Driving and Monitoring Provisional Trust Negotiation with Metapolicies

Piero Bonatti
Università di Napoli Federico II
Napoli, Italy
bonatti@na.infn.it

Daniel Olmedilla
L3S Research Center and Hanover University
Hanover, Germany
olmedilla@l3s.de

Abstract

We introduce the provisional trust negotiation framework PROTUNE, *for combining distributed trust management policies with provisional-style business rules and access-control related actions. The framework features a powerful declarative metalanguage for driving some critical negotiation decisions, and integrity constraints for monitoring negotiations and credential disclosure.*

1 Introduction

The term “policy is used in the literature in a broad sense that encompasses the following notions.

Security Policies pose constraints on the behaviour of a system. They are typically used to control permissions of users/groups while accessing resources/services.

Trust Management policy languages are used to collect user properties in open environments, where the set of potential users spans over the entire web.

Action Languages are used in reactive policy specification to execute actions like event logging, notifications etc. Authorizations that involve actions and side effects are sometimes called *provisional*.

Business Rules are ”statements about how a business is done” [2] and are used to formalize and automatize business decisions as well as for efficiency reasons. They can be formulated as *reaction rules*, *derivation rules*, and *integrity constraints* [12, 13].

All these kinds of specification interact tightly with each other: Credential-based user properties are typically used to assign access control permissions; logging, monitoring, and other actions are part of the high-level security specification documents of many organizations; many business rules—say, for granting discounts or special services—are based on the same kind of user properties that determine access control decision; moreover, this kind of business decisions and access control decisions are to be taken more or less simultaneously—e.g. immediately before service access.

Although many approaches have been described to address the above points, there is no a common solution, integrating them all in a single framework.

One technical difficulty to be addressed is that the different notions of policy are naturally described by different computational models [5], including materialization techniques (inherited from the research on deductive databases); partial evaluation, abduction, and distributed logic programming (especially for trust management); event-condition-action (ECA) rules (esp. reactive business rules).

We believe that some policies would be more naturally described under ECA, and in the long run we plan to integrate this paradigm with the popular ones for automated trust negotiation (ATN). However, in this first paper, we explore the expressiveness of the actions that can be associated to provisional predicates. Intuitively, given a rule such as “$\texttt{allow}(\mathit{Service}) \leftarrow A, B, C$” where conditions A and B are satisfied but C is not, a cooperative system may try to satisfy C by executing suitable actions (e.g. asking the client for more credentials, directing the client to a registration service etc.)

The advantage of this approach—that we call *provisional approach*—is that it fits nicely some of the current ATN frameworks, and allows to combine smoothly trust negotiation and business rules in a single policy.

A second technical problem to be addressed is related to flexibility. Automated negotiators have to make a number of decisions (e.g. How should a request for credentials be formulated? Which credentials should be disclosed among those that match a server’s request?) As the optimal choices are typically application dependent, the negotiators should adapt their behavior to the given scenario.

In this paper we introduce PROTUNE (PROvisional TrUst NEgotiation): an approach at integrating the above aspects into a coherent and flexible framework. PROTUNE’s rule language extends two previous languages: PAPL [4], that until 2002 was one of the most complete policy languages for trust negotiation [10], and PEERTRUST [6], that supports distributed credentials and a more flexible policy protection mechanism. In detail, the main contributions of this paper are (i) A trust management language supporting general provisional-style actions (possibly user-defined), (ii) An extendible declarative metalanguage for driving decisions about request formulation, information disclosure, and distributed credential collection, (iii) A parameterized negotiation procedure, that gives a semantics to the metalanguage and provably satisfies some desirable properties for all possible metapolicies, (iv) Integrity constraints for negotiation monitoring and disclosure control, and (v) General, ontology-based techniques for importing and exporting metapolicies and for smoothly integrating language extensions (with an example based on the RT family of credential languages [7]).

We assume the reader to be familiar with the basics of logic programming, as illustrated in [9]. In particular, we shall use the notions of atoms and literals, most general unifier (mgu), least Herbrand model, and negation as failure.

2 Negotiations

Trust negotiation in PROTUNE is directly inspired by PAPL [4] and PeerTrust [6], that build on ideas introduced in [14]. In summary, each party (client and server) makes decisions based on a set of rules that entail decision atoms such as $\texttt{allow}(X)$, based on conditions over currently available credentials and declarations (sent by the other party) and a time-dependent state, covering the negotiation state, user profiles, etc. (see [4] for further details). On the server, X is typically a service; on the client X may denote credential release, declaration release and actions execution. At each party, credential and declaration requests are automatically derived from the local rules by identifying the sets of credentials and declarations that entail $\texttt{allow}(X)$. In PROTUNE, requests may also contain more general actions, basically remote service invocations.

In general there may be multiple ways of entailing $\texttt{allow}(X)$, therefore multiple alternative requests. It is desirable to send them out in parallel, because the conditions that can be fulfilled by the other party cannot be known in advance; simultaneous requests may significantly reduce the number of messages in the negotiation.

On the other hand, the number of alternative requests may be exponentially larger than the policy, due to combinatorial explosion in compound requests. To avoid this, it is preferrable to send out the *rules* themselves, as a compact request. First, however, the rules should be suitably filtered to protect the sensitive parts of the policy (the policy itself may be confidential). In PROTUNE rules can be hidden until the other peer fulfils enough requests.

