I2-D2

Policy Language Specification

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Naples/I2-D2/D/PU/b1
Responsible editors:	P. A. Bonatti
Reviewers:	Jose Julio Alferes and Joachim Peer
Contributing participants:	Hannover, Naples, St-Gallen, Turin, Zurich
Contributing workpackages:	I2, I1
Contractual date of deliverable:	28 February 2005
Actual submission date:	28 February 2005

Abstract

This report's main goal is specifying syntax and semantics of the core of PROTUNE, the policy language and metalanguage of REWERSE. The language can specify access control policies, privacy policies, reputation-based policies, provisional policies, and a class of business rules.

The document also specifies the architecture of a distributed policy-based system, together with a suite of policy-related services.

It introduces some policy filtering methodologies needed for negotiation semantics and query processing, and proves their properties in terms of information preservation or loss.

We illustrate the language by means of numerous examples and outline a refined use case list for verbalization (i.e., formulation in controlled natural language) in the form of a representative list of sample policies.

Keyword List

Policy language, PROTUNE architecture, policy services, trust negotiation, metapolicies, policy filtering, verbalization

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Policy Language Specification

P. A. Bonatti[1] **and D. Olmedilla**[2]

[1] Università di Napoli Federico II
Email: `bonatti@na.infn.it`

[2] L3S Research Center and Hanover University
Email: `olmedilla@l3s.de`

28 February 2005

Abstract
This report's main goal is specifying syntax and semantics of the core of PROTUNE, the policy language and metalanguage of REWERSE. The language can specify access control policies, privacy policies, reputation-based policies, provisional policies, and a class of business rules.

The document also specifies the architecture of a distributed policy-based system, together with a suite of policy-related services.

It introduces some policy filtering methodologies needed for negotiation semantics and query processing, and proves their properties in terms of information preservation or loss.

We illustrate the language by means of numerous examples and outline a refined use case list for verbalization (i.e., formulation in controlled natural language) in the form of a representative list of sample policies.

Keyword List
Policy language, PROTUNE architecture, policy services, trust negotiation, metapolicies, policy filtering, verbalization

Contents

1 Introduction **1**

2 Strategic assumptions **1**

3 Architecture and policy services **2**

- 3.1 System Architecture 2
- 3.2 Peer Architecture 4
- 3.3 Negotiation Interface 5
  - 3.3.1 Access Control Queries 5
  - 3.3.2 Why Queries 8
  - 3.3.3 How-To Queries 11
  - 3.3.4 What-If Queries 12

4 The internal rule language **12**

5 Metapolicies **16**

6 Semantics-preserving policy filtering **18**

- 6.1 Removing Irrelevant Rules 18
- 6.2 Evaluating State Predicates 19
- 6.3 Compiling Private Policies 20
- 6.4 Predicate Renaming 20
- 6.5 Annotation / Information Increasing 21

7 Filtering with information loss **21**

- 7.1 Blurring 21
- 7.2 Expectation 24

8 Driving filtering with metapolicies **25**

9 Driving how-to query answering with metapolicies **28**

10 Metapolicies for credential and action selection **29**

11 Monitoring policies with constraints **30**

12 Distributed credentials **31**

13 Libraries and language extensions **33**

14 Reputation and recommendations **34**

15 Verbalization **35**

16 Future work **38**

1 Introduction

This report's main goal is specifying syntax and semantics of the core of Protune (PROvisional TrUst Negotiation), the policy language and metalanguage of REWERSE. The language can specify access control policies, privacy policies, reputation-based policies, provisional policies, and a class of business rules. The language is declarative, still it can describe actions which modify the current state.

The report specifies also the architecture of a distributed policy-based system, together with a suite of policy-related services.

This report introduces some policy filtering methodologies needed for negotiation semantics and query processing, and proves their properties in terms of information preservation or loss.

We illustrate the language by means of numerous examples and outline a refined us case list for verbalization (i.e., formulation in controlled natural language) in the form of a representative list of sample policies.

The report is organized as follows. In the next section we outline some simplifying assumptions applying to this first version of the policy language specification. Then, in Section 3, we describe the architecture of the system and informally illustrate the suite of policy-related services. Sections 4 and 5 define the policy language and the metalanguage, respectively, including their declarative semantics. Sections 6 and 7 introduce policy filtering methodologies that are used in sections 8 and 9 to specify the semantics of negotiation and query processing. The rest of the report mainly illustrates the use of the language. Sections 10, 11 and 12 show how to control and fine-tune the negotiation process; they introduce the notion of constraint in policy languages. Section 13 briefly deals with language extensions and integration and with policy libraries and ontologies. Reputation-based trust and recommendations are briefly discussed in Section 14. Sample verbalizations and desiderata for the natural language processing (NLP) front-end are reported in Section 15.

2 Strategic assumptions

The survey on business rules and reasoning about actions carried out in I2-D1 revealed that business rules and—more generally—declarative descriptions of dynamic behaviors generally require a rich combination of different inference modalities (cf. the different categories of business rules reported in I2-D1).

At the same time, the technical issues related to Automated Trust Negotiation (ATN) still involve nontrivial technical difficulties. As usual, flexibility may conflict with efficiency, and particular care must be taken in controlling the negotiation process without restricting it enough to block desirable negotiations.

In the light of these sources of complexity, we planned since the beginning a two-stage policy language design (the second stage corresponds to the forthcoming deliverable I2-D12). In the first stage—that is, in this deliverable—we choose a tradeoff that favors flexibility issues in trust negotiation—crucial for security—, and make some simplifying assumptions on actions and business rules. These assumptions should be relaxed in the second stage, whose second purpose is incorporating results and feedbacks coming from the other working groups.

More precisely, here *we assume that all actions are orthogonal*, so that their relative order of execution does not matter. More precisely, each action makes a particular condition true, and different actions affect logically independent conditions (no interference between different actions); repeated applications of a given action do not further change the current state (unless the corresponding condition is falsified in between by an external event), that is, actions are idempotent transformations.

In practice, this restriction is compatible with a number of so-called *provisional policies*, e.g. event logging and workflow activation, as well as many common business rules, such as discounts and special service activations. Our framework is general enough to attach to action-dependent predicates (called *provisional predicates*) compound actions, such as sequences or scripts.

The above restriction is partly balanced by introducing in the policy language suitable *constraints* or *denials*, that may be used to state the mutual incompatibility of a set of actions, and are also very useful to express privacy constraints. Constraints, for example, can be used to state that discounts cannot be cumulated. Further examples can be found in the rest of the report.

Reputation-based trust was another important modelling requirement identified in I2-D1. Fortunately, it seems that reputation can be modelled with pretty standard rules, plus a time-dependent "state" (a fact base) that is needed also in security-related trust negotiation frameworks. In other words, the technical requirements for supporting reputation-based decisions are aligned with the requirements arising from ATN needs.

Finally, we decided *not* to incorporate in our language the emerging high-level constructs for rich credentials (some of them are discussed in Section 13). Such languages are still evolving and it is not currently clear which approaches will be widely accepted in the future. We prefer lower-level syntax, capable to integrate the new constructs—if so desired—by means of suitable *rule libraries*, or *ontologies* if you prefer. This issue is further discussed and illustrated in Section 13.

3 Architecture and policy services

In this section, the whole architecture of the systems (peers hereafter) involved in a negotiation is described. First, in section 3.1, the overall picture is presented with a description of the systems involved and the communication among them. Section 3.2 focuses on the different modules available within a peer and specifies how the communication among them takes place and which are the interfaces as well as messages interchanged. Finally, section 3.3 describes how peers interact with each other in order to perform negotiations and shows how different queries are allowed depending on the purpose/goal of the requester in order to increase expresiveness and user friendliness.

3.1 System Architecture

Trust negotiation in Protune is directly inspired by PAPL [Bonatti and Samarati, 2000] and PeerTrust [Gavriloaie et al., 2004], that build on ideas introduced in [Winslett et al., 1997]. In summary, each party (client and server) makes decisions based on a set of rules that entail decision atoms such as `allow`(X), based on conditions over currently available credentials and declarations (sent by the other party) and a time-dependent state, covering the negotiation state, user profiles, etc. (see [Bonatti and Samarati, 2000] for further details). On the server, X is typically a service; on the client X may denote credential release, declaration release and actions execution.

At each party, credential and declaration requests are automatically derived from the local

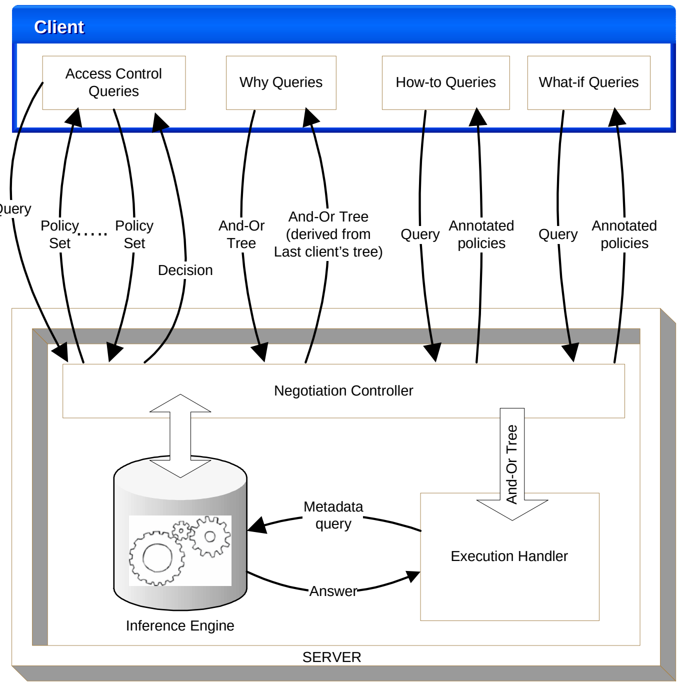


Figure 1: Framework Architecture

rules by identifying the sets of credentials and declarations that entail $\texttt{allow}(X)$. In PROTUNE, requests may also contain more general actions, basically remote service invocations.

