A Practically Implementable and Tractable Delegation Logic*

Ninghui Li
Computer Science Department
New York University
251 Mercer Street
New York, NY 10012, USA
ninghui@cs.nyu.edu

Benjamin Grosof
IBM T.J. Watson Research Center
P.O.Box 704, Yorktown Heights, NY 10598, USA
grosof@us.ibm.com
(alt.: grosof@cs.stanford.edu)
http://www.research.ibm.com/people/g/grosof

Joan Feigenbaum
AT&T Labs – Research
Room C203
180 Park Avenue
Florham Park, NJ 07932, USA
jf@research.att.com

Abstract

We address the goal of making Delegation Logic (DL) into a practically implementable and tractable trust-management system. DL [22] is a logic-based knowledge representation (i.e., language) for authorization *in large-scale, open, distributed systems.*

As introduced in [22], DL inferencing is computationally intractable and highly impractical to implement. We introduce a new version of Delegation Logic that remedies these difficulties. To achieve this, we impose a syntactic restriction and redefine the semantics somewhat. We show that, for this revised version of DL, inferencing is computationally tractable under the same commonly met restrictions for which Ordinary Logic Programs (OLP) inferencing is tractable (e.g., Datalog and bounded number of logical variables per rule). We give an implementation architecture for this version of DL; it uses a delegation compiler *from DL to OLP and can modularly exploit a variety of existing OLP inference engines. As proof of concept, we have implemented a large expressive subset of this version of DL, using this architecture.*

1 Introduction

In today's Internet, there are a large and growing number of scenarios that require authorization decisions. By an *authorization* decision, we mean one in which one party submits a *request*, possibly supported by one or more *credentials*, that must comply with another party's *policies* if it is to be granted. Scenarios that require authorization decisions include electronic commerce, health care [2, 7], content advising [28], mobile-code execution [12], public-key infrastructure [9, 30, 19, 11, 27], and privacy protection [24, 23].

Authorization in Internet services is significantly different from authorization in centralized systems or even in distributed systems that are closed or relatively small. In these older settings, authorization of a request is divided into *authentication* ("who made the request?") and *access control* ("is the requester authorized to perform the action?"). The goal of a growing body of work on *trust management* [6, 8, 11, 10, 5, 22] is to find a more expressive and "distributed" approach to authorization. The trust-management approach supports credentials that endow public keys with more than just identities or "distinguished names" of key holders, *e.g.*, with agreed-upon "authorizations" [11], with various attributes of key-holders, or with fully programmable "capabilities" [6, 8, 10, 22]. The "authorization" problem consists of deciding whether these credentials prove that a request complies with a policy. Identity information is just one kind of credential, and it may be necessary and sufficient for some applications but not for others.

*This paper appears in Proceedings of the IEEE 2000 Symposium on Security and Privacy. The expanded Research Report version of this paper gives additional details, including sample output of the implementation.

Trust management adopts a "peer model" of authorization. Every entity can be an authorizer, a third-party credential issuer, or a requester. If an entity controls some resources that others want to use, it is the authorizer for these uses. It is the ultimate source of authority over usage of these resources, and it maintains local policies that determine what uses are to be allowed. An entity can also serve as a third-party credential issuer; if an authorizer defers trust to it in a policy, then these credentials can play a role in authorization decisions. When an entity wants to use resources that other entities control, it plays the role of a requester.

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

In [22], we proposed the logic-based language DL (Delegation Logic) as a trust-management engine. DL differs from other proposed trust-management engines [6, 8, 11, 5] in providing a notion of "credentials proving that a request complies with a policy" that is not entirely *ad hoc*; rather, it is based on model-theoretic semantics (thus abstracted away from choice and details of implementation), and is smoothly extensible expressively to non-monotonicity, negation, and prioritized conflict handling. Specifically, DL starts with the notion of proof embodied in Datalog definite ordinary logic programs [20]. (For a review of standard concepts and results in logic programming, see, *e.g.*, [4].) "Ordinary" logic programs (LP's) correspond essentially to pure Prolog, but without the limitation to Prolog's particular inferencing procedure. (They are also sometimes known as "normal" LP's or "general" LP's.) "Definite" means without negation. "Datalog" means without (logical) function symbols of more than zero arity.[1]

Other notable features of DL include explicit linguistic support for delegation depth and for a wide variety of complex principals (*e.g.*, k-out-of-n thresholds). The design of DL was influenced by earlier work on trust management [6, 8, 10, 5], public-key infrastructure proposals [11, 25, 28, 30], logic-based approaches to authentication and access control in distributed systems [1, 21], and logic programming and knowledge representation. For a more thorough, but still high-level, introduction to DL and its relationship to previous work, see sections 1, 2, and 6 of [22].

In [22], we defined the syntax and semantics of a monotonic version of DL: D1LP (Delegation Logic Programs version 1). However, as introduced in [22], D1LP's semantics is defined by a transformation into Ordinary LP (OLP) that is computationally intractable: exponential in output size as well as in time. Moreover, the semantic construction there requires many iterations that each interleaves such a transformation phase with another phase of complete OLP inferencing. We observe that this makes the D1LP inferencing in [22] computationally intractable and highly impractical to implement.

In this paper, we address the goal of making Delegation Logic into a practically implementable and tractable trust-management engine. We introduce a new version of D1LP that remedies the above difficulties, preserves the previous version's attractive features, and has some extra advantages. To achieve this, we impose a syntactic restriction and redefine the semantics somewhat. Even with the restriction, the formalism has significant expressive power. We show that, for this new version of D1LP, the transformation into OLP is computationally tractable (worst-case polynomial-time) and only needs to be done once. We show that, for this new version of D1LP, inferencing is thus computationally tractable under the same commonly-met restrictions for which OLP inferencing is tractable (*e.g.*, Datalog and bounded number of logical variables per rule). We give an implementation architecture for this version that uses a *delegation compiler* from DL to OLP and can modularly exploit a variety of existing OLP inference engines. As proof of concept, we have implemented a large expressive subset of this new version of DL, using this architecture; we describe that implementation briefly.

Before going into the details of the new version of D1LP, we first briefly explain how DL can be used in authorization systems.

Entities in authorization scenarios are represented by principals in DL. They can issue policies, credentials, and requests. Policies and credentials are represented by rules in DL. Requests are represented by queries. Policies differ from credentials in that they are used by their authors.

When an authorizer gets a request that corresponds to a DL query Q along with some credentials that support this request, the authorizer creates a DL program (rule-set) P that contains all the credentials and the authorizer's local policies. DL's semantics defines a unique minimal model for P. There is a proof procedure to answer Q relative to truth in P's minimal model. Note that this reasoning is done from the authorizer's point of view. The authorizer is the trust root. In DL, the symbol "`Local`" refers to the current trust root.

In this paper, we define the syntax and semantics of the revised version of D1LP. We also give an inferencing procedure for (revised) D1LP and prove that such inferencing is tractable.

The rest of the paper is organized as follows. In section 2, we give the syntax of the revised version of D1LP. In

[1] "Arity" means number of parameters.

section 3, we give some examples of policies and inferences in D1LP. In section 4, we define the semantics for it by a transformation from a D1LP program into an OLP program. D1LP inferencing is accomplished by the combination of transformation plus OLP inferencing. Tractability results are given in section 5. In section 6, we describe our implementation and its architecture that combines a compiler implementing the transformation with a pluggable OLP inferencing engine. In section 7, we discuss the tractability motivation behind the main expressive restriction in this new version of D1LP. In section 8, we give discussion including current and future work. We conclude with a summary in section 9.

