A Nonmonotonic Delegation Logic with Prioritized Conflict Handling

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

Benjamin N. Grosof
IBM T.J. Watson Research Center
P.O.Box 704
Yorktown Heights, NY 10598
grosof@us.ibm.com
http://www.research.ibm.com/people/g/grosof

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

Abstract

We extend previous work on Delegation Logic (DL) [11, 12], a tractable and practically implementable logic-based language for authorization in large-scale, open, distributed systems. We expressively generalize the previous version of DL (called D1LP) to have nonmonotonic expressive features, including negation-as-failure, classical negation, and prioritized conflict handling. The resulting formalism is called D2LP.

We discuss the motivations and usefulness of prioritized conflict handling and some subtleties and challenges in extending DL to have it. Partly because of these subtleties, in this paper we restrict D2LP by prohibiting queries about delegation statements. Our technical approach to defining D2LP is based on tractably compiling a D2LP into a Generalized Courteous LP (GCLP) [7, 8], which is in turn tractably compiled into an Ordinary LP (OLP). We show that D2LP is thus tractable and practically implementable on top of existing technologies for OLP, *e.g.*, Prolog, SQL databases, and other rule-based systems.

1 Introduction

We address the problem of authorization in large-scale, open, distributed systems. Authorization decisions are needed in electronic commerce, mobile-code execution, remote resource sharing, content advising, privacy protection, and other applications. In distributed authorization, often an authorizer does not know the requester directly; therefore, it needs to use information from third parties who know the requester better; normally, the authorizer trusts these third parties only for certain things and to a certain degree. This multi-agent aspect makes distributed authorization different from traditional access control. We adopt the "trust-management" view of distributed authorization [3]: 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 the credentials prove that the request complies with its local policies.

"Delegation Logic" (DL) [11, 12] is a logic-based approach to representing policies, credentials, and requests in distributed authorization. DL extends logic programming languages with a delegation construct that features delegation depth and a wide variety of complex principals (including but not limited to k-out-of-n thresholds). Delegation allows a party to use other parties' information in a controllable way.

In [11, 12], D1LP, a monotonic version of DL, is defined. D1LP is based on Datalog Definite LP.[1] In this paper, we expressively generalize D1LP to have nonmonotonic expressive features, including negation-as-failure, classical negation, and prioritized conflict handling. The resulting generalization is called D2LP.

Many security policies are (logically) nonmonotonic or at least are more easily specified in a nonmonotonic formalism. In many applications, a natural policy is to make a decision in one direction, *e.g.*, in favor of authorizing H, if there is no information/evidence to the contrary, *e.g.*, no known revocation. Using *negation-as-failure* (a.k.a. *default negation* or *weak negation*) is often an easy and intuitive way to do this. Also useful in representation of many policies is *classical negation* (a.k.a. *explicit negation* or *strong negation*), which allows policies that explicitly forbid something. As argued in [9, 10], this allows more flexible security policies. Introducing classical negation leads to the potential for *conflict.* Conflict handling mechanisms are thus needed.

Nonmonotonic reasoning has been extensively studied. Existing work on logic-based language for authorization often uses formalisms and results from nonmonotonic reasoning, *e.g.*, [1, 9, 10, 19]. The main difference between D2LP and this previous work is that D2LP has a delegation construct that deals with the multi-agent aspect of distributed authorization. In defining D2LP, we have to deal with the interaction between delegation and nonmonotonic expressive features. This interaction results in some subtleties. Partly because of these subtleties, in this paper we restrict D2LP by prohibiting queries about delegation statements. D2LP is also different from the languages in [1, 9, 10, 19] in that it has prioritized conflict handling. This is especially useful in resolving conflicting advice from different, but apparently both trustworthy, sources.

Our technical approach to defining D2LP is based on tractably compiling a D2LP into a Generalized Courteous LP (GCLP) [7, 8], which is in turn tractably compiled into an Ordinary LP (OLP).[2] We show that D2LP is thus tractable and practically implementable on top of existing technologies for OLP, *e.g.*, Prolog, SQL databases, and other rule-based systems.

GCLP is based on Courteous LP (CLP) [5, 6]. CLP features negation-as-failure, classical negation, and prioritized conflict handling. In CLP, each rule can optionally have a label; conflicts between rules are resolved by priority relationships among labels defined through a reserved predicate *overrides*. GCLP extends CLP to have mutual exclusion constraints (mutex's) and (prioritized) reasoning of the predicate *overrides*. In this paper, we further generalize GCLP from the version in [7, 8] to allow rule-labels to be terms rather than constants.

The rest of this paper is organized as follows. In section 2, we discuss the motivations and usefulness of prioritized conflict handling and give the syntax and semantics of GCLP. We argue that GCLP is a useful language for expressing authorization policies. In section 3, we discuss some subtleties in extending DL to have prioritized conflict handling, define the syntax and semantics of D2LP, and show that it is tractable. We conclude in section 4.

[1]Datalog means all functions have zero arities, *i.e.*, they are constants. Definite means negation-free.