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

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

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

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

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

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

where U and P are bound to all legal (user,password) pairs. In [Bonatti and Samarati, 2000] this is avoided by a combination of *guarded rules* and a rigid two-phase negotiation protocol. Here we adopt a more flexible protocol based on *policy blurring* (Section 7.1). As an additional level of complexity, in PROTUNE the negotiator must decide when the actions associated to provisional atoms are to be dispatched to the execution handler for execution.

Similar policy filtering applies to advanced policy querying, for explanation purposes. PROTUNE provides not only typical access control queries ("is user U authorized to access service S?") but also why-queries ("why did the request from user U to access service S not succeed?"), how-to queries ("what does user U need to be granted access to service S") and what-if queries ("would user U be authorized to access service S if he disclosed his student id"). A detailed description is given in section 3.3.

Auxiliary policy-related services comprise asking for signed statements stating properties of peers and objects (a generalization of classical electronic credentials), as well as update operations that assert and retract atoms in the time-dependent state of the policy (useful for notifying events to the policy).

An overall picture of the PROTUNE architecture and its main services is depicted in Figure 1. Although we only show the interactions between two peers (client and server) in this figure, the negotiations could involve third parties as well. Auxiliary services are not illustrated.

3.2 Peer Architecture

PROTUNE requires that the software each peer in the network is running has some capabilities in order to be able to perform a negotiation. We have identified three modules being each one responsible for a different task. The modules and their functions are:

- *Negotiation Controller.* This module interfaces with the user and is in charge of controlling and monitoring the state of current negotiations. It is responsible for:
  - Interfacing with external peers. It receives the requests and responses from other peers and once the local process has finished, it sends the generated ones.
  - Sending requests to the inference engines and receiving the answers in order to do the reasoning over the time-dependent state and policies.
  - Deciding when a set of actions are sent to the execution handler in order to be executed.
- *Inference Engine.* Required in order to add support to policies and reason over them. The inference engine contains the policies protecting the services/resources as well as their metadata and the time-dependent state.
- *Execution Handler.* This module is responsible for the execution of actions in the system (e.g. fetch a credential or log a string into a file). It receives an And-Or-Tree from the Negotiation Controller with the actions that must be executed. The reason why the tree might include disjunctions is that in some cases, it could be left up to the execution handler the task of choosing among "equivalent" actions (e.g. according to availability

or cost). For example, if a notification to a user is required, the controller could give the execution handler the possibility to do it via e-mail or fax (in case the user did not express preference for any of them). In addition, the Execution Handler might need to query directly the inference engine for some metadata before it executes any action (e.g. action costs).

3.3 Negotiation Interface

Typically, systems in charge of authorizing users reduce their communication with users to requests that must be satisfied by them and a final notification of the authorization decision: either granted or denied. However, users usually feel frustrated with this kind of operation in case their request fails. This happens because they do not receive enough information to find out which was the reason for their request to fail and therefore they do not know how to refine it in order to be granted access. In addition, in the Semantic Web, any two parties must be able to interact even if they do not know about each other. Therefore, a requester might need to ask a server about how such an interaction could be performed. In the PROTUNE system we provide users with four different types of queries according to the goal of the user:

- *Access Control.* This query represents a typical request to get access to a service or resource. For example, a user could ask whether he is allowed to download a file from a server or to book an online course.
- *Why/Why Not.* Once a negotiation took place and a decision has been communicated to the requester (either "granted" or "denied") a user might want to get an explanation (or a longer one in case an explanation is already included in the decision) of why (respectively why not) its request was granted/denied.
- *How to.* Whenever a user does not know how a service works, he will need a description of how the service works and which are the requirements he must satisfy in order to be granted access to it. For example, a user may want to ask what he must provide to a server in order to download a file.
- *What if.* In some cases a user would like to ask whether in a hypothetical situation he would get access to a service. For example, a user would like to know if in case he subscribed to an online shop he would get a discount on a book he plans to buy.

It is important to note that while an evaluation of an access control query might involve execution of actions at the server, actions should never be executed during any kind of query answering as they do not perform a real negotiation.

In the following we will describe in detail each one of this type of queries and specify how the communication between the client and the server is performed.

3.3.1 Access Control Queries

This kind of query represents the classical query "is user U authorized to access service S?". While in typical systems this is a one-shot process (client makes a request and the server sends back the decision), in Trust Negotiation this is an iterative process. In this process, if one of the conditions a requester must satisfy is not fulfilled, the server may collaboratively try to satisfy it by executing suitable actions (e.g. asking the client for more credentials). In such a process,

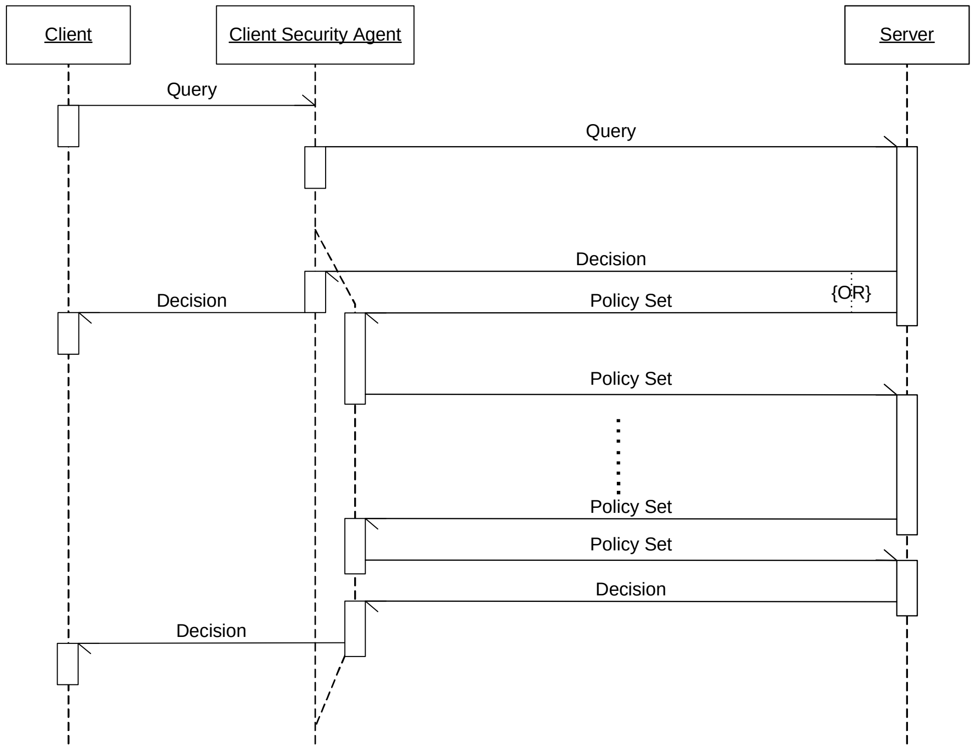


Figure 2: Sequence Diagram for an Access Control Query

the level of trust increases after each iteration. The steps performed during the negotiation are the following (also depicted in figure 2):

1. *Client's Request:* a client sends a query to the server (e.g. "allow(service(X))"). This query might already include some credentials or declarations in order to speed up the negotiation.

2. *Server Authorization Process:* the server performs the filtering of the policies (see section 8) in order to determine which conditions the requester must satisfy and whether these conditions (in case they exist) are already satisfied by the requester. Depending on the result of this process any of the following two situations may happen

   (a) If the server does not require any further interaction from the requester (e.g. it does not require any extra credential), the server sends back its final decision. This decision might be either "granted" or "denied". In this case, the process continues on step 3a.

   (b) If after the server's filtering still some conditions must be satisfied by the client, then the server will forward this information to the client in the form of a set of filtered policies (policy set). This set includes information about which alternatives (paths) the client has, what the client must provide for each one of them and whether

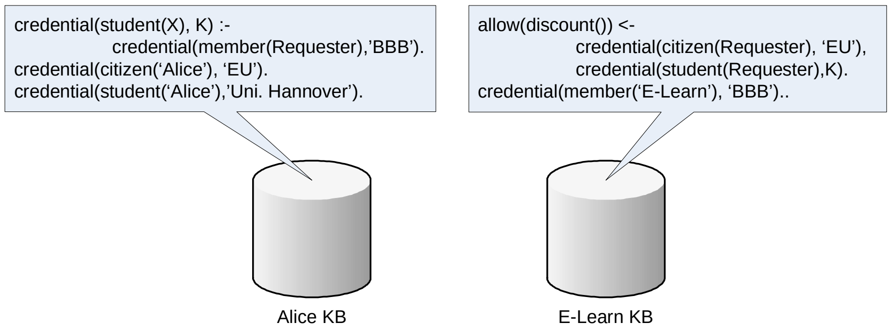

credential(student(X), K) :-
credential(member(Requester),'BBB').
credential(citizen('Alice'), 'EU').
credential(student('Alice'),'Uni. Hannover').
allow(discount()) <-
credential(citizen(Requester), 'EU'),
credential(student(Requester),K).
credential(member('E-Learn'), 'BBB')..
Alice KB
E-Learn KB

Figure 3: Example Knowledge Bases

providing that information granting to the service is guaranteed[1]. After this step the process continues on 3b.

3. *Client's Response:* according to the new message received from the server, two possibilities exist

   (a) If the client has received the final decision, then the negotiation is finished. In case access was granted, the client can acess the service. In case access was denied, the client might want to ask the server why it was denied (see section 3.3.2 below for more information on this kind of queries).

