restriction similar to that which ensures tractability of OLP inferencing. 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.

In summary, the main contribution of this paper is the TM language D1LP, which is expressive, declarative, tractable, and practically implementable. We also studied how to support delegations with integer depths and unlimited depth, delegations to complex principal structures, and inferencing of delegations. Although the practical importance of some of these features, *e.g.*, integer depth, is still debatable, we nevertheless believe that clarifying how these features work and interact is a contribution.

Acknowledgement

The first author is currently supported by DARPA through SPAWAR contracts N66001-00-C-8015 and by DoD MURI "Semantics Consistency in Information Exchange" as ONR Grant N00014-97-1-0505. We thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] Martín Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1–2):3–21, 1998.

[2] 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):706–734, October 1993.

[3] Tuomas Aura. On the structure of delegation networks. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 14–26. IEEE Computer Society Press, June 1998.

[4] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19/20:73–148, May/July 1994.

[5] 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)*, pages 175–189. IEEE Computer Society Press, 1999.

[6] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote trust-management system, version 2. IETF RFC 2704, September 1999.

[7] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer, 1999.

[8] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.

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

[10] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[11] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. REFEREE: Trust management for web applications. *World Wide Web Journal*, 2:706–734, 1997.

[12] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

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

[14] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 105–113. IEEE Computer Society Press, May 2002.

[15] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI certificate theory. IETF RFC 2693, September 1999.

[16] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. Simple public key certificates. Internet Draft (work in progress), July 1999. http://world.std.com/~cme/spki.txt.

[17] Hal Finney. Transitive trust and MLM. Post to cypherpunks mailing list, archived at http://www.inet-one.com/cypherpunks/dir.1996.05.02-1996.05.08/msg00415.html, May 1996.

[18] The XSB Research Group. The XSB programming system. http://xsb.sourceforge.net/.

[19] Joseph Halpern and Ron van der Meyden. A logic for SDSI's linked local named spaces. *Journal of Computer Security*, 9(1,2):47–74, 2001.

[20] 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*, pages 2–14. IEEE Computer Society Press, May 2000.

[21] Jonathan R. Howell. *Naming and sharing resources acroos administrative boundaries*. PhD thesis, Dartmouth College, May 2000.

[22] 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*, pages 31–42. IEEE Computer Society Press, 1997.

[23] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and Elisa Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 474–485, 1997.

[24] Trevor Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115. IEEE Computer Society Press, May 2001.

[25] Stephen T. Kent. Internet privacy enhanced mail. *Communications of the ACM*, 36(8):48–60, August 1993.

[26] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[27] Marc Langheinrich. A P3P Preference Exchange Language 1.0 (APPEL1.0). W3C Working Draft, April 2002.

[28] Ninghui Li. XD1LP: An implementation of D1LP in XSB. http://cs.nyu.edu/ninghui/xd1lp/.

[29] Ninghui Li. *Delegation Logic: A Logic-based Approach to Distributed Authorization*. PhD thesis, New York University, September 2000.

[30] Ninghui Li. Local names in SPKI/SDSI. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 2–15. IEEE Computer Society Press, July 2000.

[31] Ninghui Li, Joan Feigenbaum, and Benjamin N. Grosof. A logic-based knowledge representation for authorization with delegation (extended abstract). In *Proceedings of the 1999 IEEE Computer Security Foundations Workshop*, pages 162–174. IEEE Computer Society Press, June 1999.

[32] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. A practically implementable and tractable Delegation Logic. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 27–42. IEEE Computer Society Press, May 2000.

[33] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002.

[34] Ninghui Li, William H. Winsborough, and John C. Mitchell. Distributed credential chain discovery in trust management. To appear in *Journal of Computer Security*. Extended abstract appeared in *Proceedings of the Eighth ACM Conference on Computer and Communications Security (CCS-8)*.

[35] John W. Lloyd. *Foundations of Logic Programming, Second Edition*. Springer, 1987.

[36] Ueli Maurer. Modelling a public-key infrastructure. In *Proceedings of the 1996 European Symposium on Research in Computer Security*, volume 1146 of *Lecture Notes in Computer Science*, pages 325–350. Springer, 1997.

[37] Lee Naish. Types and the intended meaning of logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 189–216. The MIT Press, 1992.

[38] Peter Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS monographs on Theoretical Computer Science*. Springer, 1988.

[39] Frank Pfenning, editor. *Types in Logic Programming*. Logic Programming Series. The MIT Press, 1992.

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

[41] Ronald L. Rivest and Bulter Lampson. SDSI — a simple distributed security infrastructure, October 1996. http://theory.lcs.mit.edu/~rivest/sdsi11.html.

[42] Stephen Weeks. Understanding trust management systems. In *Proceedings of 2001 IEEE Symposium on Security and Privacy*, pages 94–105. IEEE Computer Society Press, May 2001.