[2]OLP is also known as "normal" (or, sometimes, "general") LP; it is definite LP plus negation-as-failure.

2 GCLP

Recently, there has been a lot of interest in language-based approach to security policies, *e.g.*, [1, 9, 10, 14, 19]. The goal is to provide a unified framework that can support multiple access control policies and achieve separation of policies from mechanisms. Most work uses Logic Programming (LP) languages; some other work [18, 20] uses languages that can be easily translated to LP languages.

Many policy languages have negative authorizations, *i.e.*, policies that explicitly forbid something. When both positive and negative authorizations can be specified, conflicts may arise. Existing authorization languages deal with conflict handling in one or more of the following ways: (1) Do not resolve conflicts; only define semantics for conflict-free policy programs, *e.g.*, [19]. (2) Use totally ordered rules to resolve conflicts, *e.g.*, [20]. (3) Define a fixed conflict resolving policy based on relative authority and/or specificity, *e.g.*, [1, 2]. (4) Add a paraconsistent layer, and use negation-as-failure to resolve conflicts. For example, in [9, 10], one can write "do(file2, s, +a) $\leftarrow$ dercando(file2, s, +a) & $\neg$dercando(file2, s, -a)." This specifies that this particular negative authorization wins over a positive one. However, one can only specify the relative priority between a pair of mutually-conflicting conclusions, not between the rules used to derive these conclusions.

We argue that these approaches are undesirably limited. In many cases, there isn't a meaningful total ordering. Relative authority and specificity are important sources of conflict-resolving information, but they are not the only sources. Other sources may also be useful, *e.g.*, recency and relative importance of rules even from a single source. GCLP has a conflict-resolving mechanism that can be used to flexibly specify conflict-resolving policies.

2.1 Syntax of GCLP

In GCLP, a *classical literal* takes the form *at* or $\neg\, at$, where *at* is an atom. A *literal* takes the form L or $\sim L$, where $\sim$ stands for negation-as-failure and L is a classical literal. A GCLP rule takes the form "$\langle lab \rangle\ L_0 \leftarrow BodyFormula$."
Here, *lab* is a term; L_0 is a classical literal; and *BodyFormula* is a formula built from literals using "," (conjunction) and ";" (disjunction). "*lab*," "L_0," and "*BodyFormula*" are called the *label*, the *head*, and the *body*, respectively, of this rule. The label of a rule can be empty, so can the body of a rule. A variable starts with a question mark. All variables in a rule's label must also appear either in the rule's head or in its body. We say that a rule is *label-range-restricted* if all variables in a rule's label also appear in the rule's head.

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

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

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

"L_1, L_2" is called the focus of the mutex and $BodyFormula$ is called the body. When $BodyFormula$ is empty, the symbol "|" is omitted. Intuitively, this mutex specifies that L_1 and L_2 conflict with each other if $BodyFormula$ is true. An atom *at* always conflicts with $\neg\, at$; this conflict does not need to be specified explicitly. Note that mutex's do not have labels, because labels are only used to resolve conflicts and nothing conflicts with a mutex. We introduce mutex's because some conflicts can not be conveniently represented using classical negations. Examples include classification of users into several mutually disjoint groups or roles and choices among some mutual exclusive actions.

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

2.2 An example of GCLP

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

Example 2.1 Consider the Database authorization model in [2]. An authorization may be specified for a single user or a group. A group may contain users and other groups as members but may not contain itself as a member, directly or indirectly. A user u has all the authorizations specified for u and all the groups that u belongs to, directly or indirectly.

Authorizations can be either positive or negative and either strong or weak. Conflicts between positive authorizations and negative ones are resolved as follows. Strong authorizations always override conflicting weak authorizations. A conflict between two strong authorizations can not be resolved; policies having such conflicts are inconsistent. In a conflict between two weak authorizations, the more specific authorization wins; if neither is more specific, the conflict can not be resolved. For example, if Alice is a member of the Scientist group, which is a member of the Researcher group, which is in turn a member of the Employee group, then authorization for the Researcher group should take precedence over a conflicting authorization for the Employee group.

This policy can be easily represented in GCLP. Giving the group *Researcher* the weak authorization to do select on the table $T5$ can be represented as the rule:

⟨auth(weak,Researcher)⟩ authorizes(?A, sel, T5) ← user(?A),member(?A, Researcher).

The conflict-resolving policies can be represented as follows:

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

2.3 Semantics of GCLP

The intuitive meaning of GCLP's conflict handling mechanism is as follows. When a rule's body is true, we say that this rule is ready and that it is a candidate for its head. A candidate R for a classical literal L_1 is refuted if there is a candidate Q for a literal L_2 such that L_2 conflicts with L_1 and $overrides(Q's\ label, R's\ label)$ is true. A literal L is true if and only if there is an unrefuted candidate for it and there is no unrefuted candidate for anything that conflicts with L. This semantics never concludes both L and something that conflicts with L. For example, when there are unrefuted candidates for both p and $\neg p$, neither p nor $\neg p$ is concluded; they "skeptically defeat" each other. This is a skeptical semantics. We can choose a paraconsistent semantics if so desired, by dropping the requirement that L is true only when there is no unrefuted candidate for anything that conflicts with L. This might be desirable for some applications.