Another reason for filtering is that the other party has no access to the local state and hence it is not able to give a meaning to state conditions in the rules. Then, state conditions should be evaluated before sending out the rules (partial rule evaluation). However, this procedure must be controlled to avoid sensitive information leakage. For example, consider the simple rule

$$\begin{aligned}&\texttt{allow}(\texttt{enter\_site}()) \leftarrow \\ &\quad \texttt{declaration}(\texttt{usr} = \texttt{U}, \texttt{passwd} = \texttt{P}), \texttt{has\_passwd}(\texttt{U}, \texttt{P})\end{aligned} \tag{1}$$

describing an old-fashioned but still very common authentication procedure based on login and password. If the state predicate `has_passwd` were evaluated before sending the policy to the client, then the client would receive all the ground rules

$$\texttt{allow}(\texttt{enter\_site}()) \leftarrow \texttt{declaration}(\texttt{usr}{=}U, \texttt{passwd}{=}P)$$

where U and P are bound to all legal (user,password) pairs. In [4] this is avoided by a combination of *guarded rules* and a rigid two-phase negotiation protocol. Here we adopt a more flexible protocol based on *policy blurring* (Section 6). As an additional level of complexity, in PROTUNE the negotiator must decide when the actions associated to provisional atoms are to be dispatched to the execution handler for execution.

The architecture of the system is described in [3].

3 The rule language

The rule language is based on normal logic program rules "$A \leftarrow L_1, \dots, L_n$" where A is a standard logical atom (called the *head* of the rule) and $L_1, \dots, L_n$ (the *body* of the rule) are literals, that is, L_i equals either B_i or $\neg B_i$, for some logical atom B_i.

A *policy* is a set of rules, such that negation is applied neither to *provisional predicates* (defined below), nor to any predicate occurring in a rule head. This restriction ensures that policies are *monotonic* in the sense of [10], that is, as more credentials are released and more actions executed, the set of permissions does not decrease. Moreover, the restriction on negation makes policies *stratified programs*; therefore negation as failure has a clear, PTIME computable semantics that can be equivalently formulated as the perfect model semantics, the well-founded semantics or the stable model semantics [1].

The vocabulary of predicates occurring in the rules is partitioned into the following categories:

- *Decision predicates*: Currrently this class comprises predicates `allow` and `sign`. These predicates are defined in the policy, that is, they occur in the head of some policy rules.

  The unary predicate `allow` is queried by the negotiator for access control decisions. The argument of `allow` can denote a service call (for access control decisions) or it can be $\texttt{release}(\textit{credential})$ or $\texttt{execute}(\textit{action})$ (for privacy protection). In response to a service request s, the negotiatior looks for a (partial) proof of $\texttt{allow}(s)$, and handles it as sketched in the previous section. Similarly, in response to a credential request or an action request r, the negotiator looks for a proof of $\texttt{allow}(r)$ and processes it appropriately.

Predicate `sign` is described in [3].

- *Abbreviation/abstraction predicates*: These are predicates defined in the policy. They have many purposes ranging from the definition of high-level client properties (e.g. by combining low-level data and/or different credentials, cf. [4]) to the specification of new credential semantics (see Section 11).
- *Constraint predicates* comprise the usual equality and disequality predicates.
- *State predicates*: Policy decisions have to be taken with respect to a time-dependent system *state*, encoding the current negotiation state, legacy data, user profiles, and so on. State predicates are further partitioned into the following subclasses.

  • *State query predicates*: These predicates read the current state without modifying it. They comprise both built-in and application dependent predicates. Built-in state predicates model the state of the negotiation, and provide a uniform interface to external packages in the style of HERMES [11]. An example of negotiation state atom is $\texttt{request}(n, R)$; it holds if R is the n-th request in the negotiation. External packages (including databases and other data sources) can be queried with atoms of the form:

  $$\texttt{in}(X, \mathit{package\_name} : \mathit{function}(\mathit{arg\_list})) \qquad (2)$$

  where the variable X ranges over the set of objects returned by the external code call $\mathit{package\_name} : \mathit{function}(\mathit{arg\_list})$.

  • *Provisional predicates*: These are predicates that may be made true by means of appropriate *actions* that may modify the current state. Such actions may be carried out by the server, by the client, or both.

  An important example is `credential`. An atom $\texttt{credential}(C, K)$ is true when the current negotiation state contains a verified credential matching C and signed by the principal whose public key is K. If this condition is not satisfied, still (an instance of) $\texttt{credential}(C, K)$ can be made true by searching for the credential (either directly or by asking the peer to provide it) and loading it into the negotiation state after verification.

  Similarly, the `declaration` predicate is satisfied if the peer releases a declaration matching the predicate arguments. The `declaration` predicate is generalized by the `do` predicate. Intuitively, $\texttt{do}(\mathit{uri\_or\_service\_request})$ can be made true if the peer connects to uri or invokes $\mathit{service\_request}$, and then carrys out some application dependent procedure. If the procedure is successfully completed, then the atom $\texttt{do}(\mathit{uri\_or\_service\_request})$ becomes true in the negotiation state.

  Sometimes, the actions associated to provisional predicates are to be executed locally, by the negotiator. A common example is $\texttt{logged}(X, \mathit{logfile\_name})$ that may be made true by recording X into $\mathit{logfile\_name}$ [3].

  Provisional predicates may be used to encode business rules. For instance, the next rule enables discounts on low_selling articles in a specific session:

  $$\begin{aligned}&\texttt{allow}(\mathit{Srv}) \leftarrow \ldots, \texttt{session}(\mathit{ID}),\\ &\quad \texttt{in}(X, \texttt{sql:query}('\texttt{select} * \texttt{from low\_selling}')),\\ &\quad \texttt{enabled}(\texttt{discount}(X), \mathit{ID})\,.\end{aligned}$$

  Intuitively, if $\texttt{enabled}(\texttt{discount}(X), \mathit{ID})$ is not yet true but the other conditions are verified, then the negotiator may execute the action associated to `enabled` and the rule becomes applicable (if $\texttt{enabled}(\texttt{discount}(X), \mathit{ID})$ is already true, no action is executed). The action associated to `enable` in this case is application dependent. In the next section we shall see how to define such application-specific provisional predicates.

