Compliance Checking for Usage-Constrained Credentials in Trust Negotiation Systems

Jinwei Hu[1], Khaled M. Khan[2], Yun Bai[3], and Yan Zhang[3]

[1] Department of Computer Science, TU Darmstadt, Germany
hu@mais.informatik.tu-darmstadt.de
[2] Department of Computer Science and Engineering, Qatar University, Qatar
k.khan@qu.edu.qa
[3] School of Computing and Mathematics, University of Western Sydney, Australia
{ybai,yan}@scm.uws.edu.au

Abstract. We propose an approach to placing usage-constraints on RT credentials; issuers specify constraints by designing non-deterministic finite automata. We show by examples that this approach can express constraints of practical interest. We present a compliance checker in the presence of usage-constraints, especially for trust negotiation systems. Given an RT policy, the checker is able to find all minimal satisfying sets, each of which uses credentials in a way consistent with given constraints. The checker leverages answer set programming, a declarative logic programming paradigm, to model and solve the problem. We also show preliminary experimental results: supporting usage-constraints on credentials incurs affordable overheads and the checker responds efficiently.

1 Introduction

Compliance checking aims to, given a policy p and a set $\mathcal{C}$ of credentials, answer the questions of whether and how a subset of $\mathcal{C}$ satisfies p. Such a subset is called a *satisfying set* of p in $\mathcal{C}$. A credential is a cryptographic certificate from a credential issuer, who asserts attributes about a principal. For example, "University says Alice is a student" is a credential, where University is the issuer, Alice is the principal, and "is-a-student" is the attribute. A policy is a statement to be proved, e.g., "Alice can access files".

Compliance checkers can be broadly categorized into three types [9]. Type-1 checkers return no satisfying set in any case but a Boolean value indicating whether p is satisfied. Type-2 checkers return one satisfying set of p, if any. Type-3 checkers are able to find *all* minimal satisfying sets of p in $\mathcal{C}$. This feature distinguishes types-3 checkers from the other two and makes them proper checkers for trust negotiation (TN) systems [6,11]. In TN, two participants, say Alice and Bob, iteratively exchange credentials to gradually increase trust between each other. When requesting (sensitive) attributes of Alice, Bob may receive a (release) policy p from Alice, which specifies attributes that Bob should exhibit before Alice discloses her attribute. In this case, Bob uses his compliance checker to search for satisfying sets of p in his credential set. There are two main reasons why type-3 checkers are more appropriate for TN than type-1 and type-2 checkers. On the one hand, a type-3 checker ensures that TN establishes trust whenever

D. Gollmann and F.C. Freiling (Eds.): ISC 2012, LNCS 7483, pp. 290–305, 2012.

possible. In event of not establishing trust with a negotiation tactic, TN may restart negotiation with an alternative tactic, owing to the use of type-3 checkers. On the other hand, it also broadens choices of negotiation strategies. For example, one may choose to disclose the set of credentials that reveals least sensitive information.

When a compliance checker finds a satisfying set of a policy, it actually composes a proof of the policy (e.g., as defined in Section 4.1), using the credentials in the set. Credential issuers, however, may not foresee all the proofs that their credentials contribute to and all the consequences that their credentials make possible, largely because of the open nature of TN systems. This uncertainty annoys credential issuers sometimes; they may want to restrict the circumstances under which their credentials are used. For example, a bank may require its credentials to be used for limited purposes set out in agreements.[1] Therefore, credential issuers may specify a constraint stating valid ways to use its credentials. For example, the bank may attach to its credentials a constraint listing proofs allowed by the agreements. Compliance checkers (e.g., Bob's checker) ought to work out satisfying sets which prove a policy in a way consistent with constraints. A principal who uses the proof (e.g., Alice) should verify the consistency.

In this paper, we propose such a compliance checker. We first devise a mechanism for credential issuers to specify usage-constraints. A constraint is defined based on NFAs (*non-deterministic finite automaton*). We employ ASP (*answer set programming*) to encode the checking problem and ASP solvers to compute all satisfying sets.

The remainder of the paper is organized as follows. We state the assumptions made in this work and review RT (a family of role-base trust-management languages) in Section 2. We discuss related work in Section 3. In Section 4 we define usage-constraints on credentials and propose the type-3 compliance checking problem in the presence of constraints. Section 5 describes an ASP encoding of the problem, while the detailed encoding is presented in Appendix A. In Section 6 we undertake a group of experiments to evaluate the performance of our approach. Finally, we conclude in Section 7.

2 Background

Assumptions When Bob searches for proofs of Alice's policy p, we assume that Bob's credential set is fixed. That is, the compliance checker works on a static set of credentials. We refer to this set as a *credential context*.

After receiving a proof, Alice is obliged to verify any usage-constraints on credentials that are used in the proof. This is because any otherwise forbidden access that results from a proof not complying with constraints may only cause loss to Alice. For example, suppose that a companyC issues a qualification certificate to Bob but puts a constraint that the certificate be used only in companyC. Suppose further Alice accepts a proof using this certificate and therefore allows Bob to access her sensitive files. The companyC is not held responsible for this access, because it already states the restriction on the certificate. On the contrary, it is Alice who did not enforce the constraint.

[1] `http://www.nbnz.co.nz/onlineservices/directtrade/PDFs/PKIagreements.pdf`

Policy language RT [8] is a family of role-base trust-management languages. We assume that TN uses two of its sub-language, RT_0 and RT_1, to represent credentials. RT_0 credentials make assertions by defining role memberships. Memberships can be defined in four ways, corresponding four kinds of RT_0 credentials. Since an NFA reads a string from left to right in convention, we reverse the arrow direction in RT.

Type-1: $D \rightarrow A.r$, where A and D are principals and r is a role name. A issues this credential to assert that D is a member of $A.r$. $A.r$ is called a *normal role*.

Type-2: $B.r' \rightarrow A.r$, where A and B are principals and r and r' are role names. By issuing this credential, A asserts that a member of $B.r'$ is also a member of $A.r$.

Type-3: $A.r_1.r_2 \rightarrow A.r$, where A is a principal and r_1 and r_2 are role names. This credential means that a principal is a member of $A.r$ if it is a member of $B.r_2$ for any principal B who is a member of $A.r_1$. $A.r_1.r_2$ is called a *linked role*.

Type-4: $B_1.r_1 \cap \cdots \cap B_n.r_n \rightarrow A.r$, where A and B_i are principals, and r and r_i are role names. The credential asserts that a principal is a member of $A.r$ if it is a member of $B_i.r_i$ for all $i \in [1..n]$. $B_1.r_1 \cap \cdots \cap B_n.r_n$ is called an *intersection role*.