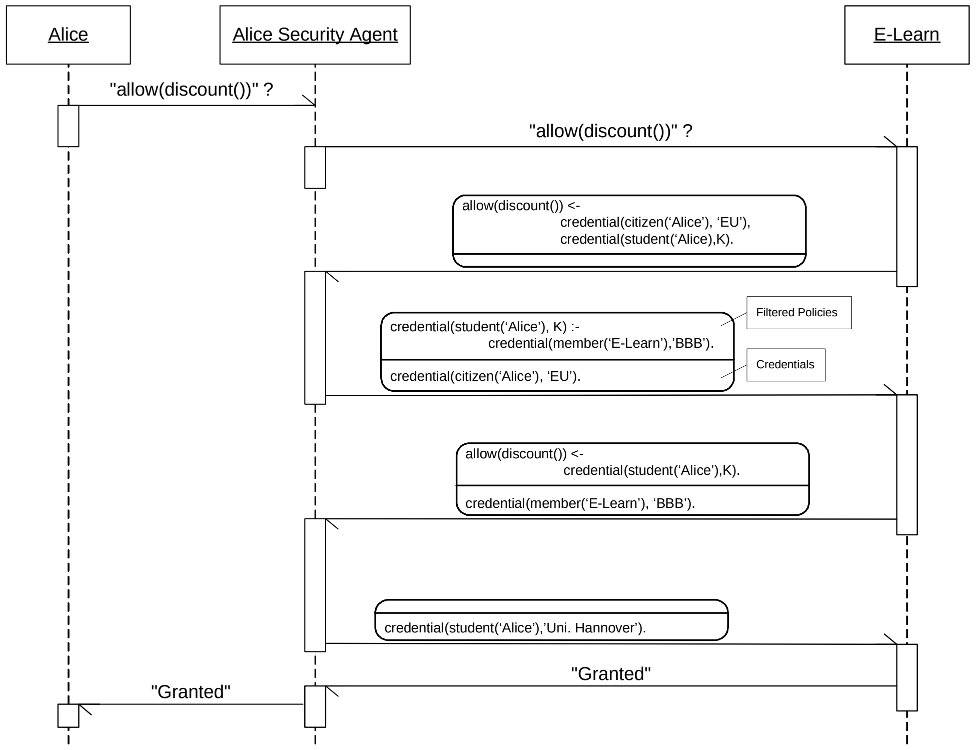
   (b) If the client received a request from the server, that means that the negotiation is still in progress. In this case, the client must choose which path he is willing to follow (or which simultaneous paths in case he wants to speed up the process) and send back a message with the information required by the server (proof). Once the server receives the message, the process continue on step 2.

As trust negotiation is a bilateral process, it may happen that in step 3b, before the client discloses e.g. a credential, it requires the server to satisfy a policy. Therefore, the message sent to the server not only contains the proof but also a policy set with the conditions the server must fulfil before it gets the information requested.

Figure 4 illustrates an example where this situation occurs given the knowledge bases depicted in figure 3. In such a scenario, Alice requests a discount from the online provider E-Learn. E-Learn notifies that in order to get a discount, Alice must prove that she is a European citizen and a student. Alice does not mind to disclose her european citizenship card to anybody but she is willing to release here student id only to companies member of the Better Business Bureau. Fortunately, E-Learn is member and it does not protect its membership credential so it can be disclosed to anybody. This way, the negotiation succeeds and a discount is granted to Alice.

Having explained the process of a negotiation, we will now describe the three types of messages exchanged between the parties in detail:

[1]Even in the case a client releases all the credentials requested by a server, still he can be denied access (e.g. the client disclosed a valid student id but the university that issued it has not an agreement with the server). Therefore, it is needed to inform the client that the disclosure of the credentials is a necessary condition and whether it is also sufficent or not (see section 7 for more information on this)



Alice
Alice Security Agent
E-Learn
"allow(discount())" ?
"allow(discount())" ?
allow(discount()) <- credential(citizen('Alice'), 'EU'), credential(student('Alice),K).
credential(student('Alice'), K) :- credential(member('E-Learn'),'BBB').
credential(citizen('Alice'), 'EU').
Filtered Policies
Credentials
allow(discount()) <- credential(student('Alice'),K).
credential(member('E-Learn'), 'BBB').
credential(student('Alice'),'Uni. Hannover').
"Granted"
"Granted"

Figure 4: Sequence of the Negotiation between Alice and E-Learn

- *Initial Query.* This is the message that inititates a negotiation. This message contains the request of the form "allow(X)". The requester can attach credentials and/or declarations to the query in order to speed up the negotiation.
- *Policy Set.* A policy set message contain a set of filtered policies (possibly empty) together with a set of credentials and declarations (possibly empty)[2]. The former gives information about which information this party must satisfy. The latter provides credentials and declarations either requested previously or given in order to speed up the negotiation.
- *Decision.* This message is the message that indicates the end of a negotiation. A decision can be "granted" or "denied". It might also provide the proof tree of the negotiation.

3.3.2 Why Queries

Whenever a client submits a request to a server and this request is denied, the user needs to receive more information than just "request denied" in order to understand why his request did

[2]This information could have been represented as an And-Or-Tree in order to provide a more intuitive representation to the other party. However, this would raise some difficulties like e.g. in case there exists recursion in the policies. The advantages of a set of policies over a tree are shorter representation (they appear only once even if they are in several branches) and non-exponential grow in case of recursive policies. In addition, the associated tree can easily be constructed from the set of policies.

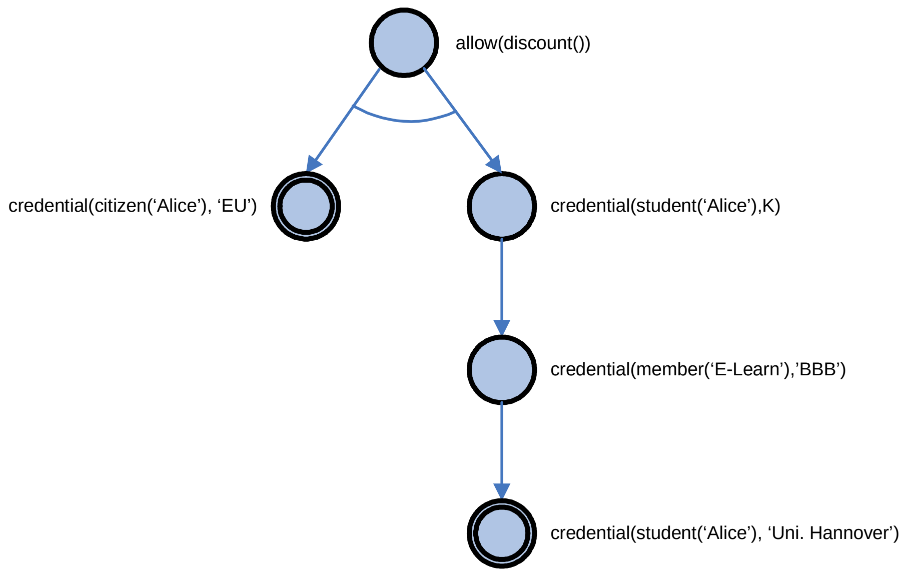

allow(discount())
credential(citizen('Alice'), 'EU')
credential(student('Alice'),K)
credential(member('E-Learn'),'BBB')
credential(student('Alice'), 'Uni. Hannover')

Figure 5: Input And-Or-Tree for a Why Query

not succeed. This way, the client may try to refine his request in order to be granted access. Common questions like "why did I not get access to the service if I provided all the information the server requested" or "My request failed when the server asked me for my student id but why does it need it in order to let me book a course?" require answers understandable not only for computers but also for humans.

As this kind of queries requests explanations of a previous negotiation, it is therefore a requirement that such a negotiation has been performed.

For example, let us assume that a negotiation like the one described in figure 4 has taken place but the final message from E-Learn to Alice is not "granted" but "denied" because E-Learn included an extra constraint in its policy where the student ID must be from a German university (in contrary to the example in the previous section where no constraint was applied to the ID). It seems reasonable that Alice would like to know why the negotiation happened to fail. In such a case, Alice would send a request to E-Learn with the And-Or-Tree depicted in figure 5 and receive the annotated And-Or-Tree (annotated tree for brevity) represented in figure 6. The annotated tree contains information in both computer and human readable form to support automatic handling and/or provide the user with nice explanations.

In scenarios with a failed provisional condition, the annotations could include a hint on how to fulfill the failed condition. Such hints could be e.g. active links pointing to some web page or service in the case of services that requires a registration procedure (possibly manual). Another example would be a payment procedure to be done on a different host (equipped for such operations); upon completion, the payment-host returns a credential (the receipt) that can be exhibited at the first host to get the desired service.

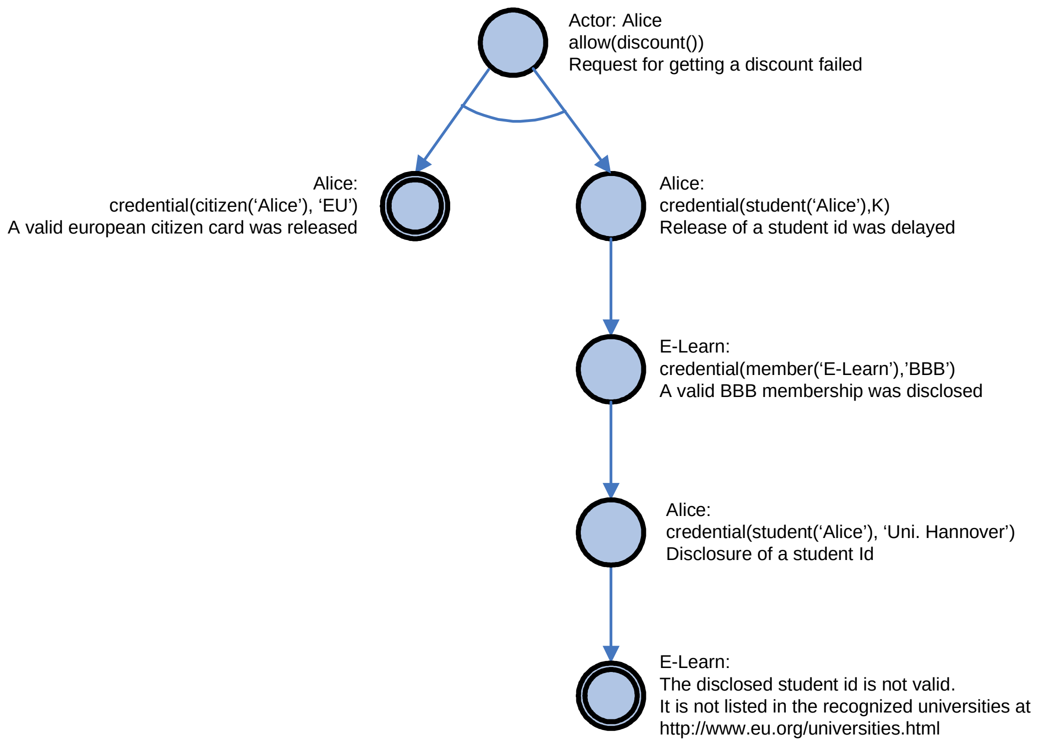

Actor: Alice
allow(discount())
Request for getting a discount failed
Alice:
credential(citizen(‘Alice’), ‘EU’)
A valid european citizen card was released
Alice:
credential(student(‘Alice’),K)
Release of a student id was delayed
E-Learn:
credential(member(‘E-Learn’),’BBB’)
A valid BBB membership was disclosed
Alice:
credential(student(‘Alice’), ‘Uni. Hannover’)
Disclosure of a student Id
E-Learn:
The disclosed student id is not valid.
It is not listed in the recognized universities at
http://www.eu.org/universities.html