  Sometimes actions should be executed *before* asking the peer for credentials. In the next rule the log action is meant to record the incoming request, and must be executed immediately and independently from the peer's response. Predicate `time` is a state query predicate, while `unlogged_allow` is an abbreviation predicate, encoding the actual access control decision for service Srv:

  $$\begin{aligned}&\texttt{allow}(\mathit{Srv}) \leftarrow \texttt{time}(T),\\ &\quad \texttt{logged}(\mathit{Srv} + '\texttt{requested at }' + T,\ \texttt{req.log}),\\ &\quad \texttt{unlogged\_allow}(\mathit{Srv})\,.\end{aligned}$$

  The following sections will show how to specify the execution time of provisional atoms with metapolicies.

Remark 1 *For simplicity, we assume in this paper that provisional atoms are orthogonal, in the sense that the action associated to any ground atom A cannot change the truth value of any other ground provisional atom.*

The rule language supports object-oriented dot syntax, compatible with semantic web standards such as RDF and OWL. Dot notation is actually syntactic sugar, e.g. term $X.\texttt{attr} : v$ abbreviates the standard atom $\texttt{attr}(X, v)$. See [3] for the translation of general O.O. terms.

Declarative Rule Semantics

Policies are interpreted in the context of a time-dependent *state*, that determines at each instant the extension of state predicates. In the abstract setting, a state is simply a consistent set of ground state literals Σ (i.e. the set of all literals that hold in the current state). In practice, of course, state predicates are evaluated on demand with a variety of techniques, as explained before.

Semantics is formulated in two stages: first, the notion of *reduct* specifies how state predicates are evaluated against the current state; then we can define the *canonical model* of the policy.

The *reduct* of a policy Pol w.r.t. Σ, denoted by Pol^{Σ}, is obtained from the ground instantiation of Pol by

- removing all rules whose body contains a literal $L \notin \Sigma$;
- removing all state literals from the remaining rules.

Note that the reduct is logically equivalent to the set of rules obtained by replacing each state literal with its truth value specified by Σ.

Let $\mathcal{H}$ denote the Herbrand base, that is, the set of all ground atoms. The *canonical model of Pol w.r.t. Σ* is

$$\mathsf{cmodel}(Pol, \Sigma) \quad = \quad \{A \in \mathcal{H} \mid Pol^{\Sigma} \models A\}\,. \qquad (3)$$

Note that the reduct is a positive program. Then—by standard results—it holds that the canonical model is the least Herbrand model of the reduct.

4 Metapolicies

Metapolicies consist of rules with a shape similar to object-level rules. The main differences are:

- The syntactic material of the object-level rule language (i.e. predicate names, constant names, variable names, rule names etc.) may occur as terms in the metapolicy. In the following, for all rules R we shall denote by $\hat{R}$ the name of R.
- The built-in predicates comprise Prolog-style metapredicates for inspecting terms, checking groundness, etc. Moreover, a predicate $\texttt{holds}(G)$ allows to call an object-level goal G against the current state, using the object-level policy. These predicates are illustrated below.
- A set of reserved attributes associated to predicates, literals and rules is used to drive the negotiator's decisions.

Here are a few examples. If p is a predicate, then $p.\texttt{sensitivity} : \texttt{private}$ means that the extension of the predicate is private and should not be disclosed. An assertion $p.\texttt{type} : \texttt{provisional}$ declares p to be a provisional predicate; then p can be attached to the corresponding action α by asserting $p.\texttt{action} : \alpha$. If the action is to be executed locally, then assert $p.\texttt{actor} : \texttt{self}$, otherwise assert $p.\texttt{actor} : \texttt{peer}$.

In most cases, the attributes of a predicate p should be inherited by all the literals with p. By default, PROTUNE handles attribute propagation for standard attributes; alternatively, attribute propagation may be expressed and controlled with simple metarules such as:

$$L.attr\!:\!Val \leftarrow \texttt{literal}(L),\ L.\texttt{predicate}.attr\!:\!Val$$

where `literal` and `predicate` are built-ins for metaterm inspection. Many of these rules do not depend on the current state and can be precompiled to improve performance (metaattribute materialization).

Metarules allow fine-grained tuning of state predicate evaluation. For example, for performance reasons, it may be useful to delay predicates with a large extension until argument instantiation restricts the number of answer substitutions. Here is a simple example: the next rule enables immediate evaluation of a predicate only if the key argument is specified.

$$\texttt{table}(Key, Data).\texttt{evaluation} : \texttt{immediate} \qquad (4)$$
$$\leftarrow \texttt{ground}(Key)\,.$$

As we pointed out before, metarules and metaattributes may be used to attach provisional predicates to the corresponding actions. The language for local actions should be flexible and powerful, to facilitate the integration of trust management in the surrounding environment. Script languages are good candidates; multiple action languages may coexist in the same policy.

As an example, recall the predicate `logged` introduced in the previous section. It can be associated to its action by a simple metafact:

$$\texttt{logged}(Msg, File).\texttt{action}\!: {'\texttt{echo}'} + Msg + {'\texttt{>}'} + File\,.$$

The exit status of the action determines whether the corresponding provisional atom is asserted.

Specifying actions for other actors is a more delicate matter. Peers cannot be assumed to execute arbitrary foreign scripts. Currently, the provisional predicate `do` is the most general way to ask peers to execute actions. This predicate accepts only URIs and performs only remote service invocation or web page downoload in a controlled way (including user approval) to prevent the use of this mechanism as a tool for DoS attacks.

5 Semantics-preserving policy filtering

We parameterize policy filtering in order to be able to modify the filtering process using metadata. For the filtering techniques reported in this section, we shall prove that the choice of the filtering criteria does not affect correctness/completeness.

5.1 Removing Irrelevant Rules

This is an instance of the *need to know principle*. The *relevant subset* of a policy Pol w.r.t. an atom A is the least set S such that:

- If the head of a rule $R \in Pol$ unifies with A, then $R \in S$;
- If the head of a rule $R \in Pol$ unifies with an atom B occurring in the body of some rule in S, then $R \in S$.

The relevant subset of Pol w.r.t A will be denoted by $\mathsf{relevant}(Pol, A)$.