2 Syntax of the revised D1LP

In this section, we give the syntax of the revised D1LP. It differs from the previous version in [22] in several regards, but overall is quite similar.

The main restriction imposed in this new version is the following: A delegatee appearing in a rule body, or in a query, must be a principal, a conjunction of principals, or a principal variable. That is, such a delegatee is not permitted to contain a disjunction, nor to contain a threshold structure (which is implicitly disjunctive in nature). We call this the "*conjunctive-delegatee-queries*" expressive restriction. This restriction is imposed to ensure tractability, as is discussed in section 7.

The revised D1LP includes several expressive generalizations from the previous version as well. We found these expressive generalizations useful in the course of designing the compiler and developing examples. And we believe these generalizations make the revised version of D1LP more useful in practice. They are discussed as we introduce them in the overall syntax which we give next.

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

   Note that we prohibit function symbols with non-zero arity: This is the *Datalog* restriction. This restriction helps enable finiteness of the semantics and of computing inferences (a.k.a. entailments).

2. A *base atom* takes the form:
   $pred(t_1, ..., t_n)$
   where $pred$ is a predicate and each t_i is a term.

   A base atom encodes a trust belief, a security action, *etc.* For example, "isBusinessKey (keyBob, Bob)," "purchase(?M, ?Price)," and "goodCredit(?X)" are all base atoms.

3. A *direct statement* takes the form:
   X `says` ba
   where X is either a principal or a principal variable, "`says`" is a keyword, and ba is a base atom. X is called the *issuer* of this statement.[2]

   Such a direct statement encodes that the issuer X believes, supports, or requests the base atom ba. For example, "keyBob says goodCredit(Carl)," "keyCA says isKey(keyBob, Bob)," and "keyTom says purchase(computer, 2000)" are direct statements.

4. A static unweighted threshold structure takes the form:
   `threshold`$(k, [A_1, \ldots, A_n])$
   where "`threshold`" is a keyword, k is a positive integer, the A_i's are principals, and $A_i \neq A_j$, for $i \neq j$. We call k the *threshold value*, and "$[A_1, \ldots, A_n]$" the *threshold pool* (note that it is a set not a bag).

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

   k-out-of-n threshold functions are common in many existing authorization systems, *e.g.*, PolicyMaker [6, 8], KeyNote [5], SPKI/SDSI [11, 27], and Delegation Networks [3]. Such threshold structures introduce fault tolerance and aid flexibility in joint authorization. A static unweighted threshold structure expresses such thresholding for explicit cases where the threshold value and threshold pool are explicit constants.

5. A static weighted threshold structure takes the form:
   `threshold`$(k, [(A_1, w_1), \ldots, (A_n, w_n)])$
   where k and the A_i's are the same as above, and the w_i's are positive integers. The w_i's are called *weights*. The set "$[(A_1, w_1), \ldots, (A_n, w_n)]$" is called a *principal-weight pair set* (abbreviated *P-W set*).

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

6. A dynamic unweighted threshold structure takes the form:
   `threshold`$(k, ?X, Prin$ `says` $pred(\ldots ?X \ldots))$

[2] The issuer is called the "subject" in [22]. Here, we changed it to "issuer" in order to conform to existing terminology, *e.g.*, that of SPKI.

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

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

This syntax is slightly more general than the one in [22], in which the predicate $pred$ is restricted to be a unary predicate. The new syntax allows multiple-arity predicates. It is more convenient and more like the convention Prolog uses for built-in predicates, *e.g.*, `setof` in Prolog.

Static threshold structures become inconvenient when the threshold pool is very large, changes very often, or both. For example, consider a policy that delegates a certain right to any two employees of a large department of a company. Use of a dynamic unweighted threshold structure yields a simple and clear policy and enables the company to change the employees in the department without changing this policy.

7. A *threshold structure* is one of the above three kinds of threshold structures.

   The version of D1LP in [22] has a fourth kind: *dynamic weighted threshold structures*. They make it difficult to achieve tractability. Although the difficulties can be overcome by imposing limits on the maximal weights allowed, we chose to leave them out in this paper in order to simplify the presentation.

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

   For example:

   $(\mathrm{threshold}(1, \mathrm{?X}, \mathrm{comA\ says\ accountant(?X)}),$
   $\mathrm{threshold}(1, \mathrm{?Y}, \mathrm{comA\ says\ manager(?Y)}))$

   is a principal structure that represents the conjunction of at least one accountant and at least one manager according to comA.

9. A *delegation statement* takes the form:

   X `delegates` ba^d `to` PS

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

   X is called the *issuer*, d is called the *delegation depth* ("$*$" means unlimited, or infinite, depth), and PS is called the *delegatee*. For example,

   Bob delegates goodCredit(?X)^1 to Carl

   is a delegation statement. It means that Bob trusts (believes) Carl about whether someone has good credit. That is, if Carl says that someone has good credit, then Bob believes it. This *transferability of belief* is the basic meaning of a delegation in DL.

   Delegation depth is used to control re-delegation. If we have the following statements:

   Alice delegates goodCredit(?X)^2 to Bob
   Bob delegates goodCredit(?X)^1 to Carl
   Carl delegates goodCredit(?X)^1 to David
   Carl says goodCredit(Jack)
   David says goodCredit(John)

   then one can infer in our DL semantics (section 4):

   Alice delegates goodCredit(?X)^1 to Carl
   Alice says goodCredit(Jack)
   Bob says goodCredit(Jack)
   Carl says goodCredit(John)

   but not:

   Bob delegates goodCredit(?X)^1 to David
   Bob says goodCredit(John)

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

   DL uses both integer depth and infinite depth to control re-delegation. SPKI designers chose to use boolean control. The rationale for this choice is documented in section 4.1 of [11]. We think that it is still unclear how re-delegation should be handled in practice, and, in particular, it is not clear whether the rationale given in [11] will be borne out in practical use of re-delegation. We choose to use DL's approach, mainly because it is more expressive. Re-delegatability being "true" and "false" in the SPKI approach can be represented in DL by depth $*$ and 1, respectively, but integer depths other than 1 or $*$ cannot be represented in SPKI. We conjecture that delegation depths such as 2 and 3 might prove useful and natural in many practical policies.

10. A *speaks_for statement* takes the form:

    Y `speaks_for` X `on` ba

    where X and Y are either principals or principal variables, and ba is a base atom.

    The above speaks_for statement intuitively means that Y has every right that X has with respect to the base atom ba, or, in other words, Y is at least as powerful as X with respect to ba. This is intuitively similar to a delegation from X to Y, but the depth is treated differently than in a delegation and the issuer of a speaks_for statement is always left implicit. We will discuss speaks_for statements in more detail at the end of this section.

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

statement, or a speaks_for statement.

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

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

    For example, the following is a body direct statement:
    threshold(2, [cardA, cardB, cardC])
    says accountGood(?X)

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

    The notions of body direct statements and body delegation statements are new relative to the previous version of D1LP. These body statements can only appear in the body of a rule.

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

16. A *clause*, also known as a *rule*, takes the form:
    H `if` F.
    where H is a head statement and F is a body formula. H is called the *head* of the clause, and F is called the *body* of the clause. The body may be empty; if it is, the "`if`" part of the clause may be omitted. A clause with an empty body is also called a *fact*.

    Terminology: The issuer of the head statement is also said to be the issuer of the rule.