We recall some notions of RT [12]. We say a credential of the form $e \rightarrow A.r$ *defines* the role $A.r$, where e can be D, $B.r'$, $A.r_1.r_2$, or $B_1.r_1 \cap \cdots \cap B_n.r_n$. Given a set $\mathcal{C}$ of RT_0 credentials, let $Prin(\mathcal{C})$ be the set of principals in $\mathcal{C}$; define $NRole(\mathcal{C})$ as the set of normal roles $A.r$ such that A is a principal in $\mathcal{C}$ and r is a role name in $\mathcal{C}$. Let $LRole(\mathcal{C})$ be the set of linked roles in $\mathcal{C}$ and $IRole(\mathcal{C})$ be the set of intersection roles in $\mathcal{C}$. Further, let $Role(\mathcal{C}) = NRole(\mathcal{C}) \cup LRole(\mathcal{C})$ be the set of roles in $\mathcal{C}$, containing normal and linked roles; let $\mathcal{E}(\mathcal{C}) = Prin(\mathcal{C}) \cup Role(\mathcal{C}) \cup IRole(\mathcal{C})$ be the set of role expressions in $\mathcal{C}$, containing principals, roles, and intersection roles.

Definition 1. *[12] The semantics of $\mathcal{C}$ is given by the least $\twoheadrightarrow_{\mathcal{C}} \subseteq \mathcal{E}(\mathcal{C}) \times \mathcal{E}(\mathcal{C})$ such that (1) for any $e \in \mathcal{E}(\mathcal{C})$, $e \twoheadrightarrow_{\mathcal{C}} e$, (2) if $e_1 \twoheadrightarrow_{\mathcal{C}} e_2$ and $e_2 \twoheadrightarrow_{\mathcal{C}} e_3$ then $e_1 \twoheadrightarrow_{\mathcal{C}} e_3$, (3) for any credential $e \rightarrow A.r$ in $\mathcal{C}$, $e \twoheadrightarrow_{\mathcal{C}} A.r$, (4) if $A.r_1.r_2 \in \mathcal{E}(\mathcal{C})$ and $B \twoheadrightarrow_{\mathcal{C}} A.r_1$, then $B.r_2 \twoheadrightarrow_{\mathcal{C}} A.r_1.r_2$, and (5) if $B_1.r_1 \cap \cdots \cap B_n.r_n \in \mathcal{E}(\mathcal{C})$ and $e \twoheadrightarrow_{\mathcal{C}} B_i.r_i$ for $i \in [1..n]$, then $e \twoheadrightarrow_{\mathcal{C}} B_1.r_1 \cap \cdots \cap B_n.r_n$. We write $\twoheadrightarrow$ if $\mathcal{C}$ is clear from the context.*

Example 1. Suppose that a parking lot provides parking services to staff of its partners (e.g., a medical center). The lot provides special service for people with disability. We let $\mathcal{C}_{lot}$ be the credential context consisting of the following credentials.

$c_1 : Bob \rightarrow Med.staff$ $c_2 : Med \rightarrow Lot.partner$ $c_3 : Lot.partner.staff \rightarrow Lot.pk$
$c_4 : Bob \rightarrow HR.dis$ $c_5 : HR.dis \rightarrow Med.dis$ $c_6 : Med.dis \rightarrow Lot.dis$
$c_7 : Lot.pk \cap Lot.dis \rightarrow Lot.spk$

We have $Bob \twoheadrightarrow_{\mathcal{C}_{lot}} Lot.pk$ and $Bob \twoheadrightarrow_{\mathcal{C}_{lot}} Lot.spk$. □

RT_1 credentials allow the use of parameterized roles. For ease of exposition, we focus mainly on RT_0 credentials.

3 Related Work

Bauer et al. [3] present an approach to constraining credential usage for *proof-carrying authorization (PCA)*. It has a heavy influence on this work: we follow the idea to place constraints on proofs. Nevertheless, we observe some essential differences between

PCA and TN; these differences dissuade us from adapting their approach to TN. First, PCA does not search for proofs but only verifies proofs submitted by access-requester. This makes it possible to define constraints as arbitrary functions over proofs. In contrast, a compliance checker of TN ought to search for proofs by itself. It appears difficult for an efficient search algorithm to work with arbitrary functions. This consideration leads to our definition of constraints as NFA properties. It seems restrictive to define constraints as NFA properties, instead of arbitrary functions over proofs; but, as it turns out, most example constraints in [3] could be expressed. Second, compliance checkers are expected to find all proofs of a conclusion. Even though some mechanisms [2] assist principals in composing proofs in PCA, they only find one proof and thus provide inadequate help for TN. Finally, PCA policy languages are seldom used in TN.

Lee and Winslett present an efficient type-3 compliance checker CLOUSEAU [6]. CLOUSEAU compiles credentials and policies into an intermediate representation that is analyzed using efficient pattern matching algorithms. Another type-3 compliance checker SSgen is proposed by Smith et al. [11]. Given an input, SSgen iteratively invokes a type-2 checker, which generates only one satisfying set with the input; SSgen feeds the type-2 checker with modified input so that previously generated proofs are excluded and an alternative proof can be found. It remains unclear, however, how to extend existing type-3 checkers like CLOUSEAU and SSgen to deal with constraints on credential usage.

To restrict credential usage, one may attempt to employ a type-3 checker to generate satisfying sets and remove the sets that violate usage-constraints; so the remaining sets respect the constraints. Here, the constraints can be arbitrary functions over proofs, as one only needs to verify proofs against them. We take a closer look at this approach with CLOUSEAU as the checker.

First, CLOUSEAU has to supply all satisfying sets but not only the minimal ones. For a counter-example, suppose that $C_1 = \{c_1, c_2, c_3, c_4, c_5\}$ is a satisfying set and $C_2 = \{c_1, c_2\}$ a minimal one. Suppose further that a constraint requires that the use of c_2 be accompanied by the use of c_3.[2] In this case, C_1 is a set conforming to the constraint, but C_2 is not. In the worst case, the number of minimal satisfying sets for a given policy can be exponential [6,11], not to speak of the number of all satisfying sets. It is challenging to efficiently verify the sets against constraints, in spite of the efficient algorithms used in CLOUSEAU.

Second, CLOUSEAU returns credential sets, instead of proofs. From a credential set, however, more than one proof might be constructed. For example, one can write two proofs using credentials in C_1, as mentioned earlier. In this case, it is ambiguous which proof a constraint concerns with.

Finally, this approach lacks an intuitive way to capture constraints over attributes required of credential issuers [3]. For example, Alice may say that her credentials can only be used in a proof involving credentials from herself or her friends. We are not aware of any simple extensions to CLOUSEAU which are able to enforce this constraint.

[2] Suppose that the given policy is $A \twoheadrightarrow C.r$ and that the credentials are $c_1 : A \rightarrow B.r$, $c_2 : B.r \rightarrow C.r$, $c_3 : C.r.r' \rightarrow C.r$, $c_4 : D \rightarrow B.r$, and $c_5 : A \rightarrow D.r'$. One can compose two distinct proofs using credentials $\{c_1, c_2\}$ and $\{c_1, c_2, c_3, c_4, c_5\}$, respectively. Note that the second proof relies on c_3 rather than just spuriously include it.