The relevant subset of Pol w.r.t A suffices to determine which instances of A are entailed by the policy in the given state:

Lemma 1 *For all ground atoms $A\theta$ (θ is a substitution), $A\theta \in \mathsf{cmodel}(Pol, \Sigma)$ iff $A\theta \in \mathsf{cmodel}(\mathsf{relevant}(Pol, A), \Sigma)$.*

5.2 Evaluating State Predicates

Next we define partial evaluation. Let E be a set of (possibly nonground) state literals. Intuitively, E specifies which literals can be evaluated. For all rules R, let $R \xrightarrow{\Sigma,E}_1 S$ iff

- $R = (A \leftarrow L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_n)$
- $L_i \in E$
- $S = \{(A \leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n)\theta \mid$ for some $L \in \Sigma,\ \theta = \mathsf{mgu}(L_i, L)\}$.

This evaluation relation is extended to policies in the natural way: For all policies Pol, define $Pol \xrightarrow{\Sigma,E}_1 Pol'$ iff

- there exists $R \in Pol$ and S such that $R \xrightarrow{\Sigma,E}_1 S$
- $Pol' = (Pol \setminus \{R\}) \cup S$.

Finally, we denote with $\xrightarrow{\Sigma,E}$ the reflexive transitive closure of $\xrightarrow{\Sigma,E}_1$.

Partial evaluation preserves the semantics of a policy Pol in all contexts Pol'':

Lemma 2 *If $Pol \xrightarrow{\Sigma,E} Pol'$, then for all Pol''*

$$\mathsf{cmodel}(Pol \cup Pol'', \Sigma) = \mathsf{cmodel}(Pol' \cup Pol'', \Sigma) .$$

The partial evaluation of a policy is a converging and non-ambiguous (confluent) process (regardless of the choice of the rule and literal to be rewritten at each step). To formalize this property, we introduce the notion of trace.

A *trace for Pol w.r.t. Σ and E* is a (possibly infinite) sequence of policies $Pol_1 \xrightarrow{\Sigma,E} Pol_2 \xrightarrow{\Sigma,E} \cdots \xrightarrow{\Sigma,E} Pol_i \xrightarrow{\Sigma,E} \cdots$. A trace is *complete* if it is infinite or for the last element Pol_n in the sequence, there exists no policy Pol' such that $Pol_n \xrightarrow{\Sigma,E} Pol'$.

Theorem 1 *For all policies Pol, states Σ and sets E,*

1. *(termination) Pol has no infinite traces w.r.t. Σ and E,*
2. *(confluence) all complete traces of Pol w.r.t. Σ and E have the same last element.*

The unique result of partial evaluation (i.e., the last element of each complete trace) will be denoted by $\mathsf{partEval}(Pol, \Sigma, E)$.

As a consequence of the above results, in order to evaluate the answer substitutions of an atom A, it suffices to use the partial evaluation of the relevant part of the policy:

Theorem 2 *For all ground atoms $A\theta$, and for all Pol, Σ, and E of the appropriate type, $A\theta \in \mathsf{cmodel}(Pol, \Sigma)$ iff*

$$A\theta \in \mathsf{cmodel}(\mathsf{partEval}(\mathsf{relevant}(Pol, A), \Sigma, E), \Sigma) .$$

5.3 Compiling Private Policies

The *immediate consequences* of a rule R w.r.t. Pol and Σ are the heads of the (ground) rules $R' \in \{R\}^{\Sigma}$ whose body is true in $\mathsf{cmodel}(Pol, \Sigma)$. The set of all immediate consequences of R w.r.t. Pol and Σ is denoted by $\mathsf{cons}(R, Pol, \Sigma)$. This operator is extended to policies in the natural way:

$$\mathsf{cons}(Pol', Pol, \Sigma) = \bigcup_{R \in Pol'} \mathsf{cons}(R, Pol, \Sigma).$$

Intuitively, cons *compiles* the subpolicy Pol' and replaces it with its immediate consequences. In this way, the results of the policy may be released to the peer without disclosing the internal structure of the rules.

This transformation preserves the semantics of the given policy, no matter what rules are compiled:

Theorem 3 $\mathsf{cmodel}(Pol \cup Pol', \Sigma) =$

$$\mathsf{cmodel}(Pol \cup \mathsf{cons}(Pol', Pol \cup Pol', \Sigma), \Sigma) .$$

6 Filtering with information loss

Policies and states are both sensitive resources. In general it may be necessary to hide part of them, which necessarily causes some information loss.

Some rules R may have to be hidden and blocked until the client is trusted enough. This is accomplished by means of suitable metastatements:

$$\hat{R}.\mathtt{sensitivity} : \mathtt{not\_applicable} \leftarrow \ldots .$$

(where $\hat{R}$ is R's name). As more credentials arrive, R may become visible and extend negotiation opportunities. In this framework, policy disclosure has a reactive flavour, as opposed to the predefined graph structure adopted in [15].

Similarly, sensitive state predicates may have to be blocked until their evaluation does not disclose confidential information.

However, they cannot simply be left in the policy and sent to the client[1] because

[1]Hereafter by "client" we mean the peer that submitted the last request, and by "server" we denote the peer that is evaluating its local policy to decide whether the request should be accepted and whether a counter-request is needed.

- the client does not know how to evaluate them, since it has no access to the server's state, and
- the syntax of protected conditions may suffice to disclose some confidential information about the structure of the policy.

Removing these occurrences from the rules is not a good solution either, because then the client would not be aware that some conditions that lie beyond its control shall be checked later by the server. The client should be able to see that even if all credentials occurring in the policy were supplied, still the requested access might be denied. More precisely, the client should be able to distinguish the credential sets that satisfy the server's request with no additional checks, from the credential sets that are subject to further verification.

The solution adopted here consists in *blurring* the state conditions that cannot be evaluated immediately and cannot be made true by the other party. Such conditions are blurred by replacing them with a reserved propositional symbol.