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

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

As usual, an expression (*e.g.*, term, base atom, statement, clause, or program) is called *ground* if it does not contain any variables.

Although DL's syntax expresses beliefs from multiple principals, there is always a single, distinguished viewpoint in DL: that of the principal who is doing reasoning and making decisions, *i.e.*, the current trust root "`Local`." Every DL rule or statement is implicitly regarded as a belief of `Local`.

Example 1: Determining credit status

The following is an example D1LP program about determining credit status.

Alice delegates order(?M, ?P)^1 to ?X
 if Alice says goodCredit(?X).
Alice delegates goodCredit(?X)^2 to Bob.

Bob says goodCredit(?X)
 if threshold(2,[cardA,cardB,cardC])
 says accountGood(?X).
cardC says accountGood(Carl).
cardC says accountGood(David).
cardA says accountGood(Carl).
cardB says accountGood(Ed).

The first two rules are Alice's policies. Alice allows anyone who is believed to have good credit to make an order. Alice trusts Bob in determining who has good credit. The third rule is from Bob: Bob 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. The rest of the rules are facts about accounts in good standing.

From these rules (policies and facts), it is inferrable (concluded) according to the DL semantics (section 4) that

Alice says goodCredit(Carl).
Alice delegates order(?M, ?P)^1 to Carl.

but not that

Alice says goodCredit(David).
Alice says goodCredit(Ed).

Discussion of Speaks_for statements

The speaks_for statement "B `speaks_for` A `on` *ba*" is similar to the delegation statement "A `delegates` *ba*`^*` `to` B." The main difference is that conclusions drawn from a speaks_for statement don't consume any delegation depth. Also, the issuer of a speaks_for statement is always implicitly the principal "`Local`", *i.e.*, the trust root.
For example, given the following:

Alice delegates goodCredit(?X)^1 to CB1.
keyCB1 speaks_for CB1 on goodCredit(?X).
keyCB1 says goodCredit(Carl).

then one can conclude that "Alice says goodCredit(Carl)."
But if one changes the second statement to

CB1 delegates goodCredit(?X)^* to keyCB1.

then one can no longer conclude that "Alice says goodCredit(Carl)," because Alice only delegates to CB1 with depth 1.

One main reason for having speaks_for statements is to handle delegations to principals that can not make (*i.e.*, sign) statements directly, *e.g.*, distinguished names in X.509 or local names in SPKI/SDSI.

In example 1, there is a credential issued by Bob about goodCredit(?X). In many scenarios, Bob is a name and can not issue statements; the credential is most likely signed by a public key of Bob. Let us call this key keyBob. Assume that Alice also knows that keyBob is Bob's public key for business purpose, *e.g.*, Alice has the following statement:

Alice says isBusinessKey(keyBob, Bob).

Then by adding the following statement, the trust root Alice can derive the same conclusions as in example 1. ?Key speaks_for ?X on goodCredit(?Y)
if Local says isBusinessKey(?Key, ?X).

Using this speaks_for statement, Alice can delegates to the name of a business and separate this delegation from the binding of keys with the names.

DL's speaks_for notion is similar to the speaks for notion in [1, 21]. However, there are two differences. First, DL's speaks_for relation is defined on a per-base-atom basis. Principal B may speak for principal A with respect to one thing but not another. In [1, 21], if B speaks for A, then B speaks for A with respect to everything. Second, in DL, the issuer of a speaks_for statement is always implicitly the principal "`Local`". But in [1, 21], "B speaks for A" is true if A says so. This is more like DL's delegation relationship in which A delegates to B if A says so. However, the speaks_for relation in [1, 21] is unrestrictedly transitive, *i.e.*, it has no ability to restrict (*i.e.*, to control) re-delegation; it is thus different from the delegation relation in DL.

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

3 More Examples

In this section, we show several further examples of the use of D1LP to represent authorization policies and credentials in different applications.

Example 2: Using multiple certification systems

The following example is modified from an example in [22].

```
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 certificate systems. They all have at most three levels of certificate authorities. 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 trusts anyone who is a "trusted friend" (unconditionally) on binding public keys with sites. The third rule says that Bob is a trusted friend of Alice. The fourth rule says that Bob thinks certification by system Z is 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 some facts. From the above rules plus the facts, it is a conclusion that

Alice says isSiteKey(MKey,MSite)

— this follows from Alice's trust of Bob — but it is not a conclusion that

Alice says isSiteKey(LKey,LSite) — because it is not a conclusion that

XRCA says isSiteKey(LKey,LSite)

Example 3: Accessing medical records

This is an example of controlling access to medical records. It is based on an example in [18]. 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 hospital it knows to certify that a principal is the physician of a patient. HM knows some hospitals by itself; furthermore, it believes that a principal is a hospital if two known hospitals certify that this principal is a hospital. The following D1LP program represents this policy and includes some facts.