Figure 6: Annotated And-Or-Tree generated as a response to a Why Query

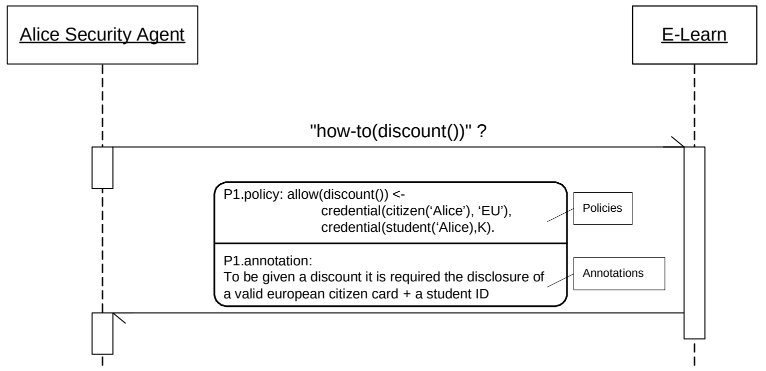

Alice Security Agent
E-Learn
"how-to(discount())" ?
P1.policy: allow(discount()) <-
credential(citizen('Alice'), 'EU'),
credential(student('Alice),K).
P1.annotation:
To be given a discount it is required the disclosure of
a valid european citizen card + a student ID
Policies
Annotations

Figure 7: Example Sequence Diagram of a How-To Query

3.3.3 How-To Queries

How-To queries are designed to help users who do not know how to interact with a service or do not know which requirements they have to fullfil in order to be granted access to it. Therefore, this kind of queries provides a computer and human understandable description of the service. This could also be automatically used by a software in case it has to select among different available services.

As in this case there is no past negotiation to refer to, the server must evaluate the query against its portfolio and return as a result a set of policies annotated in a similar way as the why queries presented in section 3.3.2.

Following with the example depicted in previous sections, suppose before any negotiation starts, Alice sends E-Learn a query "how-to (discount())". In such a case, E-Learn will generate a set of annotated policies with information of how Alice could get such a discount. The sequence diagram between both of them is depicted in figure 7.

It is important to mention that the main difference from the previous types of queries is the absence of credentials. Therefore the purpose of this type of queries is merely informative. This information is particularly useful in the case of software agents that must find services fulfilling a goal and choose among the available ones according to whether it is possible it will get access to it (service discovery).

An important property of how-to queries is the treatment of actions. During negotiations the actions associated to provisional predicates are executed, and the explanations associated to why-queries can report the result of these actions. On the contrary, no actions are executed during the evaluation of answering queries. In How-to queries, the server may use a notion of "expected outcome" to treat these predicates as (presumably) true or (presumably) false. The answer given to a client may be taken as guidance; if all conditions reported in the answer are satisfied, then the request is likely to be authorized but there is no 100% guarantee. In general, the explanations attached to provisional predicates give users information that would not emerge from the messages exchanged along negotiations.

3.3.4 What-If Queries

In some of the previous examples we dealt with situations that make use of the portfolio of the client in order to find out whether he is able to access a resource or not, that is the server requests credentials and the client either provides them or not. Commonly, a client may not yet have a valid credential, but would like to know whether the negotiation would succeed in case he had got the credential (e.g. he might be willing to obtain it if it helps him). In such a case, a client might want to know which would be the final result of the negotiation in case he had that credential as well as whether that would be sufficent for him to get access to the resource.

For example, suppose Bob wants to download a research paper from the ACM digital library website. As he is not an ACM member he would not be authorized access to the resource. Therefore, Bob decides to send the following what-if query to the server: "would I be authorized to access the paper 'Trust Negotiation with Metapolicies' if I were ACM member?". The server will in this case evaluate Bob's request "as if" the credentials were availabe and returns a similar answer to a how-to query. This way Bob could evaluate the returned policies against his own portfolio to see if the negotiation has a chance to succeed.

This type of query is really useful in case the client does not yet have the required credentials but is willing or hesitating to obtain them. However, some sensitive policies in the server might require a real disclosure of a credential before the evaluation goes on. Therefore, at the server, release policies should be evaluated according to the actual information available (i.e., the pseudo-credentials in the argument of the what-if query should not be used to release rules/facts). A second difference with an actual negotiation is that in what-if queries the server is not actually applying any action to fulfill provisional predicates. Like in how-to query processing, the server may use a notion of "expected outcome" to handle these predicates (see the previous section).

4 The internal rule language

The rule language is based on normal logic program rules

$$A \leftarrow L_1, \dots, L_n \tag{2}$$

where A is a standard logical atom (called the *head* of the rule) and $L_1, \dots, L_n$ (the *body* of the rule) are literals, that is, L_i equals either B_i or $\neg B_i$, for some logical atom B_i.

A *policy* is a set of rules shaped like (2), such that negation is applied neither to *provisional predicates* (defined below), nor to any predicate occurring in a rule head. This restriction ensures that policies are *monotonic* in the sense of [Seamons et al., 2002], that is, as more credentials are released and more actions executed, the set of permissions does not decrease. Moreover, the restriction on negation makes policies *stratified programs*; therefore negation as failure has a clear, PTIME computable semantics that can be equivalently formulated as the perfect model semantics, the well-founded semantics or the stable model semantics [Baral, 2003].

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

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

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

Predicate `sign` is used to issue statements signed by the principal owning the policy. The argument of `sign` can be any term, possibly consisting of attribute-value pairs. This feature is useful to issue new credentials stating domain-dependent properties. In response to a statement request r, a (partial) proof of `sign`(r) is searched for.

- *Abbreviation/abstraction predicates*: These are predicates defined in the policy. They have many purposes ranging from the definition of high-level client properties (e.g. by combining low-level data and/or different credentials, cf. [Bonatti and Samarati, 2000]) to the specification of new credential semantics (see Section 13).
- *Constraint predicates* comprise the usual equality and disequality predicates.
- *State predicates*: Policy decisions have to be taken with respect to a time-dependent system *state*, encoding the current negotiation state, legacy data, user profiles, and so on. State predicates are further partitioned into the following subclasses.
  - *State query predicates*: These predicates read the current state without modifying it. They comprise both built-in and application dependent predicates. Built-in state predicates model the state of the negotiation, and provide a uniform interface to external packages in the style of HERMES [Subrahmanian et al., 1995]. An example of negotiation state atom is `request`(n, R); it holds if R is the n-th request in the negotiation. External packages (including databases and other data sources) can be queried with atoms of the form:

    $$\texttt{in}(X, \mathit{package\_name} : \mathit{function}(\mathit{arg\_list})) \tag{3}$$

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

    $$\texttt{access} : \texttt{query}('\texttt{select}\ T\ \texttt{where}\ A = c'),$$

    then conceptually speaking the local state contains all the ground instances of (3) such that X is bound to a tuple of table T with attribute $A = c$. In practice, the implementation needs suitable wrappers for the packages and appropriate solution caching techniques.
  - *Provisional predicates*: These are predicates that may be made true by means of appropriate *actions* that may modify the current state. Such actions may be carried out by the server, by the client, or both.

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

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

By means of another kind of built-in provisional atoms, $\mathtt{authenticates\_to}(K)$, the peer can be asked to prove to be the owner of the private key associated to K, through a standard challenge procedure.

Sometimes, the actions associated to provisional predicates are to be executed locally, by the negotiator. A common example is $\mathtt{logged}(X, \mathit{logfile\_name})$ that may be made true by recording X into *logfile_name*. The following sample rule R records in `ac.log` that access to service `Srv` has been granted by R itself:

$$\begin{aligned}&\mathtt{allow}(\mathit{Srv}) \leftarrow \\ &\quad \ldots, \mathtt{logged}(\mathit{Srv} + {}'\mathtt{granted\ by}\ R',\ \mathtt{ac.log})\,.\end{aligned}$$

In order to prove $\mathtt{allow}(\mathit{Srv})$ from the client's credentials, the system can write the logfile, thereby triggering the rule.

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

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

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

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

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

The following sections will show how to specify the execution time of provisional atoms. In particular, Section 5 illustrates metalevel directives to the negotiator and Section 8 illustrates how such directives are applied by the negotiator (i.e. their semantics).

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

The rule language supports object-oriented dot syntax that, however, is only an abbreviation for standard first-order syntax. One can express by $X.\mathtt{attr} : v$ the fact that X has an attribute `attr` with value v. Actually, $X.\mathtt{attr} : v$ abbreviates the standard atom $\mathtt{attr}(X, v)$. This representation allows multi-valued attributes. This attribute semantics is compatible with semantic web standards such as RDF and OWL (in particular $X.\mathtt{attr} : v$ corresponds to an RDF triple).