One might consider specifying credentials in more complicated languages. For example, besides RT_0 and RT_1, the RT family includes more expressive languages such as RT^T. Similarly one might redesign policy languages so that principals can specify in more detail how their assertions and judgements are to be used. This could entail the introduction of new modalities [4] and of complex inference rules [3]. This approach is less attractive for the following reasons. First, apart from the compliance checker, it may result in changes to other TN components. Since a TN system uses its policy language to represent credentials and release policies [9], this representation might need revisions. Second, one has to design a new compliance checker that is able to efficiently find all minimal satisfying sets in the presence of the new, more complicated language. Finally, in this manner, only a set of predefined constraints can be enforced [3].

To sum up, based on these these observations, an alternative is arguably worth investigating. The challenge is to support constraints that capture practical requirements while enabling efficient compliance checking. As such, we propose a novel definition of constraints based on NFAs and employ ASP to enforce constraints.

Note that we do not concern ourselves with the information leakage and hiding problem in TN; instead, we simply focus on how to find all minimal proofs which use credentials in a way consistent with constraints.

4 Usage-Constraints on Credentials

To support usage-constraints on credentials, we follow an idea similar to the one in [3]: allow credential issuers to specify constraints on proofs where their credentials are to be used. A proof is decomposed into a set of strings. Constraints are defined based on NFAs; a proof is required to be accepted by NFAs, either partially or entirely as specified. A compliance checker is designed to return all such minimal proofs.

4.1 Defining Proofs

Suppose $e_1 \twoheadrightarrow_{\mathcal{C}} e_2$; there is a proof justifying this statement. The proof can take various forms. For example, Fig. 1 shows a proof tree of $Bob \twoheadrightarrow_{\mathcal{C}_{lot}} Lot.spk$. A node of a proof tree is a pair of principal-role, which means the principal becomes a member of the role. A proof like this hinders a simple definition of constraints. First, it could

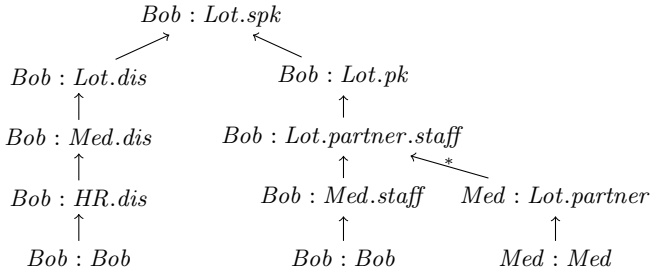

Fig. 1. A proof tree of $Lot.spk \twoheadrightarrow_{\mathcal{C}_{lot}} Bob$. The star edge denotes a linked membership derived from the outgoing node of the edge.

contain redundant reasoning. For example, the tree in Fig. 1 also serves as a proof of $Bob \twoheadrightarrow_{\mathcal{C}_{lot}} Lot.pk$; in this case, it includes superfluous applications of credentials. Second, placing constraints on trees raises the bar for credential issuers to correctly specify constraints.

A slight revision will provide a simpler solution but no less capability. In brief, we borrow an idea of model checking linear temporal logic properties [5]. We obtain a set of sequences of principal-role pairs, when a tree is traversed: a traversal starts from a leaf node and proceeds until it encounters a star edge or the root node. Consider again the tree in Fig. 1; we have the following sequences.
$[Bob : Bob, Bob : HR.dis, Bob : Med.dis, Bob : Lot.dis, Bob : Lot.spk]$
$[Bob : Bob, Bob : Med.staff, Bob : Lot.partner.staff, Bob : Lot.pk, Bob : Lot.spk]$
$[Med : Med, Med : Lot.partner]$
Observe that each sequence concerns only one principal; it could be shortened as a *role path* defined below.

Definition 2. *A* role path *in $\mathcal{C}$ is a tuple $[e_0, e_1, \cdots, e_n]$ where (1) $e_0 \in Prin(\mathcal{C})$, (2) $\{e_1, \cdots, e_n\} \subseteq Role(\mathcal{C})$, and (3) if $e_i = e_j$ then $i = j$.*

Condition 1 says that a role path begins with a principal. Condition 2 says that a principal is followed by a sequence of roles. Condition 3 says there is no cycle in a role path.[3] For example, we have the following role paths in $\mathcal{C}_{lot}$; there, h_1, h_2, and h_3 correspond to the tree in Fig. 1.
$h_1 : [Bob, HR.dis, Med.dis, Lot.dis, Lot.spk]$
$h_2 : [Bob, Med.staff, Lot.partner.staff, Lot.pk, Lot.spk]$
$h_3 : [Med, Lot.partner]$
$h_4 : [Bob, Med.staff, Lot.partner.staff, Lot.pk]$

Definition 3. *Let $\mathcal{H}$ be the set of role paths in $\mathcal{C}$. We define a relation $\vdash \subseteq 2^{\mathcal{H}} \times 2^{\mathcal{C}} \times \twoheadrightarrow$; for any $(H, C, e_1 \twoheadrightarrow e_2) \in \vdash$, we write $H \vdash_C e_1 \twoheadrightarrow e_2$. For any $H \subseteq \mathcal{H}$ and $C \subseteq \mathcal{C}$, the followings hold:*

1. *If $e \in h$ for some $h \in H$, then $H \vdash_C e \twoheadrightarrow e$.*
2. *If $H \vdash_C e_1 \twoheadrightarrow e_2$ and $H \vdash_C e_2 \twoheadrightarrow e_3$, then $H \vdash_C e_1 \twoheadrightarrow e_3$.*
3. *For any $[\cdots, e_i, e_{i+1}, \cdots] \in H$, if $e_i \rightarrow e_{i+1} \in C$ then $H \vdash_C e_i \twoheadrightarrow e_{i+1}$.*
4. *If $H \vdash_C B \twoheadrightarrow A.r_1$, then for any $[\cdots, B.r_2, A.r_1.r_2, \cdots] \in H$, $H \vdash_C B.r_2 \twoheadrightarrow A.r_1.r_2$.*
5. *For any $B_1.r_1 \cap \cdots \cap B_n.r_n \rightarrow A.r \in C$, if for $i \in [1..n]$ both $[\cdots, e, B_i.r_i, A.r, \cdots] \in H$ and $H \vdash_C e \twoheadrightarrow B_i.r_i$ hold, then $H \vdash_C e \twoheadrightarrow A.r$.*

Example 2. Continue with Example 1. Consider the sets $H_{pk} = \{h_3, h_4\}$ and $C = \{c_1, c_2, c_3\} \subset \mathcal{C}_{lot}$. One can derive $H_{pk} \vdash_C Bob \twoheadrightarrow Lot.pk$. H_{pk} shows how $Bob \twoheadrightarrow Lot.pk$ is concluded using credentials in C: From h_4, Bob first becomes a $Med.staff$; from h_3, Med is a $Lot.partner$; continuing with h_4, Bob turns into a $Lot.partner.staff$ because of Med's membership in $Lot.partner$, and finally becomes a $Lot.pk$.

