Flexible Resolution of Authorisation Conflicts in Distributed Systems*

Changyu Dong, Giovanni Russello, Naranker Dulay

Department of Computing, Imperial College London
180 Queen's Gate, London, SW7 2AZ, UK
{changyu.dong,g.russello,n.dulay}@imperial.ac.uk

Abstract. Managing security in distributed systems requires flexible and expressive authorisation models with support for conflict resolution. Models need to be hierarchical but also non-monotonic supporting both positive and negative authorisations. In this paper, we present an approach to resolve the authorisation conflicts that inevitably occur in such models, with administrator specified conflict resolution strategies (rules). Strategies can be global or applied to specific parts of a system and dynamically loaded for different applications. We use Courteous Logic Programs (CLP) for the specification and enforcement of strategies. Authorisation policies are translated into labelled rules in CLP and prioritised. The prioritisation is regulated by simple override rules specified or selected by administrators. We demonstrate the capabilities of the approach by expressing the conflict resolution strategy for a moderately complex authorisation model that organises subjects and objects hierarchically.

1 Introduction

In modern enterprise systems restricting access to sensitive data is a critical requirement. However, this usually involves a large number of objects which may have different security requirements. The complexity of managing such systems results in high administrative costs and long deployment cycles. This worsens as a system expands because the effort and time required for management becomes a burden. It is important that security management procedures are simplified and automated to reduce administrative costs [1, 2]. Policy-based management is potentially a more appropriate solution where security management includes support for the specification of authorisation policies, and the translation of these policies into information which can be used by enforcement mechanisms.

Early authorisation models were usually monotonic and supported only one type of policy. Most recent authorisation models are non-monotonic and support both positive and negative authorisation policies [3–7]. The advantage of

*This research was supported by the UK's EPSRC research grant EP/C537181/1 and forms part of CareGrid. We also acknowledge financial support in part from the EC IST EMANICS Network of Excellence (26854). The authors would like to thank the members of the Policy Research Group at Imperial College for their support and to Jorge Lobo at IBM T. J. Watson research center for his advice.

supporting both is greater flexibility, expressiveness and convenience. However, conflicts can arise when both positive and negative policies are applicable at the same time. This type of conflict is referred to as a *modality conflict* [8]. When applying a non-monotonic authorisation model in large distributed systems, modality conflicts are hard to avoid. They can arise due to many reasons, such as omissions, errors or conflicting requirements. Therefore, conflict resolution is an important practical requirement in systems that support non-monotonic authorisation.

Many conflict resolution strategies have been developed. The most primitive conflict resolution rules for authorisation policies include:

- Negative (positive) takes precedence. If a conflict arises, the result will be negative (positive).
- Most specific (general) takes precedence. When policies defined at different levels in a hierarchy conflict, the most specific (general) one wins.
- Strong and weak. Some policies are marked as strong authorisations and others are marked as weak. In case they conflict, strong policies win.

However, real application may become so complex that using only one rule may not guarantee solving the conflict (for example, when two strong policies conflict). In these cases, either the conflict ends up undecided or further resolution rules must be applied to resolve the conflict. As a consequence, more and more sophisticated conflict resolution rules are designed to fulfill the requirements posed by applications.

On the other hand, different applications usually have different need for conflict resolution. For example, negative takes precedence would be one requirement in military systems while positive takes precedence may be preferred in open systems. To make an authorisation model flexible, it is better not to fix the conflict resolution strategy at design time, but leave it to the system administrators to decide what is the appropriate strategy for a system or sub-system. This raises the questions: how to express the variety of conflict resolution requirements that exist in real-world applications and how to make it easy for administrators to express such requirements. In this paper, we show how to capture and define authorisation conflict resolution strategies using small Courteous Logic Programs (CLP) [9].

We chose CLP for three main reasons: (1) it has a clear well-defined semantics that is easy to understand; (2) the conflict handling in CLP is context-neutral, i.e. it does not depend on what the application is, or on what hierarchies are being used. This feature is vital to many policy frameworks, which must be flexible in order to serve different applications; (3) the declarative nature of logic programs makes it possible to separate the conflict resolution rules from the implementation details, and therefore makes it possible to define resolution rules as reusable meta-policies that can be published and used by different organisations with only minimal changes to their existing policy management system.

