A Rule-based Trust Negotiation System

P.A. Bonatti, Dipartimento di Scienze Fisiche, Università “Federico II” di Napoli,
J.L. De Coi, Forschungszentrum L3S,
D. Olmedilla, Telefonica R&D,
L. Sauro, Dipartimento di Scienze Fisiche, Università “Federico II” di Napoli

Abstract—Open distributed environments such as the World Wide Web facilitate information sharing but provide limited support to the protection of sensitive information and resources. Trust negotiation (TN) frameworks have been proposed as a better solution for open environments, in which parties may get in touch and interact without being previously known to each other. In this paper we illustrate PROTUNE, a rule-based TN system. By describing PROTUNE, we will illustrate the advantages that arise from an advanced rule-based approach in terms of deployment efforts, user friendliness, communication efficiency, and interoperability. The generality and technological feasibility of PROTUNE’s approach are assessed through an extensive analysis and experimental evaluations.

Index Terms—Protune, trust negotiation, rule-based policies, sensitive policies, policy exchange, explanations, privacy.

1 INTRODUCTION

The protection of sensitive information and resources plays a crucial role in raising the level of trust in web applications and hence in enabling the potential of the web. For example, even though recent developments such as Web 2.0 demonstrated that many users are willing to share their information publicly, recent experiences with Facebook’s “beacon” service and Virgin’s use of Flickr pictures have shown also that users are not willing to accept every possible use (or abuse) of their data. Therefore, the application of suitable policies for protecting services and sensitive data may determine success or failure of a new service.

Trust negotiation [32], [34], [38] has been introduced to tackle access control requirements as well as privacy preferences in open, distributed environments. The main ideas are the following: (i) access control is attribute-based, that is, based on digital credentials that encode properties (such as subscription ownership, membership to certain organizations, date of birth, etc.) that suffice to fulfil an access control policy without necessarily disclosing the identity of a user completely; (ii) users can formulate policies to control the disclosure of any piece of sensitive information that may be encoded in their credentials. Consequently, in trust negotiation (TN for short), clients and servers have a specular role: for example clients can ask servers to certify their good on-line business practices by exhibiting a membership credential of some auditing organizations, such as eTrust or the Better Business Bureau (hereafter BBB). Moreover, the use of digital credentials – whose integrity and validity can be cryptographically verified – in principle allows to increase information reliability (consider that the information about users encoded in most of the existing on-line accounts is not verified, nor verifiable). Summarizing, all parties eventually enjoy more guarantees, and less sensitive information is disclosed.

In this paper we give an overall picture of the rule-based, PROvisional TrUst NEgotiation system PROTUNE [11]. We are going to: (i) illustrate the advantages that may arise from an advanced rule-based approach to TN in terms of deployment efforts, user friendliness, communication efficiency, and interoperability; and (ii) prove the technological feasibility of the approach through experimental evaluations. PROTUNE intersects the focus of this special issue in several ways: It adopts meta-rules as an extensibility mechanism and as a means to control its behavior. PROTUNE agents exchange rules to interoperate. Moreover, PROTUNE’s policies can be regarded as semantic markup: logical axioms that constrain behavior and usage of web resources. By describing PROTUNE, we will show both how these methods and technologies have been profitably integrated to meet the needs of TN frameworks, and the resulting improvements w.r.t. simpler rule-based approaches that do not support them. For this purpose, a complete view of the system and the interplay of its different components is crucial. The generality of PROTUNE as well as some limitations of the current framework will be evaluated with a fine-grained analysis along multiple dimensions, and through an extensive comparison with competing approaches.

The paper starts with a standard scenario motivating TN (Sec. 2) followed by a discussion of several issues raised by the scenario (Sec. 3). Then we describe PROTUNE’s architecture (Sec. 4), language (Sec. 5), and negotiations (Sec. 6). The details of policy protection
will be explained in Sec. 6.2. The explanation facility of PROTUNE will be illustrated in Sec. 7. Sec. 8 will give a few details on the implementation. Next we devote two sections to an extensive comparison with other policy languages and frameworks, and to an evaluation of PROTUNE according to multiple criteria (Sec. 9 and Sec. 10). A final section summarizes the overall discussion and draws some conclusions and directions for further work.