If we let $H_{spk} = \{h_1, h_2, h_3\}$ and $C = \mathcal{C}_{lot}$, we have $H_{spk} \vdash_C Bob \twoheadrightarrow Lot.spk$. H_{spk} explains $Bob \twoheadrightarrow Lot.spk$ in the same way as the proof tree in Fig. 1 does. Path h_2

[3] This is not to be confused with policy cycles [7].

seems to indicate that Bob gains $Lot.spk$ simply because of his membership in $Lot.pk$, which is granted to any $Lot.partner.staff$ without the requirement of a membership in $Lot.dis$. As indicated by the fact that $\{h_2, h_3\} \vdash_C Bob \twoheadrightarrow Lot.spk$ does not hold, however, this understanding is not correct. Actually, the intuition here is that Bob's membership in $Lot.spk$ is preceded by his membership in $Lot.pk$. □

We write $H_1 \leq H_2$ if $H_1 \subseteq H_2$ or for all $h \in H_1$, there exists $h' \in H_2$ such that h is a prefix of h',[4] and $H_1 < H_2$ if $H_1 \leq H_2$ and $H_1 \neq H_2$.

Definition 4 (Proof). *We say H is a* proof *of $e_1 \twoheadrightarrow e_2$* using *$C$ if $H \vdash_C e_1 \twoheadrightarrow e_2$ and for all $H' < H$ it does not hold that $H' \vdash_C e_1 \twoheadrightarrow e_2$.*

For example, $\{h_3, h_4\}$ is a proof of $Bob \twoheadrightarrow Lot.pk$ using $\{c_1, c_2, c_3\}$.

4.2 Defining Constraints

Semantically, a constraint defines a set of allowable proofs. A proof, in turn, is a set of role paths. Further, a role path, when viewed as a string, is accepted or denied by an NFA (non-deterministic finite automaton). Hence, credential issuers could design NFAs to define allowable proofs.

Definition 5. *[10] An NFA N is a tuple $(S, \Sigma, \delta, s_0, F)$, where S is a finite set of states, Σ is a finite alphabet, $\delta : S \times (\Sigma \cup \{\epsilon\}) \mapsto 2^S$ is the transition function, $s_0 \in S$ is the start state, and $F \subseteq S$ is the set of accept states. For $\Sigma' \subseteq \Sigma$, we write $\delta(s_1, \Sigma') = s_2$ as a shorthand for the set $\{\delta(s_1, v) = s_2 \mid v \in \Sigma'\}$ for any $s_1, s_2 \in S$.*

Let w be a string over the alphabet Σ; we say N accepts *w if we can write w as $v_1 v_2 \cdots v_m$, where each v_i is a member of $\Sigma \cup \{\epsilon\}$ and a sequence of states $q_0, q_1, \cdots, q_m$ exists in S with three conditions: (1) $q_0 = s_0$, (2) $q_{i+1} \in \delta(q_i, v_{i+1})$ for $i \in [0..m-1]$, and (3) $q_m \in F$. Let $L(N)$ be the set of strings that N accepts.*

Example 3 (Final-usage constraint). Continue with Example 1. Recall that the medical center issues the credential $HR.dis \rightarrow Med.dis$ defining the role $Med.dis$. Suppose that the center confines the usage of the memberships of $Med.dis$ to limited purposes. For example, it can be used in proving entitlement to special parking service; rather, it cannot be used in any commercial promotions where, for instance, people with disability are given coupons. To this end, the center requires that each role path of a proof H be accepted by the NFA N_{lot} in Fig. 2. Consider a path $[e_0, e_1, \cdots, e_n] \in H$. N_{lot} says that e_0 must be a principal such as Bob. Next, if N_{lot} does not confront $Med.dis$ (i.e., no credential defining $Med.dis$ is used) all the way through the path, it will accept the path. Suppose otherwise that $e_i = Med.dis$ (i.e, a credential of the form $e_{i-1} \rightarrow Med.dis$ is used); N_{lot} accepts the path if it ends with $Lot.spk$. Consequently, H is a proof where credentials affirming disability by Med is only used for special parking service if and only if all role paths in H are accepted by N_{lot}. □

One may have noticed that N_{lot} works only in the context of $\mathcal{C}_{lot}$. It is unlikely for the center to specify a similar NFA for every situation where a credential defining $Med.dis$

[4] A path $[e_0, \cdots, e_n]$ is a prefix of a path $[e_0, \cdots, e_n, \cdots, e_{n+m}]$ where $m \geq 0$.

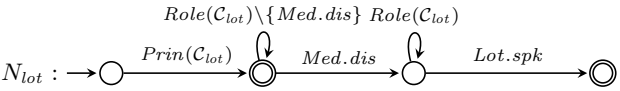

Fig. 2. An NFA N_{lot} that restricts the usage of credentials defining $Med.dis$ to special parking service in the context of $\mathcal{C}_{lot}$

may be used. Instead, the center is more willing to write a special NFA with a credential context placeholder. In event of compliance checking, as the credential context is fixed, a context like $\mathcal{C}_{lot}$ is substituted for the placeholder.

Definition 6. *A* context-dependent NFA*, denoted as γ, is an NFA $(S, \Sigma, \delta, s_0, F)$ where the alphabet Σ consists of the following:*

1. *a finite set* `Prin` *of principals and a finite set* `Role` *of roles, and*
2. *a set of special symbols $\{Prin(\underline{\mathcal{C}}), Role(\underline{\mathcal{C}})\} \cup \{Role(\underline{\mathcal{C}})\backslash A \mid A \subset \texttt{Role}\}$, where $\underline{\mathcal{C}}$ denotes the credential context placeholder.*

The set `Role` contains roles that a principal is aware of, when designing a context-dependent NFA; likewise, `Prin` contains principals. Take Example 3 for instance; the center knows the roles $Med.dis$ and $Lot.spk$, for it means to restrict the usage of credentials like $HR.dis \rightarrow Med.dis$ to the special parking service that is represented by $Lot.spk$. In this case, $\texttt{Prin} = \{Med, Lot\}$ and $\texttt{Role} = \{Med.dis, Lot.spk\}$.

A special symbol is treated as a single unit. Consider for example the NFA γ_{lot} in Fig. 3; it accepts the following string.

$$Prin(\underline{\mathcal{C}})\ Role(\underline{\mathcal{C}})\backslash\{Med.dis\}\ Med.dis\ Role(\underline{\mathcal{C}})\ Lot.spk \tag{1}$$

A special symbol turns into a set expression after a credential context $\mathcal{C}$ is substituted for the placeholder. For example, if $\mathcal{C}_{lot}$ takes place of $\underline{\mathcal{C}}$, $Role(\underline{\mathcal{C}})$ turns into $Role(\mathcal{C}_{lot})$, which is the set of roles in $\mathcal{C}_{lot}$. Supposing that v is a special symbol, we write the set expression obtained from the substitution as $v|_{\mathcal{C}}$.