For example, consider again the login policy (1). To avoid information leakage we postpone the evaluation of $\mathtt{user}(\mathtt{U},\mathtt{P})$ and send the client a modified rule:

$$\begin{aligned}&\mathtt{allow}(\mathtt{enter\_site}()) \leftarrow \\ &\quad \mathtt{declaration}(\mathtt{usr}=\mathtt{U},\mathtt{passwd}=\mathtt{P}), \mathtt{blurred}\end{aligned}$$

where r is the name of rule (1). From this rule, a machine may realize that sending the declaration does not suffice to enter the site; first the server is performing a check of some sort. Blurring is formalized below.

Let B be a set of literals, specifying which literals have to be blurred. For all rules $R = (A \leftarrow Body)$ with name r, let $\mathsf{blur}(R, B) = (A \leftarrow Body')$ where

- $Body' = Body$ if $Body \cap B = \emptyset$, and
- $Body' = (Body \setminus B) \cup \{\mathtt{blurred}\}$ otherwise.

Then for all policies Pol, define

$$\mathsf{blur}(Pol, B) = \bigcup_{R \in Pol} \mathsf{blur}(R, B)\,.$$

To prove the effectiveness of blurring in protecting the internal state, we show that under suitable conditions, the blurred partial evaluation of any given policy Pol is invariant across all possible contents of the protected part of the state. As a consequence, from the result of the blurring one cannot deduce any protected state literal.

To formalize this, say two states are equivalent if they have the same non-blurred (public) part:

$$\Sigma \equiv_B \Sigma' \text{ iff } \Sigma \setminus B = \Sigma' \setminus B\,.$$

Theorem 4 (Confidentiality) *For all Pol, Σ, Σ', E and B of the appropriate type, if $E \cap B = \emptyset$ and $\Sigma \equiv_B \Sigma'$ then*

$$\mathsf{blur}(\mathsf{partEval}(Pol, \Sigma, E), B) = \mathsf{blur}(\mathsf{partEval}(Pol, \Sigma', E), B)\,.$$

The precondition $E \cap B = \emptyset$ is very important; if it were violated, then some protected literal might be evaluated during filtering. If this happens, one can find counterexamples to the above theorem where some protected state literals can be deduced from the filtered policy.

Moreover, for a correct negotiation, $E \cup B$ *should cover all state literals that cannot be made true by the client*. This guarantees that the result of the filtering contains only predicates that can be understood and effectively handled by the client. This discussion gives us a method for determining B:

Let LSL be the set of all *local state literals*, that is, those with a predicate p such that

- $p.\mathtt{type}$ is $\mathtt{state\_predicate}$,
- $p.\mathtt{actor}$ is not $\mathtt{peer}$

(a more formal definition is given in the next section.) Then let $B = LSL \setminus E$.

Note that both LSL and E are determined by the metadata, and hence B is, as well.

Another important question is: are there any pieces of *certain* information that the client may extract from a blurred program? More concretely:

- Can the client ever be sure that some credentials fulfill a request expressed as a blurred program? Then the client may prefer to send immediately such credentials, in order to minimize useless disclosure.
- Can the client detect when its credentials do not suffice to satisfy the server's request? Then the client may immediately abort the transaction, without any further unnecessary disclosure.

Fortunately, the answer to such questions in many cases is *yes*, and the reasoning needed to carry out this kind of analysis has the same complexity as plain credential selection, because reasoning boils down to computing two canonical models.

Theorem 5 *For all blurred policies Bl, let $Bl^{\max} = Bl \cup \{\mathtt{blurred}\}$ and $Bl^{\min} = Bl$. Then, for all states Σ and all sets of state predicates B,*

$$\begin{aligned}\mathsf{cmodel}(Bl^{\max}, \Sigma) &= \bigcup \{\mathsf{cmodel}(P, \Sigma) \mid \mathsf{blur}(P, B) = Bl\}\,,\\ \mathsf{cmodel}(Bl^{\min}, \Sigma) &= \bigcap \{\mathsf{cmodel}(P, \Sigma) \mid \mathsf{blur}(P, B) = Bl\}\,.\end{aligned}$$

Informally speaking, this theorem says that Bl contains *all* the information that does not depend on blurred conditions. More precisely, the policies P such that $\mathsf{blur}(P, B) = Bl$ are those that might have originated Bl; $Bl^{\min}$ captures the consequences that are true in all these possible policies P, and the complement of $Bl^{\max}$ contains the facts that are false in all possible P.

As a corollary of the above theorem, every consequence of $Bl^{\min}$ is also a consequence of the original non-blurred policy, and every atom that cannot be derived with $Bl^{\max}$, cannot be derived from the non-blurred policy either. This is what the client can deduce from Bl.

Blurring is used also to deal with delayed actions. Delayed provisional predicates must be evaluated after the response of the client, and in general cannot be understood by the client, just like private predicates. Therefore it is appropriate to treat delayed state predicates like private predicates. Nonetheless, distinguishing the two classes of predicates is useful to keep track of why their evaluation is delayed.

7 Driving filtering with metapolicies

On each party, the policy filtering process is determined by several parameters: (i) a request Req from the client, requiring a decision about access control, or portfolio information release, (ii) an access control or portfolio release policy Pol, (iii) a metapolicy $Mpol$, (iv) the current state Σ. With the exception of Req, all the parameters are local to the peer which is to make the decision. The metapolicy is evaluated against the current state, yielding the *current canonical metamodel* MM:

$$MM = \mathsf{cmodel}(Mpol, \Sigma)$$

which is inspected to read the metaproperties of rules and predicates. Policy filtering is carried out in several phases, based on the theoretical transformations introduced in Section 5 and Section 6:

1. First, all non-applicable rules and all irrelevant rules (w.r.t. the current request Req) are discarded. The remaining rules R are those that belong to

$$\mathsf{relevant}(Pol, \mathtt{allow}(Req))$$

and such that $\hat{R}.\mathtt{sensitivity} : \mathtt{not\_applicable}$ does *not* hold, that is,

$$\hat{R}.\mathtt{sensitivity} : \mathtt{not\_applicable} \notin MM\ .$$

Denote the result of the first phase with P_1.

