University "Politehnica" of Bucharest
Faculty of Automatic Control and Computer Science

Hannover University, L3S Research Center

Policy Adaptation and Exchange in Trust Negotiation

Denisa Ghita
gdenisa@gmail.com
September 4, 2006

***Supervisors**:*
Prof. Dr. Valentin Cristea
Prof. Dr. Wolfgang Nejdl
Dipl. Inf. Daniel Olmedilla

Typeset by the author with the LaTeX 2ε Documentation System.

Contents

1 Introduction **2**
Overview 5

2 Motivation **7**
2.1 Example 7
2.2 Problems and Solutions 8

3 Policy-Driven Negotiations **11**

4 Protune Language Specification **13**
4.1 The Syntax of Protune 13
4.1.1 Complex terms 14
4.1.2 Predicates 14
4.1.3 Metapolicy 21
4.2 Policy Example 26

5 Protune Negotiation Framework **31**
5.1 Negotiation Model 31
5.2 Policy Filtering 38
5.2.1 Policy Filtering Steps 38
5.2.2 Filtering Example 45

6 Implementation **54**
6.1 Architecture 54
6.2 Protune-Prolog Translators 54
6.2.1 JavaCC 54
6.2.2 Protune to Prolog Translator 58
6.2.3 Prolog to Protune Translator 62
6.3 Policy Filtering 63
6.4 Credential Selection 72

7 Conclusions and Open Issues **75**

References **78**

A Protune grammar **80**

B Policy and metapolicy **84**

C Monotonicity of the filtering process **94**

1 Introduction

The majority of the people agree that the Internet and the World Wide Web, which is accessible via the Internet, are among the greatest discoveries of the past century. As the Internet evolves, the need for improvement of service offerings over the Internet becomes obvious. People want to be able to make online transactions in such a manner that their security and privacy are ensured. Therefore, a general system, that provides both security and privacy to each user, for managing sensitive business interactions carried out in the Internet between strangers is required.

Every business interaction consists of at least two entities: the client, who requires a service or a certain good, and the service provider, who is able to fulfill the necessity of the client. The above notions extend the concepts of buyer and seller used in every-day life as a client may not have to pay for a service which the service provider offers free of charge under some conditions.

The most common method employed for closing a transaction between strangers is face-to-face purchase. For example, people go in bookstores to buy books or in supermarkets to buy food. In this case, the buyer can pay cash for the goods he[1] wants. This is believed to be a safe transaction as the buyer can see and even touch the good before paying for it and the seller receives the money immediately and can use it at his own will. Some stores even allow the client to return the good within a certain period of time if he is not satisfied or they offer a guarantee certificate providing support in case the good should get damaged. Furthermore, the privacy of the client is ensured by the fact that no information regarding his identity is stored and usually, the seller does not know the client or any personal information about him.

Since 1950, credit cards have been introduced as another approach of paying for goods or services. Although the credit card system presents opportunities for fraud, it is widely spread and, in most developed countries, paying via credit card is more frequent than paying cash. Face-to-face purchase via credit card is comfortable and offers several methods for protection. Each credit card has a hologram against forgery attacks. For each purchase, the credit card is checked if it has not expired or if it has not been revoked, so stolen or invalid credit cards cannot be used. Also, the credit on the card is checked if it covers the price. Sellers request buyers to prove the credit card belongs to them, they ask the buyer to introduce a pin or to sign the receipt which is compared with the signature on the back of the card. In what concerns privacy, credit card companies have information about every

[1]Throughout the thesis, he is to be interpreted as he or she

transaction made via credit card by any of their clients and they may use this information for their own purposes.

Although far from perfect, face-to-face business transactions via cash or via credit card work pretty good. In general, people feel more free from risk of loss by taking this option instead of using online transactions.

Transactions over Internet can save time and allow access to a larger market with more opportunities. There already exist online purchases via credit cards but they offer low security especially for the client who has to unconditionally disclose his credit card information to the service provider who in many cases is unknown to the client.

In order to transform a face-to-face business transactions into an online transaction, some requirements must be fulfilled as stated in [15]. First, a digital version of a credit card is needed. Both the client and the service provider must be able to interpret correctly the information in the digital representation. The digital credit card must be unforgeable and checked for validity for each purchase. Second, the service provider must specify rules expressed by means of a policy about which credit cards he can accept and how he deals with them. The policy should contain:

- credit card issuers that are trusted
- how the client can demonstrate the ownership of the credit card
- how to check that the credit card has not expired
- how to check that the credit card has not been revoked and that the new charges do not exceed the current credit on the card

Third, in order to support automatic transactions, the client should also have a policy stating under what conditions he is willing to show his credit card. This helps the client avoid blindingly providing his credit card information to any service provider. For example, someone may only want to show the credit card to sellers trusted by his bank. Finally, a protocol is needed such that the buyer and the seller can establish trust and successfully close the transaction.

Business interactions do not necessarily require that the client should buy goods from a seller, a client may also access a service if he can prove he has the authorization asked by the service provider. For example, students may have free access to online books from a certain library while regular clients must pay a fee for this right. In this case, students need only provide a student membership card to gain access. Therefore, digital representation for credentials, not only for credit cards, is required.

Some credentials, like credit cards, are sensitive resources and, as discussed earlier, the owner of the credential must express in a policy conditions under which credentials can be released.

Service providers need policies to specify different methods of getting access to their services and means by which credentials received from clients are to be checked and processed.

Finally as described in [11], the client and the service provider must negotiate in order to make a transaction. This means that a level of trust must be established between the client and the service provider so that the transaction can be performed. Trust negotiation is an iterative process based on policies. Parties exchange credentials and policy requirements until they manage to establish the certain level of trust needed to close the transaction or discover that this cannot be done. For example, a client may have a policy stating he discloses his credit card only to service providers which are trusted by Visa. When he negotiates for buying a good over the Internet and he is requested to pay, he asks the service provider to prove he is trusted by Visa. The service provider may have a credential stating this and does not mind showing it to clients. He sends it to the client and the client sends his credit card.

Of course, issues related to autonomy, scalability and robustness against attacks are still to be considered. Basic needs for trust negotiation as identified in [15] include

- expressive languages for writting policies
- tools for inferring information from a policy for each trust negotiation
- representation and storage of credentials
- strategies for establishing trust [e.g., decisions about disclosure of credentials]
- automatic trust establishment, such that trust is established by agents acting on behalf of users and is transparent to the users

This thesis provides solutions to some of this aspects. First, "Protune", a language for writing policies which is also able to represent credentials, is described. Then, the behavior of tools developed for inferring information from a policy is explained. At last, a general negotiation model for trust establishment is provided.

Overview

The first section contains a general introduction to transactions over the Internet and states how security and privacy integrate in this domain. Also, it gives a short explanation of trust negotiation and describes the requirements needed in order to support trust establishment between two parties.

Section 2 talks more in detail about the need of trust negotiation in today's Internet. It provides an example and, based on it, the existing problems are identified and a brief overlook on how they can be solved is given. It discusses security based on identities versus security based on properties and bilateral security in contrast to unilateral security. In addition, comments about the need for explanations during negotiations is included.

In section 3, the purpose of policies in what concerns transactions over the Internet is discussed. It contains some basic information about what a policy is and what it can be used for. Also, it introduces the concept of metapolicy, which provides additional information on how the policy should be handled. Finally, it explains how policies can be used in trust negotiation in peer-to-peer networks and compares security and privacy in traditional environment with policy-driven negotiations.

Section 4 contains the specifications for Protune, a language especially developed for writing policies. It starts by presenting general features of the language and then, the whole syntax is explained: the notion of complex terms, the different types of predicates andthe attributes and their values predefined in the metapolicy. At last, a policy written in Protune together with its associated metapolicy are presented and commented.

Section 5 presents the overall framework and gives an overview on how different parts of the application interact. It includes a formalization of a general negotiation model that describes the behavior of one party during the negotiation process. This introduces the notions of state, message, transitions, strategy, credential selection and termination criterion in regards to negotiations. Also, it explains in detail the filtering process which is applied to the policies. Formalizations and examples are provided for each step of the filtering.

In section 6 the implementation is explained in detail. First, an overview of the programming languages and the platforms used is given. Then, the architecture of the whole application and how the different parts are going to interact are specified. The translators from Protune to Prolog and from Prolog to Protune for transforming policies are described. It contains the Prolog implementation of the filtering process applied on policies. Finally, it specifies how selection of credentials that need to be sent to the other party is implemented.

At last, section 7 contains conclusions upon the current work and possible future work that is still to be done.

2 Motivation

A new challenge for the Semantic Web is to address the issue of access control for sensitive resources such as services in peer-to-peer architectures.

Currently, in traditional distributed environments, service providers and requesters find out about each other via shared information in the environment, which describes the services offered by providers and the clients that are entitled to make use of the services. In this case, trust between parties is based on previously published conditions for getting access to services. In the traditional client-server model, the only question is whether the server should trust the client. This is handled by creating client accounts: clients have to register and then sign in each time they want to access the service. Immediately after signing in, based on their usernames, users are mapped to permissions and a decision is taken regarding whether they can access the requested service or not.

However, as mentioned in [11] and in [9], the Semantic Web is different from traditional based systems. In the Semantic Web parties can interact without previously having known each other. In this case, none of the approaches described above is suitable.

2.1 Example

To understand how access control can be handled in the Semantic Web, first an example about how online purchases are done so far is provided. Let us consider Bob, a student, and assume he needs a book. Bob is on holidays, visiting his parents in a very small village, and he cannot find the book in the only bookshop in town. He decides to look for the book in Internet. He uses a search engine and discovers a site `http:\\www.ebookstore.com` from where he can download the digital format of the book for a certain price. The price is not so high and since he needs the book, he would like to buy it. He goes to the web site but in order to buy the book he needs to sign in. However, this is the first time Bob visits this site so he does not have an account; his alternative is to register. Apart from having to fill in a very long form, the registration also requests some private information disclosure from Bob which is irrelevant for his purchase. Why should they need his telephone number or his physical address if he only wants to download the book? The information the site requests for registration is equivalent with having the client disclose his identity card. In a real bookstore, they do not ask you to provide this information and if this should happen, probably the clients would not feel very comfortable with it. Bob really wants to buy the book so he decides to go through the registration process. He knows the web site is

not entitled to ask him all that information and he doesn't feel secure to give his home address and telephone to a site he has never heard of before, so he fills in the form with fake information as the bookstore is not able to check it anyway. The registration is now over, but a bigger surprise awaits for Bob: he can only pay by credit card. Access to his credit card information means possibility to extract money from his account. At this point, Bob has two choices: provide his credit card data and hope to be among the fortunate people that do not get fooled or not give away his credit card information which means he does not get the book. Bob decides the risk is too high so he chooses not to provide the credit card and abandons the transaction.

2.2 Problems and Solutions

It is clear that the transaction described above was not a successful one. We need to analyze the situation in order to identify the problems and find solutions.

The web site uses traditional security which expresses access control through statements like

> "USER is allowed
> to perform ACTION
> on RESOURCE"

This means requesters are previously known through registration and identified at sign in so they can be mapped to permissions. As explained in [11], identity does not scale on the Web and is many times not necessary, what matters are some properties like if the requester is a student in order to get a discount, if he is an employee or if he belongs to an association. Security decisions could be based only on properties which can be proved via digital credentials. Therefore, even Bob's name becomes irrelevant and is not needed for closing the transaction.

Another problem is that only the user needs to fulfill conditions and provide information and the server accepts it or not. A bilateral approach would be better then a unilateral one, as observed in [11]. Users would be allowed to specify conditions that providers must fulfill as well. This leads to trust negotiations that have an agreement as a goal.

Let us consider the example again but this time assume that `http:\\www.ebookstore.com` is opened to trust negotiation. The web site has a policy which states that in order to be able to download the book, the client must provide a credit card for charging the fee and, optionally, a student card in

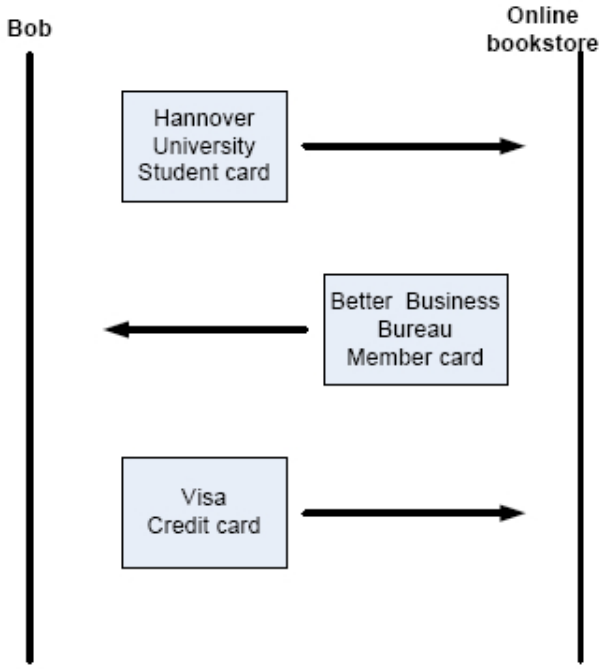

Figure 1: Trust Negotiation between Bob and the online bookstore

order to get a discount for the book. Bob still needs to provide the credit card so what is the improvement? Bob does not need to provide his credit card blindly. He has no problem with disclosing his student card but he will only disclose his credit card to members of the Better Business Bureau. Therefore, he sends the book store his student card and the policy protecting his credit card. The online book store has a credential that proves it is a member of the Better Business Bureau and it does not mind disclosing it to its clients, so it sends this credential to Bob and in reply Bob sends his credit card information. The website charges Bob for the book he wants and then Bob can download the book. This approach, based on trust negotiation, makes the transaction succeed and both Bob and the book store do not feel that their security is threatened. The trust negotiation done in this case is depicted in figure 1.

Trust negotiation does not imply that the transaction will always succeed, it may as well fail if the conditions stated in the policies are not satisfied. After having provided some credentials the negotiation can fail and the peer that provided the credentials might get frustrated if he or she does not understand why this happened. A possible solution for this situation is including explanations in policies, as described in [4].

For example, the online book store might not accept Bob's credit card as it is not a Visa card and it had specified in the policy that it can only be a card issued by Visa. Bob needs to know that the negotiation failed for this reason.

As stated in [4] a user may want to ask why the negotiation failed or how he can get access to a service or maybe he wants to know what will happen in case he would provide a certain type of credential. This questions can all be answered thanks to explanations generated from policies.

In summary, in the Semantic Web trust must be established from a zero-knowledge base. Trust can be established by having the two parties involved exchange information like digital credentials. The trust establishment process should be bi-directional as both parties may hold sensitive information which they do not feel comfortable to share unless the other party proves to be trustworthy.

Furthermore, trust negotiation allows for additional features which cannot be expressed in traditional security, like specifying business rules, e.g., discounts for students, or providing explanations for clients.

3 Policy-Driven Negotiations

As described in [10], Trust management is the activity of collecting, encoding, analyzing and presenting evidence relating to competence, honesty, security or dependability with the purpose of making assessments and decisions regarding trust relationships".

Currently, main approaches in managing trust are policy-based and reputation-based as observed in [5]. A policy-based trust management approach deals with negotiations that finish with a boolean decision, whether the requester is trusted or not. In reputation-based systems, trust is computed from feedback provided by the entities in the network.

A policy specifies the behavior of a system, it can describe how decisions are taken and how actions are executed, and it plays a crucial role in security, privacy and service usability. For example, access control policies protect open systems on the Internet and privacy policies help users interact with web services.

Most of every-day systems adopt policies formulated as rules: "If event then system_reaction" where event describes the event that can happen and system_reaction specifies the system behavior in response to that event. In conclusion, rule-based languages are, currently, the best approach for policies, which are required for trust negotiation.

As described in [5], trust management should focus on different policy languages and engines in order to specify and reason with rules for establishing trust. Based on declarative policies and exchange of credentials, a negotiation decides whether an unknown user is trusted or not.

However, in order to be able to reason with policies, additional information is needed stating how a policy is to be handled. As stated in [7], this is the purpose of the metapolicy, which is associated with a policy and contains information about how a policy should be handled. Metapolicies consist of rules that make reference to the elements, like rules and predicates, defined in the policy.