More generally, $X.\mathtt{a}_1.\cdots.\mathtt{a}_n : v$ abbreviates

$$\mathtt{a}_1(X, V_1), \mathtt{a}_2(V_1, V_2), \ldots, \mathtt{a}_\mathtt{n}(V_{n-1}, v) \tag{4}$$

where $V_1, \ldots, V_{n-1}$ are fresh variables (not used elsewhere) that are meant to be existentially quantified. In practice, this means that when $X.\mathtt{a}_1.\cdots.\mathtt{a}_n : v$ occurs in a rule body it abbreviates exactly (4), while asserting $X.\mathtt{a}_1.\cdots.\mathtt{a}_n : v$ as a fact causes the atoms in (4) to be asserted as individual facts after replacing $V_1, \ldots, V_{n-1}$ with $n-1$ Skolem constants.

Finally, an atom $A = p(\ldots, X.\mathtt{path}{:}v, \ldots)$ is expanded to two atoms $p(\ldots, X, \ldots)$, $X.\mathtt{path}{:}v$. If A occurs in the body of a rule R, then it suffices to expand A in the body of R. If A is the head of R, then the first atom in the expansion is the actual head and the second atom is to be inserted in the body:

$$p(X.\mathtt{path} : v) \leftarrow \mathit{Body} \text{ abbreviates } p(X) \leftarrow X.\mathtt{path} : v, \mathit{Body}\,.$$

Declarative Rule Semantics

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

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

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

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

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

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

$$\mathsf{cmodel}(\mathit{Pol}, \Sigma) \quad = \quad \{A \in \mathcal{H} \mid \mathit{Pol}^{\Sigma} \models A\}\,. \tag{5}$$

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

Table 1: The core metaattributes

Attribute	Domain	Range
`action`	provisional predicates	commands
`actor`	provisional predicates	`self`, `peer`
`aggregation_method`	cost and sensitivity attributes	`max`, `min`, `sum`, `adopt`(*Predicate*)
`cost`	provisional predicates	number
`evaluation`	state predicates	`immediate`, `delayed`, `concurrent`
`expected_outcome`	provisional predicates	`success`, `failure`, `undefined`, `unknown`
`explanation`	literals and rules	string expression
`ontology`	abbreviation predicates, credentials, declarations, actions	URI
`predicate`	literals	predicate names
`selection_method`	`negotiator`	`certain_first`, `order`(*attribute_list*), `adopt`(*Predicate*))
`sensitivity`	predicates, literals, rules	`public`, `private`, `not_applicable`
`type`	predicates, literals	`abbreviation`, `constraint`, `decision`, `state_predicate`, `provisional`, `state_query`

5 Metapolicies

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

- The syntactic material of the object-level rule language (i.e. predicate names, constant names, variable names, rule names etc.) may occur as terms in the metapolicy. In the following, for all rules R we shall denote by $\hat{R}$ the name of R.
- The built-in predicates comprise Prolog-style metapredicates for inspecting terms, checking groundness, etc. Moreover, a predicate `holds`(G) allows to call an object-level goal G against the current state, using the object-level policy. These predicates are illustrated below.
- A set of reserved attributes associated to predicates, literals and rules is used to drive the negotiator's decisions (see table 1).

Here are a few examples. If p is a predicate, then p.`sensitivity` : `private` means that the extension of the predicate is private and should not be disclosed. An assertion

$$p.\texttt{type} : \texttt{provisional}$$

declares p to be a provisional predicate; then p can be attached to the corresponding action α by asserting p.`action` :α. If the action is to be executed locally, then assert p.`actor` : `self`, otherwise assert p.`actor` : `peer`.

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

$$L.\mathit{attr}\colon \mathit{Val} \leftarrow \texttt{literal}(L),\ L.\texttt{predicate}.\mathit{attr}\colon \mathit{Val}$$

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

Metarules allow fine-grained tuning of state predicate evaluation. For example, a strategy that selects negative (state) literals for immediate evaluation only if they are ground can be expressed simply with:

$$(\neg A).\texttt{evaluation} : \texttt{immediate} \leftarrow \texttt{ground}(A)\,. \tag{6}$$

The rule below enables immediate evaluation of a credential only if it is available and ground:

$$\begin{aligned}&\texttt{credential}(C,K).\texttt{evaluation} : \texttt{immediate}\\ &\qquad \leftarrow \texttt{ground}(C),\ \texttt{holds}(\texttt{credential}(C,K))\,.\end{aligned} \tag{7}$$

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

$$\begin{aligned}&\texttt{table}(Key, Data).\texttt{evaluation} : \texttt{immediate}\\ &\qquad\qquad\qquad \leftarrow \texttt{ground}(Key)\,.\end{aligned} \tag{8}$$

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

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

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

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

The next example shows how to check credential revocation and notify violations to the administrator:

$$A \leftarrow \ldots, \texttt{credential}(C), \texttt{peer}(P), \texttt{check\&notify}(C,P)\,.$$

```
check&notify(X, Y).action :
    'if(revoked((X))
        sendmail(admin, X + ‘from‘ + Y);
        exit(FAIL);
    else exit(SUCCESS);
    endif'
```

The `exit`(`FAIL`) command prevents the engine from asserting the fact `check&notify`(C, P). This blocks the above rule for A.

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

Finally, the `explanation` attribute can be used to attach human-readable explanations to literals and rules. For example, given a rule like

$$\begin{aligned}\texttt{accepted\_student\_credential}(S,U) \quad \leftarrow \quad & \texttt{credential}(\texttt{student}(S),U), \\ & \texttt{recognized\_university}(U)\end{aligned}$$

one may formulate an explanation as follows:

$$\begin{aligned}&\texttt{accepted\_student\_credential}(S,U).\texttt{explanation}: \\ &\qquad\qquad\qquad\qquad\text{“Is } S \text{ a student of a recognized university } U\text{?”}\end{aligned}$$

The explanation strings are collected to produce the annotated proof trees returned as answers to advanced queries, as described in Section 3. In the presence of an `explanation` attribute, the query answering interface may give the user a direct explanation of what that particular literal is about. In the absence of an explicit explanation—or upon a request for more details—the literal can be expanded with its definition to produce more specific explanations for the success or failure of the literal.

Currently explanations are strings. However in the next version of the policy language we are planning to have more structured objects, to support more flexible forms of natural language generation. In perspective, rules will be produced by translating natural language sentences (see Section 15); then some of the explanations will be derived automatically from the natural language formulation.

6 Semantics-preserving policy filtering

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

6.1 Removing Irrelevant Rules

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

1. If the head of a rule $R \in Pol$ unifies with A, then $R \in S$;

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

The relevant subset of *Pol* w.r.t A will be denoted by

$$\mathsf{relevant}(Pol, A)\,.$$

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

Lemma 1 *For all ground atoms $A\theta$ (θ is a substitution),*

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

6.2 Evaluating State Predicates

Next we define partial evaluation. Let E be a set of (possibly nonground) state literals. Intuitively, E specifies which literals can be evaluated.

For all rules R, let $R \overset{\Sigma,E}{\longrightarrow}_1 S$ iff

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

The evaluation step relation is extended to policies in the natural way:

For all policies Pol, define $Pol \overset{\Sigma,E}{\longrightarrow}_1 Pol'$ iff

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

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

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

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

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

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

A *trace for Pol w.r.t. Σ and E* is a (possibly infinite) sequence of policies

$$Pol_1 \overset{\Sigma,E}{\longrightarrow} Pol_2 \overset{\Sigma,E}{\longrightarrow} \cdots \overset{\Sigma,E}{\longrightarrow} Pol_i \overset{\Sigma,E}{\longrightarrow} \cdots.$$

A trace is *complete* if it is infinite or for the last element Pol_n in the sequence, there exists no policy Pol' such that $Pol_n \overset{\Sigma,E}{\longrightarrow} Pol'$.

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

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

The unique result of partial evaluation (i.e., the last element of each complete trace) will be denoted by

$$\mathsf{partEval}(Pol, \Sigma, E)\,.$$

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

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

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

6.3 Compiling Private Policies

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

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

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

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

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

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

6.4 Predicate Renaming

In policy engineering, a good principle is using suggestive and meaningful predicate names. However, when rules are disclosed during negotiation, meaningful predicate names may disclose confidential information about the policy.

A simple solution, already adopted in PAPL [Bonatti and Samarati, 2000], consists in uniformly renaming predicates with mechanically generated symbols carrying no particular meaning.

Predicate renaming applies only to abbreviation predicates because:

- standard predicates such as decision predicates, credentials, declarations etc. must be understood by the other peer and cannot be distorted;

- state predicates are either eliminated by the filtering process, or protected with a special technique called *blurring* described in the next section, in order to let the client reconstruct all public information.

Clearly, predicate renaming preserves the semantics of the non-renamed predicates.

6.5 Annotation / Information Increasing

This operation defines how annotations are included into a policy in order to provide some human understandable explanations.

Let U be the set of all literals L such that

- $L.\texttt{sensitivity} : \texttt{S} \mid \texttt{S} \neq \texttt{private}$
- $L.\texttt{explanation} : \texttt{Val}$

where Val is a string with a human understandable description of the literal[3].

Intuitively, let us add the explanations attached to any of the literals in the rule set. Formally, for all rules R of the form $(r : A \leftarrow B)$ where r is the name of the rule, A is the head and $B = (L_1, \dots, L_{i-1}, L_i, L_{i+1}, \dots, L_n)$ is the body. Let $\mathsf{annot}(R, U) = R \cup S$ where

- $S^A = \{r[0].explanation : Val | A.explanation : Val, A \in U\}$
- $S^B = \{r[i].explanation : Val | B_i.explanation : Val, B_i \in U\}$
- $S = S^A \cup S^B$

Then for all policies Pol, define

$$\mathsf{annot}(Pol, U) = \bigcup_{R \in Pol} \mathsf{annot}(R, U)\,.$$

7 Filtering with information loss

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

7.1 Blurring

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

$$\hat{R}.\texttt{sensitivity} : \texttt{not\_applicable} \leftarrow \dots\,.$$

(where $\hat{R}$ is R's name). As more credentials arrive, R may become visible and extend negotiation opportunities. In this framework, policy disclosure has a reactive flavour, as opposed to the predefined graph structure adopted in [Yu et al., 2001].

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

However, they cannot simply be left in the policy and sent to the client[4] because

- the client does not know how to evaluate them, since it has no access to the server's state, and

[3]We are currently working on verbalization (see section 15) in order to provide more flexible forms of natural language generation

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

- the syntax of protected conditions may suffice to disclose some confidential information about the structure of the policy.