GCLP's semantics is formally defined by the following transformation to Ordinary LP. Given a GCLP program $\mathcal{P}$, we compile it into an OLP $\mathcal{O}$ through the following steps.

1. For each predicate $pred/z$ in $\mathcal{P}$ (z is the arity of the predicate $pred$, *i.e.*, the number of arguments it takes), introduce a new predicate n_pred/z to represent $pred$'s classical negation, and add the mutex "$\bot \leftarrow pred(?x_1, \ldots, ?x_z), n\_pred(?x_1, \ldots, ?x_z)$." Then, in $\mathcal{P}$, replace each literal $\neg pred(t_1, \ldots, t_z)$ with $n\_pred(t_1, \ldots, t_z)$. Denote the result of the transformation $\mathcal{P}'$. Let $\mathcal{P}_m$ be the set of all mutex's in $\mathcal{P}'$; let $\mathcal{P}_1$ be the set of rules in $\mathcal{P}'$; and let $\mathcal{O}$ be an empty set.

2. For each predicate $opred/z$ in $\mathcal{P}_1$ (including the new predicates introduced for classical negation), introduce two new predicates: $opred^u/z$ and $opred^s/z$. For any literal $L = opred(t_1, \ldots, t_z)$ in $\mathcal{P}_1$, define L^u to be $opred^u(t_1, \ldots, t_z)$, which is true when there is an unrefuted candidate for L, and define L^s to be $opred^s(t_1, \ldots, t_z)$, which is true when L is skeptically defeated, *i.e.*, when there is an unrefuted candidate for some literal that conflicts with L.

3. For each mutex μ in $\mathcal{P}_m$, let μ be "$\bot \leftarrow L_1, L_2 \mid body_\mu$." Define $ready_\mu$ to be the atom $ready\_pred_\mu(?x_1, \ldots, ?x_w)$, in which $ready\_pred_\mu$ is a new predicate and "$?x_1, \ldots, ?x_w$" are the variables in "L_1, L_2." Then add to $\mathcal{O}$ the rule "$ready_\mu \leftarrow body_\mu$."

   For each rule R in $\mathcal{P}_1$, let R be "$\langle lab_R \rangle \;\; head_R \leftarrow body_R$." Define $ready_R$ to be the atom $ready\_pred_R(?x_1, \ldots, ?x_w)$ and define $refuted_R$ to be the atom $refuted\_pred_R(?x_1, \ldots, ?x_w)$, in which $ready\_pred_R$ and $refuted\_pred_R$ are new predicates and "$?x_1, \ldots, ?x_w$" are the variables in "$\langle lab_R \rangle \;\; head_R$." Then add the following three rules to $\mathcal{O}$.

   $$ready_R \leftarrow body_R. \quad head^u_R \leftarrow ready_R, \sim refuted_R. \quad head_R \leftarrow head^u_R, \sim head^s_R.$$

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

4. For each rule R in $\mathcal{P}_1$, let R be "$\langle lab_R \rangle \;\; head_R \leftarrow body_R$." Then for each rule Q in $\mathcal{P}_1$, let Q be "$\langle lab_Q \rangle \;\; head_Q \leftarrow body_Q$." Without loss of generality, assume that the variables in Q do not appear in R; one can always rename variables when necessary. Then for each mutex μ in $\mathcal{P}^m$, let μ be "$\bot \leftarrow L_1, L_2 \mid body_\mu$." Again, assume that the variables in μ do not appear in R or Q. Let θ be the most general unifier (mgu) of "$(head_R, head_Q)$" and "(L_1, L_2)." If θ exists, add the following two rules to $\mathcal{O}$.

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

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

   The first rule means that R is refuted if rule Q's head conflicts with R's head, Q is ready, and Q's label overrides R's label. The second rule means that R is skeptically defeated if Q's head conflicts with R's head and Q's head has an unrefuted candidate.

The resulting program $\mathcal{O}$ is an OLP and it has size $|\mathcal{O}| = O(|\mathcal{P}|^3)$, because the transformation does a three-level loop. By "size" of a program, we mean the number of symbols, *i.e.*, variables, constants, predicate symbols, logical operators, *etc.*

The semantics of a GCLP is defined by the semantics of its corresponding OLP. There are two leading semantics for OLP: well-founded semantics (WFS) [4] and stable model semantics. Some OLP programs do not have a stable model and some programs have more than one. Furthermore, even for a propositional program, determining whether it has a stable model is NP-complete [15].

On the other hand, WFS assigns a unique three-valued model to every program. For finite ground programs, the complexity of compute the well-founded model is worst-case quadratic in the size of the program. Therefore, we choose to use WFS.