2 A Motivating Scenario for TN

Bob's birthday is next week and Alice plans to use today's lunch break for buying an on-line novel of Bob's favorite writer. She finds out that an on-line bookshop she never heard about before sells the book at a very cheap price. By interacting with the bookshop's server Alice learns that she has to provide either her credit card number or a pair (userId,password) for a previously created account. Alice does not want to create a new account on the fly, so releasing her credit card is the only option. However she is willing to give such information only to trusted on-line shops (let's say, belonging to the BBB), therefore she asks the bookshop to provide such information. The bookshop belongs indeed to the BBB and is willing to disclose such credential to anyone. This satisfies Alice, who provides her credit card number. After having interacted with the VISA server to check that the credit card is valid, the bookshop asks Alice for other information, in order to understand which customer category she belongs to and apply the corresponding sale strategy. Sale strategies typically depend on personal data (country, age, profession, . . . ) and purchase-related data (frequency, payment preferences, reputation, . . . ) and may determine discounts and other advantages. However, the lunch break is already over and Alice decides to abort the transaction.

3 Expressing/Handling Evidence in TN

The above scenario illustrates important aspects of TN. First, information exchange is bidirectional and users should be allowed to pose counter-requests. Second, in modern policies such exchange typically involves an ample range of data. Third, the need of collecting security-related information conflicts with usability: the large amount of information required to complete a transaction may become a disincentive not only because of privacy concerns, but also because of the effort needed to provide and control it manually.

Accordingly, the goal of trust negotiation frameworks is automating information exchange by having two software agents release and request information, based on (a machine understandable formulation of) client and server policies. The two agents play specular roles, due to the bidirectional nature of trust negotiation, therefore we will call them peers even if they may act on behalf of a client and a server, respectively.

To see the advantages of automated negotiations, consider that a policy we wrote for the above scenario specifies 15 different ways of getting the resource depending on whether: (i) the resource is public; (ii) the user exploits an existing account; (iii) the user pays with Visa, Mastercard, FastPay, or PayPal; (iv) the ID is a passport, a driving licence, a student ID; (v) the user presents a Gold Customer card to the website. Users should not be presented with such long lists. Moreover, users should be able to state once and for all that sensitive credentials should be disclosed only after a server has released a suitable certification. Then a user of a TN system like PROTUNE would only have to click a link on a web page to get a resource; all information exchange and service certification controls would be handled transparently by software agents. Remarkably, a single automated credit card release suffices to replace the manual input of up to 9 values, including a 16-digit number.

We pointed out above that policies can be based on a variety of user properties. Some of them can be encoded into digital credentials; we call such information strong evidence as its validity and authenticity can be certified by means of cryptographic techniques. On the opposite side of the spectrum (weak evidence) we find unsigned declarations such as those we commonly issue by accepting copyright agreements just by clicking a button on a pop-up window. Further forms of weak evidence consist in reputation scores that a user or provider may be given by an on-line community.

TN agents may request evidence in many ways. At one end of the spectrum we find agents that ask for information items one by one. This kind of negotiation is inefficient and poor from a privacy perspective: For example, Alice may be asked first for her credit card and later for her ID, and if she has no ID credential then the negotiation has to backtrack and follow the alternative path (create a new account); moreover, Alice has unnecessarily disclosed her credit card number. Another drawback is that Alice does not know at any step whether the current request of the server – if refused – would be replaced after backtracking by another request that she considers preferable from a privacy perspective.