Definition 7. *Given a role path $h = [e_0, \cdots, e_n]$ in a context $\mathcal{C}$, we say $(\gamma, \mathcal{C})$* allows *h if γ accepts a string $v_0 \cdots v_n$ such that for $i \in [0..n]$, either $v_i = e_i$ or v_i is a special symbol and $e_i \in v_i|_{\mathcal{C}}$. Denote the set of paths allowed by $(\gamma, \mathcal{C})$ as $L(\gamma, \mathcal{C})$.*

For instance, $(\gamma_{lot}, \mathcal{C}_{lot})$ allows the path $[Bob, HR.dis, Med.dis, Lot.dis, Lot.spk]$, for γ_{lot} accepts the string in (1). The center can specify the context-dependent NFA γ_{lot} in Fig. 3 so as to restrict the usage of credentials defining $Med.dis$ to special parking service in any credential context. As discussed in Example 3, those credentials will not be used for other purposes as long as role paths are allowed by $(\gamma_{lot}, \mathcal{C})$.

We notice that $L(\gamma, \mathcal{C})$ is still a regular language, which can be directly captured by an NFA. For example, N_{lot} in Fig. 2 recognizes $L(\gamma_{lot}, \mathcal{C}_{lot})$ (i.e., $L(N_{lot}) = L(\gamma_{lot}, \mathcal{C}_{lot})$). When designing a context-dependent NFA, credential issuers could first specify an NFA for a specific context, and abstract it away later. Throughout the rest of this paper, unless otherwise stated, references to an NFA imply a context-dependent NFA. When $\mathcal{C}$ is clear from the context, we say γ allows a role path instead of $(\gamma, \mathcal{C})$.

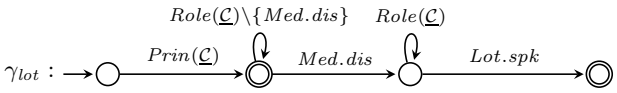

Fig. 3. A context-dependent NFA γ_{lot} restricting the usage of credentials defining $Med.dis$ to special parking service

Definition 8 (Usage-constraint). *A* credential constraint *is a tuple* $\langle \gamma, \pi \rangle$ *where* γ *is an NFA and* $\pi \in \{\forall, \exists\}$. *We say a proof* H *in* $\mathcal{C}$ *is* valid with respect to (wrt) $\langle \gamma, \pi \rangle$ *if* $H \subseteq L(\gamma, \mathcal{C})$ *when* $\pi = \forall$ *and* $H \cap L(\gamma, \mathcal{C}) \neq \emptyset$ *when* $\pi = \exists$. *A constraint* $\langle \gamma, \forall \rangle$ *requires that each role path of* H *be allowed by* $(\gamma, \mathcal{C})$. *A constraint* $\langle \gamma, \exists \rangle$ *says there is at least one role path of* H *allowed by* $(\gamma, \mathcal{C})$.

Example 4 (Delegation depth). Recall that the parking lot delegates the judgement of disability to the center by credential $c_6 : Med.dis \rightarrow Lot.dis$. No restriction is placed on the delegation; the center could re-delegate the judgement to any principal. The lot may deem this over-permissive and want to control the delegation. For example, the lot might decide that a principal's disability should be asserted directly by the center; namely, it allows no re-delegation from the center. As such, the lot designs an NFA γ_{d0} in Fig. 4 and put a constraint $\langle \gamma_{d0}, \forall \rangle$. When designing γ_{d0}, the lot is aware of credential c_6 and thus of the sets $\texttt{Prin} = \{Lot, Med\}$ and $\texttt{Role} = \{Lot.dis, Med.dis\}$.

We now examine the semantics of $\langle \gamma_{d0}, \forall \rangle$. Consider for example the credential context $\mathcal{C}_{lot}$. According to the constraint, all role paths of a valid proof wrt it should be allowed by γ_{d0}. A path is allowed if it meets one of the conditions: (1) $Lot.dis$ does not show up and (2) $Lot.dis$ is preceded by $Med.dis$, which in turn is preceded by a principal. In the latter case, $Med.dis$ follows immediately a principal; this indicates the use of a credential $e \rightarrow Med.dis$, where e is a principal. Hence, a proof involving $Lot.dis$ uses no delegation of $Med.dis$ if and only if it is valid wrt the constraint. Consider a proof H_{depth} containing a path $[Bob, HR.dis, Med.dis, Lot.dis, Lot.spk]$. Since γ_{d0} does not allow this path, the proof H_{depth} is not valid. On the other hand, H_{depth} does use a delegation that the lot tries to prevent (e.g., a credential $HR.dis \rightarrow Med.dis$).

Suppose that the lot now relaxes its requirement: it permits the center to re-delegate to another principal, but disallows any further delegation. This time the lot designs an NFA γ_{d1}, as shown in Fig. 4. In comparison with γ_{d0}, γ_{d1} permits an optional role between a principal and $Med.dis$; this models a possible one-step re-delegation. □

One can proceed to define more constraints using logical connectives as below and define their semantics as in the propositional logic.

$$con ::= \langle \gamma, \forall \rangle \mid \langle \gamma, \exists \rangle \mid (\neg con) \mid (con \wedge con) \mid (con \vee con) \mid (con \Rightarrow con)$$

We could also define a constraint like $\langle \neg\gamma_1 \vee \gamma_2, \forall \rangle$ so that a path is allowed by γ_2 if allowed by γ_1. Since regular languages are closed under the operations union, intersection, difference, and complement, there is a constraint $\langle \gamma, \forall \rangle$ to the same effect.

Definition 9. *Given a credential context* $\mathcal{C}$, *a set* Γ *of credential constraints, and a goal* $D \twoheadrightarrow A.r$, *we say* $C \subseteq \mathcal{C}$ *is a* proving set *of* $e_1 \twoheadrightarrow e_2$ *if for all* $C' \subset C$ *there is no*

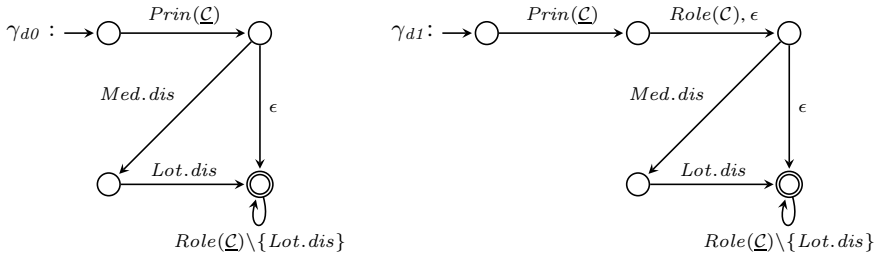

Fig. 4. NFA γ_{d0} forbids re-delegation. NFA γ_{d1} allows an optional one-step re-delegation.

proof H' of $e_1 \twoheadrightarrow e_2$ using C' such that H' is valid wrt all constraints in Γ. Type-3 compliance checking problem in the presence of constraints is to find all the proving sets $C_1, \ldots, C_n$. We denote this problem as $\langle \mathcal{C}, \Gamma, D \twoheadrightarrow A.r \rangle$.