As seen in the previous section, the disadvantages of traditional security, as explained in [11], are

- The client must unconditionally disclose information to the server or else he does not get access to the service. The clients cannot state their own conditions that the service providers have to fulfill, so the clients cannot know if the server can be trusted.

- The service provider has no means to verify if the information provided by the client in the registration form is valid

- The information required for registration may not be related to the service the client want to get access to, but the client has no other choice than to provide it if he wants access to the service.

Trust negotiation is an iterative process where trust is established gradually by disclosing credentials and policies for credentials. The main aspects of trust negotiation as identified in [11] are

- Trust between strangers is established based on properties, not identities. Properties can be proved via digital credentials.
- Every peer may define a policy for access control to its sensitive resources. Service providers can specify policies with conditions for getting access to the services or the resources they offer. Clients can specify policies for describing under which circumstances a certain credential is to be released. Of course, service providers may also have policies protecting their credentials.
- Two strangers can establish trust directly without involving a third trusted party responsible for match-making.

Trust negotiation is conducted by security agents that act on behalf of peers. Peers only need to define a policy to specify the conditions under which credentials can be released or how access to services and to resources can be obtained.

4 Protune Language Specification

Protune, PROvisional TrUst NEgotiation, first defined in [7], is a language especially developed for writting different types of policies, like access control policies or privacy policies. An access control policy enables one to specify ways of accessing resources or services, for example, it can state how a user can download a mp3 file from a server. A privacy policy is meant to protect private information, like credentials. It can ask the other party to fulfill some requirements before releasing a credential, something that in natural language can be expressed as: "I will give you my credit card if you show me first your Better Business Bureau membership card".

Protune is based on logic programming and has a declarative structure. Nevertheless, as describe in [7] the language has other features that make it more dynamic: it is able to invoke calls to other systems, like databases, and then integrate the results, it can describe actions that can be executed and can include explanations about elements defined in the policy. Most of this features are possible thanks to the metapolicy, associated to every policy. The metapolicy contains metadata about the policy and it specifies how the policy can be handled.

4.1 The Syntax of Protune

A policy written in Protune is a set of rules similar to normal logic program rules, as specified in [6]. The general form of a rule is

$$H \leftarrow L_1, L_2, ..., L_n.$$

where

- $H \equiv$ the head of the rule, a logic atom that it is not negated
- $L_i \equiv$ a literal that represents a logic atom or a negated logic atom

The set of all literals L_i is called the body of the rule. The body of rule can as well be empty, then the rule it is said to be a fact.

The rule states that if all literals $L_i, i = 1, ..n$ are true then H holds, that is, H is also true.

As explained in [6], the restriction regarding negated rule heads ensures that as more credentials are released the set of permissions does not decrease, which means the policy is monotonic in the sense of [13]

Throughout the rest of this section we will denote by Σ the state associated to a party, client or service provider, during the trust negotiation process. The state contains grounded literals that are to be used with the policy rules and provides information usefull within every negotiation.

4.1.1 Complex terms

One feature that makes Protune different from normal logic programs is the notion of complex terms, which introduces object oriented characteristics. A literal can be a complex term, this stands for an object that has some attributes together with their associated values. The general syntax of a complex term is

$$Id[Attribute_1 : Value_1, Attribute_2 : Value_2, ..., Attribute_n : Value_n]$$

where

- $Id \equiv$ identifier for the object
- $Attribute_i \equiv$ an attribute of the object
- $Value_i \equiv$ the value for the attribute $Attribute_i$

A complex terms allows to represent related information in a structured form. For example

$$myCreditcard[type : \text{“}creditcard\text{”}, issuer : \text{“}Visa\text{”},\\ owner : \text{“}Maria\ Ines\text{”}, serial\_number : \text{“}2752176\text{”}].$$

represents a Visa creditcard, which belongs to Maria Ines and has the serial number 2752176, the credit card is identified by myCreditcard and can be referenced later in the policy using this identifier.

It must be stated that complex terms are not essential for the language, they are only syntactic sugar. Nevertheless, they are an elegant way for presenting information and make the syntax easier.

4.1.2 Predicates

Apart from complex terms, a literal can be a predicate with or without arguments.

$$Predicate(Arg_1, Arg_2, ..., Arg_n)$$

$$Predicate()$$

$$Predicate^2$$

where

- $Predicate \equiv$ the name of the predicate

[2] A constant that is a string is considered to be a predicate without arguments

- $Arg_i \equiv$ the i-th argument of the predicate

An argument can be a constant, a variable, a complex term or even a predicate.

As specified in [7], Protune allows different kinds of predicates, each one with its special semantics, though most of them share the same syntax as described above. The vocabulary of predicates contains the following categories

- **Decision predicates**

  Decision predicates are predicates that can be queried by clients. This type of predicates occurs in the head of a rule and specify a decision that the service provider must take, like to allow access to a certain resource. The decision is positive if the body of the rule whose head is the decision predicate is proved to be true. Currently, there are only two examples of predefined decision predicates *allow* and *sign*.

  Allow is used in order to request something from the other party like:

  – access to a resource or a service

  $$allow(access(\text{“}OnlineBooks''\text{)}).$$

  Here “OnlineBooks” represents the books that can be consulted online on the server.

  $$allow(download(\text{“}U2 - One.mp3''\text{)}).$$

  This specifies that the user is allowed to download the file U2-One.mp3 from the server.

  – release of one credential

  $$allow(release(Visa\_CC[type : \text{“}creditcard'', issuer : \text{“}Visa'', owner : \text{“}Maria\ Ines'', serial\_number : \text{“}2752176'']\text{)}).$$

  This allows the release of the Visa credit card with the serial number 2752176, which belongs to Maria Ines.

  – execution of an action

  $$allow(execute(print(\text{“}document.pdf''\text{)}\text{)}).$$

  The above predicate states that the file document.pdf can be printed.

Sign is used to issue statements signed by the owner of the policy, creating thus new credentials

$$sign(BBB[type : \text{“}member\text{”}, issuer : \text{“}BetterBusinessBureau\text{”}, \\ owner : \text{“}Maria\ Ines\text{”}])$$

The above predicate creates a new credential, a membership card for Maria Ines issued by the Better Business Bureau, the signature on the credential can be checked using the public key of the Better Business Bureau.

- **Abbreviation predicates**

This type of predicates are used for defining new concepts and for simplifying the structure of the policy by allowing factorization of rules.

$$allow(access(Resource)) :- \\ authenticate(User), \\ has\_subscription(User, Resource).$$

$$authenticate(User) :- \\ declaration(d1, D[user : User, passwd : Passwd]), \\ check\_passwd(User, Passwd).$$

The policy above states that anyone that is authenticated and has a subscription for a certain resource can access that resource. Authentication means that the user must provide a username and a password that are then checked against the database if they are valid. In this case, *authenticate(User)* is an abbreviation predicate. Abbreviation predicates are very useful as

– the same abbreviation predicate can be used in several rules

$$allow(access(Resource)) :- \\ authenticate(User), \\ has\_subscription(User, Resource).$$

$$sign(UH\_MC[type : \text{“}member\text{”}, issuer : \text{“}Uni - Hannover\text{”}, \\ owner : RealName]) :- \\ authenticate(User), \\ real\_name(User, RealName).$$

$$authenticate(User) :-$$
$$\quad declaration(d1, D[user : User, passwd : Passwd]),$$
$$\quad check\_passwd(User, Passwd).$$

Authentication is needed for getting access to a resource but it is also required if the user wants a signed proof that he is a member of Hannover University, in which case a membership card with the real name of the user is signed by the server. In this circumstances, the abbreviation predicate *authenticate(User)* helps with reusing the same code.

– abbreviation predicates may have more than one way of being satisfied, e.g., there may be more then one way to authenticate.

$$allow(access(Resource)) :-$$
$$\quad authenticate(User),$$
$$\quad has\_subscription(User, Resource).$$

$$authenticate(User) :-$$
$$\quad declaration(d1, D[user : User, passwd : Passwd]),$$
$$\quad check\_passwd(User, Passwd).$$

$$authenticate(User) :-$$
$$\quad real\_name(User, RealName)$$
$$\quad credential(c1, UH\_MC[type : \text{“}member\text{”},$$
$$\qquad issuer : \text{“}Uni - Hannover\text{”},$$
$$\qquad owner : RealName]).$$

The user can authenticate to the server whether by providing a username and a password or by sending one credentials which proves he or she is a member of the Hannover University. In this case the abbreviation predicate *authenticate(User)* helps display the options that the user has.

It can be argued that abbreviation predicates are not really required. In the first case, the reference to *authenticate(User)* could have been replaced from the first two rules with the body of the rule whose head is *authenticate(User)* and would obtain the same effect. In the second example, two rules could have been build: one by replacing the reference to *authenticate(User)* from the first rule with the first authenticate method and the second one for the other method of authentication.

This is true, nevertheless, imagine having the two effects together, that is: several ways of authentication and several rules that reference the authenticate predicate, if the replacing method would be used, this could lead to combinatorial explosion of the number of rules and this is undesirable.

- **State predicates**

  As mentioned in the beginning of this section, a state Σ is attached to each party, necessarily for keeping information about every trust negotiation. Predicates that refer to this state are called state predicates and are partitioned in two categories state query predicates and provisional predicates.

  – **State query predicates**

    This type of predicates simply query the state and do not modify it: they read the current state and integrate the information contained in it.

    There is also one built-in state query predicate *in* used for querying external packages in the style of [14]. The syntax of *in* is given below

    $$in(Predicate(Arg_1^1, Arg_2^1, ..., Arg_n^1),\\ Package : Function(Arg_1^2, Arg_2^2, ..., Arg_m^2))$$

    where

    * $Package \equiv$ the name of the external package that is being accessed.
    * $Function(Arg_1^2, Arg_2^2, ..., Arg_m^2) \equiv$ the nomenclature of the function that is being called from the external package
    * $Predicate(Arg_1^1, Arg_2^1, ..., Arg_n^1) \equiv$ the result of the query in the shape of one predicate.

    For example, one can decide to have a database with a table of users for holding the users profiles. In order to check if the password provided by one user is valid, it makes a query to the database to retrieve the password for that user by using the *in* predicate

    $$in(passwd(maria, Passwd),$$

$database : query$
$(\text{“}select\ Passwd\ from\ Users\ where\ User = \$maria\text{”}))$.

In this way, a uniform interface can be used for accessing different kinds of packages, like databases or other sources of data, as long as there are wrappers for the packages.

– **Provisional predicates**

Provisional predicates appear in the body of a rule and have both an action and an actor associated with each of them. A provisional predicate can be made true by executing the associated action. The actor can be self, which means that the action is to be executed locally or it can be peer, in this case the action should be executed by the other party and some way is needed to verify whether the action was executed. The action can be expressed in a script language or any other method, specific to the application. The result of the action is asserted in the state.

A commune feature of most applications is to log messages in files, therefore a provisional predicate which takes as parameters the path to a filename and a message and has the associated action of writting the message in the file, seems natural.

$log(\text{“}log.txt\text{”},$
$\text{“}User\ \${User}\ requested\ a\ signed\ member\ card.\text{”})$

where $User is a variable instantiated to a certain username. The action and the actor associated with the log predicate are expressed in the metapolicy, but for this the syntax will be explained later in this section.

As discussed above one peer might also ask another to execute an action, which means to try to prove true a provisional predicate. For example a server might ask a client to register on some webpage before it gets access to a resource. Of course, the server must be able to check if the specified client has successfully registered. In this case, the metapolicy contains information about the register predicate stating that the actor is the peer and describing the action that must be executed.

$register(\text{“}www.uni - hannover.de/OnlineBooks/$
$register.php\text{”})$.

Apart from making policies more dynamic, provisional predicates are necessary for delegating authority. Suppose students from Hannover University have free access to the library. It is not recommended, nor necessarily that the library has a database with all the students. Instead, whenever a student requests access for the first time, the library server may ask him to contact the student office server from the university, in order to obtain a digital signed proof that he is a student. The metapolicy specifies the actor and action that describes what the student is supposed to do.

$$get\_signed\_proof(\text{“}www.uni-hannover.de/StudentOffice/index.php\text{”}).$$

There are two predefined provisional predicates *credential* and *declaration*. A credential means signed information, while a declaration is not signed. The mentioned predicates share the same syntax

$$Predicate(Ref, Id[Attribute_1 : Value_1, .., Attribute_n : Value_n])$$

or

$$Predicate(Ref, Id)$$

where

* $Predicate \equiv$ credential or declaration
* $Ref \equiv$ a reference to the context in which the declaration was made or the credential given
* $Id \equiv$ the id of the object, this can also be a variable
* $Attribute_i \equiv$ an attribute of the object
* $Value_i \equiv$ the value of $Attribute_i$

As explained above the second argument can be a complex term describing one credential or a declaration, or it can also be an id of a complex term. The semantics is the following: whenever the *credential* predicate is encountered in the policy it means that one credential matching the second argument is searched for a cached one in the state, if none exists the credential is requested from the other party. The same applies for declaration, whenever the *declaration* predicate is encountered in the policy then a declaration

matching the second argument is searched for a cached one in the state, if none exists, the other party is requested to provide the declaration.

Since provisional predicates specify an action that is executed, the syntax of the language does not allow negation on this type of predicates if the actor is peer as it cannot be verified that an action was not performed.

- **Constraint predicates**

  This type of predicates can appear only in the body of a rule and are the regular equality and disequality predicates like: $=, != , >, <, >=, <=$ together with two built-in predicates *is* and *ground*. Of course, the equality and disequality predicates do not follow the general form of a predicate. The predicates *is* and *ground* have the same meaning as in the logic programming language Prolog, that is

  - A is B
    checks whether A and B are instantiated to the same value
  - ground(Arg)
    checks whether the argument given, Arg, is instantiated

In conclusion, the head of a rule can be a complex term or a predicate of the type: decision predicate, abbreviation predicate or state predicate, while a literal in the body can be a predicate of any type or a complex term, both of then can also appear in negated format.

4.1.3 Metapolicy

As mentioned in section 3, the metapolicy contains metadata about the policy and provides information on how the policy should be handled. The metapolicy is structured as a set of declarative rules, more like an object oriented language. Each declaration in the metapolicy must refer to a rule in the policy, by using the identifier for that rule or a predicate defined in the policy, which may appear in the head of a rule or in the body of a rule or both.

The general form of a declaration in the metapolicy is

$$[Id].attribute : value : -Metabody.$$

or

$$Predicate(Arg_1, Arg_2, ..., Arg_n).attribute : value : -Metabody.$$

where

- $Id \equiv$ the identifier of the rule to which the metarule refers
- $Predicate \equiv$ the name of the predicate to which the metarule refers
- $Arg_i \equiv$ the i-th argument of the predicate to which the metarule refers
- $attribute \equiv$ the attribute of the referenced rule or predicate
- $value \equiv$ the value for the specified attribute
- $Metabody \equiv$ the metabody of the metarule, which is optional

The metabody of a metarule has the same syntax as the body of a rule, that is a set of literals

$$Metabody \equiv L_1, L_2, ..., L_n$$

where a literal can be

- a predicate with or without arguments, that can also appear in a negated form

$$Predicate(Arg_1, Arg_2, ..., Arg_n)$$

$$Predicate()$$

$$Predicate$$

$$not(Predicate(Arg_1, Arg_2, ..., Arg_n))$$

$$not(Predicate())$$

$$not(Predicate)$$

- a complex term, which can also be negated

$$Id[Atr_1 : Val_1, Atr_2 : Val_2, ..., Atr_n : Val_n]$$

$$not(Id[Atr_1 : Val_1, Atr_2 : Val_2, ..., Atr_n : Val_n])$$

- a comparison

$$Arg_1\ Op\ Arg_2$$

$$not(Arg_1\ Op\ Arg_2)$$

- a metaliteral, that is the head of another metarule

$$[Id].attribute : value$$

$$Predicate(Arg_1, Arg_2, ..., Arg_n).attribute : value$$

$$not([Id].attribute : value)$$

$$not(Predicate(Arg_1, Arg_2, ..., Arg_n).attribute : value)$$

The semantics of the metarule is that the rule or the predicate referenced has the given value for the specified attribute if the metabody holds, that is the metabody is empty or all the literals in the metabody are proved to be true. It must be mentioned that in order to prove that a metahead is true it must exist a metarule unifying with the metahead, whose metabody is true. In order to prove that a complex term is true it must be checked if it exists in the state. Also, within a metabody it is not allowed to include provisional actions.

