

# Protune

J. L. De Coi, D. Ghita, D. Olmedilla

September 18, 2006

## 1 Protune Negotiation Model

This document aims at formalizing the negotiation between two entities, namely  $E_1$  and  $E_2$ . For the rest of the paper we assume that  $E_1$  is the initial requester while  $E_2$  is the provider, i.e.  $E_1$  is assumed to send a request to  $E_2$  thus starting a negotiation. Notice that during a negotiation both entities  $E_1$  and  $E_2$  may both request or provide information to each other.

Let our Policy language be a rule-based language. Such a 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, i.e.  $L_i$  equals either  $B_i$  or  $\neg B_i$ , for some logical atom  $B_i$ .

**Definition 1 (Policy)** *A Policy is a set of rules, such that negation is not applied to any predicate occurring in a rule head.*

This restriction ensures that policies are *monotonic* in the sense of [?], i.e. 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 [?].

**Definition 2 (Negotiation Message)** *A Negotiation Message is an ordered pair*

$$(p, C)$$

where

- $p \equiv$  a Policy
- $C \equiv$  a set of credentials

We will denote with  $M$  the set of all possible Negotiation Messages.

**Definition 3 (Message Sequence)** *A Message Sequence  $\sigma$  is a list of Negotiation Messages*

$$\sigma_1, \dots, \sigma_n \mid \sigma_i \in M$$

We will denote with  $|\sigma|$  the length of  $\sigma$  and with  $\sigma_i$  the  $i$ -th element of the Message Sequence  $\sigma$ .

**Definition 4 (Negotiation History)** Let  $E_1$  and  $E_2$  be the two entities involved in the negotiation. Let  $E_1$  be the initiator of such a negotiation, i.e. the sender of the first message in the negotiation. A Negotiation History  $\sigma$  for the entity  $E_j$  ( $j = 1, 2$ ) is a Message Sequence

$$\sigma_1, \dots, \sigma_n \mid \sigma_i \in M$$

Moreover let

- $M_{snt}(\sigma) = \{\sigma_i \mid i = 2k - (j \bmod 2), 1 \leq k \leq \lfloor n/2 \rfloor\}$
- $M_{rvd}(\sigma) = \{\sigma_i \mid i = 2k - 1 + (j \bmod 2), 1 \leq k \leq \lfloor n/2 \rfloor\}$

We will refer to a Negotiation History also as a *Negotiation State*.

Intuitively, messages among parties are sent alternatively, i.e. a message sent by  $E_j$  is followed by the reception of a message, which is in turn followed by the sending of a new message and so on. Therefore,  $M_{snt}(\sigma)$  (resp.  $M_{rvd}(\sigma)$ ) represents the set of messages sent (resp. received) by  $E_j$ .

Notice that according to this definition, the Negotiation History  $\sigma$  is shared by the two entities  $E_1$  and  $E_2$ , but the sets  $M_{snt}(\sigma)$  and  $M_{rvd}(\sigma)$  are swapped among them. Therefore it holds that

$$M_{snt}(E_1, \sigma) = M_{rvd}(E_2, \sigma)$$

and

$$M_{rvd}(E_1, \sigma) = M_{snt}(E_2, \sigma)$$

In order to ease the notation in the rest of the paper, given a Negotiation History  $\sigma$ , we define the following entities

- $C_{snt}(\sigma) = \bigcup \{C_i \mid \exists p_i (p_i, C_i) \in M_{snt}(\sigma)\}$
- $C_{rvd}(\sigma) = \bigcup \{C_i \mid \exists p_i (p_i, C_i) \in M_{rvd}(\sigma)\}$
- $lp_{snt}(\sigma) = p_{i_{max}} \mid i_{max} = \max\{i \mid (p_i, c_i) \in M_{snt}(\sigma)\}$
- $lp_{rcv}(\sigma) = p_{i_{max}} \mid i_{max} = \max\{i \mid (p_i, c_i) \in M_{rcv}(\sigma)\}$

Intuitively,  $C_{snt}$  (resp.  $C_{rvd}$ ) represents the set of all credentials sent (resp. received) and  $lp_{snt}$  (resp.  $lp_{rvd}$ ) represents the last policy sent (resp. received).