5 ASP Representation

To solve the type-3 compliance checking problem in the presence of constraints, we encode it in ASP. The use of ASP is motivated by, among others, its ability to return all solutions to a problem. Intuitively, we view RT_0 credentials as *actions* of adding principals to roles' member sets. For example, the application of a credential $D \rightarrow A.r$ adds the principal D to $A.r$'s member set; another credential $A.r \rightarrow B.r'$, if applied, further makes D a member of $B.r'$. Therefore, a credential context is considered as a set of actions that may be executed to grant principals role-memberships. Now, to decide if a principal is a member of a role is to decide if there exists an action plan which ultimately adds the principal to the role's member set. The ASP encoding is parameterized on proof size. Given a proof H, define its size as $size(H) = \sum_{h \in H} size(h)$, where, assuming $h = [e_0, \cdots, e_n]$, $size(h) = n$. Given $\langle \mathcal{C}, \Gamma, D \twoheadrightarrow A.r \rangle$ and a proof size parameter k, for any answer set that the ASP program returns, it corresponds to a proving set with a proof H such that $size(H) < k$; on the other hand, for any proving set C with a proof H such that $size(H) < k$, then the program returns an answer set corresponding to C.

6 Experimental Results

In this section, we evaluate the performance of our compliance checker and the overheads resulting from the support of usage-constraints on credentials. Our concerns lie mainly in the computing time required to find all proving sets for a given policy. Experiments were carried out on a Windows 7 laptop with Intel Core 2.66GHz i5-560M processor and 4GB RAM. ASP programs were executed with the grounder gringo 3.0.3 and the solver clasp 2.0.3.[5] In each test, the compliance checking is performed in a credential context of 50 RT_0 credentials, i.e., $|\mathcal{C}| = 50$. All results for a specific parameter setting were averaged over 5 independent tests.

[5] `http://sourceforge.net/projects/potassco/`

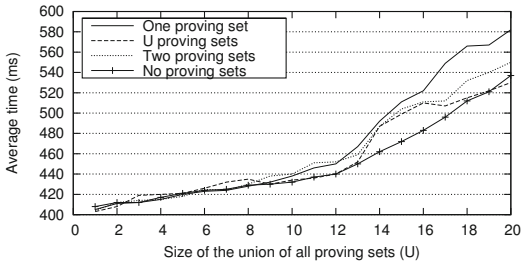

Fig. 5. The computing time as a function of the size of the union of all proving sets

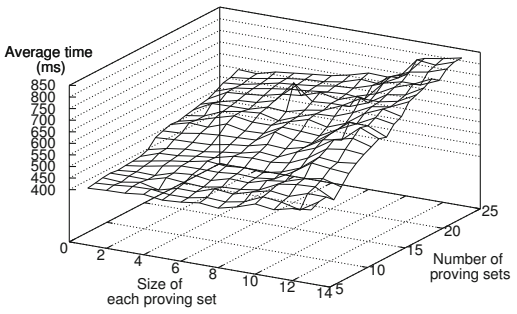

Fig. 6. Computing time as a function of the number of proving sets and the size of each set

We conducted three groups of experiments. First, we evaluated the checker's performance with respect to the size of proving sets. We considered three cases: (i) the policy had one proving set of size U, (ii) the policy had U proving sets of size one, (iii) the policy had two proving sets of size $\frac{3U}{4}$, where U is the size of the union of all proving sets. These are the most interesting cases explored in literature [6,11]. We placed on credentials 20 manually created constraints of types in [3]; these constraints concerned at least 80% of the credentials in $\mathcal{C}$. We set k as 25; namely we only searched for proofs of size smaller than 25. Fig. 5 shows the results. In all cases, it took the checker less than 600 ms to find all proving sets. Besides, we also tested the cases when no proving sets existed. To perform such tests, when we obtained the single proving set in cases of (i), we included one more constraint to rule it out so that no such set existed. In all four cases, the running time grew linearly.

Further we examined the computing time as a function of the number of proving sets and the size of each proving set. Again we put 20 manually created constraints and set k, the limit of proof size, as 25. Fig. 6 shows the results, which confirm that the computing time grew linearly with respect to the number and the size of proving sets.

Similar experiments were conducted on CLOUSEAU in [6]. Although our checker is several times slower than CLOUSEAU, it responded within 850 ms in all previous tests.

Second, we examined the overheads incurred by supporting constraints; we varied the number of constraints $|\Gamma|$. In this experiment, we set $k = 25$. Initially, we had $|\Gamma| = 0$ and 10 proving sets each of size 15. Later, we incrementally added constraints.

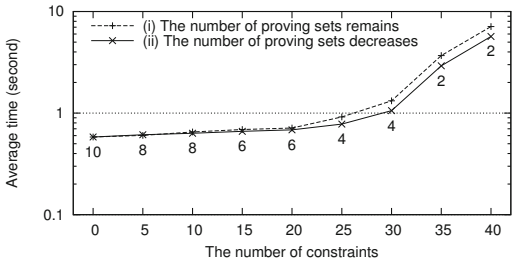

Fig. 7. Computing time as a function of the number of constraints. Below the data points of case (ii) is labelled the number of proving sets in each case of $|\Gamma|$.

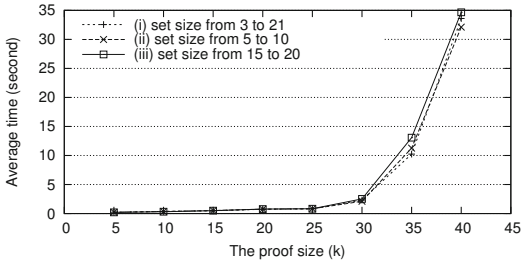

Fig. 8. Computing time as a function of the proof size parameter k

We considered two cases: (i) All proving sets remained, i.e., constraints did not invalidate any of them; (ii) some sets were not valid wrt some added constraints. Recall that a proof should be valid wrt every constraint in Γ; hence, the ASP program encoded all constraints in Γ. Fig. 7 shows the results. In both cases, the checker performed well before $|\Gamma|$ grew to 30 and turned impractical afterwards. Comparing the times for $|\Gamma| = 0$ and $0 < |\Gamma| < 30$, we note that constraints incurred overheads less than 1 second.

Finally, we evaluated the influence of proof size on performance by varying the parameter k. In this experiment, we had 20 constraints and 10 proving sets. We considered three cases according to the size of proving sets: (i) the size ranged from 3 to 21, (ii) the size ranged from 5 to 10, and (iii) the size ranged from 15 to 20. Fig. 8 shows the results. We see that k had a heavy influence on performance. The indistinguishable time difference between the three cases also implies that k played a major role in performance. The checker responded within 3 seconds when $k = 30$; but the performance degenerated rapidly as k grew. This is in accordance with our ASP encoding, which is parametric to k. We observed from practical proofs that their size seldom exceeds 30.