At the opposite end of the spectrum, agents tell all the alternatives at once. Not only this approach reduces the need of backtracking; as a further advantage, users can choose among all the alternative ways of fulfilling a server's policy those that better fit their privacy preferences (e.g., a user may prefer to disclose an IEEE membership card rather than a credit card number, and a student ID rather than a complete passport). This approach, however, has important implication on message size, because the list of all the alternative sets of information items that fulfill a policy may grow significantly with the size of the policy due to combinatorial effects. Consider for example a request for a credit card and an ID, where the credit card can be VISA, Mastercard, . . . , and the ID can be a passport, a driving licence, a student card, and so on. In order to express more efficiently
and succinctly multiple alternative and compound information requests, agents should actually send parts of their policies, as a compact representation of the conditions that need to be fulfilled. Another drawback of the approach based on extensional credential set listing, is that it gives no information about the policy’s structure, thereby preventing accurate automated explanations.

Under the approach based on sending policies, the negotiation between Alice and the bookstore would be mediated by two software agents A and B acting on their behalf. The negotiation would look like the following:

1) A sends to B a purchase request, buy(book123);
2) B sends to A a set of rules \(P_B\) encoding B’s local policy (possibly extended with some auxiliary definitions) for purchasing book123;
3) A looks in Alice’s portfolio for one or more sets of credentials \(C_A\) such that the authorization \(\text{allow}(\text{buy(book123)})\) can be inferred from \(P_B \cup C_A\); in the above scenario \(C_A\) contains a digital representation of Alice’s credit card;
4) A sends to B a set of rules \(P_A\) encoding A’s local policy + auxiliary definitions for releasing \(C_A\);
5) B looks in the server’s portfolio for the sets of credentials \(C_B\) such that \(\text{allow}(\text{release}(C_A))\) can be inferred from \(P_A \cup C_B\); in the above scenario \(C_B\) contains a digital BBB membership certificate;
6) B can prove \(\text{allow}(\text{release}(C_B))\) from its local policy, therefore \(C_B\) is sent to A;
7) A can now prove \(\text{allow}(\text{release}(C_A))\) from its local policy \(P_A \cup C_B\), therefore \(C_A\) is sent to B;
8) B can now prove \(\text{allow}(\text{buy(book123)})\) from its local policy and \(C_A\), and the purchase transaction succeeds.

PROTUNE supports the widest range of (weak and strong) evidence varieties, and its agents communicate by exchanging parts of their policies and local information as in the above example, ensuring that these messages do not leak any confidential information.

5 PROTUNE’S LANGUAGE

PROTUNE’s language [11] is based on normal logic program rules \(A \leftarrow L_1, \ldots, L_n\) where \(A\) is a standard logical atom (called the head of the rule) and \(L_1, \ldots, L_n\) (the body of the rule) are literals, that is, \(L_i\) equals either \(B_i\) or \(\neg B_i\) for some logical atom \(B_i\). This basic syntax is enhanced with some syntactic sugar as discussed below.

For example, the policy that allows to buy a book by giving a credit card can be encoded with a set of rules including:

\[
\begin{align*}
&\text{allow(buy(Resource))} \leftarrow \\
&\text{credential(C), valid_credit_card(C), accepted_credit_card(C).}
\end{align*}
\]

\[
\begin{align*}
&\text{valid_credit_card(C)} \leftarrow \\
&\text{credit_card(C), C.expiration : Exp, date(Today), Exp > Today.}
\end{align*}
\]

Alice’s credential release policy can be defined with the same language:

\[
\begin{align*}
&\text{allow(release(my_credit_card))} \leftarrow \\
&\text{credential(C), C.issuers:'BBB', C.expiration : Exp, date(Today), Exp > Today.}
\end{align*}
\]

Note that PROTUNE rules are meant to define access control and release policies (rules (1) and (3)) as well as any auxiliary concept needed to formulate the policy (rule (2)).

In order to avoid undecidability problems, PROTUNE rules are essentially restricted to function-free rules with some syntactic sugar: function symbols are allowed in the special predicate allow to model parameterized resource requests. The nesting level is limited to 1. An atom like allow(buy(Resource)) is treated like the function-free atom allow(buy, Resource).