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

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

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

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

where r is the name of rule (1). From this rule, a machine may realize that sending the declaration does not suffice to enter the site; first the server is performing a check of some sort.[5] Blurring is formalized below.

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

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

Then for all policies Pol, define

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

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

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

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

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

[5] Normally declarations result in a pop-up window where the user can directly type in the requested information or click an *accept* button. If the declaration is to be handled automatically, the client's policy should encode enough information to relate the appropriate user-password pair to the current service request. Moreover, appropriate policy rules are needed to decide whether the user should be queried or the declaration should be handled automatically.

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

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

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

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

- p.`type` is `state_predicate`,
- p.`actor` is not `peer`

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

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

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

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

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

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

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

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

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

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

7.2 Expectation

Contrary to how it works in an actual negotiation, how-to and what-if queries require the server to evaluate a request without executing inmediate actions during such an evaluation. Releasing this information to the client is useless as he does not know how it should be evaluated nor which would be the result. Just removing them is also not a valid solution since in some cases these actions could be likely to fail and therefore the client would not be aware of some conditions that might make the negotiation fail.

A solution is to make use of metadata of the form

$$\hat{R}.\texttt{expected\_outcome} : \texttt{Val} \mid Val \in \{success, failure, undefined, unknown\}$$

where the expected outcome can take the values *success* (it is expected to succeed), $failure$ (it is expected to fail), $undefined$ (it can both succeed or fail and it is not possible to say one of them is more likely than the other) or *unknown* (it is not explicitely specified). Therefore, inmediate actions are not executed but substituted with its expected outcome unless it is "failure" in which case the rule is removed. We call this process *expectation* and it is formalized below.

Let X be the set of all literals L which contain the following metadata

- $L.\texttt{type} : \texttt{provisional}$,
- $L.\texttt{actor} : \texttt{self}$,
- $L.\texttt{evaluation} : \texttt{immediate}$.

and let

$$\texttt{X}^{fail} = \{\texttt{R} \in \texttt{X} \mid \hat{\texttt{R}}.\texttt{expected\_outcome} : \texttt{failure}\}$$

Then, for all rules R, define $R \xrightarrow{X}_1 S$ iff

- $R = (A \leftarrow L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_n)$
- $L_i \in X$
- $\hat{L}_i.\texttt{expected\_outcome} : \texttt{Val} \mid Val \in \{success, failure, undefined, unknown\}$
- $S = \begin{cases} \emptyset & \text{if } \texttt{L}_\texttt{i} \in \texttt{X}^{\texttt{fail}}, \\ \{\texttt{A} \leftarrow \texttt{L}_1, \ldots, \texttt{L}_{\texttt{i}-1}, \texttt{Val}, \texttt{L}_{\texttt{i}+1}, \ldots, \texttt{L}_\texttt{n}\} & \text{if } \texttt{L}_\texttt{i} \notin \texttt{X}^{\texttt{fail}}. \end{cases}$

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

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

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

The unique result of substitution by expected outcome will be denoted by

$$\mathsf{subst}(Pol, X)\,.$$

8 Driving filtering with metapolicies

On each party, the policy filtering process is determined by several parameters:

- a request *Req* from the client, requiring a decision about access control, or portfolio information release,
- an access control or portfolio release policy *Pol*,
- a metapolicy *Mpol*,
- the current state Σ.

With the exception of *Req*, all the parameters are local to the peer which is to make the decision. The metapolicy is evaluated against the current state, yielding the *current canonical metamodel MM*:

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

which is inspected to read the metaproperties of rules and predicates. Policy filtering is carried out in several phases, based on the theoretical transformations introduced in Section 6 and Section 7:

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

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

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

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

   Denote the result of the first phase with P_1.

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

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

   The result of this phase is then

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

3. The selected public rules are partially evaluated. The result of this phase is

   $$P_3 = \mathsf{partEval}(P_2, \Sigma, E)$$

   where E (the set of literals to be evaluated) consists of all the literals L such that all the following conditions hold:

- L.`type` : `state_predicate` $\in MM$,
- L.`type` : `provisional` $\notin MM$,
- L.`sensitivity` : `private` $\notin MM$,
- L.`evaluation` : `immediate` $\in MM$.

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

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

4. The immediate actions occurring in P_3 are executed. More precisely, let E' be the set of all literals A such that:

   - A.`type` : `provisional` $\in MM$,
   - A.`actor` : `self` $\in MM$,
   - A.`evaluation` : `immediate` $\in MM$.

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

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

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

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

   (E' is defined in the previous step.)

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

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

   B is determined as specified in Section 7.1 as a function of E and LSL. Here LSL is the set of all literals L such that

   - L.`type` : `state_predicate` $\in MM$,
   - L.`actor` : `peer` $\notin MM$.

7. Some policies in P_6 might not be relevant anymore due to the blurring process in which some literals are now blurred literals. Therefore, the remining rules are

$$P_7 = \mathsf{relevant}(P_6, \mathtt{allow}(Req))$$

8. Provisional state predicates that may be satisfied by the other peer are replaced with the corresponding action. More precisely, for each literal L occurring in P_7 such that

   - L.`type` : `provisional` $\in MM$,

   - $L.\texttt{actor} : \texttt{peer} \in MM$,
   - $L.\texttt{action} : \alpha \in MM$,

   replace L with $\texttt{do}(\alpha)$. Let P_8 denote the result of this transformation.

9. Finally, all abbreviation predicates are anonymized by renaming them. Denote by P_9 the result of this last phase.

The final policy P_9 can be sent to the peer. The important properties of P_9 are:

- It contains only standard predicates (such as `credential`, `declaration`, `do`, constraint predicates, etc.), (renamed) abbreviation predicates and `blurred`. With the exception of `blurred` (whose semantics is deliberately obfuscated), the client knows how to handle all these predicates. The only non standard predicates are the abbreviation predicates that, however, come with their (filtered) definition.

- Its rules do not contain any instance of a private rule, nor any values computed from a private predicate. Delayed predicates are not evaluated, either.

- Evaluating $\texttt{allow}(\mathit{Req})$ in P_5 is equivalent to evaluating it in the currently applicable subset of the "true" policy *Pol*, by the theorems in Section 6.

  Phase 8 preserves the meaning of the policy, too, to the extent that the successful execution of the actions α makes the corresponding literals L true. Morever, phase 9 preserves the derivability of $\texttt{allow}(\mathit{Req})$.

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

  As a consequence, the final policy P_9 carries all the access control information that depends neither on non-applicable rules nor on private or delayed predicates.

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

$$\texttt{allow}(\mathit{Req}) \in \mathsf{cmodel}(P_5, \Sigma_{new}) \,, \tag{9}$$

then the request *Req* is permitted (be it a request for services, credentials, or actions).[6]

Remark 2 *Here the assumption of policy monotonicity w.r.t. the provisional predicates whose actor is the client turns out to be important. The reason is that between the release of* P_9 *and the corresponding answer there may be other interactions. This happens because in general there are multiple open requests* $\texttt{allow}(\mathit{Req})$ *in the current state, and the two parties are free to deal with any of them in any order. Due to interleaved request handling,* Σ_{new} *might be a strict superset of* $\Sigma' \cup \Delta$*, where* Σ' *is the state produced in phase 4 and* Δ *is the set of provisional atoms made true by the client to derive* $\texttt{allow}(\mathit{Req})$*. Policy monotonicity guarantees that any condition derivable in* $\Sigma' \cup \Delta$ *is derivable also in the extended state* Σ_{new}*.*

[6] Note that if a service does not exist, then the filtered policy contains no rules, and hence no proof of $\texttt{allow}(\mathit{Req})$ can be found. This tells the client that it cannot possibly obtain the requested service. When access is open (no credentials or declarations needed) the answer should be equivalent to the rule $\texttt{allow}(\mathit{Req}) \leftarrow \texttt{true}$.

9 Driving how-to query answering with metapolicies

For this kind of query, on each party, the policy filtering process is determined by the following parameters:

- a request *Req* from the client, demanding a set of annotated policies,
- an access control or portfolio release policy *Pol*,
- a metapolicy *Mpol*,
- the current state Σ.

In the same way as in the access control query, all the parameters are local to the peer which is to generate the response and the *current canonical metamodel MM* is used.

Policy filtering is carried out in the following phases:

1-3 Filtering of relevant policies, compilation of applicable non-public rules and partial evaluation are performed as in section 8. The result of these two phases is P_3.

4. In query answering we add annotations (if existing) to all non-sensitive literals in the policies. Therefore let U be the set of all literals L (as explained in section 6.5) such that

   - $L.\mathtt{sensitivity} : \mathtt{private} \notin MM$
   - $L.\mathtt{explanation} : \mathtt{Val} \in MM$

   The result is
   $$P_4 = \mathit{annot}(P_3, X).$$

5. All the immediate actions occurring in P_4 are not executed but substituted with the value of its expected outcome or removed in case their expected outcome is "failure":
   $$P_5 = \mathit{subst}(P_4, X).$$
   where X, as explained in section 7.2, is the set of literals such that

   - $L.\mathtt{type} : \mathtt{provisional} \in MM$,
   - $L.\mathtt{actor} : \mathtt{self} \in MM$,
   - $L.\mathtt{evaluation} : \mathtt{immediate} \in MM$.

6. Blurring of deferred conditions and are performed like in section 8. Note that explanations apply only to literals L in R such that $L.\mathit{sensitivity} \neq \mathit{private}$. No literal with an associated explanation is blurred. Denote by P_6 the final result of these phases.

7-9 Finally, relevant policies filtering, action replacement and anonymization of abbreviation predicates are performed as in section 8. Denote by P_9 the result of this last phase.

The final policy P_9 can be sent to the peer and provides an explanation of how access to the service can be made. P_9 contain the same properties as specified in section 8 plus:

- No actions have been executed as this is an informative query answering and not a real negotiation.