7 Conclusions

In this paper, we presented a definition of usage-constraints on credentials based on NFAs (non-deterministic finite automaton). We illustrated by examples that the definition is able to express important constraints in practice. Based on an encoding in ASP

(answer set programming), we proposed a compliance checker that is able to find all minimal sets of credentials for a given policy; each such set not only constitutes a proof of the policy but also uses the credentials in a way consistent with given constraints. Experiment results showed the efficiency of our approach.

We assumed that compliance checking is performed in a localized credential context. In practice, however, the context may be distributed. In that case, the problem is more challenging. For one thing, constraints may be stored with credentials and thus be distributed too. For another, we still need to search for multiple, if not all, proving sets for a given policy. We plan to study the compliance checking problem under distributed credential contexts in future work.

Acknowledgment. Khaled M. Khan, Yan Zhang and Yun Bai are supported by an NPRP grant (NPRP 09-079-1-013) from the Qatar National Research Fund (QNRF). Jinwei Hu is supported by CASED (www.cased.de). The statements made herein are solely the responsibility of the authors. We also thank Dieter Gollmann for shepherding the paper and the anonymous reviewers for their helpful comments.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Bauer, L., Garriss, S., Reiter, M.K.: Distributed proving in access-control systems. In: IEEE Symposium on Security and Privacy, pp. 81–95 (2005)
3. Bauer, L., Jia, L., Sharma, D.: Constraining credential usage in logic-based access control. In: CSF, pp. 154–168 (2010)
4. Becker, M.Y.: Information flow in credential systems. In: CSF, pp. 171–185 (2010)
5. Huth, M., Ryan, M.: Logic in Computer Science: modelling and reasoning about systems. Cambridge University Press (2004)
6. Lee, A.J., Winslett, M.: Towards an efficient and language-agnostic compliance checker for trust negotiation systems. In: ASIACCS, pp. 228–239 (2008)
7. Li, J., Li, N., Winsborough, W.H.: Automated trust negotiation using cryptographic credentials. ACM Trans. Inf. Syst. Secur. 13(1) (2009)
8. Li, N., Mitchell, J.C., Winsborough, W.H.: Design of a role-based trust-management framework. In: IEEE Symposium on Security and Privacy, pp. 114–130 (2002)
9. Seamons, K.E., Winslett, M., Yu, T., Smith, B., Child, E., Jacobson, J., Mills, H., Yu, L.: Requirements for policy languages for trust negotiation. In: POLICY, pp. 68–79 (2002)
10. Sipser, M.: Introduction to the Theory of Computation (2005)
11. Smith, B., Seamons, K.E., Jones, M.D.: Responding to policies at runtime in trustbuilder. In: POLICY, pp. 149–158 (2004)
12. Winsborough, W.H., Li, N.: Towards practical automated trust negotiation. In: POLICY, pp. 92–103 (2002)

A ASP Encoding

Since uppercase letters are usually taken as variables in ASP and lowercase letters as constants, we use lowercase letters to denote principals in ASP programs. We translate

a compliance checking problem $\langle \mathcal{C}, \Gamma, d \twoheadrightarrow a.r \rangle$ into an ASP program, denoted as $\Pi(\langle \mathcal{C}, \Gamma, d \twoheadrightarrow a.r \rangle, k)$. An ASP program is a finite set of rules of the form

$$a \text{ :- } b_1, \cdots, b_m, \texttt{not}\ b_{m+1}, \cdots, \texttt{not}\ b_n.$$

where $0 \leq m \leq n$, a is either an atom or $\bot$, b_i is an atom, and `not` denotes (default) negation. We say that a rule is a *fact* if $n = 0$. For simplicity, we omit :- when writing facts. Readers are referred to [1] for details.

$\Pi(\langle \mathcal{C}, \Gamma, d \twoheadrightarrow a.r \rangle, k)$ consists of four parts: $\Pi(\mathcal{C})$, $\Pi(apply)$, $\Pi(d \twoheadrightarrow a.r, k)$, and $\Pi(\Gamma)$. Except $\Pi(apply)$, the sub-programs are parametric to problems. Program $\Pi(\mathcal{C})$ models each credential in $\mathcal{C}$. Program $\Pi(apply)$ states how credentials are applied. Program $\Pi(d \twoheadrightarrow a.r, k)$ tests whether a proof of size smaller than k exists. Program $\Pi(\Gamma)$ interprets the constraints in Γ and ensures that proofs extracted from answer sets are valid wrt the constraints. Program $\Pi(\mathcal{C})$ follows the original semantics of RT [8]; we omit its presentation here.

We first present $\Pi(d \twoheadrightarrow a.r, k)$. Fact (2) below declares our goal as proving that principal d can obtain a membership in $a.r$. Rule (3) says that we prove the goal at step T if the goal holds at T. Rule (4) says that once the goal is proved at T it is proved afterwards. Rule (5) requires that an answer set should contain the atom $prove(k)$ (i.e., the membership is proved within k steps).

$$goal(m(d, r(a.r))). \tag{2}$$

$$prove(T) \text{ :- } hold(M, T), goal(M), step(T). \tag{3}$$

$$prove(T+1) \text{ :- } step(T), T < k, prove(T). \tag{4}$$

$$\bot \text{ :- } \texttt{not}\ prove(k). \tag{5}$$

A term $m(d, r(a.r))$ denotes d's membership in $a.r$. Expression $r(a.r)$ is short for a term $r(a, r, \rho)$, where ρ is a special constant used in our ASP programs. The term $r(a, r, \rho)$ denotes a role $a.r \in NRole(\mathcal{C})$. Similar shorthands are used below.

We proceed to describe $\Pi(apply)$ (i.e., how credentials are applied), as shown in Fig. 9. At each step, some credentials are applicable while others are not. As stated in rule (6), a credential is applicable if there is a principal ready to accept the membership it offers. Rule (7) says that, a principal is ready when his memberships satisfy the credential's condition and he is not forbidden from being so. Rules (8) and (9) list the two cases of a principal being not ready: one is when the principal already owns the membership that the credential offers; the other is when another principal is ready. Rules (10) and (11) are two auxiliary rules defining the satisfiability of conditions.