```
HM says authorized(?X, read(medRec(?Y)))
  if HM says inRole(?X, physician(?Y)).
HM delegates inRole(?X, physician(?Y))^1
  to threshold(1,?Z, HM says inRole(?Z,hospital)).
HM delegates inRole(?H, hospital)^1
  to threshold(2,?Z, HM says inRole(?Z,hospital)).
HM says inRole(HC, hospital).
HM says inRole(HB, hospital).
HB says inRole(HA, hospital).
HC says inRole(HA, hospital).
HA says inRole(Alice, physician(Peter)).
```

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

Example 4: Controlling delegation

Suppose that Alice wants to delegate to Bob the right to access something and allows Bob to further delegate this right as long as the principals to which Bob delegates are members of some organization `orga`, where this membership must be certified by Carl. 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 organization's members. In D1LP, Alice can represent this policy by the following two delegation statements.

$\mathrm{Alice\ delegates\ access\hat{}*\ to\ (Bob,tmpKey).}$
$\mathrm{tmpKey\ delegates\ access\hat{}1}$
$\quad\mathrm{to\ threshold(1,?X,Carl\ says\ member(?X,orga)).}$

Here, `tmpKey` is a new "dummy" principal created by Alice.[3] Alice can first generate a new pair of public-private keys as tmpKey, then sign the second statement with the new private key and use the new public key in the first statement. After signing the second statement, Alice can throw the new secret key away and not 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 a member, and tmpKey does not allow re-delegation. Therefore, this achieves the intended policy.

Suppose further that we have

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

Then the delegation "Alice delegates access ˆ1 to David" is a conclusion, but "Alice delegates access ˆ1 to John" is not a conclusion.

4 Semantics

In this section, we give the formal semantics of the revised D1LP.

4.1 Overview

The semantics of D1LP defines a minimal model for every D1LP program $\mathcal{P}$ and an answer to every query Q relative to $\mathcal{P}$.

The D1LP program $\mathcal{P}$ is first transformed (essentially, compiled) into a definite OLP $\mathcal{O}$ in a sorted (typed) OLP language $\mathcal{LO}_\mathcal{P}$. This transformation is defined in such a manner that it corresponds straightforwardly to an algorithm. According to the usual 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.

[3]Note that this is a different kind/use of "dummy" principal than the "dummy" principals employed in the definition of the semantics of delegation, discussed in section 4 and section 7.

The minimal D1LP model of $\mathcal{P}$, denoted by $M_\mathcal{P}$, is obtained by reverse-transforming $M_\mathcal{O}$ back into D1LP syntax. $M_\mathcal{P}$ is a set of entailed ground conclusions expressed in D1LP. The reverse transformation is defined in such a manner that it too corresponds straightforwardly to an algorithm. This inferencing procedure to compute the entire model $M_\mathcal{P}$ is called *exhaustive* (forward) inferencing. It is useful when one wants to check all the conclusions of a program.

As in OLP, one does not always want to perform exhaustive inferencing. For example, one may wish to answer a particular query Q. In section 4.6, we give a query answering procedure that avoids computing the entire minimal D1LP model. Query answering is also called *backward* inferencing.

The most complex and innovative part of our redefinition of D1LP semantics is the definition of a tractable transformation from $\mathcal{P}$ to $\mathcal{O}$. We now specify the details of that transformation.

In specifying the transformation and calculating the size of $\mathcal{O}$, 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 finite delegation depth used in $\mathcal{P}$. Because it is difficult to imagine an authorization decision that distinguishes between depth, say, 12 and 13, we expect D to be a very small integer, *e.g.*, 3 to 5. For any $d \in [0..D]$, we define that $d < *$. We also define $[0..*] = [0..D] \cup \{*\}$ and $[*..*] = \{*\}$. For $d_1, d_2 \in [0..*]$,

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

We continue specifying the transformation in the next three subsections, first showing how to transform a program $\mathcal{P}$ without threshold structures, then showing how to handle threshold structures as well.

4.2 Transformation Without Threshold Structures

As mentioned in the beginning of section 4.1, the language $\mathcal{LO}_\mathcal{P}$ is sorted, *i.e.*, typed. In a sorted LP language, each variable has a sort (type), and each function symbol has a type signature. The type signature of a given function symbol $func$ specifies, firstly, the type of each of $func$'s arguments. The signature specifies, secondly, the type of $func$'s "return value," *i.e.*, the type of any term of the form $func(...)$. Variables of one sort can only be instantiated to terms of the same sort. There are simple techniques to translate programs from a sorted language to a unsorted language. See [20] for a standard reference.

The symbols of $\mathcal{LO}_{\mathcal{P}}$ include all the constants of $\mathcal{P}$ plus a bunch of new predicate and function symbols introduced by the transformation. The language $\mathcal{LO}_{\mathcal{P}}$ has only two sorts. All the variables and constants coming from $\mathcal{P}$ are in one sort. All the terms introduced during the transformation are in the other sort. All the variables in $\mathcal{LO}_{\mathcal{P}}$ actually come from $\mathcal{P}$, *i.e.*, appeared in $\mathcal{P}$, because the transformation does not introduce any new variables, as we will see soon. Therefore, all the variables in $\mathcal{LO}_{\mathcal{P}}$ are in the first sort.

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

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

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

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

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

Here, dep stands for depth and $dele$ stands for delegatee. The domains of $issuer$ and ba are the same as they are in $holds$; the domain of dep is $[1..*]$; the domain of $dele$ is $Principals$, the same as the $issuer$ field; the domain of len is $[0..*]$. Note that the len field of a $delegates$ atom can be 0; this will be the case for "speaks_for" statements.

Function PSFormula:

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

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

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

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

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

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

The function $PSFormula$ transforms statements that have complex principal structures as issuers to equivalent formula of statements whose issuers are either principals or principal variables. This function enables body statements to be more general than head statements.

For now, $PSFormula$ simply returns a formula. When we start to deal with threshold structures in sections 4.3 and 4.4, $PSFormula$ will have side effects as well as returning a formula; it will generate some additional rules and introduce some new constants.

Finally, we can define the transformation. It is divided into two steps: *body transformation* and *head transformation*.

Transformation I: Body transformation

Transformation I both changes rules in $\mathcal{P}$ and constructs some new rules. The result of changing $\mathcal{P}$ is called $\mathcal{P}_1$. The set of new rules is called $\mathcal{P}_1^{add}$.

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

1. Replace each body direct statement
   AS `says` ba
   with "$PSFormula(AS, holds(ba, *))$,"
   where AS is a principal structure or a principal variable.

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

2. Replace each body delegation statement
   AS `delegates` ba`^`d `to` B
   with "$PSFormula(AS, delegates(ba, d, B, *))$,"
   where AS is a principal structure or a principal variable and B is a principal or a principal variable.

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