Since digital credentials and unsigned declarations (frequently modeled as html forms) are semistructured objects, PROTUNE is enhanced with a FLORA-like object-oriented syntax. Expression \(X.\text{attr}: v\) means that \(X\) has an attribute \(\text{attr}\) with value \(v\); e.g. in the previous example \(C.\text{expiration:Exp}\) is an O.O. expression meaning that \(\text{Exp}\) is the value of \(C\)'s attribute expiration. Such O.O. expression are only abbreviations of standard first-order syntax; actually \(X.\text{attr}: v\) abbreviates the standard atom \(\text{attr}(X, v)\). This representation allows multi-valued attributes – as in \(X.\text{sibling: john, X.sibling:mary}\) – and is compatible with the semantic web standards RDF and OWL. \(X.\text{attr}: v\) corresponds to an RDF triple. We support attribute concatenation, like \(X.\text{expiration.year:2010}\). In the body, this is expanded to \(\text{expiration}(X, V).\text{year}(V, 2010)\), where \(V\) is a fresh variable. Similarly, a rule head \(p(X.\text{attr})\) would be replaced by \(p(Z)\) where \(Z\) is a fresh variable, and the rule body would be extended with \(\text{attr}(X, Z)\).

Remark 5.1: Adopting a standard function-free logic programming language as internal format brings two benefits: (i) we can directly adopt well-established and understood semantics, and (ii) we can exchange rules
by encoding them with the emerging rule interchange formats such as W3C RIF, whose core already covers PROTUNE rules. It is worth mentioning that RIF’s use case collection includes our rule exchange scenario.

Rules are interpreted according to the stable model semantics [23], a two-valued\(^3\) semantics where negation faithfully models underivability.

Negation is restricted in PROTUNE. First, it should be stratified, in order to keep the data complexity of inference polynomial (in general it is NP-hard), and in order to avoid ambiguous policies with multiple stable models (cf. the survey in [13]). We further restrict negation to state predicates (conceptually defined by sets of ground facts only), in order to simplify the reasoning tasks related to credential selection and explanations.

Second, negation should not be applied to any predicate that depends on some evidence (such as credentials) because it is impossible to check reliably whether a peer does not have a certain digital token. This argument has been first raised in [32], and caused this restriction to be systematically adopted thereafter.

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

- **Decision Predicates** represent policy outcomes. Currently, PROTUNE supports allow, for access control decisions, and sign, for issuing signed statements (to define dynamic certificate release policies).
- **Logical Predicates** are abbreviation predicates defining auxiliary concepts such as valid\_credit\_card(C), or state-query predicates that read the current state without modifying it, for example date(Today).\(^4\)
- **Constraint Predicates** comprise the usual arithmetic predicates (e.g. Exp > Today).
- **Provisional Predicates** may change their value during negotiation as a result of an action carried out by some peer. PROTUNE has a few built-in provisional predicates including credential(C) and declaration(D) that, roughly speaking, become true when a credential C or an unsigned declaration D (respectively) are released by the other peer. It is possible to define new provisional predicates through PROTUNE’s metalanguage (described later).

Most policies make decisions based on the contents of the information system they protect; for example access control to a digital library typically takes into account the database of customers, their subscriptions, and profile information. Therefore, a TN framework must be able to interact with legacy data and software. PROTUNE tackles this need by adopting a uniform syntax supported by Impact’s agents [35]. The built-in provisional predicate in/2 provides a uniform interface to external packages (including databases and other data sources) that can be queried with atoms of the form in([X\(_1\), ..., X\(_n\)], packageName:function(Arg\(_1\), ..., Arg\(_n\))) where the variable list [X\(_1\), ..., X\(_n\)] ranges over the fields of the set of objects returned by the code call packageName:function(Arg\(_1\), ..., Arg\(_n\)). For example if the code call is access\_query('select A, B from C where D = e') then X\(_1\) (resp. X\(_2\)) is bound to field A (resp. B) of the tuples of table C such that D = e. In practice, the implementation needs a suitable wrappers for each package and appropriate caching techniques. Currently, PROTUNE has wrappers for file systems, RDF stores, LDAP servers, and relational DBMS via JDBC.