The GCLP model of $\mathcal{P}$ is computed as follows. First compute $\mathcal{O}$, then compute the WFS model of $\mathcal{O}$, and, finally, translate the conclusions in $\mathcal{O}$'s WFS model back into GCLP by discarding the new predicates introduced in steps 2-4 and translating $n\_pred(t_1, \ldots, t_z)$ to $\neg pred(t_1, \ldots, t_z)$. One can also compile GCLP queries into OLP queries and evaluate them in OLP. To compile a GCLP query into OLP, one only needs to replace $\neg pred$ with n_pred. In OLP reasoning, one can detect whether there is a conflict about a given literal lit by checking whether both lit^u and lit^s are true.

2.4 Complexity results

Theorem 1 *The transformation from a GCLP $\mathcal{P}$ to the corresponding OLP takes time $O(N^3)$, and it generates an output program of size $O(N^3)$. Here, $N = |P|$ is the size of $\mathcal{P}$.*

Sketch of proof. The $O(N^3)$ size bound follows from the definition of the transformation. The definition corresponds straightforwardly to an algorithm linear in the output size. Note that there are linear-time algorithms for unification (see [13]). ∎

The $O(N^3)$ worst-case size is reached when there are $O(N)$ mutex's and almost all pairs of rules are potentially in conflict. This is highly unlikely in practice.

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

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

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

We now show that the transformation maintains the variable bound. Each new rule added in step 3 has at most v variables, because it only has variables from one rule in $\mathcal{P}$. Now consider the two rules added in step 4: "$refuted_R\theta \leftarrow ready_\mu\theta,\ readyQ\theta,\ overrides(lab_Q, lab_R)\theta$." and "$head^s_R\theta \leftarrow ready_\mu\theta,\ head^u_Q\theta$." Because each rule in $\mathcal{P}$ is label-range-restricted, the only variables in the two new rules before applying θ are the variables in "$(head_R, head_Q)$" and "(L_1, L_2)." Note that, under the Datalog restriction, when θ unifies two terms t_1 and t_2, $nv((t_1\theta, t_2\theta)) \leq \min(nv(t_1), nv(t_2))$, in which $nv(t)$ is the number of variables in t. Similarly, when θ unifies t_1, t_2, and t_3, $nv((t_1\theta, t_2\theta, t_3\theta)) \leq \min(nv(t_1), nv(t_2), nv(t_3))$.

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

Inferencing for a Datalog OLP that has variable-per-rule bound v takes time $O(N^{2(1+v)})$, and so the worst-case GCLP inferencing complexity is equivalent to adding two variables per rule. This is not so bad, because, logic programs with high variable bounds are often used in practice. Although they have high theoretical worst-case complexity; the practical running times of many programs are often acceptable.

3 D2LP: A Nonmonotonic Delegation Logic

In this section, we extend Delegation Logic to have the nonmonotonic expressive features in GCLP. We call this nonmonotonic version of Delegation Logic D2LP.

3.1 Subtleties of integrating delegation and nonmonotonicity

Delegation is an important concept in many distributed authentication and authorization systems. In Delegation Logic, each delegation has a depth, which is either a positive integer or the symbol "$*$" (infinite depth). DL interprets a depth-1 delegation "Alice delegates p^1 to Bob" as "Alice says p if Bob says p." A depth-2 delegation "Alice delegates p^2 to Bob" implies a depth-1 delegation and also "Alice delegates p^1 to ?X if Bob delegates p^1 to ?X." In D1LP, the delegation relation can also be queried. That is, D1LP's semantics answers both "who says what?" and "who delegates to whom?". Answering delegation queries and simultaneously resolving conflicts is subtle. Consider the following example:
Example 3.1

⟨A1⟩ Alice delegates p^2 to Bob. ⟨B1⟩ Bob delegates p^1 to Carl.
⟨B2⟩ Bob says !p. ⟨B3⟩ Bob says overrides(B2, B1).
⟨C1⟩ Carl says p.

Should one conclude "Alice delegates p^1 to Carl"? By chaining the delegation ⟨A1⟩ and ⟨B1⟩ , it is true. Then, because of the fact ⟨C1⟩ , one should also conclude "Alice says p." However, this is counter-intuitive. Intuitively, the conclusion p propagates from Carl through Bob to Alice. However, by ⟨B2⟩ and ⟨B3⟩ , the conclusion p is blocked at Bob; therefore it should not reach Alice.

3.2 The Delegation-Query-Free Restriction

Because of the subtlety discussed in the previous section, in this paper, we restrict D2LP by prohibiting delegation statements from appearing in queries or rule-bodies (since rule-bodies are queries by nature). We call this the "delegation-query-free (DQF) restriction." How to answer delegation queries with conflict resolution is a topic for further research. We expect it to have a much more complex syntax and semantics than the one we present here.

The DQF restriction is stronger than the conjunctive-delegatee-query restriction that ensures that D1LP is tractable [12]. There, delegation statements in queries and rule-bodies are required to have a single principal or a conjunction of principals as delegatee.