- The client knows how to handle all predicates in the policy with the exception of `blurred` (whose semantics is deliberately obfuscated) and replacements with expected outcomes (which the client can still handle as expected results of the operations they are substituting).

- The policy contains explanations for literals in order to provide human readable statements of the processes behind the policy. Note that explanations also let users understand conditions (corresponding to non-sensitive delayed state predicates) that would be hidden (blurred) during negotiations.

10 Metapolicies for credential and action selection

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

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

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

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

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

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

For example, a credential c can be associated to a sensitivity level l (e.g. `low`, `medium`, `high`) with assertions of the form

$$c.\texttt{sensitivity} : l\,.$$

Similarly, actions can be given a cost with assertions like

$$\textit{action}.\texttt{cost} : \textit{value}\,.$$

More attributes relevant to candidate selection may be added if needed.

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

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

Then a few standard selection methods can be selected with the attributes of a reserved entity `negotiator`, for example:

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

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

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

$$\texttt{negotiator.selection\_method} : \texttt{adopt}(P)\,.$$

Another important feature of Protune is the support of *metalevel constraints*. They are formulae of the form:

$$\leftarrow L_1, \ldots, L_n\,. \tag{10}$$

A constraint like (10) is *satisfied* w.r.t. a (meta)policy *Pol* and a state Σ, iff no ground instance of $\{L_1, \ldots, L_n\}$ is contained in $\mathsf{cmodel}(\mathit{Pol}, \Sigma)$.

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

Such directives can be easily expressed with constraints of the form:

$$\leftarrow \texttt{credential}(c_1, \_), \ldots, \texttt{credential}(c_n, \_)\,.$$

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

11 Monitoring policies with constraints

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

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

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

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

$$\leftarrow \ldots, \texttt{logged}(\mathit{log\_message})\,.$$

Constraint verification can be implemented efficiently by means of standard algorithms for event-condition-action rules, such as RETE or TREAT [Forgy, 1982].

12 Distributed credentials

Credentials need not be stored at their owner's site nor at their issuer's. Moreover, there is no unique way of searching for a credential, and the responsibility of the search may be of the server, of the client, or even shared [Li et al., 2003]. Therefore, in general, the following entities are distinct:

- the credential issuer,
- the credential repository,
- the credential owner,
- the actor(s) responsible for fetching the credential.

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

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

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

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

If the actor is `self`, then the local engine has to fetch and verify the credential. Search may be nontrivial, as in general it may require a navigation through several servers [Li et al., 2003].

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

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

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

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

Table 2: Translating RT_0 credential types into PROTUNE

Type 1	$A.r \leftarrow D$	$A.r : D \leftarrow \texttt{credential}(C.\texttt{contents} : 'A.r', K)$
Type 2	$A.r \leftarrow B.r_2$	$A.r : X \leftarrow \texttt{credential}(C.\texttt{contents} : 'A.r \leftarrow B.r_2', K),\ B.r_2{:}X$
Type 3	$A.r \leftarrow A.r_1.r_2$	$A.r : X \leftarrow \texttt{credential}(C.\texttt{contents} : 'A.r \leftarrow A.r_1.r_2', K),\ A.r_1.r_2{:}X$
Type 4	$A.r \leftarrow A_1.r_1 \cap \ldots \cap A_n.r_n$	$A.r : X \leftarrow \texttt{credential}(C.\texttt{contents} : 'A.r \leftarrow \mathit{Body}', K),\ \texttt{in\_RT0\_body}(\mathit{Body}, X)$ $\texttt{in\_RT0\_body}(B_1 \cap B_2, X) \leftarrow \texttt{in\_RT0\_body}(B_1, X), \texttt{in\_RT0\_body}(B_2, X)$ $\texttt{in\_RT0\_body}(A.\mathit{Path}, X) \leftarrow A.\mathit{Path} : X$

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

Roughly speaking, the concurrent method is a sort of prefetch strategy that may shorten the response time in some applications.

A more general treatment of the `concurrent` modality can be easily integrated in the negotiator. It suffices to split the actions associated to concurrent provisional predicates; for example, in the case of credentials we assert:

$$\texttt{credential(\_).action\_1} : \textit{fetch\_action}\,,$$
$$\texttt{credential(\_).action\_2} : \textit{verification}\,.$$

The provisional atom is asserted only after both actions have been successfully completed

Then set E' in phases 4 and 5 must be slightly modified to include all `action_1` of the concurrent predicates occurring in the policy. Finally, the negotiator should evaluate `action_2` upon successful completion of the corresponding `action_1`.

13 Libraries and language extensions

Untrained users may find it difficult to formulate autonomously appropriate metapolicies, balancing confidentiality and cooperativeness. Such users would benefit from a library of standard metapolicies that protect their access control policy from the most common forms of information leakage, for instance by setting the `sensitivity` attributes of all state literals to `private`, by default, to protect the local state. At the same time, to reduce blurring, a standard metapolicy may enable early state predicate evaluation when appropriate, cf. Section 5.

The user may personalize the negotiation method by overriding the standard predicate attributes, which is a much simpler task. A standard library of monitoring constraints on metaattributes may protect the user from some common mistakes, such as wrong attribute multiplicity.

As an alternative form of personalization, libraries may be organized in small modules that may be combined together to join their features and compose the desired protection profile.

Some abbreviation predicates of common interest may be defined once and for all in a library. For example, a predicate defining credential chains, that needs some expertise in formulating recursive definitions (see [Bonatti and Samarati, 2000]).

Abbreviation libraries constitute also a means for language extensions, which is of great importance in a growing field like trust management. Some recent extensions are particularly interesting: for example the family of languages *RT* [Li et al., 2002] adopts rich credentials, encoding general facts and rules about role membership and role containment. PEERTRUST supports something similar via signed rules [Gavriloaie et al., 2004]. The semantics of PROTUNE's credentials is only apparently weaker. For instance, the semantics of the four types of

RT_0 credentials [Li et al., 2002] can be encoded with a small PROTUNE library as shown in Table 2.

In this way, PROTUNE becomes a candidate target architecture for a variety of credential languages. The metalanguage enriches those approaches with provisional predicates and declarative negotiation control.

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

Abbreviations and credentials can be linked to the ontologies that specify their meaning by means of a suitable metaattribute: $\mathit{Obj}.\texttt{ontology} : \mathit{URI}$. This attribute may have multiple values because the contents of *Obj* may use symbols defined in different ontologies.

Metapolicy and abbreviation libraries can be exported and stored in standard formats, using RuleML and RDF/OWL.

14 Reputation and recommendations

Reputation-based trust can be formalized by relations between *trustors, trustees, actions*, and *trust levels* [Staab et al., 2004]. For instance, a fact like

$$\texttt{trust}(P, S, \texttt{diagnosis}(\texttt{viral}), 80{-}100)$$

would model the fact that patient P trusts specialist S on diagnosis of viral diseases with an estimated confidence level belonging to the interval $80 - 100$.

Such trust statements can be the basis for trust propagation (e.g. via rules such as "trust X as a bike mechanic if X is trusted as a car mechanic"), for access control decisions such as:

$$\begin{aligned}&\texttt{allow}(\texttt{download}(\texttt{contents/pre\_release}))\\ &\qquad\leftarrow\ \texttt{user}(X),\\ &\qquad\ \texttt{trust}(\texttt{self}, X, \texttt{download}(\texttt{contents/pre\_release}), 90{-}100)\,.\end{aligned}$$

as well as *recommendations*, that is, statements like

$$\begin{aligned}&\texttt{recommend}(\texttt{AmbulanceSupervisor}, \_\mathit{Paramedic}, \texttt{JoinResponseTeam}, \texttt{high})\\ &\qquad\qquad\leftarrow\ \texttt{employed}(\texttt{LondonAmbulance}, \_\mathit{Paramedic})\,.\end{aligned}$$

Such decisions may consider a notion of *risk*, as in

$$\begin{aligned}&\texttt{trust}(\texttt{ProgramX}, \texttt{Server}, \texttt{storeData}(\texttt{Server}), 80{-}100)\\ &\qquad\qquad\leftarrow\ \texttt{Server.owner:CoXYZ},\\ &\qquad\qquad\ \texttt{risk}(\texttt{fail}(\texttt{Server}), 0{-}0.1)\,.\end{aligned}$$

These examples (taken from [Staab et al., 2004]) show how trust and recomendations can be modelled and applied through a small set of predicates. The problem is: How should the basic *facts* about trust and risk be gathered and maintained?

In some case, such facts can be defined by standard policy rules, for example:

$$\begin{aligned}\texttt{trust}(A, B, \texttt{download}(\texttt{file}), 80{-}100) \quad \leftarrow \quad & \texttt{credential}(X, \texttt{VISA}),\\ & X.\texttt{type} : \texttt{credit\_card},\\ & X.\texttt{owner} : B\,.\end{aligned}$$

However, the main current approaches are based on numerical models (see I2-D1 for an extensive illustration of the main approaches) and ad-hoc algorithms for gathering, processing, and propagating historical data about past interactions and the resulting trust measures. In perspective, it may be possible to apply probabilistic, possibilistic or annotated logics to handle such numbers, but so far there is no clear indication that this is the right direction, nor any hint on how to do it.

Further difficulties are: (i) data are application dependent, as well as the procedures for obtaining them; (ii) trust is a dynamic concept, i.e., it changes over time.

The above difficulties suggest a modular approach, namely, the computation and distribution of the basic facts on reputation and risk are delegated to suitable external packages. The results of their processing can be imported via HERMES-like state predicates such as

$$\texttt{in}(\texttt{trust}(X,\, Y, A, L),\ \texttt{reputation\_pckg} : \texttt{eval\_trust}()))$$

$$\texttt{in}(\texttt{risk}(X, L),\ \texttt{reputation\_pckg} : \texttt{eval\_risk}()))$$

(cf. (3) in Section 4). In the above examples the functions eval_trust() and eval_risk() wrap queries to the underlying reputation management and risk assessment algorithms, whatever they are. The two wrappers collect and return the results of those subsystems as a set of terms matching the first argument of the in predicate. Then non-rule-based reputation and risk models can be integrated in PROTUNE policies without any ad-hoc language primitives.