Last but not least, ‘in’ can be used to gather reputation evidence by invoking a suitable web service or by extracting the information from a web page [9].

We can assume that in is the only state-query predicate as all other state queries can be defined on top of in. For example, the date predicate can be defined by the rule date(D) ← in([D], shell:exec('date')), where shell is a wrapper for shell commands. The declarative semantics of the in predicate is the (possibly infinite) set of all true ground facts in(α, β); they are implicitly added to the policy.

The two built-in provisional predicates credential and declaration deserve a few more words on their implementation. When an X.509 credential χ is received, a peer checks its validity; if χ passes the validity check,
<table>
<thead>
<tr>
<th>meta-attribute</th>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>atoms</td>
<td>provisional, logical</td>
</tr>
<tr>
<td>actor</td>
<td>provisional atoms</td>
<td>self, peer</td>
</tr>
<tr>
<td>ontology</td>
<td>atoms, literals</td>
<td>URI</td>
</tr>
<tr>
<td>explanation</td>
<td>atoms, literals</td>
<td>immediate, deferred</td>
</tr>
<tr>
<td>sensitivity</td>
<td>atoms, rules</td>
<td>string expression</td>
</tr>
<tr>
<td>blurred</td>
<td>literals</td>
<td>private, public</td>
</tr>
<tr>
<td></td>
<td></td>
<td>true, false</td>
</tr>
</tbody>
</table>

**TABLE 1**

Table of meta-attributes

then it is converted into a set of facts $\text{attr}(c_X, \text{value})$ that represent its attributes, where $c_X$ is a constant that represents the entire credential. These facts and $\text{credential}(c_X)$ are stored locally for the rest of the current negotiation and can be used thereafter as if they were part of the policy. The treatment of unsigned declarations is similar; the only difference is that cryptographic validity checks are not needed.

Unsigned declarations can be used to encode a traditional password-based authentication procedure as in:

\[
\text{authenticated} \leftarrow \text{declaration}(D),
\text{valid_login_data}(D.\text{username}, D.\text{password})
\]

where $\text{valid_login_data}$ is a suitable state predicate specifying correct login-password pairs (it may be defined by exploiting in to query a legacy database table).

**Remark 5.2:** In order to interoperate, \textsc{protune} agents need to share only the procedural understanding of a few built-in predicates (such as allow, credential, declaration) and rule semantics. Policy authors are free to define and use high-level abstractions (e.g. “registered user” or “accepted credit card”) as far as they are linked by appropriate rules to the concrete evidence they represent. This represents a fundamental step beyond classical standardization processes that require prior agreements on shared specifications for each and every extension of a standard (sometimes called profiles).

The metalanguage

\textsc{protune} needs a metalanguage for several purposes: (i) as an extensibility mechanism to define application-dependent provisional predicates and explanations; (ii) as a means of annotating the confidential parts of policies. Meta-annotations may be included in the messages that release policies to other peers.

**Metapolicies** consist of rules similar to object-level rules with the difference that built-in predicates comprise Prolog-style metapredicates to inspect terms and check groundness. Metarules can use a set of reserved attributes in order to describe properties of the atoms occurring in the object policy (see table 1). For example a piece of metapolicy could be:

\[
\log(X) \rightarrow \text{type} : \text{provisional}.
\]

\[
\log(X) \rightarrow \text{ontology} : \text{http : //www.protune.com/logAction}.
\]

\[
\log(X) \rightarrow \text{actor} : \text{self}.
\]

Here “\$\rightarrow\$” connects a metaterm to its metaproperties (we use it in place of ‘\$’ in order to disambiguate the grammar and distinguish meta-atoms from complex terms). The meta-attribute type establishes the type of a user defined atom, i.e. logical or provisional. For provisional atoms the meta-attribute ontology determines the action to be performed. If an action $p$ must be executed locally we assert $p \rightarrow \text{actor} : \text{self}$, otherwise $p \rightarrow \text{actor} : \text{peer}$.