Under the DQF restriction, only "who says what?" can be answered, not "who delegates to whom?" Despite this restriction, D2LP still has significant expressive power. One way to use D2LP is to use direct statements to represent attributes of principals; for example, groups, roles, and authorizations can all be viewed as attributes. Using D2LP, a principal can bind attributes to principals, delegate to other principals the authority to bind attributes to principals, and reason about attributes of principals.

Example 3.2: We now give an example of D1LP. In this program, Alice authorizes anyone she believes to have good credit to do transactions and delegates the right to determine who has good credit to credit bureaus and allows them to further delegate one more step.

Alice says authorizes(?P, transaction) if Alice says credit(?P, good).
Alice delegates credit(?P, good)ˆ2 to ?B if Alice says creditBureau(?B).

3.3 Syntax of D2LP

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

2. A *direct statement* takes the form "X `says` *lit*." A *delegation statement* takes the form "X `delegates` *lit*ˆ*d* `to` XS." A *speaks_for statement* takes the form "Y `speaks_for` X `on` *lit*." Here X and Y are *principal terms* and X is called the *issuer* of this statement. A principal term is either a principal or a variable. *lit* is a base literal. XS is a *complex principal term*, *i.e.*, either a *principal structure* or a variable; it is called the delegatee of the delegation statement. Principal structures are constructed from principals and threshold structures using ","(conjunction) and ";"(disjunction). See [12] for definition and discussion of threshold structures and discussion of delegations and speaks_for statements.

   A *mutex statement* takes the form "X `says` *lit*1 `opposes` *lit*2." Here X is a principal term and is called the issuer of this statement; *lit*1 and *lit*2 are base literals. Intuitively, this statement means that, in X's view, *lit*1 and *lit*2 conflict with each other, *i.e.*, "X says *lit*1" and "X says *lit*2" are mutually exclusive.

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

4. A rule takes the form: $\langle lab \rangle$ *head* `if` *body*.
   Here *head* is a direct statement, a delegation statement, a speaks_for statement, or a mutex statement, and *body* is a formula constructed from body statements using ","(conjunction) and ";"(disjunction). Note that delegation statements, speaks_for statements, and mutex statements are not allowed to appear in rule-bodies. The rule labels are used to resolve conflicts between direct statements derived from rules.

Example 3.3 Continuing example 3.2, Alice also delegates the ability to determine that someone has bad credit to any fraud expert, and bad-credit information overrides good-credit information. Finally there is an expert Bob whom Alice fully trusts.

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

⟨trusted⟩ Alice delegates credit(?P, ?Status) to Bob.
⟨good⟩ Alice delegates credit(?P, good)^2 to ?X if Alice says creditBureau(?X).
⟨bad⟩ Alice delegates credit(?P, bad)^1 to ?X if Alice says fraudExpert(?X).
Alice says credit(?P, good) opposes credit(?P, bad).
Alice says overrides(bad, good).
Alice says overrides(trusted, good).
Alice says overrides(trusted, bad).

3.4 Semantics of D2LP

D2LP's semantics is define by transforming a D2LP $\mathcal{P}$ into a GCLP $\mathcal{G}$. This transformation is, for the most part, similar to the transformation from D1LP into OLP in [12].

In addition to the reserved predicate *overrides*, $\mathcal{G}$ has one more predicate: *holds*. It is used to represent direct statements that appear in $\mathcal{P}$ or are derived in the inference process. An atom of *holds* takes the form: $holds(X, lt, len)$, where X is a principal term (a principal or a principal variable), lt is a term that represents a base literal, and $len \in [1..*]$. For any integer d, we have $d < *$ and $[d..*] = [d..D] \cup \{*\}$, where D is the largest integer delegation depth used in $\mathcal{P}$. For $d_1, d_2 \in [0..*]$, we define

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

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

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.

In the transformation, we need to use the function *PSFormula* defined in [12] to expand statements with complex principal structures as issuers. For example,
the function call $PSFormula((A, (B; C)), holds(p(a), *)$
returns $(holds(A, p(a), *), (holds(B, p(a), *); holds(C, p(a), *)))$.

Transformation 0: Label and Negation transformation (D2LP specific)
This transformation changes rules in $\mathcal{P}$; the result is called $\mathcal{P}_0$.

- For each rule R in $\mathcal{P}$, let R be "$\langle labc(t_1, \ldots, t_n)\rangle$ $head_R$ `if` $body_R$"; change its label to "$labc(X, t_1, \ldots, t_n)$," where X is the issuer of $head_R$.
- In $\mathcal{P}$, replace each base literal $\neg\, pred(\ldots)$ with $nd\_pred(\ldots)$.

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

- Replace each body direct statement "XS `says` lt" with "$PSFormula(XS, holds(lt, *))$."
- (D2LP specific) Replace each negation-as-failure statement "$\sim XS$ `says` lt" with the "De-Morganization" of "$\sim holds(XS, lt, *)$," *i.e.*, the negation $\sim$ is pushed inside conjunctions and disjunctions.