2. Applicable, non-public rules are compiled. Let

$$\begin{array}{rcl} P_1^{prv} & = & \{R \in P_1 \mid \hat{R}.\mathtt{sensitivity} : \mathtt{private} \in MM\}\ , \\ P_1^{pub} & = & P_1 \setminus P_1^{prv}\ . \end{array}$$

The result of this phase is then

$$P_2 = P_1^{pub} \cup \mathsf{cons}(P_1^{prv}, P_1, \Sigma)\ .$$

3. The selected public rules are partially evaluated. The result of this phase is $P_3 = \mathsf{partEval}(P_2, \Sigma, E)$, where E (the set of literals to be evaluated) consists of all the literals L such that all the following conditions hold:

   - $L.\mathtt{type} : \mathtt{state\_predicate} \in MM$,
   - $L.\mathtt{type} : \mathtt{provisional} \notin MM$,
   - $L.\mathtt{sensitivity} : \mathtt{private} \notin MM$,
   - $L.\mathtt{evaluation} : \mathtt{immediate} \in MM$.

   Note that if L occurs in a rule R and $L.\mathtt{sensitivity} : \mathtt{not\_applicable} \in MM$, then R is not applicable; therefore there can be no such literal at this stage.

   The metaproperties `sensitivity` and `evaluation` associated to predicates are handled implicitly (recall that they are inherited by literals).

4. The immediate actions occurring in P_3 are executed. More precisely, let E' be the set of all literals A such that MM contains the atoms: (i) $A.\mathtt{type} : \mathtt{provisional}$, (ii) $A.\mathtt{actor} : \mathtt{self}$, (iii) $A.\mathtt{evaluation} : \mathtt{immediate}$.

   Collect and execute all actions α such that, for some literal $L \in E'$ occurring in P_3, $L.\mathtt{action} : \alpha \in MM$. As a result, the current state may change. Denote the new state with Σ'.

   Immediate actions may fail, that is, they are not guaranteed to make true all the provisional literals occurring in P_3. Then we need the next evaluation phase.

5. The local provisional literals of P_3 are evaluated against the new state Σ'. The result is

$$P_5 = \mathsf{partEval}(P_3, \Sigma', E')$$

   (E' is defined in the previous step.)

6. All state conditions whose evaluation must be deferred are blurred:

$$P_6 = \mathsf{blur}(P_5, B)\ .$$

   B is determined as specified in Section 6 as a function of E and LSL. Here LSL is the set of all literals L such that: (i) $L.\mathtt{type} : \mathtt{state\_predicate} \in MM$, (ii) $L.\mathtt{actor} : \mathtt{peer} \notin MM$.

7. Provisional state predicates that may be satisfied by the other peer are replaced with the corresponding action. More precisely, for each literal L occurring in P_6 such that: (i) $L.\mathtt{type} : \mathtt{provisional} \in MM$, (ii) $L.\mathtt{actor} : \mathtt{peer} \in MM$, (iii) $L.\mathtt{action} : \alpha \in MM$, replace L with $\mathtt{do}(\alpha)$. Let P_7 denote the result of this transformation.

8. Finally, all abbreviation predicates are anonymized by renaming them, as in [4]. Denote by P_8 the result of this last phase.

The final policy P_8 can be sent to the peer. The important properties of P_8 are:

- It contains only standard predicates (such as `credential`, `declaration`, `do`, constraint predicates, etc.), (renamed) abbreviation predicates and `blurred`. With the exception of `blurred` (whose semantics is deliberately obfuscated), the client knows how to handle all these predicates. The only non standard predicates are the abbreviation predicates that, however, come with their (filtered) definition.
- Its rules do not contain any instance of a private rule, nor any values computed from a private predicate. Delayed predicates are not evaluated, either.
- Evaluating $\texttt{allow}(\mathit{Req})$ in P_5 is equivalent to evaluating it in the currently applicable subset of the "true" policy Pol, by the theorems in Section 5.

  Phase 7 preserves the meaning of the policy, too, to the extent that the successful execution of the actions α makes the corresponding literals L true. Morever, phase 8 preserves the derivability of $\texttt{allow}(\mathit{Req})$.

  Phase 6 (blurring) may lose information. However, all the information that does not depend on blurred predicates is preserved and can be recovered from the min and max versions of the policy, as stated by Theorem 5.

  As a consequence, the final policy P_8 carries all the access control information that depends neither on non-applicable rules nor on private or delayed predicates.

After the client returns a set of credentials and/or executes a set of actions associated to the goal $\texttt{allow}(\mathit{Req})$, the private and delayed predicates occurring in P_5 can be evaluated in the new state Σ_{new} . If

$$\texttt{allow}(\mathit{Req}) \in \mathsf{cmodel}(P_5, \Sigma_{new}) \,, \tag{5}$$

then the request Req is permitted (be it a request for services, credentials, or actions).

Remark 2 *Here the assumption of policy monotonicity w.r.t. the provisional predicates whose actor is the client turns out to be important. The reason is that between the release of P_8 and the corresponding answer there may be other interactions. This happens because in general there are multiple open requests $\texttt{allow}(\mathit{Req})$ in the current state, and the two parties are free to deal with any of them in any order. Due to interleaved request handling, Σ_{new} might be a strict superset of $\Sigma' \cup \Delta$, where Σ' is the state produced in phase 4 and Δ is the set of provisional atoms made true by the client to derive $\texttt{allow}(\mathit{Req})$. Policy monotonicity guarantees that any condition derivable in $\Sigma' \cup \Delta$ is derivable also in the extended state Σ_{new}.*

8 Metapolicies for credential and action selection

When a party receives a (filtered) policy P with a goal G, it should look for a way of proving goal G using P and whatever credentials and actions (registration procedures, challenges, etc.) the party is willing to apply. For each proof of G, the set of credentials and actions occurring in the proof will be called a *candidate set*.