Metapolicy attributes

Current version allows any attribute in the metapolicy but only some are in use for the moment. This ones are explained below:

- **type**
  This attribute only applies to predicates and it specifies their type. All the values for this attribute have already been explained before.
  - *decision_predicate*
  - *abbreviation_predicate*
  - *provisional_predicate*
  - *state_predicate* [3]
  - *constraint_predicate*

- **sensitivity**
  This attribute denotes the sensitivity of a rule or of some literal and it can be
  - *public*
    This means that the predicate or the rule referenced can be disclosed to the other party.

[3]This refers to state query predicates.

  - *private*
    This is opposite to public, it specifies that the information is secret and should be hidden from the other party. This is the default values used.
  - *non-applicable*
    This type of sensibility is only used for rules to specify that a certain rule is not applicable under some circumstances expressed in the metabody. This is currently not used.

- **evaluation**
  Some predicates, like state query predicates and provisional predicates, need to be evaluated. In the case of state predicates, the state needs to be queried and the result asserted or in the case of provisional predicates some action must be executed. This evaluation can be done in several ways.

  - *immediate*
    which means that the evaluation of the predicate must be done immediately
  - *deferred*
    this means that the evaluation can be delayed until the negotiation finishes
  - *concurrent*
    which specifies that the evaluation can be done concurrently This is currently not used.

- **action**
  This is only used in the case of provisional predicates, for specifying the action attached to the predicate. Since the way the action is specified is application dependent, further on, only natural language statements are used to specify it in examples. Real applications may choose any script language to specify actions.

- **actor**
  Provisional predicates must also have an actor attached, the party executing the action. This can be self or peer:

  - *self*
    which means the action is to be executed locally
  - *peer*
    which specifies that the action should be executed by the other party.

- **explanation**
  This attribute is used in order to attach explanations to predicates or to rules. Explanations are important as they are sent to the other party in order to provide more information about the requirements for satisfying a policy and help throughout the negotiation process. Explanations are natural language statements that can be understood by human beings.

Since the language is very flexible other attributes and values can be added in time. Below, a summary is given of the predefined attributes and their values.

Attribute	Applied on	Possible values
type	literals	decision_predicate, abbreviation_predicate, state_predicate, provisional_predicate, constraint_predicate
sensitivity	literals, rules	public, private,
evaluation	provisional_predicates, state_predicates	immediate, deferred,
action	provisional_predicate	script language
actor	provisional_predicate	self, peer
explanation	rules, literals	natural language statement

4.2 Policy Example

For better understanding the syntax of Protune described in the previous subsection, a policy together with its associated metapolicy, is explained here.

The policy shown below is meant to be an access control policy for an online library with three sections: books, videotec and sonotec. Each of the three sections is considered a resource and access is granted to all of them in a similar manner.

```
[r1]allow(access(Resource)):-
        credential(sa,Student_card[type:student,issuer:I,public_key:K]),
        valid_credential(Student_card,I),
        recognized_university(I),
        challenge(K).

[r12]valid_credential(C,I):-
        public_key(I,K),
        verify_signature(C,K).

[f1]recognized_university(upb).
[f2]recognized_university(hu).
[f3]recognized_university(epfl).

        /*Metapolicy*/
allow(_).sensitivity:public.

public_key(_,_).type:provisional_predicate.
public_key(I,_).evaluation:immediate:-ground(I).
public_key(_,_).actor:self.
public_key(I,K).action:"connect to $C server and get $I's public key $K".
public_key(_,_).sensitivity:private.

verify_signature(_,_).type:provisional_predicate.
verify_signature(C,K).evaluation:immediate:-ground(C),ground(K).
verify_signature(_,_).actor:self.
verify_signature(C,K).action:"verify the signature on the credential $C
                using public key $K".
verify_signature(_,_).sensitivity:private.

recognized_university(_).type:state_predicate.
```

```
recognized_university(_).evaluation:immediate.
recognized_university(_).sensitivity:public.

valid_credential(_,_).type:abbreviation_predicate.
valid_credential(_,_).sensitivity:public.

challenge(_).type:provisional_predicate.
challenge(K).evaluation:immediate:-ground(K).
challenge(_).actor:self.
challenge(K).action:
        "challenge peer to prove that he has the private key
        corresponding to public key $K".
challenge(_).sensitivity:public.
```

The first rule, r1, states that access to a resource can be granted if the user provides a valid credential stating that he is a student at an university recognized by the library.

Each time a credential is received from the other party, like the student card, it is checked whether it is valid, which basically means verifying the signature on it against the public-key of the issuer. This proves that the credential is valid, that it was indeed issued by the required institution. Furthermore, another check is necessary to prove that the other party is the real owner of the credential and this is done via a challenge sent to the other party, the challenge uses the public key of the user and it can only be answered if the user holds the private key.

The library has a list of univeristies that collaborates with, this is expressed via the recognized_university predicate.

The metapolicy contains information regarding the policy. For the rest of the example, the metapolicy is not shown in this section but is included in B.

```
[r2]allow(access(Resource)):-
        authenticate(U),
        has_subscription(U,Resource).

[r3]authenticate(U):-
        declaration(ad,D[username:U,password:P]),
        passwd(U,P).
```

[f4]passwd(mirela,alerim).
[f5]passwd(dragos,sogard).
[f6]passwd(alina,anila).

[f7]has_subscription(dragos,books).
[f8]has_subscription(dragos,videotec).
[f9]has_subscription(mirela,sonotec).
[f10]has_subscription(mirela,books).
[f11]has_subscription(alina,books).

Rules r2 and r3 from the policy specify another method for getting access to a resource. The use of a resource is allowed for a user which is authenticated and has a subscription for the requested resource. In this case, authentication means providing a username and a password that are then checked against the database (like any regular sign in).

Information about user profiles like the users, the password for each user and their subscriptions are kept in the state Σ.

```
[r4]allow(access(Resource)):-
        european_citizen(X),
        paid(X,Resource),
        register(X,U),
        assert(has_subscription(U,Resource)).

[r5]european_citizen(X):-
        credential(ea,European_id[owner:X,type:european_citizen,
        issuer:I,public_key:K]),
        valid_credential(European_id,I),
        trusted_organization(I),
        challenge(K).

[r6]paid(X,Resource):-
        price(Resource,P),
        credit_card_payment(X,P),
        logged("$X paid $P for the resource $R").

[r7]credit_card_payment(X,P):-
        credential(pc,Credit_card[type:credit_card,issuer:visa,owner:X]),
        valid_credential(Credit_card,visa),
        charged(Credit_card,P).
```

```
[r8]charged(C,P):-not_revoked(C),
        transfer_money(C,P).

[r9]register(X,U):-
        declaration(rd,D[username:U,password:P]),
        check(U,P,X).

[r10]check(U,_,X):-
        passwd(U,_),
        register(X).

[r11]check(U,P,X):-
        not(passwd(U,_)),
        assert(passwd(U,P)),
        logged("New user: $X registered as $U").

[f12]trusted_organization(ec).
[f13]trusted_organization(euh).

[f14]price(books,5).
[f15]price(videotec,9).
[f16]price(sonotec,7).
```

The third method that can be used in order to gain access to a resource is by providing an european citizen membership card issued by an organization trusted by the library and paying the price for the requested resource with a Visa credit card. This method also implies that the user is now a registered user and gets a subscription for that resource. The next time he wants to access the resource he can use authentication, the second method for getting access described by rules r2 and r3.

Before a user pays with a Visa credit card, the credit card is checked whether it was not revoked, in order to do this a provisional predicate is used *not_revoked(C)*. The action associated to this predicate, "connect to Visa server and check to see if $C was revoked", is evaluated as soon as C is instantiated, which is expressed in the metarule.

$$not\_revoked(C).evaluation : immediate : -ground(C).$$

After every payment for a resource, data about the transaction is written in a log. This is one type of action which does not need to be executed

immediately and maybe deferred until the end of the negotiation.

After having paid for a resource a user must register with a username and a password. This is done in a loop until a username that is not yet in the database is chosen. Finally, after registration, the user gets the subscription for the requested resource and can access it.

The entire policy and its metapolicy are given in the appendix B.

5 Protune Negotiation Framework

In order to describe online negotiations, we need to specify how the entities involved interact and also what are the processes that take place at each peer for every negotiation. Trust negotiation means exchange of policies and credentials such that a certain level of trust is established so the transaction can be performed. Since policies are sent to the other party, they must be processed first. For example, a policy may contain sensitive information needed for reasoning but which should be hidden to the other party. Processing a policy is called filtering the policy. This section describes in detail the negotiation model and the process of filtering a policy.

5.1 Negotiation Model

In the following, we formalize the negotiation between two entities. For each party, a negotiation model is required so that it describes the behavior of the peer during the negotiation process. Two parties involved in a negotiation may use the same model with different parameters. In order to formalize a general negotiation model, we first introduce some notations.

We consider two parties E_1 and E_2 that take part in a negotiation. We assume that E_1 requests a service or a resource that E_2 can provide. If E_2 also wants to request a resource or a service from E_1, not as part of the negotiation, then it must start a new negotiation, not use the current one started by E_1. Notice that during a negotiation both entities E_1 and E_2 may request or provide information to each other.

Each peer has its own policy for allowing access to its resources and protecting its credentials. Each policy has an associated metapolicy used during the filtering process to protect the private information contained in the policy, before sending it to the other party. We assume that a policy does not change during the negotiation process.

The policy language used,e.g., Protune, is a rule-based language. Such a language is based on normal logic program rules

$$H \leftarrow L_1, \ldots L_n$$

where H is a standard logical atom (called the *head* of the rule) and $L_1, \ldots L_n$ (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.*

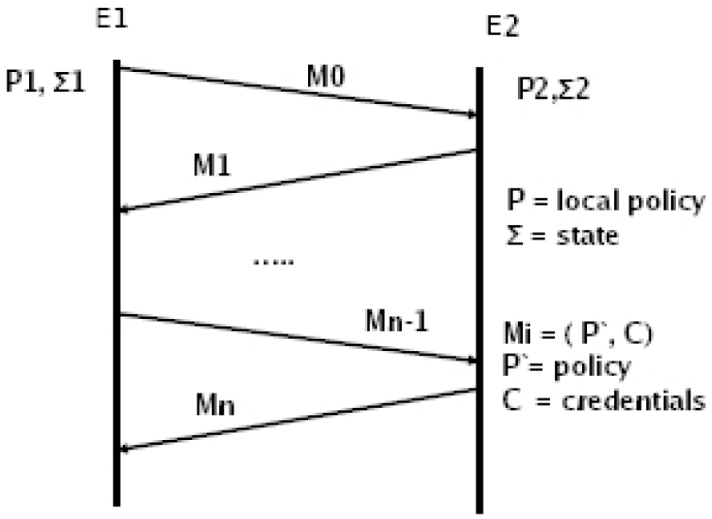

Figure 2: Interaction between entities

This restriction ensures that policies are *monotonic* as explained in section 4.

Each peer has a set of credentials protected by the policy, that may be exchanged during the negotiation. From the logical point of view, declarations are the same as credentials, so through the rest of this section we assume declarations are included in the set of credentials.

The interaction between two entities that are negotiating is depicted in figure 2.

A negotiation advances by sending and receiving negotiation messages.

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 denote with M the set of all possible Negotiation Messages.

The initial message contains the initial request and, also, it is possible to contain some credentials, that the requester knows they are needed in the negotiation, in order to speed up the negotiation process.

Definition 3 (Message Sequence) *A Message Sequence σ is a list of Negotiation Messages*

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

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

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 σ for the entity E_j ($j = 1, 2$) is a Message Sequence*

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

Moreover let

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

We refer to a Negotiation History also as a *Negotiation State*. The initial Negotiation State, for each peer, is an empty list, this means that in the beginning nothing was disclosed and nothing was received.

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 σ 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 σ, 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) during the negotiation.

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* $S = (\sigma_1, \ldots \sigma_n)$ *then* $t(S, \sigma) = (\sigma_1, \ldots \sigma_n, \sigma_{n+1})$ *(transition function)*

Intuitively a Negotiation State Machine models how an entity evolves during the negotiation by the exchange of messages. Σ contains both sent and received Negotiation Messages and can therefore be partitioned into two subsets Σ_{snd} and Σ_{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* (Σ, 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 local credentials
- p_0 represents the local policy protecting the local credentials and allowing access to the local resources
- S represents the set of states which the peer can have
- s_0 represents the initial state, i.e. the state in which the peer is at the beginning of the negotiation
- ff 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

Regarding the filtering process, which must be applied on the local policy before sending it to the other party, it must be specified that $p^{'}_{snd}$, a disclosed policy, is the result of the filtering process which uses as input the original policy p_0 and the information in the local state s_i

$$p^{'}_{snd} = ff(p_0, s_i)$$

Since p^{i}_{snd} is included in s_i this implies $\mathcal{H}(p^{i}_{snd}) \subseteq \mathcal{H}(p^{'}_{snd})$ as demonstrated in the appendix C.

Since it is possible, that, after all, the negotiation does not succeed because of some policy not being satisfied, this would lead to deadlock. For this case we must specify a termination criterion as part of the negotiation strategy.

Definition 7 (Negotiation Termination) *A negotiation termination is a function*

$$ta : S \rightarrow \{true, false\}$$

The termination function we consider for the application is

$$ta : S \rightarrow \{true, false\}$$

- f is true if $\sigma_{n-1} \cup \sigma_{n-2} = \phi$
- f is false otherwise

In our case, we consider that the negotiation terminates when two empty messages are exchanged in a row, one after another.

A policy can specify several ways for getting access to a resource or allowing the release of one credential. This implies that there are several possibilities to choose which credentials to disclose and when each credentials should be disclosed. For automatically taking this decisions we need a way of selecting the next credentials that will be disclosed. Credentials selection is also part of the negotiation strategy since it selects the next credentials that are to be in the next message sent to the other party.

Valid negotiation models

Let us consider our example where E_1 requested access to a resource from E_2. We will denote by NM_1^E, E_1's negotiation model and by NM_2^E, E_2's negotiation model. The two models NM_1^E and NM_2^E represent a valid negotiation model as stated in [12] if

- For each message sent from E_1 to the E_2, an identical message, that is with the same parameters, is received by E_2 from E_1. Similar for each message sent from E_2 to E_1.
- The negotiation models should not allow neverending exchanges of empty messages. As specified above, in our case, no more than two empty message in a row are allowed.
- Each party should send information that offers the other party the opportunity to release credentials or that enables the other party the expansion of hidden parts of the policy. This is possible since, one party can deduct from the policy the credentials requested by the other party and it can also deduce variables on which hidden parts of the policy depend.
- It is possible that a message repeats information which has previously been disclosed, in this case the message is considered empty.
- A message cannot contain a request for a credential if this has already been provided by the other party.
- A message must contain at least either a set of credentials or a filtered policy disclosed to the other party.

Valid negotiation models are required as two valid models make a negotiation successfully if the policies on both sides allow it.

5.2 Policy Filtering

As already explained in section 4, a policy written in Protune is organized as a set of rules. In order to describe access control information or protect private credentials, it is likely that a policy contains some sensitive data. This information is necessary in order to take decisions during the negotiation process, but special care must be taken so it is not disclosed to other parties. Furthermore, a policy needs to be preprocessed before sending it to the other peer, like removing some irrelevant parts and adding some information needed by the other party. In order to fulfill this goals, the policy must always be filtered before sending it to any client as described in [6]. A filtered policy does not contain any information that is sensitive for its owner.

5.2.1 Policy Filtering Steps

The filtering of a policy is a sequential process with the following parameters as explained in [7]:

- $R \equiv$the request of the client for access to a resource or for the release of one credential.
- $P \equiv$the policy, together with the associated metapolicy
- $\Sigma \equiv$the current state used for keeping information related to the current negotiation.

The steps of the filtering process, extended form the ones described in [6] and [7], are explained in detail below.