**Definition 5 (Negotiation State Machine)** A *Negotiation State Machine* is a tuple

$$(\Sigma, S, s_0, t)$$

such that

- $S \equiv$  a set of Negotiation States
- $s_0 \equiv$  the empty list (initial state)

- $\Sigma \equiv$  a set of Negotiation Messages.
- $t \equiv$  a function  $S \times \Sigma \rightarrow S$  such that if  $S = (\sigma_1, \dots, \sigma_n)$  then  $t(S, \sigma) = (\sigma_1, \dots, \sigma_n, \sigma_{n+1})$  (transition function)

Intuitively a Negotiation State Machine models how an entity evolves during the negotiation by the exchange of messages.  $\Sigma$  contains both sent and received Negotiation Messages and can therefore be partitioned into two subsets  $\Sigma_{snd}$  and  $\Sigma_{rcv}$ .

**Definition 6 (Negotiation Model)** A Negotiation Model is a tuple  $(C, P, p_0, NSM, ff, ns)$  where

- $C \equiv$  a set of credentials
- $P \equiv$  a set of Policies
- $p_0 \equiv$  a Policy (local Policy)
- $NSM \equiv$  a Negotiation State Machine  $(\Sigma, S, s_0, t)$
- $ff \equiv$  a function  $S \rightarrow P$  (Filtering Function)
- $ns \equiv$  an ordered pair  $(csf, ta)$  where
  - $csf \equiv$  a function  $S \rightarrow C$  (Credential Selection Function)
  - $ta \equiv$  a function  $S \rightarrow \{true, false\}$  (Termination Algorithm)
 (Negotiation Strategy)

Each occurrence of  $S$  is supposed to refer to the same set of Negotiation States.

The intended meaning is as follows

- $C$  represents the set of the credentials local to the Peer
- $p_0$  represents the Peer's policy protecting the local credentials and allowing access to the local resources
- $S$  represents the set of states in which the Peer can be
- $s_0$  represents the initial state, i.e. the state in which the Peer is at the beginning of the negotiation
- $f$  represents the process of filtering the Peer's Policy according to the current state
- $csf$  represents the process of selecting the Peer's credentials to send to the other Peer
- $ta$  represent the Peer's decision about whether going on or terminating the current negotiation

## 2 Protune to Prolog Rewriting Rules

Translation is needed between the Protune representation and the implementation languages used for the inference. Currently we have chosen Prolog as implementation language and the following are the translation rules used in order to transform Protune rules into Prolog rules.

The Protune to Prolog parser, implemented in JavaCC, receives as input a policy written in Protune. It processes each directive, rule and metarule contained in the policy given and then, the rules and the metarules are translated to a Prolog representation. Finally, it stores the directives and the translated rules and metarules in separate vectors.

General rules are transformed as follows:

$$\begin{aligned}
 [Id]Head \leftarrow L_1, \dots, L_n. \\
 \longrightarrow \\
 rule(Id, Head, [L_1, \dots, L_n]).
 \end{aligned}$$

If the rule does not have a body then the third argument of the rule predicate, which is a list, is empty. Identifiers for rules written in Protune are optional, they can be specified or not. However during the filtering, it is essential that each rule has an identifier in order to distinguish among them while processing the policy. The translator is then responsible for assigning default different identifiers to each rule missing one.

Metarules are transformed as follows:

$$\begin{aligned}
 [Id].Attribute : Value \leftarrow L_1, \dots, L_n. \\
 \longrightarrow \\
 metarule(id, Attribute(Id, Value), [L_1, \dots, L_n]). \\
 Head.Attribute : Value \leftarrow L_1, \dots, L_n \\
 \longrightarrow \\
 metarule(pred, Attribute(Head, Value), [L_1, \dots, L_n]).
 \end{aligned}$$

If a metarule is without a body, then the third argument of the metarule predicate in the Prolog representation, which is a list, is empty. When translating metarules, two particles are introduced “id” and “pred” in order to mark that each metarule refers to a rule using the rule id or to a predicate using a literal which unifies with it. This is necessary as an id which is a simple string constant is identical with a predicate without any arguments, so it can be ambiguous.