2 Courteous Logic Programs

2.1 Overview

Courteous Logic Programs (CLP) is motivated largely by extended logic programs [10] where classical negation is permitted in rule heads and bodies. Extended logic programs are more expressive than normal logic programs where the negative information is implicit. For example the statement: "Penguins do not fly" can be expressed naturally as:

$$\neg fly(X) \leftarrow penguin(X).$$

However, explicit negation may also lead to contradictions. For example, with the above statement and the following ones:

$$\begin{aligned} fly(X) &\leftarrow bird(X). \\ bird(X) &\leftarrow penguin(X). \\ penguin(tweety). & \end{aligned}$$

We can derive both $\neg fly(tweety)$ and $fly(tweety)$, which contradict each other. The design goal of CLP is to preserve the expressive power brought by explicit negation in extended logic programs while also guaranteeing a consistent and unique set of conclusions. This is done by labelling the logic rules and prioritising the labels. If two conflicting conclusions can be derived, the conclusion being derived from a rule with a higher priority label will override the one with a lower priority label. In the case that each conclusion is derived from multiple rules with different priorities, the conclusion from the rule with the highest priority wins.

We illustrate the intuition with a simple example. If we label the rule "birds fly" with a label $\langle bird \rangle$, and the rule "Penguins do not fly" with a label $\langle penguin \rangle$. Furthermore is we assume that there exists a super-penguin which can fly, and its name is Tweety, then we have the following additional rules:

$$\begin{aligned} \langle superPenguin \rangle\ fly(X) &\leftarrow superPenguin(X). \\ superPenguin(tweety). & \end{aligned}$$

Of course by knowing X is a penguin, we can draw a more precise conclusion about X than only knowing X is a bird. We would decide that $\langle penguin \rangle$ has a higher priority than $\langle bird \rangle$, and $\langle superPenguin \rangle$ has a higher priority than $\langle penguin \rangle$.

The conclusion $fly(tweety)$ can be drawn by two rules with labels $\{\langle bird \rangle, \langle superPenguin \rangle\}$ and $\neg fly(tweety)$ can be drawn by one rule with label $\{\langle penguin \rangle\}$. Although $\langle penguin \rangle$ overrides $\langle bird \rangle$, it is overriden by $\langle superPenguin \rangle$. Therefore the conclusion $fly(tweety)$ is the winner because it is supported by the rule with the highest priority.

CLP is also computationally tractable for the (acyclic) propositional case, e.g. under the Datalog restriction. The entire answer set can be computed in

$O(m^2)$time, where m is the size of the ground-instantiated program. This makes CLP more attractive especially in our case because most of the authorisation policies can be expressed in Datalog.

2.2 Syntax

A courteous logic program can be viewed as a union of two disjoint parts: the *main sub-program* and the *overrides sub-program*.

The main sub-program is defined as labelled rules. A labelled rule has the form:

$$\langle lab \rangle \; L_0 \leftarrow L_1 \wedge \ldots \wedge L_m \wedge \sim L_{m+1} \wedge \ldots \wedge \sim L_n$$

where lab is an optional label for the rule, each L_i is a literal. If a rule has a label, the label is preserved during instantiation, all the ground instances of the rule have the same label. A literal can be of the form A or $\neg A$ where A is an atom and $\neg$ is the classical negation operator. $\sim$ stands for negation-as-failure.

A special binary predicate *overrides* is used to specify prioritisation. $overrides(i, j)$ means that the label i has strictly higher priority than the label j. *overrides* is syntactically reserved and cannot appear in the rule body. Given a set Lab of all the labels in the program, *overrides* must be a strict partial order over Lab, i.e. transitive, antisymmetric and irreflexive.

The program must be acyclic and stratified which means one can restructure the program into separate parts in such a way that references from one part refer only to previously defined parts.

2.3 Semantics

The semantics of CLP is defined using the concept of an *answer set*. Let $\mathcal{C}$ be a courteous logic program, it has a unique answer set S which is defined as follows.

We use $\mathcal{C}^{instd}$ to denote the logic program that results from each rule in $\mathcal{C}$ having variables been replaced by the set of all its possible ground instantiations. Let$\rho = p_1, ..., p_m$ be a sequence of all the ground atoms of $\mathcal{C}^{instd}$ such that ρ is a reverse-direction topological sort of the atom dependency graph. Reverse means that body comes before head. We call ρ a total atom stratification of $\mathcal{C}$. For example, given the following logic program:

$$p(a) \leftarrow q(a).$$
$$p(a) \leftarrow p(b).$$

A total atom stratification is $q(a), p(b), p(a)$. There may be multiple total stratifications, but the answer set is independent of the total stratification choice.