In general, G may have several proofs, hence multiple candidate sets. Then the party should choose one candidate, as privacy issues suggest to minimize the amount of information disclosed, and in particular the number of credentials released. In practice, the number of executed actions should be minimized, too, as many of the common actions in trust negotiation involve information disclosure.

Minimizing the number of disclosed credentials and the number of action executions is not the only criterion in this framework. Clearly, different credentials have different sensitivity, depending on the information they encode, and disclosing two "safe" credentials may be preferrable to disclosing a sensitive one.

Note that attaching privacy-related information to individual credentials and actions is just the first step. The preferences over individual entities must be extended to candidate sets.

Another important aspect arises from blurring: a proof from P^{min} guarantees that the credentials and actions in the proof suffice to satisfy the server's conditions, while the credentials and actions in a proof from P^{max} are subject to further verification on the server (the details of this verification are not known to the client). Choosing a proof from P^{max} may lead to unnecessary information disclosure; then, in some cases, a proof from P^{min} can be preferred to a proof from P^{max}.

In order to increase flexibility in candidate selection, the metalanguage of PROTUNE supports a few attributes for deriving preferences over credential and action sets.

For example, a credential c can be associated to a sensitivity level l (e.g. `low`, `medium`, `high`) with assertions of the form $c.\texttt{sensitivity} : l$. Similarly, actions can be given a cost with assertions like $action.\texttt{cost} : value$. More attributes relevant to candidate selection may be added if needed.

To compute the sensitivity and the cost of a *set* of credentials and actions, the above attributes must be combined using appropriate functions. The aggregation method can be specified with assertions like

$$\texttt{credential}(\_).\texttt{sensitivity}.\texttt{aggregation\_method} : \texttt{max}$$
$$\texttt{do}(\_).\texttt{cost}.\texttt{aggregation\_method} : \texttt{sum} \,.$$

Then a few standard selection methods can be selected with the attributes of a reserved entity `negotiator`, e.g.:

$$\texttt{negotiator}.\texttt{selection\_method}{:}\texttt{order}(\texttt{sensitivity}, \texttt{cost})$$
$$\texttt{negotiator}.\texttt{selection\_method} : \texttt{certain\_first}$$

The first assertion states that the main preference ordering is by sensitivity, and the secondary is cost (the list of parameters may be longer if needed). The second assertion forces the negotiator to try the candidates extracted from P^{min} before trying those extracted from P^{max} (because the former are guaranteed to satisfy the condition G).

This works for the simplest cases. In general, since the nature of sensitivity and costs is application dependent, it may be necessary to define ad-hoc comparison criteria using the rule language. The standard selection method can be replaced with an ad-hoc predicate P by means of the assertion: $\texttt{negotiator.selection\_method} : \texttt{adopt}(P)$.

Another important feature of PROTUNE is the support of *metalevel constraints*. They are like metapolicy rules without head, like $\leftarrow L_1, \ldots, L_n$.

Such constraints are *satisfied* w.r.t. a (meta)policy Pol and a state Σ, iff no ground instance of $\{L_1, \ldots, L_n\}$ is contained in $\mathsf{cmodel}(Pol, \Sigma)$.

Constraints are very useful in identity protection. It is well known that simple combinations of individual attributes (such as birth date and zip code) may disclose a user's identity. In the framework of trust negotiation, this means that some combinations of credentials, $\{c_1, \ldots, c_n\}$, should never be disclosed.

Such directives can be expressed with constraints of the form: $\leftarrow \texttt{credential}(c_1, \_), \ldots, \texttt{credential}(c_n, \_)$.

More precisely, the disclosure decision procedure, given a candidate set Δ of credentials and actions (sufficient to prove G) checks whether all the release constraints are satisfied w.r.t. the local metapolicy $Mpol$ and the state $\Sigma \cup \Delta$. If some constraint is violated in this context, then the candidate Δ is discarded.

9 Monitoring policies with constraints

Metalevel constraints may also be used to monitor policies and metapolicies at runtime. By checking contraints at each state change, one can detect conflicts and inconsistencies in the specification. This is particularly important when metapolicies consist of nontrivial rules; then statically checking that for *all* states the consequences of the metapolicy are meaningful may be computationally too hard.

Below is an example of a monitoring constraint. It verifies that no action is associated to more than one actor:

$$\leftarrow X.\texttt{action} : A,\ A.\texttt{actor} : Y,\ A.\texttt{actor} : Z,\ Y \neq Z\ .$$

If ad-hoc actions (e.g. logging) are to be executed upon constraint violations, then it suffices to include suitable provisional atoms in the constraint

10 Distributed credentials

Credentials need not be stored at their owner's site nor at their issuer's. Moreover, there is no unique way of searching for a credential, and the responsibility of the search may be of the server, of the client, or even shared [8]. Therefore, in general, the following entities are distinct: (i) the credential issuer, (ii) the credential repository, (iii) the credential owner, and (iv) the actor(s) responsible for fetching the credential.

The issuer is encoded in the credential, and ownership can be checked via challenges. The remaining two properties are encoded with suitable metaattributes:

- $Credential.\texttt{location} : URI$
- $Credential.\texttt{actor} : X$

where X can be `self`, `peer`, or a reference to a third party credential collection service.

If the actor is `peer`, then the credential is not evaluated; it is sent to the other peer who shall decide whether to fetch it (if necessary) and disclose it.

If the actor is `self`, then the local engine has to fetch and verify the credential. Search may be nontrivial, in general it may require navigation through several servers [8].

Finally, if the actor is a reference to a third party service, then the local engine has to call the service and verify the returned credential (if any).