3. Replace each speaks_for statement
   B `speaks_for` A `on` ba
   with "$delegates(A, ba, *, B, 0)$."

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

4. Replace each body delegation statement
   AS `delegates` $ba\hat{}d$ `to` $(B_1, \ldots, B_n)$
   with "$PSFormula(AS, delegates(ba, d, B_{new}, *))$," where $B_1, \ldots, B_n$ are principals, and B_{new} is a newly created principal.

   For each B_i, $i = 1..n$, add the following fact to $\mathcal{P}_1^{add}$:
   $delegates(B_i, ba, *, B_{new}, 0)$.

   This transformation enables tractable inference of delegations that have conjuncts of principals as delegatees. Remember that the $dele$ field of the predicate $delegates$ is required to be a principal, rather than a conjunction of principals, as in [22]. Here, we introduce a dummy principal B_{new} to represent the principal structure "$(B_1, \ldots, B_n)$." That B_{new} is equivalent to "$(B_1, \ldots, B_n)$" is fully characterized by the relationships that B_{new} speaks 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 program $\mathcal{O}$ is sufficient.

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

Transformation II: Head transformation

The input to Transformation II is $\mathcal{P}_1$. The transformation changes rule heads in $\mathcal{P}_1$; the result is called $\mathcal{P}_2$. Transformation II 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.

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

2. **Holds length-weakening meta-rule:**
   For each $len \in [1..D]$, add the following rule:
   $holds(A, ba, len \oplus 1)$ `if` $holds(A, ba, len)$.
   This meta-rule states that, if something can be derived with smaller length, then it can also be inferred when larger length is allowed.

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

Sub-case a: If R's head is a delegation statement:
A `delegates` $ba\hat{}d$ `to` BS,
i.e., a depth-d delegation from A to BS: let ll be 1, and B_{new} be B if B is a single principal or a principal variable; otherwise let B_{new} be a newly created principal (dummy principal).

Sub-case b: If R's head is a speaks_for statement:
B `speaks_for` A `on` ba:
let d be $*$; ll be 0, and B_{new} be B.

For both sub-cases, do the following.

3. **Delegates head translation:**
   Replace R's head with
   $delegates(A, ba, d, B_{new}, ll)$.

4. **Holds propagation meta-rule:**
   For each $len \in [1..d]$, add the following rule:
   $holds(A, ba, len \oplus ll)$
   `if` $delegates(A, ba, d, B_{new}, ll)$,
   $PSFormula(BS, holds(ba, len))$.
   This meta-rule propagates direct statements through delegation as follows: If the delegation in R's head is true (by $II.3$, it is true when the body of R is true) and the delegatee BS supports something within len delegation steps, then the issuer A supports the same thing within $len \oplus ll$ steps, where ll is 1 if R's head is a delegation and 0 if R's head is a speaks_for statement.

5. **Holds length-weakening meta-rule:**
   For each $len \in [d \oplus 1..D]$, add the following rule:
   $holds(A, ba, len \oplus 1)$ `if` $holds(A, ba, len)$.
   This meta-rule is the same as $II.2$. It appears again, because it is also needed for case two.

6. **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.

7. **Delegates length-weakening meta-rule:**
   For each $C \in Principals$, for each $dep \in [1..d]$, and for each $len \in [0..D]$, add the following rule:
   $delegates(A, ba, dep, C, len \oplus 1)$
   `if` $delegates(A, ba, dep, C, len)$.
   This meta-rule states that any delegation that is derived within a certain length can also be derived within a larger length.

8. **Delegates depth-weakening meta-rule:**
   For each $C \in Principals$, for each $len \in [0..*]$, and for each $dep \in [1..D]$, add the following rule:
   $delegates(A, ba, dep, C, len)$
   `if` $delegates(A, ba, dep \oplus 1, C, len)$.
   This meta-rule states that a smaller-depth delegation can be derived if a corresponding larger-depth delegation is derived.

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:
$delegates(A, ba, \min(d \ominus len, dep), C, ll \oplus len)$
`if` $delegates(A, ba, d, B_{new}, ll)$,
$PSFormula(BS, delegates(ba, dep, C, len))$.
where for any $d1, d2 \in [0..D]$: "$* \ominus * = *$," "$*\ominus d1 = *$," "$d1\ominus d2 = d1-d2$," and "$d1\ominus * = -*$."

This is the most complex and the most expensive (in terms of the size of the new rules added) meta-rule. It chains two delegations to derive a new one. The derived delegation's depth is bounded both by the depth of the second delegation in the chain and the depth of the first delegation minus the number of delegation steps used to derive the second delegation.

The above meta-rules may seem unnecessarily complicated, especially in the way they deal with length and delegation depth. They are so because we are avoiding introducing new variables in the transformation; this is essential in ensuring tractability. In fact, we exclude dynamic weighted threshold structures in this version of D1LP precisely because they require introducing new variables.

The result of the transformation is:

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

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

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

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

In Transformation I, a body statement is replaced by the result of a corresponding $PSFormula$ call. Therefore, $|\mathcal{P}_1|/|\mathcal{P}| = O(N)$. If the 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, $|\mathcal{P}_1^{add}|/|\mathcal{P}| = O(N)$.

In Transformation II, if a rule has a direct statement in the 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 $\mathcal{P}_1$, so $|\mathcal{P}_2|/|\mathcal{P}| = O(N)$.

In Transformation II, if a rule R has a delegation statement or a speaks_for statement in the head, several meta-rules apply; each adds a set of rules to $\mathcal{P}_2^{add}$, but the size of $\mathcal{P}_2$ remain unchanged from $\mathcal{P}_1$. Transformation $II.9$ (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 $PSFormula$ to change parts of it. Therefore, the growth factor for transformation $II.9$ is $O(N^2D^2)$.

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

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. Then we extend the definition of $PSFormula$ to include the following:

$PSFormula(threshold(k, [A_1, \ldots, A_n]), holds(ba, l))$
$= holds(suth(k, [A_1, \ldots, A_n]), ba, l)$
and a similar definition in which a delegates atom replaces the holds atom.

The function of $PSFormula$, for calls of such form, results in side effects besides returning an atom: it adds the following new rules to $\mathcal{P}_2^{add}$. These rules reason about atoms that have issuers of the form "$suth(i, [A_1, \ldots, A_n])$." The integer i is the remaining threshold value that needs to be satisfied.

- For $i = k$ to 1, for $j = 1$ to $n - i$, add the rule:
  $holds(suth(i, [A_j, A_{j+1}, \ldots, A_n]), ba, l)$
  `if` $holds(suth(i, [A_{j+1}, \ldots, A_n]), ba, l)$.

- For $i = k$ to 1, for $j = 1$ to $n - i + 1$, add the rule:
  $holds(suth(i, [A_j, A_{j+1}, \ldots, A_n]), ba, l)$
  `if` $holds(A_j, ba, l)$,
  $holds(suth(i - 1, [A_{j+1}, \ldots, A_n]), ba, l)$.

- For $j = k + 1$ to n, add the fact:
  $holds(suth(0, [A_j, \ldots, A_n]), ba, l)$.

- Add three analogous sets of rules for $delegates$. Here, we omit the details.

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

Static weighted threshold structures are handled similarly; a new function symbol "$swth$" is introduced. Handling them doesn't change the growth factor of $PSFormula$, it is still $O(N^2)$. We omit the details here.

4.4 Transformation with Dynamic Threshold Structures

To handle dynamic threshold structures, we need a list of all principals in $Principals$, the set of all principals in $\mathcal{P}_1 \cup \mathcal{P}_1^{add}$. Let $M = |Principals|$ and let $[G_1, G_2, \ldots, G_M]$ be one such list.