Given a ρ such that all of the overrides atoms come before all the other atoms and let p_i be the i^{th} ground atom in ρ, the answer set is constructed iteratively:

$$S_0 = \emptyset$$
$$S_i = \bigcup_{j=1,\ldots,i} T_j, i \geq 1$$

$$S = \bigcup_i T_i$$

where $\emptyset$ is the empty set and T_i is defined as follows:

$$T_i = \{\sigma p_i | Cand_i^\sigma \neq \emptyset, \forall k \in Cand_i^{\neg\sigma}. \exists j \in Cand_i^\sigma . overrides(j,k) \in S_{i-1}\},$$
$$Cand_i^\sigma = \{j | labels(j,r), Head(r) = \sigma p_i, Body(r) \subseteq S_{i-1}\}$$

Here σ stands for a modality, either positive (+, usually omitted) or negative ($\neg$). $\neg\sigma$ means the reverse modality of σ. $label(j,r)$ means j is the label for rule r. $Head(r)$ is a ground literal in the head of the rule r, $Body(r)$ is the set of all ground literals in the body of r. The set T_i either consists of a single grounded literal, or is empty. The literal is of positive sign (p_i), or of negative sign ($\neg p_i$).

According to above definition, $\mathcal{C} \models p$ means that p is in the answer set of $\mathcal{C}$.

3 Overview

Our approach is depicted in Figure 1. When a CLP-enabled policy system is asked to decide on whether a $(subject, target, action)$ request should be permitted, it translates the authorisation policies associated with the request from the underlying authorisation model (e.g. XACML) into labelled rules in CLP (the main sub-program). The labelled rules for the request are then forwarded to the policy decision point that makes the authorisation and uses the conflicts resolution rules (the overrides sub-program) to resolve any conflicts by choosing the rules with the highest priorities. If a system supports multiple authorisation models, then a translation module is needed for each. Systems can also select the conflict resolution rules needed on an application-by-application basis.

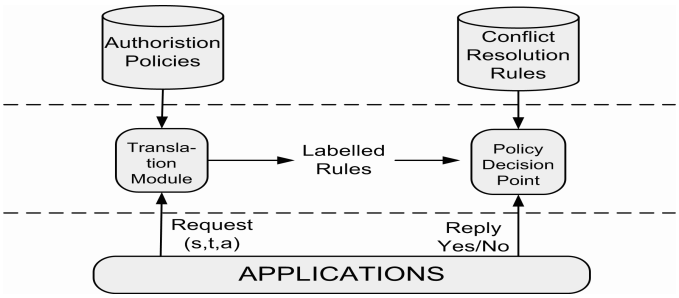

Fig. 1. Architecture of CLP-enabled policy system

4 Case Study: Hierarchical Conflict Resolution

To demonstrate the capabilities of CLP-based conflict resolution, we apply it to a moderately complex authorisation model, Ponder2 [11]. Ponder2 supports

hierarchies that are used to define objects, entities and organisational structures at different levels of scale from small embedded devices to complex services and virtual organisations. It supports both positive and negative authorisation policies. In Ponder2, managed objects (MOs) are organised in a tree-like domain hierarchy according to various criteria such as geographical boundaries, object type, responsibility and authority. Each domain in the hierarchy contains other domains or MOs. Instances of MOs are allowed to be present in more than one domain. In this way, if a domain represents a role then an instance of a MO can be associated with multiple roles. The rationale for the authorisation model and conflict resolution strategy are described in more detail in [12].

4.1 Domain Hierarchy and Authorisation Policies

Figure 2 shows a small concrete example of a hierarchically organised set of Ponder2 authorisation policies for printers in a department. In the figure, each circle represents a domain and the squares are MOs. A domain or an MO can be addressed by its domain path which is a Unix-style path from the root domain to the specified domain or MO. An MO may have multiple domain paths since it can be in more than one domain. For example, there are two domain paths for $cd04$: $/Doc/DSE/Stud$ and $/Doc/Stud/PhD$.

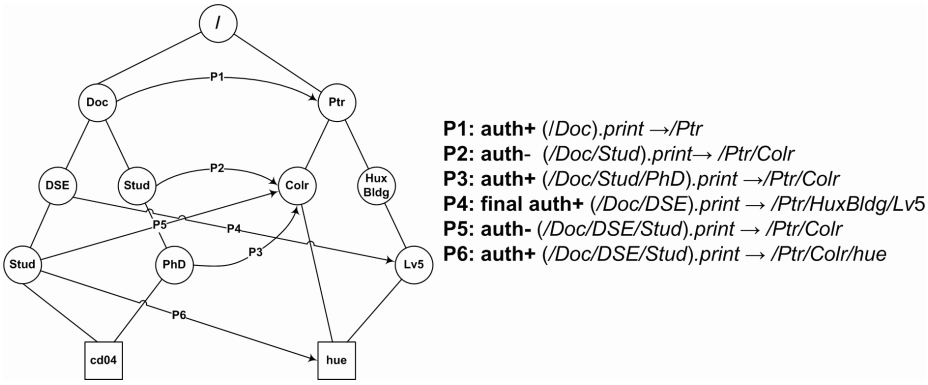

Fig. 2. Domains and Authorisation Policies for Printers in a Department

Authorisation policies define what actions a subject can(not) invoke on a target. If a domain is used as a subject or as a target, then the member MOs of the domain inherit the policies defined for the domain. For example, in the domain hierarchy shown in Figure 2, an authorisation policy $P1$ exists with subject "$/Doc$", target "$/Ptr$" and action "print", then when MO $cd04$ sends a request for action print to MO hue, the policy must be considered in the authorisation process.