Selecting relevant rules

A request from one client only asks access to a specific resource or requires the release of only one credential. Since the policy of the server may include rules for many resources and for different credentials, the rules that are related to the request of the client need to be selected and the other ones removed. Apart from this, some rules, even if relevant for the request, must be hidden until the client proves to be trustworthy.

Definition 8 (Relevant rules subset) *The relevant rules subset of the server's policy P in regards to the client's request R is the set of rules S, where each rule*

$$[Id]H \leftarrow L_1, L_2, ..., L_n.$$

- *it is not marked as non-applicable, that is there is no metarule defined for this rule which has the value non_applicable for the sensitivity attribute*

$$[Id]sensitivity : non\_applicable.$$

- *and fulfills at least one of the two conditions below*
  - *the head of the rule H unifies with the client's request R.*
  - *the head of the rule H unifies with a literal L_i which occurs in the body of some rule that is in S.*

The notation used to denote this step will be:

$$P^{''} = relevant\_rules(P^{'}, R, \Sigma)$$

Partial evaluation

The state Σ contains information related to the current negotiation, like credentials disclosed by the other party. In order to integrate this information in the policy, we need to evaluate the literals contained in the state against the policy.

Definition 9 (Partial evaluation) *The state is a set of grounded literals*

$$\Sigma = \{L|\ L\ is\ a\ grounded\ literal\}$$

For a rule in the policy

$$H \leftarrow L_1, ..., L_i, ..., L_n.$$

partial evaluation is applied if one literal in the body of the rule L_i unifies with one grounded literal L in the state Σ.

After partial evaluation the rule becomes:

$$(H \leftarrow L_1, ..., L_{i-1}, L_{i+1}..., L_n.)\theta$$

where

$$\theta = most\_general\_unifier(L_i, L)$$

This means that a substitution is applied on the rule, propagating the value of the variables that became grounded after the unification of L and L_i.

This step is referenced as:

$$P^{''} = partial\_evaluation(P^{'}, \Sigma)$$

Actions execution

As mentioned before, a policy written in Protune integrates actions through provisional predicates. There is one step in the filtering dedicated to executing the actions. First, the actions that are ready for execution are selected, then the actions are performed and last, the action results are asserted to the state.

Definition 10 (Actions execution) *Each literal L that is a provisional predicate with immediate evaluation and actor self is selected.*

$$L.type : provisional\_predicate.$$

$$L.evaluation : immediate.$$

$$L.actor : self.$$

The action associated with L

$$L.action : Action$$

will be in the ImmediateActions set. Actions in this set will be executed and the execution results, which are grounded literals, are gathered in the ActionResults set. The literals in the ActionResults set are asserted to the state.

This is written as:

$$ImmediateActions = select\_actions(P, \Sigma)$$

$$\Sigma^{'} = \Sigma \cup ActionResults$$

where

- $ImmediateActions \equiv$ the set of all the actions that are ready for execution
- $ActionResults \equiv$ the set of results after performing the actions
- $\Sigma^{'} \equiv$ the new state associated to the current negotiation

Actions need to be executed within a loop. This is required as after integrating in the policy the action results, new actions may be ready for execution.

Blurring

The policy can contain sensitive information that needs to be hidden, e.g., state predicates providing information about user profiles may include passwords. These predicates cannot be simply removed from the policy as the client would not be aware that something hidden from him is being checked by the server. The solution is blurring the literals that refer to private information. Blurring means wrapping the sensitive literals in a special particle "blurred" and renaming the predicate. By blurring a rule, the rules whose heads unify with the sensitive literals are discarded from the policy so the definition for blurred predicates is not given to the other party.

Definition 11 (Blurring) *Let B be the set of literals that need to be blurred, B is composed of*

- *provisional_predicates that must be evaluated immediately and whose actor is self, that is every L such that*

$$L.type : provisional\_predicate.$$

$$L.evaluation : immediately.$$

$$L.actor : self.$$

- *all other types of predicates that are private, that is every L such that*

$$L.sensitivity : private.$$

Every rule

$$Head \leftarrow Body$$

is transformed into

$$Head \leftarrow Body^{'}$$

where

- $$Body^{'} \equiv Body,\ if\ Body\ \cap\ B = \phi$$

  that is, Body does not contain any predicate that is in the blurred set.

- $$Body' \equiv (Body \backslash \{L | L \subseteq B\}) \cup \{blurred(L) | L \subseteq B\}$$

  that is every literal which is in the blurred set B will be wrapped with the "blurred" particle and renamed.

For example, if a policy contains a predicate mapping usernames to passwords, like passwd(User,Password), this would be a private state predicate and after blurring, every occurrence of passwd(User,Password) is replaced with blurred(predicate(User,Password)).

Further on this step is referred to as:

$$P_{''} = blurring(P_{'}, \Sigma)$$

Selecting peer actions

The metapolicy can also contain information necessary for the other party, like actions, associated to provisional predicates in the policy, that the other party must perform. Since only the filtered policy and not also the metapolicy is going to be sent to the other party, this information needs to be extracted and sent to the other party as well.

Definition 12 (Peer Actions) *The set of peer actions A in policy P is the set of all provisional predicates with actor peer that are referenced in the policy P. That is every L such that*

$$L.type : provisional\_predicate$$

$$L.actor : peer$$

and $L \subseteq P$.
Actions associated to predicates that are in the set of peer actions

$$L.action : Action.$$

where $L \subseteq A$*, are prefixed with a "do" and sent to the other party.*

Further on, this step is referred to as:

$$P_{''} = peer\_actions(P_{'})$$

Adding explanations

Protune also support explanations, these are meant to provide information for the other party. Explanations can be very important as they provide human understandable statements for clarifying the meaning of predicates or rules. Explanations need to be extracted and send together with the policy.

Definition 13 (Explanations) *The set of explanations E in a policy P is the set of metapolicies such that*

$$L.explanation : Explanation.$$

and $L \subseteq P$.

This is written as:

$$P'' = add\_explanations(P')$$

Rename abbreviation predicates

Usually, when a policy is written, meaningful names are given to literals. The problem is that sometimes predicate names can disclose information about the goal of the predicate or the structure of the policy. The solution for this, adopted in Protune, is renaming all the predicates to an uniform name like "predicateN" where N is the number of the predicate. In fact, only abbreviation predicates and state predicates need to be renamed as the other types of predicates either need to be understood by the other party (like decision predicates, credential, declaration), have been taken out from the policy or have been blurred.

Definition 14 (Predicate Renaming) *Let A be the set of all the predicates that need to be renamed.*

$$A = \{L|L.type : abbreviation\_predicate\} \cup \{L|L.type : state\_predicate\}$$

Every rule

$$Head \leftarrow Body$$

is transformed into

$$Head \leftarrow Body'$$

where

- $$Body' \equiv Body,\ if\ Body\ \cap A = \phi$$

  that is Body does not contain any predicate that needs to be renamed.

- $$Body' \equiv (Body \backslash \{L|L \subseteq A\}) \cup \{rename(L)|L \subseteq A\}$$

  that is every literal which is in the A set will be renamed.

This is referenced later as:

$$P_{\prime\prime} = rename\_predicates(P_{\prime})$$

After having described in detail all steps of the filtering, we can now formalize the whole process as a sequence of steps.

- $P_1 = relevant\_rules(P, R, \Sigma)$
- $P_2 = partial\_evaluation(P_1, \Sigma)$
- $P_3 = relevant\_rules(P_2, R, \Sigma)$
- $ImmediateActions = select\_actions(P_3, \Sigma)$
  $\Sigma^{'} = \Sigma \cup ActionResults$
  where the ActionResults set contains the results of the actions in the ImmediateActions set that were executed.
- $P_4 = partial\_evaluation(P_3, \Sigma^{'})$
- $P_5 = relevant\_rules(P_4, R, \Sigma^{'})$
- $P_6 = blurring(P_5, \Sigma)$
- $P_7 = relevant\_rules(P_6, R, \Sigma^{'})$
- $P_8 = add\_info(P_7)$
- $P_9 = rename\_predicates(P_8)$

The filtered policy is $F = P_9$. The relevant rule step appears several times during filtering process in order to remove irrelevant parts of the policy and improve performance. The whole filtering process is depicted in 3

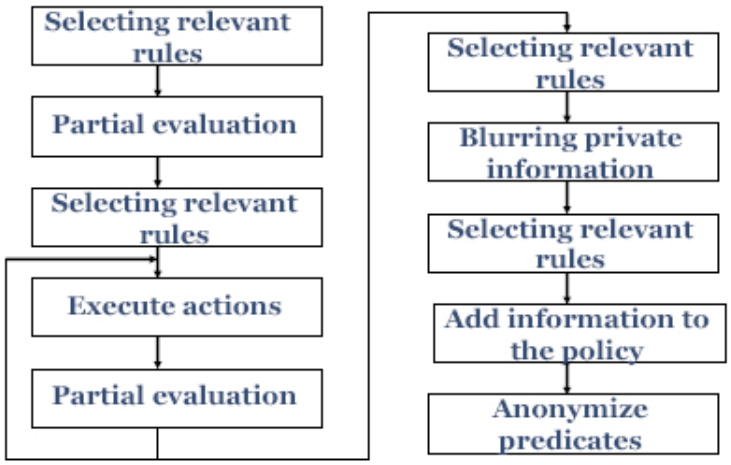

Figure 3: The filtering process

5.2.2 Filtering Example

For better understanding the filtering process, we will apply the filtering, as formalized above, on the example policy given in section 4. The metapolicy for this policy can be found in the appendix B.

We assume that the client request is

$$allow(access(books)).$$

Furthermore, in order to explain in detail the filtering, we consider that the client, together with the request, also sends one credential stating he is a student of Hannover University ('HU').

$$credential(studentcard[type : student, issuer : hu, public\_key : 5272117]).$$

Step1. Selecting relevant rules

In this step rules that are related to other resources than the one requested are taken out from the policy. There is no rule marked as non_applicable, so each rule relevant to the request is provided.

```
[r1]allow(access(books)):-
        credential(sa,Student_card[type:student,issuer:I,public_key:K]),
        valid_credential(Student_card,I),
        recognized_university(I),
        challenge(K).
```

```
[r12]valid_credential(C,I):-
        public_key(I,K),
        verify_signature(C,K).

[f1]recognized_university(upb).
[f2]recognized_university(hu).
[f3]recognized_university(epfl).

[r2]allow(access(books)):-
        authenticate(U),
        has_subscription(U,books).

[r3]authenticate(U):-
        declaration(ad,D[username:U,password:P]),
        passwd(U,P).

[f4]passwd(mirela,alerim).
[f5]passwd(dragos,sogard).
[f6]passwd(alina,anila).

[f7]has_subscription(dragos,books).
[f10]has_subscription(mirela,books).
[f11]has_subscription(alina,books).

[r4]allow(access(books)):-
        european_citizen(X),
        paid(X,books),
        register(X,U),
        assert(has_subscription(U,books)).

[r5]european_citizen(X):-
        credential(ea,European_id[owner:X,type:european_citizen,
        issuer:I,public_key:K]),
        valid_credential(European_id,I),
        trusted_organization(I),
        challenge(K).

[r6]paid(X,Resource):-
        price(Resource,P),
        credit_card_payment(X,P),
        logged("$X paid $P for the resource $R").
```

```
[r7]credit_card_payment(X,P):-
        credential(pc,Credit_card[type:credit_card,issuer:visa,owner:X]),
        valid_credential(Credit_card,visa),
        charged(Credit_card,P).

[r8]charged(C,P):-not_revoked(C),
        transfer_money(C,P).

[r9]register(X,U):-
        declaration(rd,D[username:U,password:P]),
        check(U,P,X).

[r10]check(U,_,X):-
        passwd(U,_),
        register(X).

[r11]check(U,P,X):-
        not(passwd(U,_)),
        assert(passwd(U,P)),
        logged("New user: $X registered as $U").

[f12]trusted_organization(ec).
[f13]trusted_organization(euh).

[f14]price(books,5).
[f15]price(videotec,9).
[f16]price(sonotec,7).
```

Step 2. Partial evaluation

Partial evaluation on the policy is done against the state. For the moment, the state only contains the credential sent by the other party, the student card, which is required in order to prove rule [r1].

```
[r1]allow(access(books)):-
        valid_credential(student_card,hu),
        recognized_university(hu),
        challenge(5272117).
```

Step 3. Relevant rules

This step removes from the policy the rules about the recognized universities that do not unify anymore.

[f1]recognized_university(upb).
[f3]recognized_university(epfl).

Also in this step, while selecting the relevant rules, the instantiation of the valid_credential predicate for the student card is done.

[r12]valid_credential(student_card,hu):-
public_key(hu,K),
verify_signature(student_card,K).

As one can notice, the rules describing all methods to get access to the requested resource are still in the policy even if one way was chosen already by providing a credential. This ensures that later the client can follow another way if this one does not succeed.

Step 4. Select actions and Partial evaluation

The provisional predicates which specify actions that are ready for execution are:

$$public\_key(hu, K).$$

$$challenge(5272117).$$

We will assume actions associated with them were successfully executed, that is we get the public key of the Hannover University and the client fulfills the challenge, proving he holds the private key corresponding to the public key "2172705". As one can imagine this cannot be a real public key, it is used only as example. The results of the actions are asserted to the state:

$$successful(public\_key(hu, 1721275)).$$

$$successful(challenge(5272117)).$$

As already mentioned, actions are executed in a loop. After partial evaluation on the new state, another provisional predicate is ready for execution

$$verify\_signature(student\_card, 1721275).$$

The following two rules change due to partial evaluation.

[r1]allow(access(books)):-
valid_credential(student_card,hu),
recognized_university(hu).

[r12] valid_credential(sa, student_card):-
verify_signature(student_card,1721275).

Verifying the signature on the student card against the university public key ends well, so we assert the result to the state. So far, we have proved that the studentcard given is valid, furthermore the client is entitled to get access to the 'books' resource since rule [r1] has been proved.

[f2]recognized_university(hu).

[r1]allow(access(books)):-
valid_credential(student_card,hu),
recognized_university(hu).

[r12] valid_credential(sa, student_card).

Step 5. Relevant rules

This step leaves the policy unchanged as there are no rules that need to be excluded from the policy.

Step 6. Blurring

Blurring means wrapping sensitive information inside the "blurred" particle and renaming private predicates. In this way, definitions for sensitive concepts is not given to the other party. The policy after blurring the private parts is shown below.

[f2]recognized_university(hu).

[f4]passwd(mirela,alerim).
[f5]passwd(dragos,sogard).
[f6]passwd(alina,anila).

```
[f7]has_subscription(dragos,books).
[f10]has_subscription(mirela,books).
[f11]has_subscription(alina,books).

[f12]trusted_organization(ec).
[f13]trusted_organization(euh).

[f14]price(books,5).


[r1]allow(access(books)):-
        valid_credential(student_card,hu),
        recognized_university(hu).

[r2]allow(access(books)):-
        authenticate(U),
        blurred(has_subscription(U,books)).

[r3]authenticate(U):-
        declaration( ad,D[username:U,password:P]),
        blurred(passwd(U,P)).

[r4]allow(access(books)):-
        european_citizen(X),
        paid(X,books),
        register(X,U).
        blurred(assert(blurred(has_subscription(U,books)))).

[r5]european_citizen(X):-
        credential( ea, European_id [owner:X,type:european_citizen,
            issuer:I,public_key:K]),
        valid_credential(European_id,I),
        trusted_organization(I),
        blurred(challenge(K)).

[r6]paid(X,books):-
        price(books,P),
        credit_card_payment(X,P),
        blurred(logged("$X paid $P for the resource books"))/
```

[r7] credit_card_payment(X,P):-
credential(pc, Credit_card[type:credit_card, issuer:visa, owner:X]),
valid_credential(Credit_card,visa),
charged(Credit_card,P).

[r8] charged(C,P):-
blurred(not_revoked(C)),
blurred(transfer_money(C,P)).

[r9] register(X,U):-
declaration(rd, D[username:U,password:P]),
check(U,P,X).

[r10] check(U,_,X):-
blurred(passwd(U,_)),
register(X,U).

[r11] check(U,P,X):-
not(blurred(passwd(U,_))),
blurred(assert(passwd(U,P))),
blurred(logged('New user: $X registered as $U')).

[r12]valid_credential(sa, student_card).