Another meta-attribute which applies to provisional predicates is execution: in general an action may require some input parameter(s), therefore the \textsc{protune} framework must make sure that any attempt to perform such action is carried out only if all its input parameters have been provided (i.e., have been instantiated). A policy author can specify the input parameters of an action by defining “execution” metarules like the following one.

\[
\log(X) \rightarrow \text{execution} : \text{immediate} \leftarrow \text{ground}(X).
\]

This metarule states that $\log(X)$ can be immediately executed only if $X$ has been instantiated.

Furthermore, atoms and rules may be sensitive data that cannot be freely disclosed. For example, in a social network scenario, disclosing that only my best friends can see these pictures is undesirable, as some friends may realize they are not “best friends”. In business applications, a policy may reveal confidential relationships with a business partner. Then meta-attribute sensitivity is used to establish whether an atom or a rule should be kept private when a policy is disclosed to another peer.

Size (hence efficiency) is another reason for not releasing the entire policy: since some predicates are imported from potentially large database tables, it is advisable not to send their tuples across the network. We use the meta-attribute blurred to say that the definition of a given predicate might not be entirely included in the policy sent to another peer. By using these metafacts, the recipient can understand when the closed world assumption can be assumed and negation-as-failure applied (for a blurred atom, negation-as-failure is inappropriate because the lack of matching rules does not imply that the sender cannot derive that atom). Given the intended meaning of blurred, it is clear that if a literal $L$ is blurred, then all literals $L'$ more general than $L$ should be blurred as well (because any missing rule unifying with $L$ unifies also with $L'$). Therefore we implicitly assume that every metapolicy contains the rules ($L' \rightarrow \text{blurred} : \text{true} \leftarrow (L \rightarrow \text{blurred} : \text{true})$ and $(L \rightarrow \text{blurred} : \text{false} \leftarrow (L' \rightarrow \text{blurred} : \text{false})$, for all literals $L$ and $L'$ such that $L'$ is more general than $L$.

Sections 6.2 and 7 will discuss meta-attribute explanation and further details of sensitivity and blurred. In the following a metapolicy will be considered part of the policy $KB_i$ of a peer.

6 Negotiations

Conceptually, negotiations consist of sequences of steps that we will identify with an initial segment of the natural numbers. At each step $s$, every peer\$^5\$ is associated

\[5.\text{So far we have talked about 2-peer negotiations, however the framework can be extended to 3 or more peers.}\]
to the following set of logic programs:

- $KB_i(s)$, comprising the policy of peer $i$, the auxiliary definitions such as rule (2) and the definition of $i$’s state predicates such as $\text{in}$;
- $\text{Prov}_i(s)$, a set of ground atoms that defines the provisional predicates of $i$ at step $s$; the credentials and declarations of $i$ are not included here; $\text{Prov}_i(s)$ may include an encoding of some credentials and declarations from other peers;
- $\text{Ev}_i(s)$, the peer’s portfolio of evidence, that contains the logical encoding of $i$’s own credentials and unsigned declarations;
- $R_{i,j}(s)$, the last set of policy rules, auxiliary definitions and metarules that $i$ has received from $j$, as a formulation of $j$’s requests and counter-requests.

Note that both $\text{Prov}_i$ and $\text{Ev}_i$ are sets of ground atoms, that encode semistructured information as explained in Section 5. Parameter $s$ will sometimes be omitted when clear from the context or irrelevant.

In PROTUNE the above programs must satisfy the following constraints, for all steps $s, s'$ such that $s < s'$:

- **C1:** $KB_i(s) = KB_{i}(s')$;
- **C2:** $\text{Prov}_i(s) \subseteq \text{Prov}_i(s')$ and $\text{Ev}_i(s) \subseteq \text{Ev}_i(s')$.

In other words, we assume that during a single negotiation, the policy and the local state of any peer $i$ remain constant (accordingly, we will often omit the parameter $s$), while provisional predicates – including the evidence released by the other peers – may grow. The foreign policies $R_{i,j}(s)$ may have some nonmonotonic aspects related to the metarules they contain.