The syntax of an authorisation policy is simple. **auth+** means this policy is a positive authorisation policy and **auth-** means it is negative. **final** is an

optional keyword which gives authorisation policies more priority than non-final ones. Constraints can be used to limit the applicability of policies. In the example shown in Figure 2, we can see there are six policies defined at different levels between different domains. We summarise the policies below.

$P1$ is a general policy which gives all the members in the Department of Computing (Doc) access to all the printers (Ptr). Policy $P2$ is a negative policy which prevents students (Stud) from using colour printers (Colr). However, $P3$ says that if a student is registered as a PhD student (PhD), then he can use colour printers. $P4$ is a final policy which permits all the members of the distributed software engineering (DSE) group to use all the printers located at level 5 of the Huxley Building. $P5$, $P6$ together say that the students in the DSE group cannot access colour printers other than hue.

4.2 Translating Authorisation Policies to CLP rules

When cd04 sends a print job to hue, the policies in Figure 2 are translated into the labelled rules shown in Figure 3. The translation algorithm is specific to the authorisation model. The translation module for Ponder2 evaluates all the possible domain paths. The domain paths for cd04 are: $\{/Doc/DSE/Stud/cd04$, $/Doc/Stud/PhD/cd04\}$. There are two domain paths for hue: $\{/Ptr/Colr/hue$, $/Ptr/HuxBldg/Lv5/hue\}$. For simplicity, we refer to these paths as p_{s1}, p_{s2}, p_{t1}, p_{t2} afterwards. For each combination (p_{si}, p_{tj}), two rules are added: one with a label $\langle(p)\rangle$ and the other with a label $\langle(n)\rangle$. The first 8 rules in Figure 3 apply the principle that the overall authorisation is the aggregation of the path authorisation (see section 4.3). The translation module also generates a rule for the default authorisation policy labelled $\langle(d)\rangle$.

The remaining 9 labelled rules in Figure 3 are generated for the actual policies applied along a specific path combination. For each policy, the translation module translates it into a labelled rule with head $auth(p_{si}, p_{tj}, action)$ if the policy is $auth+$, and $\neg auth(p_{si}, p_{tj}, action)$ if the policy is $auth-$. These labelled rules are used to derive the path authorisation for each path combination. The label is of the form $(type, tdis, sdis, mode)$. $type$ is determined by the policy type. If the policy is a final policy, then $type = f$. Otherwise $type = n$, which means it is a normal policy. $tdis, sdis$ are the distances regarding the path combination (p_{si}, p_{tj}). If we view the domain hierarchy as a graph and each policy defined as an arc between two nodes, then $tdis$ is the number of nodes traversed from the subject to the target through the "policy arc" and $sdis$ is the number of nodes traversed from the subject to the node where the arc starts. Finally, $mode$ is the modality of the policy. For an $auth+$ policy, $mode = p$, for an $auth-$ policy, $mode = n$.

Let's take the (p_{s1}, p_{t1}) combination as an example to show how the policies are mapped into labelled rules. The policies relevant to this combination are $P1, P5, P6$. $P1$ is a positive authorisation policy, so it is mapped into a rule with head $auth(/doc/dse/stud/cd04, /ptr/colr/hue, print)$. The label of this rule is $\langle(n, 5, 3, p)\rangle$. Recall that $P1$ is a normal positive policy defined over $(/doc, /ptr, print)$. In the label, n means a normal policy, p means positive. The

$\%auth(cd04, hue, print)$ defined for each path combination.
$\langle(p)\rangle\ auth(cd04, hue, print) \leftarrow auth(/doc/dse/stud/cd04, /ptr/colr/hue, print).$
$\langle(n)\rangle\ \neg auth(cd04, hue, print) \leftarrow \neg auth(/doc/dse/stud/cd04, /ptr/colr/hue, print).$

$\langle(p)\rangle\ auth(cd04, hue, print) \leftarrow auth(/doc/dse/stud/cd04, /ptr/huxBldg/lv5/hue).$
$\langle(n)\rangle\ \neg auth(cd04, hue, print) \leftarrow \neg auth(/doc/dse/stud/cd04, /ptr/huxBldg/lv5/hue).$