Transformation II: Head transformation

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

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

Case one:(D2LP specific) $head_R$ is a mutex statement "X `says` $lt1$ `opposes` $lt2$."

- Remove R; and for each $len1, len2 \in [1..*]$, add the rule:
  $\bot \leftarrow holds(X, lt1, len1), holds(X, lt2, len2) \mid body_R$.

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

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

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

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

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

- **Delegation expansion:** Remove R and for each $len \in [1..d]$, add the rule:
  $\langle lab_R \rangle$ $holds(A, lt, len \oplus ll) \leftarrow body_R,\ PSFormula(BS, holds(lt, len))$.

For both Case two and Case three, also do the following.

- For each $len \in [1..D]$, add the rule
  $holds(X, lt, len \oplus 1) \leftarrow holds(X, lt, len)$.
- For $len1, len2 \in [1..*]$, add the mutex: (D2LP specific)
  $\bot \leftarrow holds(A, lt, len1), holds(A, \overline{lt}, len2)$.
- If lt is "$overrides(labc_1(t_{11}, \ldots, t_{1u}), labc_2(t_{21}, \ldots, t_{2v}))$." Add the rule: (D2LP specific)
  $overrides(labc_1(X, t_{11}, \ldots, t_{1u}), labc_2(X, t_{21}, \ldots, t_{2v}))$
  `if` $holds(X, overrides(labc_1(t_{11}, \ldots, t_{1u}), labc_2(t_{21}, \ldots, t_{2v})), *)$.

Threshold structures are handled similarly to that in [12]. Here, we omit the details.

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

An important property of this transformation is that it does not introduce any new variables; more precisely: for each rule in $\mathcal{G}$, all the variables in it come from one rule in $\mathcal{P}$.

This transformation introduces new logical function symbols ("new" in the sense that they did not appear in the original D2LP). In particular, even if the D2LP is Datalog, the generated GCLP $\mathcal{G}$ is non-Datalog. For each predicate *pred* in $\mathcal{P}$, $\mathcal{G}$ has one or two corresponding function symbols (*pred* and *nd_pred*). The transformation also introduces several pre-defined functions for threshold structures. Thus, we cannot directly use the tractability results for Datalog GCLP in section 2.4. The same problem exists in the transformation from D1LP to definite OLP; in [12], this problem is dealt with by generating a typed OLP $\mathcal{O}$. Essentially, we wish to ensure that, for each variable in $\mathcal{O}$, there are $O(N)$ ground terms that can instantiate it. Because variables in $\mathcal{G}$ 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. Typing ensures this restriction of the instantiation. Here, we use the same technique. The transformation generates a typed GCLP. The transformation from GCLP to OLP simply passes through the typing, to the generated OLP as well. For more discussions of types, see appendix A.

3.5 Inferencing and Complexity Results

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

Sketch of proof. By the counting argument in [12], the function $PSFormula$ has an $O(N^2)$ growth factor. The delegation expansion step generates the largest output among all steps. It generates $O(D)$ rules, and each one uses $PSFormula$. Therefore, this step has an $O(N^2D)$ growth factor. Thus the output program has size $O(N^3D)$. ∎

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

Example 3.3 (continued) We now add the following facts to example 3.3.

Alice says creditBureau(cb1).	Alice says fraudExpert(Carl).
Bob says credit(John, good).	cb1 says credit(Jack, good).
Carl says credit(John, bad).	Carl says credit(Jack, bad).

One can conclude "Alice says credit(John, good)" and "Alice says credit(Jack, bad)."

Recall that we say that a LP is VBLD(v) if it is Datalog, each rule in it has at most v variables, and for each rule the variables in its label also appears in the head. In practice, the bound v for most programs is a small constant. We also expect that D will usually be much smaller than N.

Theorem 4 *Inferencing of a D2LP $\mathcal{P}$ that is VBLD(v) takes time polynomial in $(ND)^v$. When v is a constant and $D = O(N)$, inferencing of a D2LP takes time polynomial in N.*

Sketch of proof. Given a D2LP $\mathcal{P}$ that is VBLD(v), the output GCLP program $\mathcal{G}$ has size $O(N^3D)$ and G is VBL(v). Typing ensures that variables in $\mathcal{G}$ will only be instantiated to constants in $\mathcal{P}$ (note that $\mathcal{P}$ is Datalog). When we compile $\mathcal{G}$ into an OLP $\mathcal{O}$, $|\mathcal{O}| = O((N^3D)^3)$ and $\mathcal{O}$ has the same variable bound. Instantiating $\mathcal{O}$ increases its size by $O(N^v)$, and inferencing of $\mathcal{O}$ takes time quadratic in the size of instantiated $\mathcal{O}$. Therefore, the inferencing time is polynomial in $(ND)^v$. [4] ∎

4 Discussion and Conclusions

We defined the syntax and semantics of a nonmonotonic version of Delegation Logic: D2LP. D2LP extends D1LP, the earlier version of Delegation Logic given in [11, 12], so as to equip it with negation-as-failure (as in OLP) and prioritized conflict handling, in a manner similar to Generalized Courteous Logic Programs (GCLP) [7, 8]. As one of its features for prioritized conflict handling, D2LP includes not only classical negation, but also mutual exclusion integrity constraints (mutex's)