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 "$duth(i, j, c)$," where i and j are integers, and c is a newly introduced constant. The integer i is similar to that in $suth$. And the integer j is an index to the list of principals "$[G_1, G_2, \ldots, G_M]$." We also extend the definitions of $PSFormula$ to include the following:

$$PSFormula(threshold(k, ?X, Prin\ \texttt{says}\ pred(\ldots ?X \ldots)), holds(ba, l)) = holds(duth(k, 1, c), ba, l)$$

and a similar definition for $delegates$.

Each time $PSFormula$ is called with a dynamic unweighted threshold structure argument, a new constant c and a set of new rules are generated. The constant c is used to uniquely identify the dynamic threshold pool defined by "?X, Prin says pred(... ?X ...)." The new rules are as follows.

1. For $i = k$ to 1, for $j = 1$ to $M - i$, add the rule:
   $holds(duth(i, j, c), ba, l)$
   `if` $holds(duth(i, j + 1, c), ba, l)$.

2. For $i = k$ to 1, for $j = 1$ to $M - i + 1$, add the rule:
   $holds(duth(i, j, c), ba, l)$
   `if` $holds(Prin, pred(\ldots G_j \ldots), *)$,
   $holds(G_j, ba, l)$,
   $holds(duth(i - 1, j + 1, c), *)), ba, l)$.

3. For $j = k + 1$ to n, add the rule:
   $holds(duth(0, j, c), ba, l)$.

4. Add an analogous set of rules for $delegates$. Here, we omit the details.

For each dynamic threshold structure, $O(\min(k, M)M)$ rules are added, where k is the threshold value. Recall that $M = |Principals| = O(N)$. Thus, the worst-case growth factor of $PSFormula$ with dynamic threshold structures is still $O(N^2)$, the same as that 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, was the size of one static threshold pool).

4.5 Reverse Transformation of Conclusions

We next define a simple reverse transformation that maps an OLP model of $\mathcal{O}$ to a D1LP model of $\mathcal{P}$. This reverse transformation is useful if one wants all the D1LP conclusions entailed by $\mathcal{P}$.

- For each atom of the form: $holds(A, ba, len)$, where A is a principal, include the D1LP-conclusion:
  A `says` ba.
- For each atom of the form: $delegates(A, ba, *, D, 0)$, where A and D are principals, include the D1LP-conclusion:
  D `speaks_for` A `on` ba.
- For each atom of the form:
  $delegates(A, ba, dep, D, len)$,
  where A and D are principals, and where $len > 0$, include the D1LP-conclusion:
  A `delegates` $ba\hat{\ }dep$ `to` D.
  (Note that, because of the way the semantic transformation is defined, there are no atoms with both $len = 0$ and $dep < *$.)

Notice here that length is ignored after the OLP conclusions are drawn.

4.6 Query answering

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

Next, we give an **algorithm** to answer the query Q relative to $\mathcal{P}$, that avoids exhaustive inferencing:

1. Transform Q into an OLP query, using the same procedure as the one used to transform rule-bodies, *i.e.*, Transformation I (see section 4.2). This transformation changes Q into an OLP query Q' and generates a new set of OLP rules $\mathcal{Q}^{add}$ (possibly empty).
2. Form a OLP program $\mathcal{O}' = \mathcal{O} \cup \mathcal{Q}^{add}$.

3. Answer the OLP query 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 Q relative to $\mathcal{P}$.

5 Tractability Results; Algorithms

In this section, we give upper-bound results on the worst-case computational complexity of the transformation from D1LP to OLP (which is defined in the previous section) and of overall D1LP inferencing using this approach. We show that the transformation is tractable and that, under commonly-met restrictions, overall D1LP inferencing is also tractable.

Theorem 1 (Tractable Transform Size)
The size of the output program $\mathcal{O}$ is $O(N^4D^2)$, where $N = |\mathcal{P}|$, and D is the maximal delegation depth in $\mathcal{P}$.

Proof. From the counting arguments in the previous section, it follows straightforwardly that $|\mathcal{O}|/|\mathcal{P}|$ is $O(N^3D^2)$. Therefore, $|\mathcal{O}| = O(N^4D^2)$. ∎

We expect that D will usually be much smaller than N, *e.g.*, less than 10.

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 not all rules grow by the factor of $O(N^3D^2)$.

Consider a rule R that does not contain any threshold structures; let S_R be the size of the largest principal structure in R. Certainly $S_R < |R| < N$. Often, S_R will be a small constant. If R's head is a direct statement, R's growth factor is $\max(S_R, D)$. Otherwise, R's growth factor is "MS_RD^2." Recall that $M = |Principals| = O(N)$ and $Principals$ is the set of all principals in $\mathcal{P}_1 \cup \mathcal{P}_1^{add}$. Often, this "$MS_RD^2$" factor is much smaller than the worst-case factor $O(N^3D^2)$.

Transformation of rules whose heads contain threshold structures is more expensive. Consider such a rule R. Let K_R be the largest threshold value in R's head; let L_R be the size of the largest threshold pool in R's head. If R's head contains any dynamic threshold structure, $L_R = |Principals| = M$. The growth factor of R is "$\min(K_R, L_R)L_RMD^2$," which, in the worst case, is N^3D^2. However, often, K_R will be a small constant. Also, static threshold structures with small threshold pools are much less expensive than dynamic ones, because their L_R factor is much smaller. Moreover, although M is typically of the same order of N, often, it will be significantly smaller than N. Also note that having threshold structures in one rule doesn't affect the growth factor of other rules. We expect that, in most scenarios, rules that use threshold structures will be relatively rare compared to simple rules and facts.

In the previous section, we described the transformation by defining its output. We observe that the transformation (plus the reverse transformation) corresponds straightforwardly to an **algorithm** for transformation (and reverse transformation). We observe further that this algorithm for transformation takes time linear in the size of the output OLP.

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

Proof. Follows from theorem 1, along with the above observation that the transformation from D1LP can be implemented by an algorithm whose time is proportional to the output size. ∎

Next, we review some previously known results about OLP (see, *e.g.*, [4] [20]). We say that a 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). A fact about LP's is that: given a sorted definite OLP program $\mathcal{O}$ that is VB(v), if the Herbrand universe for each sort of variable is bounded by N, then the ground instantiation of $\mathcal{O}$ has size $O(|\mathcal{O}|N^v)$. This is because for each variable, there are at most $O(N)$ ground terms that can be used to instantiate it. Then for each rule, there are at most $O(N^v)$ ways to instantiate it. Another fact about LP's is that: for a definite OLP, exhaustive inferencing (*i.e.*, computing its entire model) takes time (and space) that is worst-case linear in the size of its instantiated version. This is a major reason why OLP inferencing is very practical, *e.g.*, as in SQL/RDB and Prolog, and is often done with these restrictions (*e.g.*, Datalog restriction common in SQL/RDB).

A straightforward **algorithm** for computing the minimal D1LP model of a given D1LP program $\mathcal{P}$ is: transform the D1LP $\mathcal{P}$ into the OLP $\mathcal{O}$, then compute the minimal OLP model of $\mathcal{O}$, then reverse-transform this model. There are a number of existing procedures for computing a minimal OLP model, *e.g.*, Smodels [26].

Theorem 3 (Tractable D1LP Inferencing)
Given a D1LP $\mathcal{P}$ that is VB(v), computing the minimal D1LP model of $\mathcal{P}$ has time complexity $O(N^{v+4}D^2)$.

Proof. If $\mathcal{P}$ is VB(v), then $\mathcal{O}$ is also VB(v), because the transformation from $\mathcal{P}$ to $\mathcal{O}$ doesn't introduce any new variables. All (logical) variables in $\mathcal{O}$ are in one sort; and this sort has a Herbrand universe of size N – because all of the logical function symbols in $\mathcal{P}$ have zero arity, *i.e.*, they are all constants. Therefore, the instantiated size of $\mathcal{O}$

is $O(|\mathcal{O}|N^v) = O(N^{v+4}D^2)$. Then, computing the minimal OLP model of $\mathcal{O}$ takes time $O(N^{v+4}D^2)$, and the size of this model is $O(N^{v+4}D^2)$. The reverse transformation takes time linear in the size of this model. So computing the minimal D1LP model of $\mathcal{P}$ has time complexity $O(N^{v+4}D^2)$. ∎