$\langle(p)\rangle\ auth(cd04, hue, print) \leftarrow auth(/doc/stud/phd/cd04, /ptr/colr/hue, print).$
$\langle(n)\rangle\ \neg auth(cd04, hue, print) \leftarrow \neg auth(/doc/stud/phd/cd04, /ptr/colr/hue, print).$

$\langle(p)\rangle\ auth(cd04, hue, print) \leftarrow auth(/doc/stud/phd/cd04, /ptr/huxBldg/lv5/hue).$
$\langle(n)\rangle\ \neg auth(cd04, hue, print) \leftarrow \neg auth(/doc/stud/phd/cd04, /ptr/huxBldg/lv5/hue).$

%default authorisation
$\langle(d)\rangle\ \neg auth(cd04, hue, print).$

%the first path combination,policies P1, P5 and P6
$\langle(n, 5, 3, p)\rangle\ auth(/doc/dse/stud/cd04, /ptr/colr/hue, print).$
$\langle(n, 2, 1, n)\rangle\ \neg auth(/doc/dse/stud/cd04, /ptr/colr/hue, print).$
$\langle(n, 1, 1, p)\rangle\ auth(/doc/dse/stud/cd04, /ptr/colr/hue, print).$

%the second path combination,policies P1, P4
$\langle(n, 5, 3, p)\rangle\ auth(/doc/dse/stud/cd04, /ptr/huxbldg/lv5/hue, print).$
$\langle(f, 3, 2, p)\rangle\ auth(/doc/dse/stud/cd04, /ptr/huxbldg/lv5/hue, print).$

%the third path combination,policies P1, P2, P3
$\langle(n, 5, 3, p)\rangle\ auth(/doc/stud/phd/cd04, /ptr/colr/hue, print).$
$\langle(n, 3, 2, p)\rangle\ \neg auth(/doc/stud/phd/cd04, /ptr/colr/hue, print).$
$\langle(n, 2, 1, p)\rangle\ auth(/doc/stud/phd/cd04, /ptr/colr/hue, print).$

%the fourth path combination,policy P1
$\langle(n, 6, 3, p)\rangle\ auth(/doc/stud/phd/cd04, /ptr/huxbldg/lv5/hue, print).$

Fig. 3. Labelled Rules for the Example

distance from cd04 to the domain $/doc$ is 3, the distance from hue to the domain $/ptr$ is 2, therefore $tdis$ is the sum, 5, and $sdis$ is 3. $P5$ is a negative authorisation policy, so the labelled rule is with a label $\langle(n, 2, 1, n)\rangle$ and head $\neg auth(/doc/dse/stud/cd04, /ptr/colr/hue, print)$. In the same way, $P6$ is translated into a labelled rule with a label $\langle(n, 1, 1, p)\rangle$.

4.3 Conflict Resolution Strategy

Although policies defined over hierarchies simplify configuration and management they also give rise to conflicts. In Ponder2, when multiple policies along the path from the subject to the target have different signs, conflict occurs.

Ponder2 resolves these types of conflict using the following rules: (1) the most specific policy takes precedence; (2) if two policies are equally specific, the negative one takes precedence. Informally, a policy that applies to a subdomain is more specific than a policy that applies to any ancestor domains. The specificity is determined by the distance from the subject to the target using the policy arc. In case 4-(a), the path from the subject to the target through $p1$ is "s, c, a, d, t" and the path from the subject to the target through $p2$ is "s, c, d, t". The second path is shorter, so $p2$ is more specific than policy $p1$. Case 4-(b) shows a not so intuitive example where both paths have the same length. In such cases, Ponder2 gives higher importance to the subject side path. Therefore, $p2$ is more specific than $p1$ because domain c is closer to s than domain a. Case 4-(c) shows an example where two paths have the same length and are defined between the same levels, i.e. they are equally specific. According to rule (2), $p1$ "wins" because it is negative.