At each step, one and only one membership is granted; as a result, if more than one membership can be granted at a step, only one of them succeeds and others are blocked. In addition, if the goal $d \twoheadrightarrow a.r$ has been reached, no more membership is needed. Rules (12)-(14) capture this idea. Specifically, rule (12) says that, when the goal has not yet been reached, a credential is applied if not blocked. A credential is blocked in event of another credential being applied at the same time (rule (13)) or a linked role membership being derived (rule (14)). Rule (14) also implies that linked role memberships take priority. Finally, rule (15) says that only applicable credentials may be applied; otherwise there arises a conflict.

$$apbl(C,T) \text{ :- } rdy(X,C,T), cred(C), step(T). \quad (6)$$
$$rdy(X,C,T) \text{ :- } \texttt{not } not\_rdy(X,C,T), sat(cond(C,X),T). \quad (7)$$
$$not\_rdy(X,C,T) \text{ :- } effect(C,m(X,R)), hold(m(X,R),T). \quad (8)$$
$$not\_rdy(X,C,T) \text{ :- } rdy(Y,C,T), X \neq Y, prin(X), prin(Y). \quad (9)$$
$$sat(D,T) \text{ :- } \texttt{not } not\_sat(D,T), in(\_,D), step(T). \quad (10)$$
$$not\_sat(D,T) \text{ :- } in(M,D), \texttt{not } hold(M,T), step(T). \quad (11)$$
$$apl(C,T) \text{ :- } cred(C), step(T), \texttt{not } goal(T), \texttt{not } blk(C,T). \quad (12)$$
$$blk(C,T) \text{ :- } cred(C;C'), step(T), apl(C',T), C \neq C'. \quad (13)$$
$$blk(C,T) \text{ :- } linked(\_,T), cred(C). \quad (14)$$
$$\bot \text{ :- } cred(C), step(T), apl(C,T), \texttt{not } apbl(C,T). \quad (15)$$
$$obt(m(X,R'),m(X,R),T+1) \text{ :- } rdy(X,C,T), effect(C,m(X,R)), in(m(X,R'), cond(C,X)), apl(C,T), T < k-1. \quad (16)$$
$$obt(M',M,T+1) \text{ :- } linked(M',M,T), T < k-1. \quad (17)$$
$$hold(M,T) \text{ :- } obt(\_,M,T'), T' \leq T, step(T;T'). \quad (18)$$
$$obt(m(X,r(X)),m(X,r(X)),1) \text{ :- } prin(X). \quad (19)$$
$$need(M) \text{ :- } goal(M). \quad (20)$$
$$need(M) \text{ :- } need(M'), obt(M,M',\_). \quad (21)$$
$$need(m(Y,r(Z.N))) \text{ :- } obt(m(X,r(Y.N')),m(X,r(Z.N.N')),T), need(m(X,r(Y.N')); m(X,r(Z.N.N'))). \quad (22)$$
$$\bot \text{ :- } \texttt{not } need(M), obt(\_,M,T), T > 1. \quad (23)$$

Fig. 9. Rules modelling how credentials are applied

A principal obtains a role membership when a credential is applied (rule (16)) or when a linked role membership is derived (rule (17)). A fact $obt(m_1, m_2, t)$ means that, following m_1 in a path, membership m_2 is obtained at step t; note that m_1 and m_2 are memberships of the same principal. The obtained memberships remain afterwards (rule (18)). Rule (19) declares facts that a principal d is a member of $r(d)$ at the beginning.

The rest of the encoding ensures that the proof is minimal. To retain only necessary memberships, we traverse back from the goal. As stated in rule (20), the goal is needed. Moreover, a membership next to a needed membership is necessary (rule (21)). Rule (22) considers the case of linked role memberships: if y's membership in $z.n$ helps derive x's membership in $z.n.n'$ from x's membership in $y.n'$, then y's membership in $z.n$ is necessary provided that the latter two are also necessary. Rule (23) says that all derived memberships should be necessary; otherwise there arises a conflict.

Finally, we present the program $\Pi(\Gamma)$. For each $con \in \Gamma$, we have a program $\Pi(con)$. Hence, $\Pi(\Gamma) = \bigcup_{con \in \Gamma}(\Pi(con))$. Here we only present the representation of a constraint $\langle \gamma, \exists \rangle$; other types of constraints can be likewise encoded. Assume that the credential context is $\mathcal{C}$ and that $L(N) = L(\gamma, \mathcal{C})$, i.e., N recognizes the language $L(\gamma, \mathcal{C})$. We work with N. First we denote the NFA N in ASP. Suppose $N = (S, \Sigma, \delta, s_0, F)$; we use a fact $start(s_0)$ to denote the start state s_0, a fact

$$read(m(X, r(X)), S) \text{ :- } start(s_0), tran(s_0, X, S), need(m(X, r(X))). \quad (24)$$

$$read(m(X, R), S) \text{ :- } read(m(X, R'), S'), tran(S', R, S), obt(m(X, R'), m(X, R), T). \quad (25)$$

$$read(M, S) \text{ :- } read(M, S'), tran(S', \epsilon, S). \quad (26)$$

$$final(M) \text{ :- } read(M, S), accept(S). \quad (27)$$

$$exist \text{ :- } final(M), last(M). \quad (28)$$

$$\bot \text{ :- } \texttt{not } exist. \quad (29)$$

$$last(M) \text{ :- } \texttt{not } not\_last(M), obt(\_, M, \_). \quad (30)$$

$$not\_last(m(X, R)) \text{ :- } obt(m(X, R), m(X, R'), \_), R \neq R'. \quad (31)$$

Fig. 10. Rules encoding $\langle \gamma, \exists \rangle$

$accept(s_f)$ to denote an accept state $s_f \in F$, and a fact $tran(s_1, e, s_2)$ to denote a transition $s_2 \in \delta(s_1, e)$ where $e \in Prin(\mathcal{C}) \cup Role(\mathcal{C}) \cup \{\epsilon\}$.

Next we test if a role path is accepted by N. Recall that a path is seen as a sequence of principal-role pairs. For example, a path $[e_0, e_1]$ corresponds to a sequence $[e_0 : e_0, e_0 : e_1]$. In our ASP programs, we do not encode the path, but the sequence with atoms of the form $obt(m(e_0, r(e_0)), m(e_0, r(e_1)), t)$, where $m(e_0, r(e_1))$ denotes the principal-role pair $e_0 : e_1$. Therefore, we verify the sequence.

In Fig. 10, rules (24)-(27) simulate N reading an input string and making state transitions accordingly. Rule (24) says, given a sequence $[e_0 : e_0, \cdots]$ as input, N starts in its start state s_0 and, if there is a transition $s \in tran(s_0, e_0)$, proceeds to another state s. In a state s', N proceeds to another state s, depending on what it reads from the input. Suppose that N reaches s' after reading $e_0 : e_i$ and that the next symbol in a sequence is $e_0 : e_j$. In this case, N transits to another state s if there is a transition $s \in tran(s', e_j)$. Rule (25) captures this idea; there, an atom $read(m(e_0, r(e_j)), s)$ means that N is in state s after reading $e_0 : e_j$. Rule (26) interprets the transition ϵ as usual: in a state s', N moves to a state s reachable from s' via ϵ. This reading and transiting process goes on until the input reaches its end or no transition is available. Rule (27) marks a membership final if an accept state is reached after N reads it. If the final membership is also the last one in a sequence, N accepts this sequence; namely, there exists a sequence accepted by N, as stated in rule (28). As required by the constraint, such a sequence must exist; hence rule (29) excludes answer sets where $exist$ is not true. Rules (30) and (31) define the last membership in a sequence.