6 Implementation

Overall, our implementation architecture for this version of DL uses a *delegation compiler* from DL to OLP that implements the semantical transformation we gave from DL to OLP. Earlier, we gave[4] not only an algorithm for this delegation compiler, but also an algorithm for using the delegation compiler to compute the entire minimal D1LP model, and (in section 4.6) an algorithm for using the delegation compiler to answer D1LP queries (without computing the whole the entire minimal D1LP model).

We have implemented, in Java, such a delegation compiler that does the transformation from D1LP to OLP (and the reverse transformation back from OLP to D1LP) described in Section 4 and does exhaustive D1LP inferencing by combining that compiler with a previously existing OLP inferencing engine. The implemented compiler can generate OLP in the syntactic formats of a variety of OLP inferencing engines, both forward reasoning ones (*e.g.*, Smodels [26]) and backward reasoning ones (*e.g.*, XSB [29], a variant of Prolog).

The compiler and D1LP inferencing engine are integrated as an extension to the previously existing IBM CommonRules system [17] [15], a Java library which among its capabilities includes a rule translation format ("interlingua", encoded in XML) and sample translators to talk to multiple OLP inferencing engines. The D1LP compiler reuses classes and code from the CommonRules core, especially for specification and inferencing.

The implementation is somewhat expressively restricted, and slightly different syntactically, from the version of D1LP given in this paper. (It was based on a preliminary version of the design in this paper.)

We were able to develop this implementation rapidly — in a few person-weeks of coding effort — by building upon existing OLP systems, largely because the delegation compiler approach provides great modularity in the software engineering sense.

We have also implemented the compiler in XSB [29], a Prolog-variant logic programming system developed at SUNY Stony Brook. This second implementation uses an alternative transformation that is different from (but similar to) the one we gave in section 4 and used in the first (Java) implementation. 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 "ungrounded transformation," and the transformation in Section 4 "grounded transformation." This implementation can handle everything in the syntax described in Section 2. With ungrounded transformation, it can also handle the *dynamic weighted* threshold structures defined in [22]. This implementation uses XSB's inference engine, so it can answer queries directly.

The expanded Research Report version of this paper gives additional details about implementation, including sample output.

7 Discussion of Syntax; Ensuring Computational Tractability

As we discussed near the beginning of section 2, the new version of D1LP syntax has the conjunctive-delegatee-queries expressive restriction, in order to ensure that it is tractable, unlike the version in [22]. In the rest of this section, we discuss in detail the tractability motivation for the conjunctive-delegatee-queries restriction.

To understand this change, it is necessary first to understand why D1LP as defined in [22] is intractable.

In [22], D1LP's semantics defines a minimal model for each D1LP program $\mathcal{P}$: as the least fixed point of an operator $\Psi_{\mathcal{P}}$. The operator $\Psi_{\mathcal{P}}$ takes as input an interpretation I of $\mathcal{P}$. First, it transforms $\mathcal{P}$ to an ordinary logic program (OLP) $\mathcal{O}^I$; this transformation depends upon I. Next, it computes $\mathcal{O}^I$'s minimal OLP model. Then, it maps this model back to a corresponding interpretation of $\mathcal{P}$ and finally returns this interpretation J. It is shown in [22] that this iteration process always terminates and yields a unique minimal model. We observe here that, unfortunately, it is very expensive computationally.

Because the transformation from D1LP to OLP depends on the interpretation I, computing the minimal model of one D1LP program requires an iterative series of steps until the (least) fixed point is reached. Each iteration step includes a transformation phase and an OLP inferencing phase. Computing the minimal D1LP model thus requires base-level OLP inferencing to be interleaved repeatedly with transforming. The obvious upper bound on the number of iterations needed to reach the fixed point is the size of the Herbrand base, which is normally quite large.

Furthermore, even one phase of the transformation is intractable. Even under the commonly-met expressive restrictions that ensure that OLP is tractable (Datalog plus bounded number of logical variables per rule), the transformation phase can generate an OLP program whose size is exponential in the size of the starting D1LP $\mathcal{P}$. The trans-

[4] at a high level of description

formation phase also instantiates $\mathcal{P}$, which is normally quite expensive.

The source of the exponential growth in program size is the transformation (defined in [22] as part of the overall transformation from a D1LP to an OLP) from a principal structure that contains disjunctions (either explicitly or implicitly through threshold structures) to its normal form, which is similar to the disjunctive normal form (DNF) of a propositional logical formula. The principal structure $((A_{11}; \ldots ; A_{1n}), (A_{21}; \ldots ; A_{2n}), \ldots , (A_{m1}; \ldots ; A_{mn}))$ contains $m \times n$ principals and thus has size $O(m \times n)$, but its normal form has size $O(n^m)$. The normal form of the threshold structure "$\texttt{threshold}(k, [A_1, \ldots , A_n])$" has size $O(\binom{n}{k})$. Both normal forms are thus worst-case exponential in size.

The source of the dependence on the interpretation is the transformation of a dynamic threshold structure to its normal form. The threshold pool of a dynamic threshold structure is decided by a direct statement and thus varies from interpretation to interpretation. This dependence makes it necessary to iterate until the interpretation "stabilizes," *i.e.*, reaches fixed point. This also means that answering a single D1LP query requires computing the minimal model of the whole D1LP program. This is undesirably different from many other knowledge representations, *e.g.*, OLP. In backward-reasoning OLP systems such as Prolog and SQL, answering a query does not require such inefficiently exhaustive computation of the whole OLP program's model, *i.e.*, of its whole set of conclusions.

The arguments in the previous two paragraphs show that the source of the intractability is in the transformation from principal structures to their normal forms. Why is this transformation needed? It is used to support reasoning about delegations, *i.e.*, to conclude a weaker delegation from a stronger one.

Such reasoning is based on 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 statements "A delegates to B" and "A delegates to C." By contrast, "A delegates to (B,C)" is a weaker delegation than either "A delegates to B" or A "delegates to C" in the sense that either of the last two delegations implies the first, but not the converse.

Note that for any two different sets of principals $\mathcal{S}_1$ and $\mathcal{S}_2$, the delegation from A to the conjunction of $\mathcal{S}_1$ is different from the delegation from A to the conjunction of $\mathcal{S}_2$. The transformation in [22] generates delegations to conjunctions of principals as conclusions and translates delegations to complex principal structures into equivalent delegations to conjunctions of principals. This requires transforming principal structures into normal forms and thus results in worst-case exponential growth. Note that the number of sets of principals in a D1LP program is worst-case exponential in the size of the program. So the number of all conclusions is worst-case exponential.

To get a flavor of what kind of inference D1LP in [22] supports, see the following example.