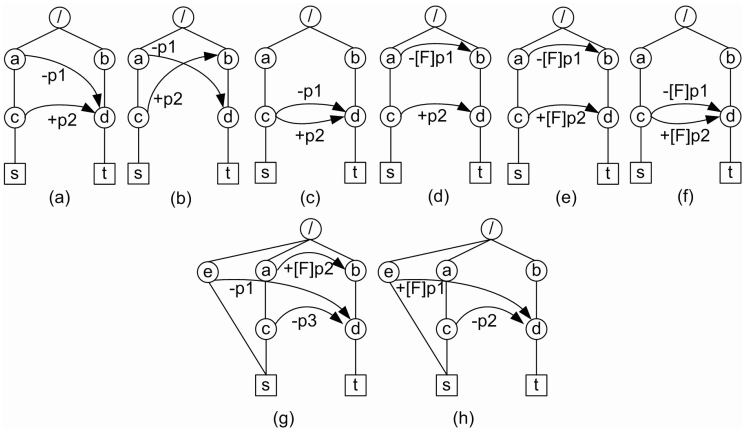

Fig. 4. Authorisation conflicts examples

Sometimes it is desirable that a general policy overrides more specific ones. When a policy is defined as a **final** policy, it has a higher priority than the normal ones. If more than one final policy exists, the conflict resolution rules for final policies are: (1) the most general final policy takes precedence; (2) if two policies are equally specific, the negative one takes precedence. In figure 4, final policies are shown with a prefix [**F**]. In case 4-(d), a final policy "wins" even though a normal policy $p2$ is more specific. In case 4-(e), the more general final policy $p1$ overrides a more specific final policy $p2$. In case 4-(f), a final policy $p1$ overrides another final policy $p2$ because they are equally general and $p1$ is negative.

When there are multiple domain paths from a MO s to MO t, the domain nesting rules cannot be applied because the policies are defined over different paths. In such situations, the path authorisation for each path combination is

derived using the above rules. The overall authorisation is then determined by aggregating the path authorisations. If one of the path authorisations is negative, then the result is negative; if all the path authorisations are positive, then the result is positive. In the examples shown in Figure 4-(g), (h), there are two path combinations: $(/e/s, /b/d/t)$ and $(/a/c/s, /b/d/t)$. In case 4-(g), the path authorisation for the first combination is negative and for the second combination is positive, therefore the overall authorisation is negative. In case 4-(h), the overall authorisation is also negative because there is a negative path authorisation. Note that the final keyword only affects conflict resolution in domain nesting cases. A final policy cannot override a normal policy when the normal policy is defined over other path combinations.

Administrators are required to define a default authorisation, either positive or negative that is applied when an authorisation request has no applicable policy.

4.4 CLP Conflict Resolution Strategy for Example

The conflict resolution rules for Ponder2 can be captured as a small overrides sub-program in CLP that is used to prioritise the labels and resolve any conflicts. Recall that we generated two kinds of labels when we translated Ponder2 policies into CLP.

For the first set of labels (p),(n),(d), we define the following overrides rules:

$$overrides((n), (p)). \tag{1}$$
$$overrides((p), (d)). \tag{2}$$
$$overrides((n), (d)). \tag{3}$$

The meaning of the above rules is that a negative path authorisation has the highest priority, then the positive ones, and finally the default authorisation which has the lowest priority.

For the set of labels of the form $(type, dis1, dis2, mod)$, we have the following overrides rules. First,

$$overrides((f, \_, \_, \_), (n, \_, \_, \_)) \tag{4}$$

states that the priority of a final policy is always higher than a normal one. Then among the final policies, the overrides rules are:

$$overrides((f, X1, \_, \_), (f, X2, \_, \_)) \leftarrow X1 > X2. \tag{5}$$
$$overrides((f, X, Y1, \_), (f, X, Y2, \_)) \leftarrow Y1 > Y2. \tag{6}$$
$$overrides((f, X, Y, n), (f, X, Y, p)). \tag{7}$$

which state that a more general final policy always has more priority, and that we always give higher priority to a negative policy if there is a "tie".

For normal policies, the rules are reversed and the more specific policy wins:

$$overrides((n, X1, \_, \_), (n, X2, \_, \_)) \leftarrow X1 < X2. \tag{8}$$
$$overrides((n, X, Y1, \_), (n, X, Y2, \_)) \leftarrow Y1 < Y2. \tag{9}$$
$$overrides((n, X, Y, n), (n, X, Y, p)). \tag{10}$$

It is easy to see that the *overrides* relations defined in the sub-program are transitive, anti-symmetric and irreflexive, and therefore meet the requirement.

4.5 Resolving Conflicts in CLP

We now explain how CLP resolves conflicts for our example. For path combination (p_{s1}, p_{t1}), i.e. $(/doc/dse/stud/cd04, /ptr/colr/hue, print)$, there are two labelled rules which permit the action. The labels of these two rules are $\{(n, 5, 3, p)$, $(n, 1, 1, p)\}$. There is also one rule which denies the action, whose label is $\{(n, 2, 1, n)\}$. It's clear that given the conflict resolution strategy in Section 4.4, the following are true:

$$overrides((n, 2, 1, n), (n, 5, 3, p)).$$
$$overrides((n, 1, 1, p), (n, 2, 1, n)).$$