Note that whenever the actor is not `peer`, the local engine has to perform some actions. Their execution time can be immediate or delayed, like the execution of any other local provisional predicate. Credential collection, however, may be significantly slow, because it involves internet navigation. PEERTRUST optimizes such distributed computations by sending out credential requests in parallel and then using the results as they arrive. In PROTUNE we enable parallelized search for specific credentials C by asserting

$$\texttt{credential}(C, \_).\texttt{evaluation} : \texttt{concurrent}\ .$$

More precisely, for all credentials whose actor is not `peer`,

- if the `evaluation` attribute is `immediate`, then the credential is fetched and verified in phase 5; the filtering process is suspended until all immediate credentials have been fetched and verified;
- if the `evaluation` attribute is `delayed`, then the credential is fetched and verified after the client's response; this procedure has the advantage of focussing search only on those credentials that together with the client's credentials prove the server's request;
- if the `evaluation` attribute is `concurrent`, then credential search starts at phase 5 and proceeds in parallel with filtering; credentials are verified as they are received.

Roughly speaking, the concurrent method is a sort of prefetch strategy that may shorten the response time in some applications. A general treatment of the `concurrent` modality (extended to user-defined predicates) can be easily integrated in the negotiator. It is not described here due to lack of space.

11 Libraries and language extensions

Untrained users may find it difficult to formulate autonomously appropriate metapolicies. Such users would benefit from a library of standard metapolicies that protect their access control policy from the most common forms of information leakage.

Abbreviation libraries constitute also a means for language extensions, which is of great importance in a growing field like trust management. In [3] it is shown how to encode the semantics of the four types of RT_0 credentials [7] with a small, simple PROTUNE library.

Note that libraries of this kind consist of logical axioms defining predicates and credential meaning with a small set of shared symbols. In fact, such libraries are nothing but *ontologies*. The fact that shared symbols are few and well identified makes the task of building shared ontologies much easier; consider that plain X.509 credentials suffice to define an incredibly rich set of policies and user categories.

Abbreviations and credentials can be linked to the ontologies that specify their meaning by means of a suitable metaattribute: $Obj.\texttt{ontology} : URI$.

12 Related and Future Work

The Ponder system [16] has metapolicies but their purpose is different as there are no negotiations. Ponder's metapolicies restrict the kind of rules that may occur in a policy, thereby addressing issues such as static separation of duties and conflict avoidance. LGI [17] has action predicates. However, while LGI actions *must* be executed, PROTUNE's actions *may*; whether an action is executed or not depends on the overall decision making process behind negotiations.

Our metalanguage is currently very effective in specifying policy filtering and credential search. It should be extended to cover other important negotiation decisions, namely, the choice of the open requests to be handled at each step. Moreover, ECA rules should be integrated in this framework, to extend the set of business rules that can be directly supported. We are working at an implementation based on PEERTRUST.

Acknowledgements. Research partially supported by the network of excellence REWERSE , IST-506779.

References

[1] C. Baral. Knowledge representation, reasoning and declarative problem solving. Cambridge University Press, 2003.

[2] Bell, J., Brooks, D., Goldbloom, E., Sarro, R., Wood, J.: Re-Engineering Case Study Recommendsations to Application Developers. US West Information Technologies Group. Bellevue Golden (1990)

[3] P.A. Bonatti, D. Olmedilla. Policy Language Specification. REWERSE Deliverable I2-D2, Feb. 2005. `http://rewerse.net/deliverables.html`

[4] P.A. Bonatti, P. Samarati. Regulating Service Access and Information Release on the Web, *Proc. of the Seventh ACM Conference on Computer and Communications Security*, 2000.

[5] P.A. Bonatti, P. Samarati. Logics for authorizations and security. In J. Chomicki, R. van der Meyden, G. Saake (eds.) Logics for Emerging Applications of Databases, Springer-Verlag, August 2003

[6] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. Seamons, M. Winslett. No Registration Needed: How to Use Declarative Policies and Negotiation to Access Sensitive Resources on the Semantic Web. 1st First European Semantic Web Symposium, 2004

[7] N. Li, J.C. Mitchell, W. Winsborough. Design of a Role-based Trust-management Framework. IEEE Symposium on Security and Privacy, 2002

[8] N. Li, W. Winsborough, J.C. Mitchell. Distributed Credential Chain Discovery in Trust Management. Journal of Computer Security, 11(1), 2003

[9] J.W. Lloyd. *Foundations of logic programming*, Springer-Verlag, 1984

[10] K. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobsen, H. Mills and L. Yu. Requirements for Policy Languages for Trust Negotiation. 3rd International Workshop on Policies for Distributed Systems and Networks, 2002

[11] V.S. Subrahmanian, Sibel Adali, Anne Brink, James J. Lu , Adil Rajput, Timothy J. Rogers, Robert Ross, Charles Ward. HERMES: Heterogeneous Reasoning and Mediator System. `http://www.cs.umd.edu/projects/hermes`

[12] Taveter, K., Wagner, G.: Agent-oriented enterprise modeling based on business rules. In: Proc. of 20th Int. Conf. on Conceptual Modeling (ER2001). LNCS, Springer-Verlag (2001)

[13] Wagner, G.: How to design a general rule markup language? In: XML Technologien für das Semantic Web - XSW 2002, Proceedings zum Workshop, 24.-25 Juni 2002, Berlin. (2002)

[14] M. Winslett and N. Ching and V. Jones and I. Slepchin. Assuring Security and Privacy for Digital Library Transactions on the Web: Client and Server Security Policies. Proceedings of ADL '97 — Forum on Research and Tech. Advances in Digital Libraries, 1997

[15] T. Yu, M. Winslett and K. Seamons. Interoperable Strategies in Automated Trust Negotiation. ACM Conference on Computer and Communication Security, 2001

[16] N. Damianou, N. Dulay, E. Lupu, M. Sloman. The Ponder Policy Specification Language, In Proc. of *IEEE ComSoc Workshop on Policies for Distributed Systems and Networks* (Policy 2001), LNCS 1995, p.18-38, Springer, 2001

[17] N. Minsky, V. Ungureanu. Law-Governed Interaction: A Coordination & Control Mechanism for Heterogeneous Distributed Systems. *ACM Transactions on Software Engineering and Methodology* (TOSEM) 9(3):273-305, 2000