```
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]).
```

Given the above D1LP program, the semantics in [22] concludes "A says do_something." From the first fact, it concludes "A delegates p^1 to (B,C)" and "A delegates p^1 to (B,D)." These two conclusions together with the fact "A delegates p^1 to (C,D)" prove the delegation "A delegates p^1 to threshold(2,[B,C,D])." Besides being expensive, such inferences from stronger delegations to weaker ones can be tricky and difficult to understand.

To be tractable, the new version of D1LP only generates as delegation conclusions: delegations *to a single principal*. Therefore, it can only answer directly those delegation queries that have a single principal as delegatee. However, the new version of D1LP permits one to have delegations to a *conjunction* of principals: in rule bodies or queries. Semantically, these are handled by introducing a new "dummy" principal to represent the conjunction, and then mapped to single-principal queries; see transformation I in section 4.1 for the details.

It is useful to have such delegations to conjunctions of principals. For example, when a request is signed by multiple principals, one may need to determine whether there is a delegation from "`Local`" to the conjunction of all the signers.

Under this restriction on delegation conclusions in the new D1LP, delegation queries that contain disjunctions (among the delegatees) can not be answered. Therefore, we forbid queries or rule bodies (since they are implicit queries) to have disjunction within delegatees: neither explicit disjunction nor threshold structures is allowed, because they contain implicit disjunction. This is the conjunctive-delegatee-queries expressive restriction we defined near the beginning of section 2.

We conjecture that this restriction leaves D1LP with all of the expressive power needed in practice. All the examples given in [22] are in the restricted class we define here. In fact, we have not yet encountered a realistic example which requires delegation queries that contain disjunctions, *i.e.*, which does not obey the conjunctive-delegatee-queries restriction.

8 Discussion, Current and Future Work

Our overall approach to semantics via transformation, and our overall approach to implementation via a compiler, was inspired in part by previous work by one of us on a similar transformation/compiler approach to prioritized conflict handling in logic programs: courteous logic programs [13, 14, 15, 16] which are transformable/compileable into OLP with negation-as-failure, and which are implemented in IBM CommonRules [17]. Our implementation also reused classes and code from IBM CommonRules.

We gave an implementation architecture for this version of DL; it uses a *delegation compiler* from DL to OLP that implements the semantical transformation we gave from DL to OLP. In particular, we gave[5] not only an algorithm for this delegation compiler, but also an algorithm for using the delegation compiler to compute the entire minimal D1LP model, and (in section 4.6) an algorithm for using the delegation compiler to answer D1LP queries (without computing the whole the entire minimal D1LP model).

There are several additional infrastructural issues, beyond what we discussed in this paper, that are practically important for developing real-world systems based on DL, and which are the subject of current and future work. Next, we discuss some of them.

One infrastructure issue is: what data structures and communication protocols to use for exchanging DL rules between distributed applications/principals/Internet-sites. An approach we are currently exploring is to encode DL in XML syntax, in a manner building upon the XML Business Rules Markup Language for OLP's that is supported by IBM CommonRules[6].

However, there are work-arounds to use DL even in the absence of such a communication infrastructure. One way is to first 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 infrastructure issue is how an authorizer obtains all the necessary credentials to make the decision. There are several possible scenarios for how such credentials should flow to the authorizer. One is that the requester submits credentials together with its request. Another is that the authorizer asks the requester for additional credentials during the evaluation of the request. Yet another is that the authorizer asks other entities for relevant credentials during the evaluation of the request. Mixes of the above are also interesting. How to obtain relevant credentials dynamically during DL inference is a topic we are exploring.

[5] at a high level of description

[6] http://www.research.ibm.com/rules/ and http://alphaworks.ibm.com

Next, we briefly outline additional areas of current and future work. These include: fuller implementation; applications, especially in the area of inter-enterprise e-commerce; and expressive generalization especially to D2LP, the extension (sketched in [22]) of D1LP that enables negation and prioritized conflict handling. We believe that our compiler approach will extend to D2LP, by tractably compiling a D2LP into a courteous logic program [13] [15], which is in turn itself tractably compilable into an OLP. It compiles into OLP with negation-as-failure, however, which is still tractable under the same restrictions we discussed here in connection with our tractability results in section 5 (VB and polynomial-size Herbrand universe (*e.g.*, Datalog)).

9 Conclusions

We made Delegation Logic (DL) into a tractable and practically implementable trust-management system by introducing a new version of D1LP that is syntactically restricted from the version in [22]. Its semantics is defined by a tractable, one-pass transformation into OLP. This transformation led us directly to an implementation approach based on compiling D1LP into OLP. The new version of D1LP also has some expressive advantages as well. The tractability and modularity of implementation make this new version of D1LP far more practical than the previous version. We implemented a large expressive fragment of this new version of D1LP using this approach. We were able to do this rapidly by building upon existing OLP systems.

The expanded Research Report version of this paper gives additional details, including sample output of the implementation.

Acknowledgements

We are grateful to Hoi Chan of IBM T.J.Watson Research Center for help in the Java implementation of the DL compiler. We also thank the anonymous reviewers whose detailed and insightful comments and suggestions have significantly improved this paper. We benefited from discussions with Yosi Mass and Amir Herzberg of IBM Haifa Research, and with Martin Abadi of Bell Labs.

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, "A Calculus for Access Control in Distributed Systems," *ACM Transactions on Programming Languages and Systems*, 15 (1993), pp. 706–734.

[2] R. Anderson, "A Security Policy Model for Clinical Information Systems," in *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Los Alamitos, 1996, pp. 30–43.

[3] T. Aura, "On the Structure of Delegation Networks," in *Proceedings of the 10th IEEE Computer Security Foundation Workshop*, IEEE Computer Society Press, Los Alamitos, 1998.

[4] C. Baral and M. Gelfond, "Logic Programming and Knowledge Representation", *Journal of Logic Programming*, 19,20 (1994), pp. 73–148. Includes extensive review of literature.

[5] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, "The KeyNote Trust-Management System, Version 2," Internet Engineering Task Force RFC 2704, September 1999.

[6] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized Trust Management," in *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Los Alamitos, 1996, pp. 164–173.

[7] M. Blaze, J. Feigenbaum, and J. Lacy, "Managing Trust in Medical Information Systems," AT&T Technical Report 96.14.1. `http://www.research.att.com/library/trs/TRs/96/96.14/96.14.1.body.ps`

[8] M. Blaze, J. Feigenbaum, and M. Strauss, "Compliance-Checking in the PolicyMaker Trust Management System," in *Proceedings of Financial Crypto '98*, Lecture Notes in Computer Science, vol. 1465, Springer, Berlin, 1998, pp. 254–274.

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

[10] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss, "REFEREE: Trust Management for Web Applications," *World Wide Web Journal*, 2 (1997), pp. 706–734.

[11] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "SPKI Certificate Theory," Internet Engineering Task Force RFC 2693.

[12] J. Gosling and H. McGilton, *The Java Language Environment, A White Paper*, Sun Microsystems, Inc., Mountain View, 1995.

[13] B. Grosof, "Prioritized Conflict Handling for Logic Programs," in *Proceedings of the International Symposium on Logic Programming*, MIT Press, Cambridge, 1997, pp. 197–212.

[14] B. Grosof, "Compiling Prioritized Default Rules Into Ordinary Logic Programs," IBM Research Report , April 1999.

[15] B. Grosof, "DIPLOMAT: Compiling Prioritized Default Rules Into Ordinary Logic Programs, for E-Commerce Applications (extended abstract of Intelligent Systems Demonstration)," in *Proceedings of AAAI-99*, Morgan Kaufmann, 1999.

[16] B. Grosof, "A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML," in *Proceedings of the 1st ACM Conference on Electronic Commerce*, ed. by M. Wellman. ACM Press, 1999.

[17] B. Grosof, H. Chan, *et al*, IBM CommonRules home pages, `http://www.research.ibm.com/rules/` and `http://alphaworks.ibm.com` .

[18] A. Herzberg, J. Mihaeli, Y. Mass, D. Naor, and Y. Ravid, "Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers," to appear in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Los Alamitos, California, 2000. Also available at: `http://www.hrl.il.ibm.com/TrustEstablishment/paper.asp`

[19] S. T. Kent, "Internet Privacy Enhanced Mail," *Communications of the ACM*, 8 (1993), pp. 48–60.

[20] J. W. Lloyd, *Foundations of Logic Programming*, second edition, Springer, Berlin, 1987.

[21] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in Distributed Systems: Theory and Practice," *ACM Transactions on Computer Systems*, 10 (1992), pp. 265–310.

[22] N. Li, B. Grosof, and J. Feigenbaum, "A Logic-Based Knowledge Representation for Authorization with Delegation", IBM Research Report RC21492, June 1999. Extended abstract (actually, a full-length conference paper) appears in *Proceedings of the 12th IEEE Computer Security Foundations Workshop*.

[23] M. Longhair (editor), "A P3P Preference Exchange Language (APPEL) Working Draft," W3C Working Draft 9, October 1998, `http://www.w3.org/P3P/Group/Preferences/Drafts/WD-P3P-preferences-19981009`.

[24] M. Marchiori, J. Reagle, and D. Jaye (editors), "Platform for Privacy Preferences (P3P1.0) Specification," W3C Working Draft 9 November 1998, `http://www.w3.org/TR/WD-P3P/`.

[25] U. Maurer, "Modelling a Public-Key Infrastructure," in *Proceedings of the 1996 European Symposium on Research in Computer Security*, Lecture Notes in Computer Science, vol. 1146, Springer, Berlin, 1997, pp. 325–350.

[26] I. Niemela and P. Simons, Smodels home page, `http://saturn.hut.fi/html/staff/ilkka.html`.

[27] R. Rivest and B. Lampson, "Cryptography and Information Security Group Research Project: A Simple Distributed Security Infrastructure," `http://theory.lcs.mit.edu/˜cis/sdsi.html`.

[28] P. Resnick and J. Miller, "PICS: Internet access controls without censorship," *Communications of the ACM*, 39 (1996), pp. 87–93.

[29] D. Warren, *et al*, "The XSB Programming System," `http://www.cs.sunysb.edu/˜sbprolog/xsb-page.html`.

[30] P. Zimmermann, *The Official PGP User's Guide*, MIT Press, Cambridge, 1995.