Although the negative rule overrides one of the positive rules, it itself is also overridden by another positive rule. Therefore $auth(/doc/dse/stud/cd04, /ptr/colr/hue, print)$ is true because a positive rule has the highest priority. In the same way, we get the following for the other path combinations:

$$auth(/doc/dse/stud/cd04, /ptr/huxBldg/lv5/hue).$$
$$auth(/doc/stud/phd/cd04, /ptr/colr/hue, print).$$
$$auth(/doc/stud/phd/cd04, /ptr/huxBldg/lv5/hue).$$

Once the authorisation status of each path combination has been decided, the overall authorisation result can be decided. Given the above results, we can derive $auth(cd04, hue, print)$ from the CLP program and this is derived from rules labelled $\langle(p)\rangle$. Although a conflicting authorisation, $\neg auth(cd04, hue, print)$, can also be derived from the default rule, the label of the default rule, $\langle(d)\rangle$, has a lower priority than $\langle(p)\rangle$. Finally the request is authorised and cd04 can print on hue.

5 Alternative Conflict Resolution Strategies

To support different resolution strategies, we simply change the prioritisation rules (the overrides sub-program). In the example negative authorisations takes precedence. If we want to change this to positive takes precedence, we only need to modify rules (1), (7), (10) in Section 4.4 into:

$$overrides((p), (n)).$$
$$overrides((f, X, Y, p), (f, X, Y, n)).$$
$$overrides((n, X, Y, p), (n, X, Y, n)).$$

The most general final policy takes precedence can be changed to the most specific final policy takes precedence by modifying rules (5),(6) into:

$$overrides((f, X1, \_, \_), (f, X2, \_, \_)) \leftarrow X1 < X2.$$
$$overrides((f, X, Y1, \_), (f, X, Y2, \_)) \leftarrow Y1 < Y2.$$

In this case, we can even combine rules (5), (6), (8), (9) into the following two rules:

$$overrides((W, X1, \_, \_), (W, X2, \_, \_)) \leftarrow X1 < X2.$$
$$overrides((W, X, Y1, \_), (W, X, Y2, \_)) \leftarrow Y1 < Y2.$$

Ponder2 defines the most specific policy as the one with the shortest path from the subject to the target and in the case that there are several ones with the same path length, the one which is closest to the subject. We can change this to allow the one closest to the target to win. In the context of the most general final/most specific normal policy takes precedence, this can be done by modifying rules (6), (9) into:

$$overrides((f, X, Y1, \_), (f, X, Y2, \_)) \leftarrow Y1 < Y2.$$
$$overrides((n, X, Y1, \_), (n, X, Y2, \_)) \leftarrow Y1 > Y2.$$

After modification, if two policies have the same distance, the one closer to the target takes precedence. Many other variations are possible as long as the override rules define a strict partial order.

Besides assigning priorities according to the domain hierarchy and inheritance, our approach can easily support conflict resolution strategies based on other factors. For example, which administrator defined this policy, who the owner of a policy is, the date and time that a policy was enabled, etc. The translation module needs to obtain all such information when an access request is made and include the information in the labels of CLP authorisation rules. Strategies would then assign priorities according to the new labels.

6 Related Work

Early authorisation models such as SeaView [13] and the Andrew File System [14] employ a simple negative takes precedence rule to resolve conflicts. However, as the authorisation models become more complex, such rules are not enough for handling all the conflicts.

Woo and Lam [15, 7] propose an access control model for distributed systems where the management of authorisation is decentralised. The authorisation requirements are specified as policy rules using a language similar to default logic. Conflicts can be resolved by either positive takes precedence or negative takes precedence. The problem of this model is that it has no hierarchical structure for organising subjects or objects. Therefore, all the policies must be defined instance by instance, which will be burdensome in large systems.

Bertino et.al. [3] propose an authorisation model for relational data management systems. In this model, subjects are grouped in a group hierarchy and authorisation policies are classified as *strong* and *weak*. Strong policies always override weak policies and conflicts among strong policies are not allowed. The administrators must be very careful to avoid the conflicts among strong policies. The conflict resolution strategy is fixed in this model, while in our framework, the resolution strategy can be tailored to meet different applications' requirements.

Jajodia et.al. [6] proposed a flexible authorisation framework and a formal language called Authorisation Specification Language (ASL). The subjects are organised into hierarchies and the authorisation is defined as rows in an authorisation table. To define a full-fledged conflict resolution strategy, an administrator needs to define the following: a propagation policy that specifies how to derive authorisations according to the hierarchies and the authorisation table; a conflict resolution policy that specifies how to eliminate conflicts; a decision policy that decides the default authorisation in the absence of explicit specifications for the access; a set of integrity constraints that impose restrictions on the content and the output. The conflict resolution in this framework is quite flexible and powerful, but complex to understand and support. Our approach is much simpler. All the administrator needs to do is to decide an ordering over policies based on the properties captured by the labels (modality, position in the hierarchy and so on), and supply a small override program.