Step 7. Relevant rules

Since blurred predicates are not expanded anymore, in this step some rules are eliminated from the policy.

[f4]passwd(mirela,alerim).
[f5]passwd(dragos,sogard).
[f6]passwd(alina,anila).

[f7]has_subscription(dragos,books).
[f10]has_subscription(mirela,books).
[f11]has_subscription(alina,books).

Step 8. Rename predicates

The last step of the filtering process is renaming the predicates as they might have meaningful names, after this is done the policy can be sent to the other party.

```
[f2]predicate0(hu).

[f14]predicate1(books,5).

[r1]allow(access(books)):-
        predicate3( student_card,hu),
        predicate0(hu).

[r2]allow(access(books)):-
        predicate4(U),
        blurred(predicate5(U,books)).

[r3]predicate4(U):-
        declaration( ad,D[username:U,password:P]),
        blurred(predicate(U,P)).

[r4]allow(access(books)):-
        predicate7(X),
        predicate8(X,books),
        predicate9(X,U).
        blurred(assert(predicate5(U,books))).

[r5]predicate7(X):-
        credential( ea, European_id [owner:X,type:european_citizen,
            issuer:I,public_key:K]),
        predicate3( European_id, I),
        predicate10(I),
        blurred(predicate11(K)).

[r6]predicate8(X,books):-
        predicate1(books,P),
        predicate12(X,P),
        blurred(predicate13("$X paid $P for the resource books"))/

[r7] predicate12(X,P):-
```

```
        predicate3( pc, Credit_card[type:credit_card, issuer:visa, owner:X]),
        predicate14(credit_card,P).

[r8] predicate14(C,P):-
        blurred(predicate15(C)),
        blurred(predicate16(C,P)).

[r9] predicate9(X,U):-
        declaration(rd, D[username:U,password:P]),
        predicate15(U,P,X).

[r10] predicate15(U,_,X):-
        blurred(predicate6(U,_)),
        predicate9(X,U).

[r11] predicate15(U,P,X):-
        not(blurred(predicate6(U,_))),
        blurred(assert(predicate6(U,P))),
        blurred(predicate13('New user: $X registered as $U')).

[r12] predicate3( studentcard, hu ).
```

From this policy, explanations can be also generated as specified in [4].

6 Implementation

This section provides details about the implementation of an application which consists of the modules that provide support for an entity to take part in negotiations, mainly, the processes that need to be executed at each peer during every trust negotiation.

6.1 Architecture

As already mentioned, the language for writing policies is Protune, whose syntax is described in section 4. However, the policies are used in order to infere new information during the filtering, as a final decision about giving access permissions or releasing credentials must be taken. For the filtering part an inference engine is necessary, the one used for this application is a Prolog based engine, JLog [3]. Since the policies are written in Protune and the filtering applied on policies is done in Prolog, a translator for policies and metapolicies from Protune to Prolog is essential. Furthermore, when messages are exchanged between two parties, the Protune representation for policies is used, therefore a translator from Prolog to Protune, for policies and metapolicies, is also required. The translators were implemented in JavaCC [1].

An overview of the application is depicted in figure 4. Every peer has a local policy written in Protune and a metapolicy which contains metadata, stating how the policy should be handled. When a request from a client is received, the local policy is translated from Protune to Prolog and then, the policy is given as input together with the request of the client to the module which is responsible for performing the filtering process. The filtered policy must be translated from Prolog into the Protune representation in order to be sent to the other party.

6.2 Protune-Prolog Translators

6.2.1 JavaCC

The translators from Protune to Prolog and from Prolog to Protune, necessary for transforming the policies during negotiations, were developed using JavaCC. JavaCC stands for Java Compiler Compiler and it is a parser generator to use with Java based applications. There is a plug-in for integrating JavaCC in the Eclipse platform, which was also used for this implementation.

JavaCC is a generator for parser and lexical analyzers, this means it reads the specification from a file, called a grammar and then, based on

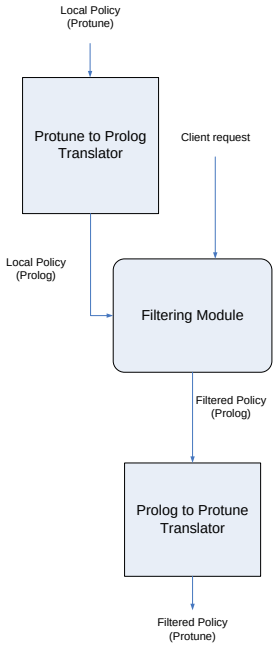

Figure 4: Trust Negotiation between Bob and the online bookstore

this grammar, it generates the lexical analyzer and the parser. Writing a parser directly in Java can be a difficult task as the interaction between rules needs to be studied carefully. By using a tool, such as JavaCC, rules can be written separately and this increases modularity so the specification files are easier to write, read, and modify compared with hand-written Java programs. Furthermore, using a parser generator, saves time and produces higher quality applications.

As explained in [2], a JavaCC grammar file contains a part dedicated to the lexical analyzer where the tokens used are declared in order to be able to identify them in the given input.

The syntax below:

$$SKIP : \{< NAME : REGULAR\_EXPRESSION >\}$$

means that the input string matching the regular expression, must not be taking into consideration by the lexical analyzer but skipped.

Declaration of a new token is:

$$TOKEN : \{< NAME : REGULAR\_EXPRESSION >\}$$

Whenever the input string matches the regular expression then this token, identified by name is returned by the lexical analyzer.

NAME is optional in both of the above rules. There can also be tokens without any name associated, but then it would not be possible to refer to them in the parser specification.

It must be stated that the order for declaring the tokens is important, as the lexical analyzer always returns the first token that matches the input stream.

The JavaCC grammar contains also a part in which the parser is specified, this consists of BNF productions. The representation of a BNF production is similar to Java method definition.

```
ReturnType MethodName() :
{ VariableDeclarations}
{ RegularExpression
        {JavaCode}
}
```

where

- $ReturnType \equiv$ the return type of the method,e.g., void, String, Vector.
- $MethodName \equiv$ the name of the method. This can be used in order to call this method from another one in support of modularity
- $VariableDeclarations \equiv$usual declarations of Java variables which can be used to process the token matched
- $RegularExpression \equiv$ the regular expression constructed with tokens that specifies the legitimate sequences of token kinds in an error free input
- $JavaCode \equiv$ the code for the method, simply Java code, which will be used in order to process the recognized sequence of tokens and return the result of the method.

The parser detects whether or not the input sequence is error free. It must include a special method called "Start" which stands for the main BNF production. Of course, this method can invoke other methods and thus expand the grammar.

Apart from the lexical analyzer and the parser, the JavaCC grammar file includes an "Options" section, where options for the parser generator can be specified and, of course, the class with the main method for calling the parser. The class must be defined between $PARSER\_BEGIN(ParseName)$ and

$PARSER\_END(ParserName)$, where ParserName is the name of the class, and it can include code for more complex methods which can be called from the parser methods in order to process the tokens. Since this is not meant to be a general description of JavaCC, but a simple presentation in order to introduce the translators, we will not go into details here.

When compiling the grammar file, JavaCC generates several Java classes

- TokenMgrError
  a class that deals with errors detected by the lexical analyzer
- ParseException
  a class that deals with errors detected by the parser
- Token
  a class for representing the input token stream. Each token is associated two important fields
  - an integer value to specify the token type, related to the name given to the token
  - a String that contains the sequence of characters from the input stream that matches the token
- SimpleCharStream
  a class for delivering caracthers to the lexical analyzer
- ParserConstants
  a interface to define the constants, implemented by the lexical analyzer and the parser
- ParserTokenManager
  the class that implements the lexical analyzer
- Parser
  the class that implements the parser

The last two specified classes do not need to have the name we used, if the name of the parser is ParserName then the lexical analyzer will be ParserNameTokenManager and the generated parser will be ParserName.

Another important aspect that must be mentioned is the lookahead. The parser mainly reads input stream and determines whether it matches the grammar description or not. This seems easy but it can be quite time consuming since the input stream can be quite long and for some parts of it more than one rule can be matched. This leads to backtracking and making

new choices and this implies a lot of time consumed. The time needed is also depended on the manner in which the grammar is written, as it can speed up the matching process.

JavaCC generated parsers make decisions at choice points based on exploration or future tokens and once a decision is taken no backtracking is made. Exploring further tokens from the input stream is called "looking ahead", from here the "LOOKAHEAD" term which is used to determine how many further tokens the parser should inspect before making a decision. The lookahead is a JavaCC option and by default it is set to one, before taking a decision the JavaCC generated parser inspects the next token. This means that the grammar is LL(1) and can be handled by top-down parsers. However, there are situations when more than one rule can match for some part of the input stream, but only one rule matches the whole stream. With only a lookahead of one this cannot be solved, but also increasing the default lookahead is not a good idea, as exploring more tokens require more time and the grammar will not be LL(1) anymore. The solutions we adopted in this case are

- factorization of rules
  The rules that have common parts are factored together in order to make just one rule. The grammar will still be LL(1), but this puts a heavy burden on the grammar writer. The grammar is more complicated, not so flexible and not easy to change anymore.

- local lookahead
  This solution introduces local hints for the parser when choices need to be made. When a choice between several rules needs to be made and a lookahead of one is not enough, we can instruct the parser to check several tokens in advance. This means using local lookahead when multiple choices are possible. By doing so, the grammar is mainly still LL(1) and is still flexible, however the parser will spend more time taking a decision when it finds a local lookahead.

In our implementation, we use both of the solutions described above, each one according to the situation that needs to be considered.

6.2.2 Protune to Prolog Translator

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.

As shown in section 4 a rule written in Protune has the general form

$$[Id]Head :- L_1, L_2, ..., L_n.$$

This is translated in Prolog as

$$rule(Id, Head, [L_1, L_2, ..., L_n]).$$

If the rule doesn't 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.

Translating of Protune metarules

$$[Id].Attribute : Value :- L_1, L_2, ..., L_n$$

or

$$Head.Attribute : Value :- L_1, L_2, ..., L_n$$

is performed as follows

$$metarule(id, Attribute(Id, Value), [L_1, L_2, ..., L_n])$$

for the first expression and

$$metarule(pred, Attribute(Head, Value), [L_1, L_2, ..., L_n])$$

for the second one.

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 introduces "id" and "pred" in order the 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

$$Id[Attribute_1 : Value_1, ..., Attribute_n : Value_n]$$

as such a construction is not allowed in Prolog. The general transformation for the above complex term is

$$complex\_term(Id, Attribute_1, Value_1), ..., complex\_term(Id, Attribute_n, Value_n)$$

However, taking in consideration Prolog semantics the translation is a little different according to the position in which the complex term appears

- the complex term is the head of a rule

  [r1]studentcard[owner:“MariaInes”,type:“Student”,
  issuer:“Hannover University”]:-Body.

  The translation for this case is:

  rule(r1,complex_term(studentcard,owner,“Maria Ines”),[Body]).
  rule(r1,complex_term(studentcard,type,“Student”),[Body]).
  rule(r1,complex_term(studentcard,issuer,“‘Hannover University”),[Body]).

- the complex term is an argument of a predicate that is a rule head

  [r2]allow(release(studentcard[owner:“MariaInes”,type:“Student”,
  issuer:“Hannover University”])):-Body.

  This is translated as:

  rule(r2, allow(release(studentcard)), [Body]).
  rule(r2,complex_term(studentcard,owner,“Maria Ines”),[Body]).
  rule(r2,complex_term(studentcard,type,“Student”),[Body]).
  rule(r2,complex_term(studentcard,issuer,“‘Hannover University”),[Body]).

- the complex term is in the body of a rule.

  [r3]Head:-studentcard[owner:“MariaInes”,type:“Student”,
  issuer:“Hannover University”].

  In this case we translate

  rule(r3,Head,
  [complex_term(studentcard,owner,“Maria Ines”),
  complex_term(studentcard,type,“Student”),
  complex_term(studentcard,issuer,“‘Hannover University”)]).

- the complex term is an argument of a predicate in the body of a rule

[r4]Head:-credential(studentcard[owner:"MariaInes",type:"Student",
issuer:"Hannover University"]).

The translation for this case is:

rule(r4,Head,[credential(studentcard),
complex_term(studentcard,owner,"Maria Ines"),
complex_term(studentcard,type,"Student"),
complex_term(studentcard,issuer,"'Hannover University")]).

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

The above different translations are necessary since in Prolog a rule head can be only one literal,it cannot be a conjunction of literals and the semantics differs for literals that are in the body of a rule, which need to be proved true, and for literals that appear in the head of the rule which hold if the body is true.

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 intresting issue is translating special built-in predicates. The syntax of the in predicate is

$$in(Predicate_1(Arg_1^1, .., Arg_n^1), \\ Package\_call : Predicate_2(Arg_1^2, .., Arg_m^2))$$

The translation performed looks like

$$in(Predicate_1(Arg_1^1, .., Arg_n^1), \\ Package\_call, Predicate_2, [Arg_1^2, .., Arg_m^2])$$

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 were 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.

6.2.3 Prolog to Protune Translator

The Prolog to Protune parser, implemented in JavaCC, receives as input the policy in the Prolog representation and produces as output a vector with all the rules and another vector holding all the metarules. The transformations performed are the reverse translations as described in the previous subsection.

When identified in the input stream, complex terms are stored as vectors and then are processed later according to the position in which they appear. Since the complex terms need to be reconstructed in the initial Protune format, the rules are stored internally as vectors with three elements : the rule's id stored as a String, the head of the rule stored as an Object since there are several possibilities and the body of the rule which is a String. After having processed all the policy, a method "building_rules" is called in order to form all the Protune rules as strings.

In order to process the complex terms several functions were defined in the parser class.

- private boolean is_complex_term(Object literal)
  As mentioned in section 4 a literal can be a predicate or a complex term or a comparison. This function checks wether the literal identified, the argument of the function, is a complex term.

- private String inline_literal(Object literal)
  This function constructs the image of the literal into a String, e.g., it maps together the predicate and its arguments.

- private String rebuild_complex_term(Vector complex_terms)
  This function rebuilds a complex term in the Protune syntax. Given a vector of elements belonging to a complex term, in the form complex_term(Id,Attribute,Value), it constructs the initial complex term, e.g.,
  $$Id[Attribute_1 : Value_1, ..., Attribute_n : Value_n]$$

- private Vector
  integrate_complex_term(Object literal,Vector complex_terms)
  This function integrates a complex term, whose elements are stored in the vector complex_terms, as argument of a predicate represented by the Object literal. The result is a vector with two elements: a String, the predicate's name and a Vector, the predicate's arguments, among which the complex term.

- private Object
  update_literal(Object literal,String id,String complex_term)
  This function is called by integrate_complex_term, it must replace the argument which matches the String id in the predicate represented by literal with the complex term provided

- private String process_body(Vector body)
  At first the body of a rule is stored as a Vector, when the whole rule is matched the body must be transformed into a String. This is the goal of the process_body function

- private void building_rules()
  This function processes the vector of rules in the policy, the rules stored as vectors will be transformed into Strings. This must be done at the end, as if a rule contains a complex term in the head then it has been splintered between several Prolog rules and the parser needs to rebuild the initial rule.

As like the Protune to Prolog translator, this translator does not change the semantics of the policy while transforming it.

6.3 Policy Filtering

The filtering process that must be applied on policies before sending them to the other party requires an inference engine. For the implementation we considered a Prolog based engine, JLog, as it is suitable for developers who need an embedded Prolog engine in Java.

In order to achieve our goal, a meta-interpreter for Prolog was developed. As explained in [8], a meta-program is a program that receives other programs as input. A meta-interpreter is a particular type of a meta-program: an interpreter for a language, written in the same language. Prolog is a powerfull language for meta-programming. By means of the developed meta-interpreter, written in Prolog, the Prolog engine can be instructed how to process policies translated in the Prolog representation.