[4]The complexity bound in the proof is a high-degree polynomial. We can get a tighter bound by doing a more detailed analysis. One observation is that not all pairs of rules in $\mathcal{G}$ re potentially in conflict with each other. However, due to space limit and focus of this paper, we omit that analysis. We also acknowledge that the practicality of D2LP needs to be tested in experiments.

that specify the scope of conflict, *i.e.*, that flesh out which particular consistency constraints are to be enforced by the semantics. We discussed some subtleties that arise when the delegation construct in DL is combined with these nonmonotonic features. In particular, care must be taken in defining the semantics of delegation so as to make appropriate provision for handling of conflict at intermediate steps in a chain of delegation. Partly as a result of these subtleties, in this paper, we avoided the non-trivial derivation of statements *about* delegation *per se*, *i.e.*, of delegation statements, and thus prohibited queries about delegation statements.

Our technical approach to defining D2LP is based on compiling a D2LP into a Generalized Courteous LP (GCLP), which is in turn compiled into an Ordinary LP (OLP). We showed that each of these compilation steps is computationally tractable and that D2LP inferencing is thus tractable under a broad restriction similar to that which ensures tractability of OLP inferencing. This compilation approach enables D2LP 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 (*e.g.*, see [8] for discussion and review of how OLP relates to current commercially important families of rule-based systems).

A major challenge in designing a knowledge representation (KR), esp. when nonmonotonic or multi-agent, is to achieve usefully rich expressiveness and intuitively natural semantics, *together with* moderate computational complexity and relative ease of incorporation into existing software environments. We believe D2LP represents significant progress along these lines.

References

[1] E. Bertino, F. Buccafurri, E. Ferrari, and P. Rullo, "A Logical Framework for Reasoning on Data Access Control Policies," in *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press, Los Alamitos, 1999, pp. 175-189.

[2] E. Bertino, S. Jajodia, and P. Samarati, "A Flexible Authorization Mechanism for Relational Data Management Systems," *ACM Transactions on Information Systems*, 17:2 (1999), pp. 101-140. A preliminary version appeared under the title "Supporting Multiple Access Control Policies in Database Systems" in the *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1996.

[3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, "The Role of Trust Management in Distributed Systems," in *Secure Internet Programming*, LNCS vol. 1603, Springer, Berlin, 1999, pp. 185-210.

[4] A. Van Gelder, K. A. Ross, and J. S. Schlipf, "The Well-founded Semantics for Logic Programming," *Journal of the ACM*, 38 (1991), pp. 620-650.

[5] B. Grosof, "Courteous Logic Programs: Prioritized Conflict Handling for Rules," IBM Research Report RC20836, May 1997. This is an extended version of [6].

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

[7] B. Grosof, "Compiling Prioritized Default Rules Into Ordinary Logic Programs," IBM Research Report RC 21472, May 1999.

[8] 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. Extended version is IBM Research Report RC 21473, May 1999.

[9] S. Jajodia, P. Samarati, and V. S. Subrahmanian, "A Logical Language for Expressing Authorizations," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Los Alamitos, 1997, pp. 31–42.

[10] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino, "A Unified Framework for Enforcing Multiple Access Control Policies," in *Proceedings ACM SIGMOD Conference on Management of Data*, 1997.

[11] N. Li, J. Feigenbaum, and B. Grosof, "A Logic-Based Knowledge Representation for Authorization with Delegation (Extended Abstract)," in *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press, Los Alamitos, CA, 1999, pp. 162-174. Full paper available as IBM Research Report RC21492.

[12] N. Li, B. Grosof, and J. Feigenbaum, "A Practically Implementable and Tractable Delegation Logic," to appear in *Proceedings of the 21st IEEE Symposium on Security and Privacy*, May 2000, Oakland CA. Also available at `http://cs.nyu.edu/ninghui/papers/oakland00.ps`.

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

[14] E. Lupu and M. Sloman, "Conflict in Policy-based Distributed Systems Management," *IEEE Transaction on Software Engineering – Special Issue on Inconsistency Management*, to appear. A preliminary version appeared under the title "Analysis for Management Policies" in *Proceedings of the Fifth IEEE/IFIP International Symposium on Integrated Network Management*, San Diego, USA, May 1997.

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

[16] L. Naish, "Types and the Intended Meaning of Logic Programs," in [17], pp. 189–216.

[17] F. Pfenning (editor), *Types in Logic Programming*, The MIT Press, Cambridge, MA, 1992.

[18] T. Ryutov and C. Neuman, "Representation and Evaluation of Security Policies for Distributed System Services," in *Proceedings of the DISCEX*, Hilton Head Island, SC, January 2000.

[19] T. Woo and S. Lam, "Authorization in Distributed Systems: A New Approach," *Journal of Computer Security*, 2 (1993), pp. 107–136.