Benferhat et.al. [16] proposed a stratification-based approach for handling conflicts in access control. They classify the information used in access control as facts, rules without exceptions, and rules with exceptions. The information is prioritised as follows: facts and rules without exceptions are always preferred to rules with exceptions; for rules with exceptions, more specific ones are preferred to the more general ones. After information has been stratified according to the priority, conflicts can be solved by probabilistic logic inference or lexicographical inference. However, this approach requires that the facts and the rules without exceptions to be consistent and can only implement one conflict resolution strategy for the rules with exceptions.

Chadha [17] argued that many application-specific runtime policy conflicts can be addressed by re-writing policies. Rather than writing policies and defining complex resolution rules, it may be simpler to just re-write the policies. The conclusions were based on the analysis of obligation policies and conflicts such as redundant conflicts, mutually exclusive configurations or inconsistent configuration. Authorisation policies and modality conflicts, which is the focus of our work, were not discussed in detail. However, the author also concluded that "conflicts such as modality conflicts that are application-independent should be resolved automatically using conflict resolution tools", which is exactly what our approach trying to do.

7 Conclusions and Future Work

In this paper we express conflict resolution strategies in terms of Courteous Logic Programs. Authorisations are dynamically translated into labelled logic rules. The label for each rule is determined by a translation module specific to each authorisation model and can use features such as the policy type, the domain hierarchy and the policy modality. The priority of the label is defined through a special *overrides* predicate and conflicts are resolved by choosing the rule with the highest priority. The conflict resolution rules are not static and can be dynamically changed according to domain-specific requirements.

The Ponder2 policy model was used as a case study and to demonstrate a variety of conflict resolution strategies and how easy it is for administrators to define their own. An implementation of our CLP translation module and PDP is available as an option for Ponder2 and can be used instead of its Java-coded conflict resolution strategy. The implementation is based on a Prolog version of CLP engine. We hope to develop an implementation and a library of conflict resolution strategies for XACML (Extensible Access Control Markup Language) [18] in the near future, and to investigate how to combine multiple strategies.

References

1. Feridun, M., Leib, M., Nodine, M.H., c. Ong, J.: Anm: Automated network management system. IEEE Network **2**(2) (March 1988) 13–19
2. Strassner, J.: Policy-based network management, solution for the next generation. MORGAN AND KAUFMANN (2004)
3. Bertino, E., Jajodia, S., Samarati, P.: A flexible authorization mechanism for relational data management systems. ACM Trans. Inf. Syst. **17**(2) (1999) 101–140
4. Bertino, E., Samarati, P., Jajodia, S.: Authorizations in relational database management systems. In: ACM Conference on Computer and Communications Security. (1993) 130–139
5. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In: POLICY. (2001) 18–38
6. Jajodia, S., Samarati, P., Sapino, M.L., Subrahmanian, V.S.: Flexible support for multiple access control policies. ACM Trans. Database Syst. **26**(2) (2001) 214–260
7. Woo, T.Y.C., Lam, S.S.: Authorizations in distributed systems: A new approach. Journal of Computer Security **2**(2-3) (1993) 107–136
8. Lupu, E., Sloman, M.: Conflicts in policy-based distributed systems management. IEEE Trans. Software Eng. **25**(6) (1999) 852–869
9. Grosof, B.N.: Courteous logic programs: Prioritized conflict handling for rules. Research Report RC 20836(92273), IBM (1997)
10. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. **9**(3/4) (1991) 365–386
11. Ponder2: The Ponder2 project. `www.ponder2.net`
12. Russello, G., Dong, C., Dulay, N.: Authorisation and conflict resolution for hierarchical domains. In: POLICY. (2007) 201–210
13. Lunt, T.F., Denning, D.E., Schell, R.R., Heckman, M., Shockley, W.R.: The seaview security model. IEEE Trans. Software Eng. **16**(6) (1990) 593–607
14. Satyanarayanan, M.: Integrating security in a large distributed system. ACM Trans. Comput. Syst. **7**(3) (1989) 247–280
15. Woo, T.Y.C., Lam, S.S.: Authorization in distributed systems: A formal approach. In: SP '92: Proceedings of the 1992 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society (1992) 33–50
16. Benferhat, S., Baida, R.E., Cuppens, F.: A stratification-based approach for handling conflicts in access control. In: SACMAT. (2003) 189–195
17. Chadha, R.: A cautionary note about policy conflict resolution. Military Communications Conference, 2006. MILCOM 2006 (Oct. 2006) 1–8
18. XACML: Extensible access control markup language. `http://xml.coverpages.org/xacml.html`