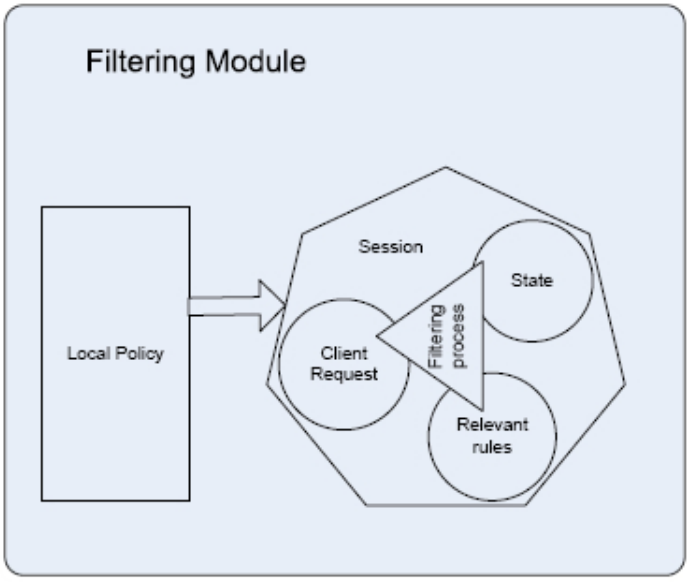

Figure 5: Filtering Module

In a peer-to-peer network it is likely that one party would have more than one on-going negotiation at a certain time. For this reason, the meta-interpreter must support sessions for each opened negotiation. Each session is identified by a number and must store the initial request. The filtering process of the policy is done within the session, since it depends on the request of the client. This means rules relevant for the client's request are saved in the session for filtering. Further on, the filtering requires additional information like credentials disclosed by the other party or the results of the actions which were executed during the filtering. This information is stored in the session state as literals. This is shown in figure 5

The local policy and metapolicy, containing all access control and privacy information, will be loaded by the meta-interpreter. As long as the policy respects the Prolog representation, the policy can be modified and loaded as many times as the peer wishes. For each session, necessary information from

the policy is used.

During the filtering, the designed meta-interpreter needs to communicate with a module. Communication is done via several functions which must be called by the controlling module. [4]

- $start\_session(+SId, +Head)$
  where
  - $SId \equiv$ the session identifier assigned for this negotiation
  - $Head \equiv$ the literal representing the request of the client

  This function is called each time a new client request arrives, therefore a new session needs to be started

- $add\_to\_state(+SId, +Literal)$
  where
  - $SId \equiv$ session id
  - $Literal \equiv$ the literal which must be added to the state session

  Information like disclosed credentials is necessary during the filtering process, so it must be stored in the session state before the filtering process starts. There are two types of literals which can be asserted to the session state
  - credentials disclosed by the other party These must be asserted as received(Literal).
  - actions results These must be asserted as action(successful(literal)) if the action was successful or action(unsuccessful(Literal)) otherwise.

- $filter\_policy1(+SId, -ActionList)$
  where
  - $SId \equiv$ session id
  - $ActionList \equiv$ a list that contains all the actions which are ready for execution and must be performed by the owner of the policy

[4] The sign "+" in front of a function argument specifies that it is an input, while the sign"-" means it is given as output of the function.

The above function starts the filtering process: it gathers the relevant rules, performs partial evaluation in case the state should contain disclosed credentials, refines the relevant rules set and then, selects the actions with actor self which are ready for execution. The list containing the actions is returned to the module which is responsible for their execution. The list contains elements of the form

$$immediate_action(Literal, Action)$$

where Literal is the provisional predicate and Action specifies the action that must be executed in order to make true Literal.

- $filter\_policy2(+SId, +ActionResults, -ActionList)$
  where
  - $SId \equiv$ session id
  - $ActionResults \equiv$ a list with the results of the successfully executed actions
  - $ActionList \equiv$ a list with all the actions which become ready for execution after partial evaluating against the results of the actions

  As explained before, the actions must be executed within a loop because, as new actions are successfully executed and partial evaluation is done against the action results, new actions may become available for execution. The "filter_policy2" function does partial evaluation against the state and selects the available actions if any. This function should be called in a loop until the ActionList is empty, that is there are no more actions ready for execution. The ActionResults list must contain elements in the form

  $$successful(Literal)$$

  or

  $$unsuccessful(Literal)$$

  depending whether the action was successfully executed or not.

- $filter\_policy3(+SId)$
  where
  - $SId \equiv$session id

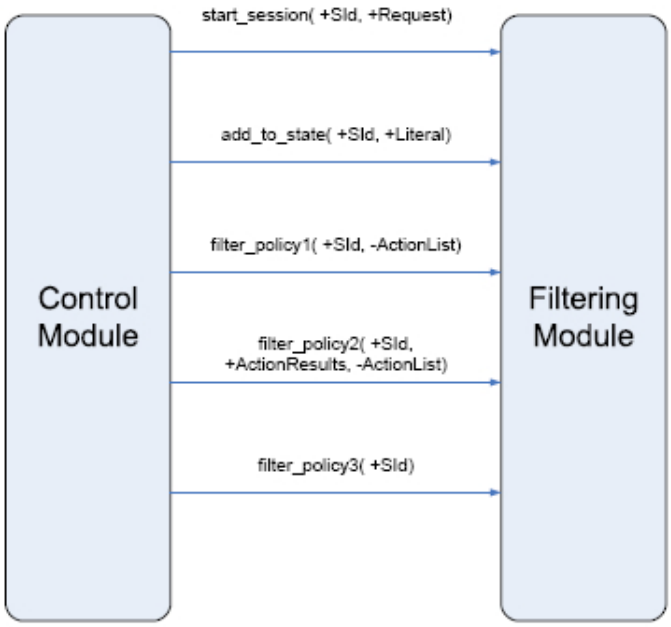

Figure 6: Interface to the filtering module

This function resumes the filtering process started by the other two functions "filter_policy1" and "filter_policy2". It performs the blurring of the predicates, adds necessary information expressed in the metapolicy, as peer actions and explanations, renames the required predicates and refines several times the relevant rules set.

The interface between the filtering module and the module which must control the filtering process is depicted in figure 6.

In a policy written in Protune, information is expressed in the form of rules, this is also valid in the Prolog transformation of a policy. The final goal of the negotiation is to take a decision, considering the request of the client. In order to do this, rules in the policy need to be inferred and a rule granting access for the client must be proved true. For this a procedure is needed. It must verify that a given rule or metarule holds, that is, its body is true. This is shown below:

```
prove_body( SId, []).
prove_body( SId, [ First |Rest]):-
        prove( SId, First),
        prove_body( SId, Rest).
```

The above procedure, "prove_body", tries to prove, sequentially, every literal in the body of the rule or metarule, which is the second argument. If a single literal in the body cannot be proved true, considering that anything which is not specified is false, then the rule or metarule does not hold. SId is the session id, this is necessary to be passed along as a literal must be checked against the session to decide if is true or not.

The "prove" procedure is the one that tries to prove a goal, Goal, against the policy and the session identified by SId.

```
prove( SId, not( Goal) ):-
        not( prove( SId, Goal) ), !.
```

This part tries to prove the negation of a goal, that is the literal Goal does not hold. Thanks to the closed-world assumption, this is achieved if trying to prove the literal Goal fails.

```
prove( SId, metarule( id, Field) ):-
        Field =..[ Attribute, Id, Value],
        metarule( SId, Id, _, Attribute, Value),!.

prove( SId, metarule( pred, Field)):-
        Field =..[ Attribute, Head, Value],
        metarule( SId, _, Head, Attribute, Value),!.
```

Metabodies, the body of the metarules, must also be proved. As seen in 4, the body of a metarule can contain a reference to another metarule, so the referenced metarule must be proved true when trying to determine if the initial metabody holds. Reference to a metarule is done via

$$metarule(id, Attribute(Id, Value))$$

for rules and via

$$metarule(pred, Attribute(Head, Value))$$

for predicates. Information upon the referenced metarule is unwrapped and a special procedure "metarule", which is responsible for determining if a metarule is in the policy, is called.

```
prove( SId, Goal):-
        session( SId, rule( _, Goal, Body) ),
        prove_body( SId, Body),!.
```

The body of a rule may contain an abbreviation predicate, this specifies that a rule whose head matches the abbreviation predicate must be proved true for the abbreviation predicate to hold. In this case both of the rules are in the relevant rule set.

```
prove( SId, Goal):-
        rule( _, Goal, Body),
        prove_body( SId, Body).
```

Metarules are not considered in the relevant rule set, but one metarule can contain in its body reference to another rule. Since this rule may not be stored in the session, the whole policy needs to be checked for a matching rule that holds.

```
prove( SId, Goal):-
        session( SId, state( Goal )),!.
```

A rule may become true if the other party has given a credential or if an action was executed successfully. This information can be founded in the session state. For this reason a goal must as well be verified against the session state.

```
prove( SId, Goal):-
        Goal.
```

At last, a goal can also be a comparison or a predicate like "ground". In order to establish this, is suffices a call to the Prolog engine.

For the above procedure "cuts", e.g., "!", were introduced for optimization.

Each of the filtering steps must perform different transformations upon the rules. In order to help with modularity and possible future changes each filtering phase finds the rules of the filtered policy in the session in the format

$$rule(Id, Head, Body)$$

and, after processing them, leaves them in the same format.

Rules in the policy are processed with the help of metarules. In order to check if a metarule exists or to retrieve values of attributes from a metarule, which refers to a predicate or a rule, a procedure was defined.

```
metarule( SId, Id, _, Attribute, Value):-
        nonvar( Id),
        F =..[ Attribute, Id, Value],
        metarule( id, F, Body),
        prove_body( SId, Body),!.

metarule( SId, _, Head, Attribute, Value):-
        nonvar( Head),
        F =..[ Attribute, Head, Value],
        metarule( pred, F, Body),
        prove_body( SId, Body).
```

The procedure first checks that the reference item, the id of the rule or the predicate, is bounded. Then is tries to find the metarule in the policy and if this succeeds, all left is to determine wether the body of the metarule is true.

A procedure was defined for each step of the filtering process described in section 6.3.

- relevant_rules1(+SId)
  This function considers the rules in the initial policy whose head match the request, which is stored in the session identified by SId, adds them to the session and then it processes each rule's body in order to retrieve other rules that are referenced by this ones. The entire body of the rules that belong to a session are processed, therefore, in the end, all the relevant rules set is included in the session.

- relevant_rules2(+SId)
  This function corresponds to the intermediate relevant rules steps in the filtering. It refines the set of rules that are in the session in order to exclude rules that are not relevant anymore due to filtering. The function has a similar behavior with the one described above, except it considers the rules from the session not the whole set of rules in the initial policy. It finds the rules in the form rule(Id, Head, Body) and after processing the relevant rules are renamed to relevant_rule(Id, Head, Body) to separate them from the ones that are now irrelevant. After erasing the irrelevant rules from the session, the relevant set of rules need to be renamed back to rule(Id, Head, Body) to ensure flexibility of the filtering process. For this purpose a function, rename_rules(+SId, +old_name, +new_name), is called; it must rename the old_name rules from the session identified by SId to new_name rules.

- partial_evaluation(+SId)
  The above function performs partial evaluation of the rules in the session against the session state: each rule's body is processed in order to determine the literals that unify with the grounded literals in the state, when this happens the matching literal is removed from the body and the propagation of variables that became instantiated, if any, is executed.

- select_actions(+SId, -ActionList)
  The body of the rules in the session are processed in search of immediate actions, that is predicates for which are defined metarules stating they are provisional_predicates with actor peer and evaluation immediate. These predicates are asserted in session as immediate_action(Predicate, Action) and then collected in a list, ActionList, which is returned by the function.

- process_action_results(+SId, +ActionResults),
  After executing the actions, the results of the successfully actions are needed in the filtering process. This function is responsible for asserting the actions results, which are given in the ActionResult list, to the session state.

- blurring_rules(+SId)
  This function processes each session rule's body in search of literals that must be blurred. For each literal that must be blurred, which is checked via a must_blurr(+SId, +Id, +Predicate) function, the literal is wrapped inside the "blurred" particle and renamed. Special care is also taken in case of nested predicates, e.g., if we have a Predicate(Literal), where Literal must be blurred then the function transforms these in Predicate(blurred(Literal)).

- select_peer_actions(+SId)
  The above function is responsible for finding the actions that need to be executed by the other party. It checks via the metarule procedure for provisional predicates with actor peer and it adds them to the session in order to be sent to the other party together with the policy.

- add_explanations(+SId)
  This function considers each rule in the session to determine whether the rule, the head of the rule or the literals in the body have explanations attached. If this should be the case the explanations from the metapolicy are stored in the session via get_explanation(+SId, +Id,

+Head) together with the policy that must be sent to the other party. Special care needs to be taken as explanations can also be attached to blurred predicates.

- rename_predicates(+SId)
  As explained in section 6.3 some predicates in the policy have to be renamed for security reasons. The purpose of this function is to consider each session rule and for every literal that needs to be renamed, which is determined via the must_rename procedure, a new general meaningless name "predicateN" is given to the predicate name. The old name and the new name are temporally stored in the session as rename_predicate(OldName,NewName) in order that all the occurrences of some predicate are changed to the same new name. Predicate renaming must also be done for the explanations of predicates in the policy.

As shown in the above examples, Prolog seems a natural choice for the inference engine needed for the filtering of policies. Furthermore, a constant goal while developing the meta-interpreter was its flexibility in order not to impose constraints and to be flexible to future changes.

6.4 Credential Selection

In a policy several methods can be specified for getting access to the same resource or releasing one credential. Someone who wants access to a specified resource or needs to obtain some credential must be aware of all his possibilities and choose the method it fits him the best. In order to support this, we developed a credential selection module in Prolog.

The credential selection module receives as input the request and the filtered policy sent by the other party, both of them in the Prolog representation. The select_credentials(-Root) procedure is then called and its output is an "And-Or" tree representing all the possible methods for satisfying the request which are included in the policy. Each node in the tree has one number for identification. The Root argument is the number given to the root of the tree.

An "And-Or" tree contains two types of nodes:

- "*And*" *node* $\equiv$ all the node's children need to proved, in our case all the credentials specified by the node's children are required
- "*Or*" *node* $\equiv$ it suffices to prove one of the node's children, in our case it suffices to release one of the credentials specified by this node's children.

An "And-Or" tree node is represented via an assertion of the form

$$tree(No, Type, List)$$

where

- $No \equiv$ the node's number, this is used to reference the node from other nodes descriptions, to establish links between nodes
- $Type \equiv$ the type of the node, this is either "and" or "or"
- $List \equiv$ the list which contains credentials that must be given, actions that must be executed and references to other nodes

Each rule in the policy will be represented as an "and" node: a conjunction of all the credentials and actions referenced in the rule's body. Since some of them will have the List argument empty, they will not be included in the tree.

The "select_credentials" procedure considers all the rules in the filtered policy that match the request and processes each one of them. After every rule is processed, it receives a number and needs not be processed again if referenced by another rule. Processing a rule means that for each literal in its body if it represents

- a credential or a declaration
  it is added to List of the rule it appears in
- an action the peer must perform
  it is added to List of the rule it appears in
- an abbreviation predicate
  the rules whose head match the abbreviation predicate are processed and represented as nodes, then their representations are gathered into an "or" node whose number is store in the List of the rule were the abbreviation predicate appears.

For the policy given as example in section 4, the "And-Or" tree is shown in figure 7.

The resulting "And-Or" tree will be used by the negotiation strategy in order to choose the next credentials to send.