[20] T. Woo and S. Lam, "Designing a Distributed Authorization Service," in *Proceedings of IEEE INFOCOM '98*, 1998.

A An approach to adding types to LP Languages

In section 3, we mentioned that typing is needed to ensure tractability of the D2LP inferencing. Typing is also implicitly present in the syntax of DL. DL has principals and principal variables. The set of principals is a subset of all the constants, and a variable is said to be a principal variable if it appears in certain places. There is thus an implicit type: principal.

The addition of types to logic programs has been studied in the logic programming community [17]. It has been argued that logic programs often make implicit assumptions about types and a logic program only satisfies the intended meaning if type information is added to the program [16]. We think that typing is potentially useful in LP-based authorization languages. In authorization, there are different types of entities, *e.g.*, subjects, objects, groups, roles, *etc.* Most predicates should only take arguments of certain types.

We now briefly describe an approach to add types to LP languages. Our approach is different from other approaches presented in [17] in that we view type information as supplementary to the logic rules. One can give partial type information or even no type information at all; every program is correctly typed. The additional type information serves two purposes: to clarify the intended meaning of policy programs and to enable more efficient reasoning. This approach can be used to add types to OLP, GCLP, or other LP languages.

Intuitively, each type corresponds to a subset of all the ground terms; these subsets do not need to be disjoint. We use symbols that start with a colon for type names. A ground term has type :x if it is contained in the subset corresponding to :x. A term may have multiple types. We require that all terms constructed by one function (a function is uniquely identified by its name and arity) have the same types. So we can say a function f has types :x and :y, or equivalently, both the type :x and the type :y contain the function f. In our approach, one can also define the argument types of functions and predicates, the types of variables, and the subtype relationship between types.

A typed (LP) program $\mathcal{P}$ consists of a set of *typed rules* (called the theory of $\mathcal{P}$) and a (possibly empty) set of *type specifications*. In a typed rule, each term (including variable) can be followed by zero or more type names; they are called type definitions of the term. There are two pre-defined types in any program: ":a" (all terms) and ":c" (constants). Every term has the type :a, and every constant has the type :c in addition to :a. These two types do not need to be explicitly added to terms. The scope of a function (including a constant) is the whole program. All the type definitions for a function in one program are taken together; this function has all these types. The scope of a variable is the rule in which it appears. All the type definitions for a variable in one rule are taken together; the variable should only be instantiated to terms that have all the required types.

Type definitions in rules can only define the "return types" of a function (the types of the terms constructed by this function). The type specification part can specify the argument types of predicates and functions. For example, "$\leftarrow declare\_pred(overrides(:l, :l))$" specifies that the predicate $overrides/2$ only takes terms of type :l as arguments. This specification is equivalent to adding a type definition :l to each argument of each use of the predicate $overrides/2$. For example, with this specification, an atom $overrides(?X, good{:}d)$ is equivalent to $overrides(?X{:}l, good{:}l{:}d)$.

Similarly, one can specify the argument types for functions, *e.g.*, "$\leftarrow declare\_func(auth(, :group):l)$." This specifies that the second argument of the function $auth/2$ has to be the type :$group$, but it doesn't specify the type for the first argument. This specification also defines the function auth/2 to have the type :l. There can be at most one specification for each predicate or function. If no specification exists for a predicate or a function, each argument has the default type $:a$.

In the specification part, one can also specify the ordering of two types. The declaration ":- declare_incl(:x, :y)" means that all the terms defined to have the type "$:y$" automatically have the type "$:x$."

The semantics of such a typed OLP $\mathcal{P}$ is defined as follows. First, find all the functions in $\mathcal{P}$; determine the functions contained in each type and the types for each variable; then determine the functions each variable can be instantiated to, this is the intersection of all the sets of functions corresponding to each type the variable has. This can be done in time N^2S^2, where $N = |\mathcal{P}|$ and S is the number of types in $\mathcal{P}$. Second, instantiate $\mathcal{P}$. To do this, we need to compute all the ground terms that can instantiate some variable. There may be infinite number of such terms; however, if the type signatures of all functions satisfy an acyclic condition, there are only finite number of such terms. Finally, we can compute the minimal model of the instantiated program.

To ensure that variables in the post-transform GCLP for a D2LP $\mathcal{P}$ will only be instantiated to terms constructed from functions and constants in $\mathcal{P}$, add the following step as the first step of the transformation from D2LP to GCLP.

Transformation S: Sort-related transformation

For each sort in $\mathcal{P}$, insert a d to its name after the colon, for example, $:a$ becomes $:da$. Change all sort names in $\mathcal{P}$ to be the new one. Also explicitly add the sort $:dc$ to all constants and $:da$ to all terms. For each sort specification of predicate *pred* in $\mathcal{P}$, replace it with sort specifications for functions *pred* and *nd_pred*.

After this transformation, each variable in $\mathcal{G}$ has the type $:da$. Therefore, it can only be instantiated to terms in $\mathcal{P}$, because only these terms have the type $:a$.