Another advantage of this approach is that a single policy may simultaneously apply different approaches to reputation simply by invoking different packages and combining their results with suitable rules. This kind of flexibility is particularly important in a stage where it is not yet clear which of the competing models of reputation-based trust will become widely accepted, and which application domains they will prove to be good for. It is also possible to change the number and type of parameters of the trust and risk predicates, if needed by a particular reputation model.

This flexible architecture is compatible both with *on-demand* trust computation and with proactive propagation of trust evaluation, as reputation packages may receive asynchronous messages from other peers, concerning warnings and reputation evaluations.

15 Verbalization

The policy language illustrated so far should eventually be intended as an internal format to be automatically generated from controlled natural language sentences, or possibly crafted by hand by specialists to adapt the framework to specific application scenarios. Specialists may resort to the internal rule format to achieve fine-grained control on the policy behavior.

A grammar for a fragment of natural language, capable of capturing the main features of the rule language lies beyond the scope of this report. Our purpose here is collecting some representative example of rules, show their verbalization (i.e., their natural language form) along with the intended internal format. This is to be intended as a set of requisites for the research thread of WG-I2 devoted to the development and adaptation of ACE. We shall highlight some natural verbalizations whose structure is significantly different from the structure of the formal counterpart; these sentences constitute a challenge for the natural language front-end.

Let us start with a simple policy:

> *The user can browse directory "articles" if he is a member of REWERSE.*

This natural language sentence can be used to produce the policy rule

$$\texttt{allow(browse(articles))} \quad \leftarrow \quad \texttt{user(X)}, \texttt{member(X}, \text{“REWERSE”}) \,.$$

In turn, REWERSE membership can be defined by:

> *A user is a REWERSE member if there is a credential signed by REWERSE_CA where the type is "membership" and the object is the public key of the user.*

The translation should look like:

$$\begin{array}{lcl} \texttt{member}(X, \text{“REWERSE”}) & \leftarrow & \texttt{user}(X), \\ & & \texttt{credential}(Y, \text{“REWERSE\_CA”}), \\ & & Y.\texttt{type} : \texttt{membership}, \\ & & Y.\texttt{object} : Z, \\ & & \texttt{public\_key\_of}(Z, X) \,. \end{array}$$

Checking that a key K is the public key of a principal requires a standard challenge procedure: a random number is crypted with K and sent to the principal, who should decrypt the number with his private key and send it back. This knowledge may be encoded in the system as part of a basic rule library. The library rule would look like:

$$\texttt{public\_key\_of}(K, X) \quad \leftarrow \quad \texttt{challenge}(X, K)$$

where `challenge` is a provisional predicate whose associated action is the challenge procedure. Alternatively, the above rule might itself be verbalized.

The challenge-based semantics of "being a public key of" is an example of the security-specific knowledge that must be encoded into the system.

The above policy might be refined to specify that the client is to provide the credential. The verbalization is:

> *The user can browse directory "articles" if he provides a credential stating he is a member of REWERSE.*

This kind of statements should produce both a (numbered) policy rule R_i

$$\begin{array}{lcl} R_i : \texttt{allow(browse(articles))} & \leftarrow & \texttt{user(X)}, \texttt{credential(C)}, \\ & & \texttt{states(C}, \texttt{member(X}, \text{“REWERSE”})) \,. \end{array}$$

and a metapolicy statement like

$$R_i[2].\texttt{actor} : \texttt{peer}$$

stating that the second literal in the body of R_i should be made true by the client (of course another sentence is needed to specify under which conditions $\texttt{states(C}, \texttt{member(X}, \text{“REWERSE”}))$ holds).

Remark 3 *This translation, based on the semantics of verb "provides", seems to require ad hoc translation rules in the natural language front-end.*

The next policy is:

The user can access directory "Kubrick" if he is older than 16

Formalization:

$$\mathtt{allow(access(Kubrick))} \;\leftarrow\; \mathtt{user(X), age(X,Y), Y > 16}\,.$$

Here we assume the translator has enough linguistic knowledge to connect predicate "older" to predicate "age"; alternatively, we might have obtained a similar effect by formulating a (possibly verbalized) suitable rule (that might be part of a basic library, as in the previous example). The above rule must be complemented by a statement like:

The user has age X if some credential signed by PA_CA has the same name as the user, birth date Y, and today-Y=X.

where PA_CA is a certification authority of the public administration. The corresponding rule would be:

$$\begin{array}{lll}\mathtt{age}(U,X) & \leftarrow & \mathtt{user}(U),\\ & & \mathtt{credential}(C, \text{"PA\_CA"}),\\ & & C.\mathtt{name} : Z,\\ & & U.\mathtt{name} : Z,\\ & & C.\mathtt{birth\_date} : Y,\\ & & \mathtt{today} - Y \text{ is } X\,.\end{array}$$

The next example comes from the section on reputation.

ProgramX trusts ServerY to store data with confidence 80-100 if the owner of ServerY is "CoXYZ" and the risk that ServerY fails is less than 0.1

In order to translate this sentence the front-end should know how to map a compound statement about trust and confidence into a single predicate:

$$\begin{array}{ll}\multicolumn{2}{l}{\mathtt{trust(ProgramX, ServerY, storeData(ServerY)}, 80{-}100)}\\ \leftarrow & \mathtt{Server.owner{:}CoXYZ},\\ & \mathtt{risk(fail(ServeYr)}, 0{-}0.1)\,.\end{array}$$

Next we show the verbalization of some constraints. Let us start with a release constraint for privacy protection:

Never release two different ID.

Formalization:

$$\leftarrow \mathtt{credential}(X,Y), \mathtt{credential}(Z,T), X.\mathtt{type} : \mathtt{ID}, Z.\mathtt{type} : \mathtt{ID}, X \neq Z\,.$$

The next constraint is a policy monitoring constraint:

No action may have two actors.

Formalization:

$$\leftarrow X.\mathtt{action} : Y,\ Y.\mathtt{actor} : Z,\ Y.\mathtt{actor} : W,\ Z \neq W\ .$$

Finally, the most common metapolicy statements can be formalized in a pretty natural way:

$$c.\mathtt{sensitivity} : l\ .$$

The sensitivity of c is l

$$\mathit{action}.\mathtt{cost} : \mathit{value}\ .$$

The cost of action is value

$$\mathtt{credential(\_).sensitivity.aggregation\_method} : \mathtt{max}$$

The aggregation method for the sensitivity of credentials is "max".

$$\mathtt{do(\_).cost.aggregation\_method} : \mathtt{sum}\ .$$

The aggregation method for the cost of "do" is "sum".

$$\mathtt{negotiator.selection\_method} : \mathtt{certain\_first}$$

The selection method of the negotiator is "certain first".

16 Future work

The expert reader has noted that this language is aiming at creating a flexible and evolving language and does not deal with completeness issues, that in this context sound like: *"Is negotiation always successful when the policies of the parties allow it?"*

The main potential sources of incompleteness lie in the metadecisions on policy disclosure (some relevant rule may remain hidden), on credential search (who is willing to search for the credentials?), and on location (where are the credentials searched for?). Some strategies clearly affect completeness in the above sense.

These issues are the planned subjects for future deliverables (see the Annex); we are already starting to investigate these topics jointly with other top experts in this area.

Moreover, now that the basic features of the policy language have been laid out, the specialization of the natural language front-end to the policy domain can proceed. Among the challenges related to NLP, we mention the automatic generation of natural language explanations from proofs and filtered policies, including the adaptation of the explicit explanations defined in the metapolicies.

Acknowledgements

REWERSE members Claudiu Duma and Nahid Shahmeri (Linköping), Joachim Peer (St. Gallen), Matteo Baldoni (Turin) and Gerd Wagner (Eindhoven) contributed to this deliverable with suggestions and ideas, discussed and collected during several virtual meetings.

References

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

[Bonatti and Samarati, 2000] Bonatti, P. and Samarati, P. (2000). Regulating service access and information release on the web. In *CCS '00: Proceedings of the 7th ACM conference on computer and communications security*, pages 134–143. ACM Press.

[Forgy, 1982] Forgy, C. L. (1982). RETE: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19:17–37.

[Gavriloaie et al., 2004] Gavriloaie, R., Nejdl, W., Olmedilla, D., Seamons, K., and Winslett., M. (2004). No registration needed: how to use declarative policies and negotiation to access sensitive resources on the semantic web. In *FirsT European Semantic Web Symposium*, Heraklion, Greece.

[Li et al., 2002] Li, N., Mitchell, J., and Winsborough, W. (2002). Design of a role-based trust-management framework. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 114. IEEE Computer Society.

[Li et al., 2003] Li, N., Winsborough, W., and Mitchell, J. (2003). Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86.

[Seamons et al., 2002] Seamons, K., Winslett, M., Yu, T., Smith, B., Child, E., Jacobson, J., Mills, H., and Yu, L. (2002). Requirements for policy languages for trust negotiation. In *POLICY '02: Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, page 68. IEEE Computer Society.

[Staab et al., 2004] Staab, S., Bhargava, B., Lilien, L., Rosenthal, A., Winslett, M., Sloman, M., Dillon, T. S., Chang, E., Hussain, F. K., Nejdl, W., Olmedilla, D., and Kashyap, V. (2004). The pudding of trust. *IEEE Intelligent Systems Journal*, 19(5):74–88.

[Subrahmanian et al., 1995] Subrahmanian, V., Adali, S., Brink, A., Lu, J., Rajput, A., Rogers, T., Ross, R., and Ward, C. (1995). *HERMES: Heterogeneous reasoning and mediator system.* http://www.cs.umd.edu/projects/hermes.

[Winslett et al., 1997] Winslett, M., Ching, N., Jones, V., and Slepchin, I. (1997). Assuring security and privacy for digital library transactions on the web: client and server security policies. In *IEEE ADL '97: Proceedings of the IEEE international forum on Research and technology advances in digital libraries*, pages 140–151. IEEE Computer Society.

[Yu et al., 2001] Yu, T., Winslett, M., and Seamons, K. (2001). Interoperable strategies in automated trust negotiation. In *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 146–155. ACM Press.