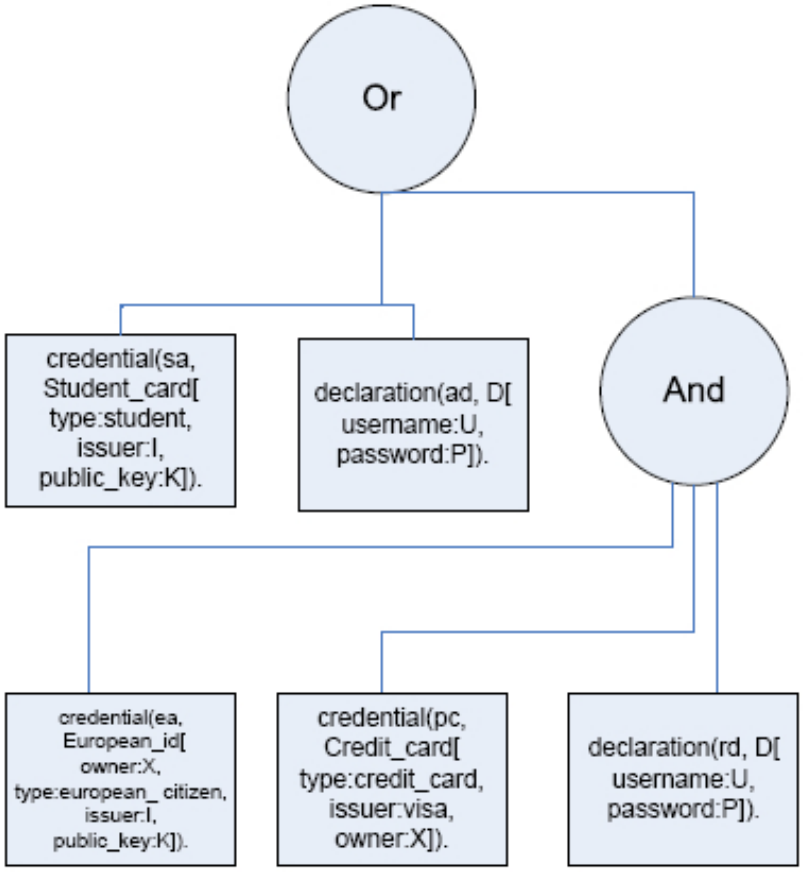

Figure 7: The "And-Or" tree for the policy given as example

7 Conclusions and Open Issues

Face-to-face business transactions have their vulnerabilities but they are the most common way for trading goods. Online interactions between strangers should be as easy to perform as if the transaction was conducted face-to-face. In order to achieve this, a certain level of trust must first be established among the two parties involved in a transaction. Trust can be established gradually by disclosure of credentials leading to trust negotiation. Negotiation of trust must be a bilateral process in which each party may define under what circumstances it is willing to release its credentials and may request for disclosure of credentials from the other party in order to allow access to a service. Access control issues and circumstances under which credentials are to be disclosed can be expressed in policies. With the use of policies trust negotiation can be done automatically by a run-time system. The purpose of the run-time system is that of establishing trust between service providers and requesters, nevertheless it must provide security and privacy to its users.

This thesis brings its contribution in the area of dynamic trust negotiation. More specifically, it describes and implements the processes that need to take place at each peer in order to support negotiation for trust establishment. This involves

- extension of Protune language and specification of its Prolog representation
- extension of the filtering process applied on policies and metapolicies before disclosing them such that private information is not unrevealed
- started work on credentials selection so that one party should be able to consider all the alternatives it can follow to satisfy the other party's policy
- development of translators for policies and metapolicies from the policy language into a logic language where information can be inferred
- specification and implementation of an interface for the filtering module

All this is going to be part of the run-time system responsible for negotiating trust between two parties on the Semantic Web.

Future work regarding interactions between parties is still required in order to complete the process of automatic trust negotiation. Online transactions between peers need to be considered carefully. Negotiation strategies are also to be described : one party can decide to follow multiple alternatives for getting access to a service in order to speed up the process while another

party may prefer to be very cautions with its credentials and disclose only one credential a time. Furthermore, the modules for integrating legacy systems like databases in the run-time system need to be added. Explanations must be refined and specifying actions that are to be executed by the system also requires further research.

In the current work several assumptions were made:

- All the credentials and complex terms, that appear in the description of the policy and need to be checked, are credentials which the other party is suppose to provide during the negotiation process. In the future, local ones will also be referenced.

- The policy language grammar does not deal with nested complex terms, e.g., the value of one attribute is another complex term. For example

$$Id_1[Attribute_1^1 : Value_1^1, .., Attribute_n^1 : \\ Id_2[Attribute_1^2 : Value_1^2, .., Attribute_n^2 : Value_n^2]]$$

  This would be usefull for allowing credential chains.

- For private predicates that need to be blurred, they are wrapped inside the blurred particle and their names are changed. Sometimes it may be necessary to also hide their arguments as they could contain sensitive information. This could be achieved by replacing the occurrences by a predefined particle, for example "blurred_n".

- The grammar defined for the policy language allows constructions like

$$Id[Attribute_1^1 : Value_1^1, .., Attribute_n^1 : Value_n].Attribute : Value$$

  However, this is currently not processed during the filtering, but was introduced for future versions that may make use of it.

  These assumptions were made in order to keep the first version of the application simple. However, they need to be considered for future development of the application.

Acknowledgments

I would like to thank Daniel Olmedilla for his advice and guidance regarding my bachelor research and for all his support during the six month I have spent at L3S Research Center. Also, I want to thank Juri Luca De Coi for his help in what concerns the formalization of the negotiation model.

References

[1] Javacc project. https://javacc.dev.java.net/.

[2] Javacc tutorial. http://www.engr.mun.ca/ theo/JavaCC-Tutorial.

[3] Jlog prolog project. http://jlogic.sourceforge.net/.

[4] Piero Bonatti, Daniel Olmedilla, and Joachim Peer. Advanced policy explanations. Technical report, Working Group I2, EU NoE REWERSE, August 2005. `http://rewerse.net/deliverables/m18/i2-d4.pdf`.

[5] Piero A. Bonatti, Claudiu Duma, Daniel Olmedilla, and Nahid Shahmehri. An integration of reputation-based and policy-based trust management. In *Semantic Web Policy Workshop in conjunction with 4th International Semantic Web Conference*, Galway, Ireland, November 2005.

[6] Piero A. Bonatti and Daniel Olmedilla. Driving and monitoring provisional trust negotiation with metapolicies. In *6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*, pages 14–23, Stockholm, Sweden, June 2005. IEEE Computer Society.

[7] Piero A. Bonatti and Daniel Olmedilla. Policy language specification. Technical report, Working Group I2, EU NoE REWERSE, February 2005. `http://rewerse.net/deliverables/m12/i2-d2.pdf`.

[8] Ivan Bratko. Prolog, programming for artificial intelligence.

[9] Rita Gavriloaie, Wolfgang Nejdl, Daniel Olmedilla, Kent E. Seamons, and Marianne Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In *1st European Semantic Web Symposium (ESWS 2004)*, volume 3053 of *Lecture Notes in Computer Science*, pages 342–356, Heraklion, Crete, Greece, May 2004. Springer.

[10] T. Grandison. Trust management for internet applications.

[11] Wolfgang Nejdl, Daniel Olmedilla, and Marianne Winslett. Peertrust: Automated trust negotiation for peers on the semantic web. In *VLDB Workshop on Secure Data Management (SDM)*, volume 3178 of *Lecture Notes in Computer Science*, pages 118–132, Toronto, Canada, August 2004. Springer.

[12] Daniel Olmedilla. Trust negotiation strategies in protune.

[13] K. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobsen, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation.

[14] V.S. Subrahmanian, Sibel Adali, Anne Brink, James J. Lu, Adil Rajput, Timothy J. Rogers, Robert Ross, and Charles Ward. Hermes: Heterogeneous reasoning and mediator system.

[15] Marianne Winslett. An introduction to trust negotiation. In *iTrust*, pages 275–283, 2003.

A Protune grammar

BASIC DATATYPES

```
<STRING_QUOTED>          ->"'"  (~["'","\\","\n","\r"])* "'"
                          | "\""  (~["\"","\\","\n","\r"])* "\""

<ANY_CHARACTER_BUT_EOF> -> (~["\n","\r"])*

<DIGIT>                  -> 0 | 1 | ... | 9

<LOWER_CASE>             -> a | b | ... | z

<UPPER_CASE>             -> A | B | ... | Z

<NUMBER>                 -> <DIGIT> <NUMBER>
                          | <DIGIT>

<STRING_EXTENDED>        -> <LOWER_CASE> <STRING_EXTENDED>
                          | <UPPER_CASE> <STRING_EXTENDED>
                          | _ <STRING_EXTENDED>
                          | <NUMBER> <STRING_EXTENDED>
                          | E

<STRING_CONSTANT>        -> <LOWER_CASE> <STRING_EXTENDED>
                          | <STRING_QUOTED>

<CONSTANT>               -> <STRING_CONSTANT>
                          | <NUMBER>

<VARIABLE>               -> <UPPER_CASE> <STRING_EXTENDED>
                          | _ <STRING_EXTENDED>

<RULE_SEP>               -> <-
                          | :-

<NEG_SYMBOL>             -> not
                          | \+

<OPERATOR>               -> =
```

```
                        | >
                        | >=
                        | <
                        | <=
                        | !=
                        | is
```

PROGRAM RULES

```
<PROGRAM>               -> <DIRECTIVE_LIST> <RULE_LIST>

<DIRECTIVE_LIST>        -> <DIRECTIVE> <DIRECTIVE_LIST>
                         | E

<DIRECTIVE>             -> include <STRING_QUOTED>

<RULE_LIST>             -> <RULE>
                         | <RULE> <RULE_LIST>
                         | <META_RULE>
                         | <META_RULE> <RULE_LIST>
                         | <COMMENT>
                         | <COMMENT> <RULE_LIST>

<COMMENT>               -> "%" <ANY_CHARACTER_BUT_EOF> <EOF>
                         | "//" <ANY_CHARACTER_BUT_EOF> <EOF>
                         | "/*" ( )* "*/"

<RULE>                  -> [ <ID> ] <HEAD_LITERAL> <RULE_SEP> <BODY> .
                         | [ <ID> ] <HEAD_LITERAL> .
                         | <HEAD_LITERAL> <RULE_SEP> <BODY> .
                         | <HEAD_LITERAL> .

<META_RULE>             -> <META_HEAD_LITERAL> <RULE_SEP> <META_BODY> .
                         | <META_HEAD_LITERAL> .

<ID>                    -> <CONSTANT>

<HEAD_LITERAL>          -> <STRING_CONSTANT>
                         | <PREDICATE_LITERAL>
                         | <COMPLEX_TERM>
```

```
<PREDICATE_LITERAL>     -> <PREDICATE> ( )
                         | <PREDICATE> ( <ARGUMENT_LIST> )

<META_HEAD_LITERAL>     -> [ <ID> ] . <FIELD>
                         | <HEAD_LITERAL> . <FIELD>

<LITERAL>               -> <HEAD_LITERAL>
                         | <NEG_SYMBOL> <HEAD_LITERAL>
                         | <TERM> <OPERATOR> <TERM>
                         | <NEG_SYMBOL> <TERM> <OPERATOR> <TERM>
                         | <SPECIAL_LITERAL>

<META_LITERAL>          -> <META_HEAD_LITERAL>
                         | <NEG_SYMBOL> <META_HEAD_LITERAL>
                         | <LITERAL>

<BODY>                  -> <LITERAL_LIST>
                         | E

<META_BODY>             -> <META_LITERAL_LIST>
                         | E

<LITERAL_LIST>          -> <LITERAL>
                         | <LITERAL> , <LITERAL_LIST>

<META_LITERAL_LIST>     -> <META_LITERAL>
                         | <META_LITERAL> , <META_LITERAL_LIST>

<TERM>                  -> <VARIABLE>
                         | <CONSTANT>

<TERM_LIST>             -> <TERM>
                         | <TERM> , <TERM_LIST>

<COMPLEX_TERM>          -> <VARIABLE> [ <FIELD_LIST> ]
                         | <STRING_CONSTANT> [ <FIELD_LIST> ]

<ANY_TERM>              -> <COMPLEX_TERM>
                         | <TERM>
```

```
<FIELD>-> <ATTRIBUTE> : <VALUE>

<FIELD_LIST>-> <FIELD>
 | <FIELD> , <FIELD_LIST>

<SPECIAL_LITERAL>       -> in ( <FUNCTION> , <PACKAGE_CALL> )
                         | declaration ( <ID> , <ANY_TERM> )
                         | credential ( <ID> , <ANY_TERM> )

<PACKAGE_CALL>          -> <PACKAGE> : <FUNCTION>

<FUNCTION>              -> <PREDICATE>
                         | <PREDICATE> ( )
                         | <PREDICATE> ( <TERM_LIST> )

<ARGUMENT>              -> <ANY_TERM>
                         | <PREDICATE_LITERAL>

<ARGUMENT_LIST>         -> <ARGUMENT>
                         | <ARGUMENT> , <ARGUMENT_LIST>

<ATTRIBUTE>             -> <STRING_CONSTANT>

<VALUE>                 -> <TERM>

<PREDICATE>             -> <STRING_CONSTANT>

<PACKAGE>               -> <STRING_CONSTANT>
```

B Policy and metapolicy

Protune policy and metapolicy

```
[f1]recognized_university(upb).
[f2]recognized_university(hu).
[f3]recognized_university(epfl).

[f4]passwd(mirela,alerim).
[f5]passwd(dragos,sogard).
[f6]passwd(alina,anila).

[f7]has_subscription(dragos,books).
[f8]has_subscription(dragos,videotec).
[f9]has_subscription(mirela,sonotec).
[f10]has_subscription(mirela,books).
[f11]has_subscription(alina,books).

[f12]trusted_organization(ec).
[f13]trusted_organization(euh).

[f14]price(books,5).
[f15]price(videotec,9).
[f16]price(sonotec,7).

[r1]allow(access(Resource)):-
        credential(sa,Student_card[type:student,issuer:I,public_key:K]),
        valid_credential(Student_card,I),
        recognized_university(I),
        challenge(K).

[r2]allow(access(Resource)):-
        authenticate(U),
        has_subscription(U,Resource).

[r3]authenticate(U):-
        declaration(ad,D[username:U,password:P]),
        passwd(U,P).

[r4]allow(access(Resource)):-
        european_citizen(X),
```

```
        paid(X,Resource),
        register(X,U),
        assert(has_subscription(U,Resource)).

[r5]european_citizen(X):-
        credential(ea,European_id[owner:X,type:european_citizen,
        issuer:I,public_key:K]),
        valid_credential(European_id,I),
        trusted_organization(I),
        challenge(K).

[r6]paid(X,Resource):-
        price(Resource,P),
        credit_card_payment(X,P),
        logged("$X paid $P for the resource $R").

[r7]credit_card_payment(X,P):-
        credential(pc,Credit_card[type:credit_card,issuer:visa,owner:X]),
        valid_credential(Credit_card,visa),
        charged(Credit_card,P).

[r8]charged(C,P):-not_revoked(C),
        transfer_money(C,P).

[r9]register(X,U):-
        declaration(rd,D[username:U,password:P]),
        check(U,P,X).

[r10]check(U,_,X):-
        passwd(U,_),
        register(X).

[r11]check(U,P,X):-
        not(passwd(U,_)),
        assert(passwd(U,P)),
        logged("New user: $X registered as $U").

[r12]valid_credential(C,I):-
        public_key(I,K),
        verify_signature(C,K).
```

```
    /* Metapolicy:*/

public_key(_,_).type:provisional_predicate.
public_key(I,_).evaluation:immediate:-ground(CA).
public_key(_,_).actor:self.
public_key(I,K).action:
        "connect to the server and get $I's public key $K".
public_key(_,_).sensitivity:private.

verify_signature(_,_).type:provisional_predicate.
verify_signature(C,K).evaluation:immediate:-ground(C),ground(K).
verify_signature(_,_).actor:self.
verify_signature(C,K).action:
        "verify the signature on the credential $C using public key $K".
verify_signature(_,_).sensitivity:private.

recognized_university(_).type:state_predicate.
recognized_university(_).evaluation:immediate.
recognized_university(_).sensitivity:public.

authenticate(_).type:abbreviation_predicate.
authenticate(_).sensitivity:public.

passwd(_,_).type:state_predicate.
passwd(_,_).evaluation:immediate.
passwd(_,_).sensitivity:private.

has_subscription(_,_).type:state_predicate.
has_subscription(_,_).evaluation:immediate.
has_subscription(_,_).sensitivity:private.

european_citizen(_).type:abbreviation_predicate.
european_citizen(_).sensitivity:public.

trusted_organization(_).type:state_predicate.
trusted_organization(_).evaluation:immediate.
trusted_organization(_).sensitivity:public.

paid(_,_).type:abbreviation_predicate.
```

```
paid(_,_).sensitivity:public.

price(_,_).type:state_predicate.
price(_,_).evaluation:immediate.
price(_,_).sensitivity:public.

credit_card_payment(_,_).type:abbreviation_predicate.
credit_card_payment(_,_).sensitivity:public.

charged(_,_).type:abbreviation_predicate.
charged(_,_).sensitivity:public.

not_revoked(_).type:provisional_predicate.
not_revoked(C).evaluation:immediate :- ground(C).
not_revoked(_).actor:self.
not_revoked(C).action:
        "connect to Visa server and check to see if $C was revoked".
not_revoked(_).sensitivity:private.

transfer_money(_,_).type:provisional_predicate.
transfer_money(C,P).evaluation:immediate:-ground(C),ground(P).
transfer_money(_,_).actor:self.
transfer_money(C,P).action:
        "connect to Visa server and transfer $P from $C
                to the library's account".
transfer_money(_,_).sensitivity:public.

logged(_).type:provisional_predicate.
logged(_).evaluation:deferred.
logged(_).actor:self.
logged(M).action:
        "write message $M in the log file".
logged(_).sensitivity:private.

register(_,_).type:abbreviation_predicate.
register(_,_).sensitivity:public.
    check(_,_,_).type:abbreviation_predicate.
check(_,_,_).sensitivity:public.

assert(_).type:provisional_predicate.
assert(X).evaluation:immediate:-ground(X).
```

```
assert(_).actor:self.
assert(X).action:
        "add to the database $X".
assert(_).sensitivity:public.

valid_credential(_,_).type:abbreviation_predicate.
valid_credential(_,_).sensitivity:public.

challenge(_).type:provisional_predicate.
challenge(K).evaluation:immediate:-ground(K).
challenge(_).actor:self.
challenge(K).action:
        "challenge peer to prove that he has the private key
        corresponding to public key $K".
challenge(_).sensitivity:public.
```