Special care must be taken when translating complex terms which in general are transformed as follows:

$$\begin{aligned}
 Id[Attribute_1 : Value_1, \dots, Attribute_n : Value_n] \\
 \longrightarrow \\
 complex\_term(Id, Attribute_1, Value_1), \dots, complex\_term(Id, Attribute_n, Value_n).
 \end{aligned}$$

However, the translation is different depending on the position of a rule where the complex term occurs:

- It is in the head of a rule:

$$\begin{aligned}
& [RId]CId[attribute_1 : Value_1, \dots, attribute_n : Value_n] \leftarrow L_1, \dots, L_n. \\
& \longrightarrow \\
& \quad rule(RId, complex\_term(CId, attribute_1, Value_1), [L_1, \dots, L_n]). \\
& \quad \dots \\
& \quad rule(RId, complex\_term(CId, attribute_n, Value_n), [L_1, \dots, L_n]).
\end{aligned}$$

- It is an argument of a predicate that is a rule head:

$$\begin{aligned}
& [RId]pred(CId[attribute_1 : Value_1, \dots, attribute_n : Value_n]) \leftarrow L_1, \dots, L_n. \\
& \longrightarrow \\
& \quad rule(RId, pred(CId), [L_1, \dots, L_n]). \\
& \quad rule(RId, complex\_term(CId, attribute_1, Value_1), [L_1, \dots, L_n]). \\
& \quad \dots \\
& \quad rule(RId, complex\_term(CId, attribute_n, Value_n), [L_1, \dots, L_n]).
\end{aligned}$$

- It is in the body of a rule:

$$\begin{aligned}
& [RId]pred() \leftarrow L_1, \dots, CId[attribute_1 : Value_1, \dots, attribute_n : Value_n], \dots, L_n. \\
& \longrightarrow \\
& \quad rule(RId, pred(), [L_1, \dots, complex\_term(CId, attribute_1, Value_1), \\
& \quad \dots, complex\_term(CId, attribute_n, Value_n), \dots, L_n]).
\end{aligned}$$

- It is an argument of a predicate in the body of a rule

$$\begin{aligned}
& [RId]pred_1() \leftarrow L_1, pred_2(\dots, CId[attribute_1 : Value_1, \dots, attribute_n : Value_n]), \dots, L_n. \\
& \longrightarrow \\
& \quad rule(RId, pred_1(), [L_1, \dots, pred_2(CId), complex\_term(CId, attribute_1, Value_1), \\
& \quad \dots, complex\_term(CId, attribute_n, Value_n), \dots, L_n]).
\end{aligned}$$

Transformations for complex terms which appear in metarules are similar to the ones described above but complex terms cannot appear in the head.

In order to make all these translations possible when a complex term is discovered in the input stream, its elements are stored in a vector and this vector is passed along as part of the object returned by a method. For example, in case the complex term appears in a predicate in the body of the rule, the id of the complex term is stored as a predicate argument and the elements of the complex term are passed along and placed after the predicate. If the complex term appears in the head of a rule then the elements of the complex term need to be passed along until the entire rule is matched. In the case where more complex terms appear as arguments of the same predicate, then their elements are merged together in a single vector and then passed along.

Another interesting issue is translating special built-in predicates. The translation is performed as follows:

$$\begin{aligned}
& in(pred(Arg_1^1, \dots, Arg_n^1), package : function(Arg_1^2, \dots, Arg_m^2)). \\
& \longrightarrow \\
& in(pred(Arg_1^1, \dots, Arg_n^1), package, function, [Arg_1^2, \dots, Arg_m^2]).
\end{aligned}$$

The arguments for the package call function are gathered in a list, so is be easier to integrate modules for making package calls.

The start method of the parser which must match the main BNF production, for the entire policy, is extended. This happens as in the policy rules and metarules can be mixed and have common tokens. Since identifying the start of a rule or a metarule is frequently done while translating, we preferred to factor together the rule and the metarule syntax instead of using local lookahead, in order not to slow down the transformation process. Local lookahead is also used in other situations where factorization of rules would have been difficult and would have made the grammar not welcome future changes very well. It must be mentioned as well that even though the rules referring to literal and metaliteral have common parts, they were implemented separately in order to favor factorization over local lookahead. An important feature of the translator is that while transforming the Protune policy in the Prolog representation the semantics of the policies is not changed, only the syntax is modified.