Policy and metapolicy in Prolog representation

```
rule(f1,recognized_university(upb),[]).
rule(f2,recognized_university(hu),[]).
rule(f3,recognized_university(epfl),[]).

rule(f4,passwd(mirela,alerim),[]).
rule(f5,passwd(dragos,sogard),[]).
rule(f6,passwd(alina,anila),[]).

rule(f7,has_subscription(dragos,books),[]).
rule(f8,has_subscription(dragos,videotec),[]).
rule(f9,has_subscription(mirela,sonotec),[]).
rule(f10,has_subscription(mirela,books),[]).
rule(f11,has_subscription(alina,books),[]).

rule(f12,trusted_organization(ec),[]).
rule(f13,trusted_organization(euh),[]).

rule(f14,price(books,5),[]).
rule(f15,price(videotec,9),[]).
rule(f16,price(sonotec,7),[]).

rule(r1,allow(access(Resource)),[credential(sa,Student_card),
    complex_term(Student_card,type,student),
    complex_term(Student_card,issuer,I),
    complex_term(Student_card,public_key,K),
    valid_credential(Student_card,I),
    recognized_university(I),challenge(K)]).

rule(r2,allow(access(Resource)),[authenticate(U),
    has_subscription(U,Resource)]).

rule(r3,authenticate(U),[declaration(ad,D),
    complex_term(D,username,U),
    complex_term(D,password,P),
    passwd(U,P)]).

rule(r4,allow(access(Resource)),[european_citizen(X),
    paid(X,Resource),
    register(X,U),
```

```
    assert(has_subscription(U,Resource))]).

rule(r5,european_citizen(X),[credential(ea,European_id),
    complex_term(European_id,owner,X),
    complex_term(European_id,type,european_citizen),
    complex_term(European_id,issuer,I),
    complex_term(European_id,public_key,K),
    valid_credential(European_id,I),
    trusted_organization(I),
    challenge(K)]).

rule(r6,paid(X,Resource),[price(Resource,P),
    credit_card_payment(X,P),
    logged("${X} paid ${P} for the resource ${R}")]).

rule(r7,credit_card_payment(X,P),[credential(pc,Credit_card),
    complex_term(Credit_card,type,credit_card),
    complex_term(Credit_card,issuer,visa),
    complex_term(Credit_card,owner,X),
    valid_credential(Credit_card,visa),
    charged(Credit_card,P)]).

rule(r8,charged(C,P),[not_revoked(C),
    transfer_money(C,P)]).

rule(r9,register(X,U),[declaration(rd,D),
    complex_term(D,username,U),
    complex_term(D,password,P),
    check(U,P,X)]).

rule(r10,check(U,_,X),[passwd(U,_),
    register(X)]).
rule(r11,check(U,P,X),[not(passwd(U,_)),
    assert(passwd(U,P)),
    logged("New user: ${X} registered as ${U}")]).

rule(r12,valid_credential(C,I),[public_key(I,K),
    verify_signature(C,K)]).

metarule(pred,sensitivity(allow(_),public),[]).
```

```
metarule(pred,type(public_key(_,_),provisional_predicate),[]).
metarule(pred,evaluation(public_key(I,_),immediate),[ground(I)]).
metarule(pred,actor(public_key(_,_),self),[]).
metarule(pred,action(public_key(I,K),
    "connect to ${C} server and get ${I}'s public key ${K}"),[]).
metarule(pred,sensitivity(public_key(_,_),private),[]).

metarule(pred,type(verify_signature(_,_),provisional_predicate),[]).
metarule(pred,evaluation(verify_signature(C,K),immediate),
    [ground(C),ground(K)]).
metarule(pred,actor(verify_signature(_,_),self),[]).
metarule(pred,action(verify_signature(C,K),
    "verify the signature on the credential ${C} using public key
     ${K}"),[]).
metarule(pred,sensitivity(verify_signature(_,_),private),[]).

metarule(pred,type(recognized_university(_),state_predicate),[]).
metarule(pred,evaluation(recognized_university(_),immediate),[]).
metarule(pred,sensitivity(recognized_university(_),public),[]).

metarule(pred,type(authenticate(_),abbreviation_predicate),[]).
metarule(pred,sensitivity(authenticate(_),public),[]).

metarule(pred,type(passwd(_,_),state_predicate),[]).
metarule(pred,evaluation(passwd(_,_),immediate),[]).
metarule(pred,sensitivity(passwd(_,_),private),[]).

metarule(pred,type(has_subscription(_,_),state_predicate),[]).
metarule(pred,evaluation(has_subscription(_,_),immediate),[]).
metarule(pred,sensitivity(has_subscription(_,_),private),[]).

metarule(pred,type(european_citizen(_),abbreviation_predicate),[]).
metarule(pred,sensitivity(european_citizen(_),public),[]).

metarule(pred,type(trusted_organization(_),state_predicate),[]).
metarule(pred,evaluation(trusted_organization(_),immediate),[]).
metarule(pred,sensitivity(trusted_organization(_),public),[]).

metarule(pred,type(paid(_,_),abbreviation_predicate),[]).
metarule(pred,sensitivity(paid(_,_),public),[]).
```

```
metarule(pred,type(price(_,_),state_predicate),[]).
metarule(pred,evaluation(price(_,_),immediate),[]).
metarule(pred,sensitivity(price(_,_),public),[]).

metarule(pred,type(credit_card_payment(_,_),abbreviation_predicate)
    ,[]).
metarule(pred,sensitivity(credit_card_payment(_,_),public),[]).

metarule(pred,type(charged(_,_),abbreviation_predicate),[]).
metarule(pred,sensitivity(charged(_,_),public),[]).

metarule(pred,type(not_revoked(_),provisional_predicate),[]).
metarule(pred,evaluation(not_revoked(C),immediate),[ground(C)]).
metarule(pred,actor(not_revoked(_),self),[]).
metarule(pred,action(not_revoked(C),
    "connect to Visa server and check to see if ${C} was revoked")
     ,[]).
metarule(pred,sensitivity(not_revoked(_),private),[]).

metarule(pred,type(transfer_money(_,_),provisional_predicate),[]).
metarule(pred,evaluation(transfer_money(C,P),immediate),
    [ground(C),ground(P)]).
metarule(pred,actor(transfer_money(_,_),self),[]).
metarule(pred,action(transfer_money(C,P),
    "connect to Visa server and transfer the money ${P} from ${C}
     to the library's account"),[]).
metarule(pred,sensitivity(transfer_money(_,_),public),[]).

metarule(pred,type(logged(_),provisional_predicate),[]).
metarule(pred,evaluation(logged(_),deferred),[]).
metarule(pred,actor(logged(_),self),[]).
metarule(pred,action(logged(M),
    "write message ${M} in the log file"),[]).
metarule(pred,sensitivity(logged(_),private),[]).

metarule(pred,type(register(_,_),abbreviation_predicate),[]).
metarule(pred,sensitivity(register(_,_),public),[]).

metarule(pred,type(check(_,_,_),abbreviation_predicate),[]).
metarule(pred,sensitivity(check(_,_,_),public),[]).
```

```
metarule(pred,type(assert(_),provisional_predicate),[]).
metarule(pred,evaluation(assert(X),immediate),[ground(X)]).
metarule(pred,actor(assert(_),self),[]).
metarule(pred,action(assert(X),"add to the database ${X}"),[]).
metarule(pred,sensitivity(assert(_),public),[]).

metarule(pred,type(valid_credential(_,_),abbreviation_predicate)
    ,[]).
metarule(pred,sensitivity(valid_credential(_,_),public),[]).

metarule(pred,type(challenge(_),provisional_predicate),[]).
metarule(pred,evaluation(challenge(K),immediate),[ground(K)]).
metarule(pred,actor(challenge(_),self),[]).
metarule(pred,action(challenge(K),
    "challenge peer to prove that he has the private key
     corresponding to public key ${K}"),[]).
metarule(pred,sensitivity(challenge(_),public),[]).
```

C Monotonicity of the filtering process

In this section, a demonstration is provided in order to prove that the filtering process is monotonic in regards to the state

Let us assume the following notations

- $P \equiv$ the policy together with the associated metapolicy
- $R \equiv$ the initial request for a resource
- $\Sigma \equiv$ the state containing information about the current negotiation

The filtering process can be described as a sequence of steps

- $P_1 = relevant\_rules(P, R, \Sigma)$
- $P_2 = partial\_evaluation(P_1, \Sigma)$
- $P_3 = relevant\_rules(P_2, R, \Sigma)$
- $ImmediateActions = select\_actions(P_3, \Sigma)$
  $\Sigma' = \Sigma \cup ActionResults$
  where the ActionResults set is dependent on the ImmediateActions set.
- $P_4 = partial\_evaluation(P_3, \Sigma')$
- $P_5 = relevant\_rules(P_4, R, \Sigma')$
- $P_6 = blurring(P_5, \Sigma')$
- $P_7 = relevant\_rules(P_6, R, \Sigma')$
- $P_8 = add\_info(P_7)$
- $P_9 = rename\_predicates(P_8)$

The filtered policy $F = P_9$

It must be proved that if i is a given step in the negotiation process, identified by the P, R and Σ_i, and i+1 is a negotiation step ulterior to i, identified by P, R and Σ_{i+1}, then all the facts that can be inferred from F_i, the filtered policy P in step i, can also be inferred from F_{i+1}, the filtered policy P in step i+1. If we use the Herbrand notation then we write this as $\mathcal{H}(F_i) \subseteq \mathcal{H}(F_{i+1})$

All during the negotiation process, nothing is taken out of the state, so the state can only be monotonically increasing, that is expressed as $\Sigma_i \subseteq \Sigma_{i+1}$

To prove that $\mathcal{H}(F_i) \subseteq \mathcal{H}(F_{i+1})$ holds we must prove it for each phase of the filtering process.

- selecting relevant rules

$$P_1^i = relevant\_rules(P, R, \Sigma_i)$$

$$P_1^{i+1} = relevant\_rules(P, R, \Sigma_{i+1})$$

Since we assumed in the beginning that $\Sigma_i \subseteq \Sigma_{i+1}$, then we can write $P_1^i \subseteq P_1^{i+1}$. This means that if the sate is only monotonically increasing then the set of relevant rules selected in step i is included in the step of relevant rules selected in step i+1. We can conclude for this phase that $\mathcal{H}(P_1^i) \subseteq \mathcal{H}(P_1^{i+1})$

- partial evaluation

$$P_2^i = partial\_evaluation(P_1^i, \Sigma_i)$$

$$P_2^{i+1} = partial\_evaluation(P_1^{i+1}, \Sigma_{i+1})$$

As discussed earlier $\Sigma_i \subseteq \Sigma_{i+1}$ so $\mathcal{H}(P_2^i) \subseteq \mathcal{H}(P_2^{i+1})$ holds for this phase also.

- selecting relevant rules

$$P_3^i = relevant\_rules(P_2^i, R, \Sigma_i)$$

$$P_3^{i+1} = relevant\_rules(P_2^{i+1}, R, \Sigma_{i+1})$$

Since $\Sigma_i \subseteq \Sigma_{i+1}$ and $P_1^i \subseteq P_1^{i+1}$ then $P_2^i \subseteq P_2^{i+1}$ this concludes that $\mathcal{H}(P_3^i) \subseteq \mathcal{H}(P_3^{i+1})$

- selecting the immediate actions, adding the action results to the state and partial evaluation

Due to the state only being monotonically increasing, the immediate actions selected in step i+1 can only be a superset of the immediate actions selected in step i. From this results that $ActionResults_i \subseteq ActionResults_{i+1}$

$$\Sigma_i^{'} = \Sigma_i \cup ActionResults_i$$

$$\Sigma_{i+1}^{'} = \Sigma_{i+1} \cup ActionResults_{i+1}$$

Since we have $\Sigma_i \subseteq \Sigma_{i+1}$ and $ActionResults_i \subseteq ActionResults_{i+1}$, we get that

$$\Sigma_i^{'} \subseteq \Sigma_{i+1}^{'}$$

$$P_4^i = partial\_evaluation(P_3^i, \Sigma_i^{'})$$
$$P_4^{i+1} = partial\_evaluation(P_3^{i+1}, \Sigma_{i+1}^{'})$$

From the last three equations we get

$$\mathcal{H}(P_4^i) \subseteq \mathcal{H}(P_4^{i+1})$$

- selecting relevant rules

$$P_5^i = relevant\_rules(P_4^i, R, \Sigma_i)$$

$$P_5^{i+1} = relevant\_rules(P_4^{i+1}, R, \Sigma_{i+1})$$

This step is proved similar to the other relevant rules step.

- blurring

$$P_6^i = blurring(P_5^i, \Sigma_i)$$
$$P_6^{i+1} = blurring(P_5^{i+1}, \Sigma_{i+1})$$

We can consider

$$P_6^i = blurring(P_5^i, \Sigma_i) = B^i \cup NB^i$$

where B^i corresponds to the set of rules that have been blurred, according to the current state and NB^i represents the set of non-blurred rules. Obviously, $B^i \cap NB^i = \phi$.

Further, we can write

$$P_5^{i+1} = S \cup T$$

where

$$S = \{P | \mathcal{H}(P) = \mathcal{H}(P_5^i)\}$$

and

$$S \cap T = \phi$$

that is the set of rules that are found in P_5^{i+1} and from which we can inffer as much as we can from P_5^i.

After blurring P_5^{i+1} we have

$$P_6^{i+1} = blurring(S, \Sigma_{i+1}) \cup blurring(T, \Sigma_{i+1})$$

We can write

$$blurring(S, \Sigma_{i+1}) = B^S \cup NB^S$$

where B^S is the set of rules from S that have been blurred.

Since the state can only be monotonically increasing and blurring is only done in regards to the state, we get $B^S \subseteq B^i$. This means that the set of blurred rules in P_5^i is greater or equal to the set of blurred rules in S.

We now have

$$\mathcal{H}(S) = \mathcal{H}(P_5^i)$$
$$B^S \subseteq B^i$$

so we can conclude that

$$\mathcal{H}(P_6^i) = \mathcal{H}(blurring(P_5^i, \Sigma_i)) \subseteq \mathcal{H}(blurring(S, \Sigma_{i+1}))$$

.

Since

$$P_6^{i+1} = blurring(P_5^{i+1}, \Sigma_{i+1}) \supseteq blurring(S, \Sigma_{i+1})$$

we have finally proved that

$$\mathcal{H}(P_6^{i+1}) \supseteq \mathcal{H}(P_6^i)$$

The remaining steps of the filtering process, add_info and renaming_predicates, don't actually modify the policy in regards to what can be inferred from it, so we have now proved that

$$\mathcal{H}(F^i) = \mathcal{H}(F^{i+1})$$

holds all during the filtering process, this proves our goal that the filtering process is monotonic in regards to the state.