**Remark 6.1:** C1 and C2 are necessary to ensure that the authorizations that allow information disclosure cannot be invalidated by any action executed later during the same negotiation. There is an interesting issue: when a state change invalidates an authorization granted during a previous negotiation, what should recipients do with the disclosed information? This is one of the toughest open issues in the field of usage control and its solution is beyond the scope of this paper. An extreme solution is freezing policies and states, but this solution would be too rigid in most practical settings.

### 6.1 Access and disclosure control

The above logic programs evolve as a result of peer actions, that change the value of provisional predicates. For instance, this is the case of $\text{release}(e)$ that denotes the action of releasing a credential or declaration $e$. All the actions executed by a peer $i$ should be allowed by $i$’s policy. A peer $i$ can make such access control and/or information release decisions using the logic program $P_i(s) = KB_i \cup \text{Prov}_i(s)$. Intuitively, $i$ checks whether a certain request is authorized or a certain piece of evidence can be released by applying the local policy and its auxiliary definitions and data to the evidence gathered from the other peers. More formally, an action $a$ is permitted at step $s$ iff $\text{allow}(a)$ belongs to the unique stable model of $P_i(s)$.

With the above notion of permission, we can formulate the third constraint that negotiations should satisfy:  

**C3:** (Safety) Let $p \in \{\text{credential, declaration}\}$. For all pieces of evidence $e$ such that $p(e) \in \text{Prov}_i(s+1) \setminus \text{Prov}_i(s)$, there must be a peer $j$ such that $p(e) \in \text{Ev}_j(s)$ and $\text{allow}(\text{release}(e))$ belongs to the unique stable model of $P_j(s)$.

The actions executed by each peer (and the actual pieces of evidence released) depend on the negotiation strategies adopted by each peer. In the next section we illustrate the reasoning tasks that strategies may use.

### 6.2 Filtering and blurring

Each peer $i$ at step $s$ has a list of open goals $OG_i(s)$ (a set of ground atoms). If $i$ is a server, then $OG_i(s)$ contains the ground atom $\text{allow}(\text{Req})$, where $\text{Req}$ is the initial request received from a client. Moreover, each peer $i$ maintains in $OG_i(s)$ the set of ground atoms $\text{allow}(\text{release}(e))$ such that $e$ is a piece of evidence in $\text{Ev}_i(s)$ that $i$’s strategy has selected as a candidate for release, but at step $s$ the action $\text{release}(e)$ is not yet permitted. To “unlock” these actions, $i$ has to construct a request for further evidence.

For each goal $G \in OG_i(s)$, the strategy may need several pieces of information as an input, for example: (i) Is it at all possible to derive $G$, given enough further evidence? (ii) Which are the local actions that may help in deriving $G$? (iii) Which rules have to be sent off to request the missing evidence for $G$? In the rest of this section we illustrate the reasoning tasks associated to the above questions.

We start with the first question in the above list. It can be answered by solving an abduction problem: is there any evidence $E$ that makes it possible to derive $G$ from the current program $P_i(s)$ (so as to permit the action encoded in $G$)? This question can be solved by means of particular SLD derivations. We need an auxiliary definition first: an atom $t.\text{attr}:u$ and its negation are called semi-provisional w.r.t. a set of literals $S$ iff there exists a sequence of terms $t_1, \ldots, t_n$ such that (i) $t_1$ is an argument of some provisional atom of $S$, (ii) for $i = 1, \ldots, n - 1$, there exists attr’ such that $S$ contains an atom $t_i.\text{attr'}:t_{i+1}$, and (iii) $t_n = t$.

**Definition 6.1:** A derivation of type 1 of $G$ in a context $(i, s)$ is an SLD derivation $\Delta$ of $G$ from $P_i(s)$ whose last goal $G_\Delta$ contains only (i) provisional predicates and semi-provisional atoms $t.\text{attr}:u$ w.r.t. $G_\Delta$; and (ii) negative literals. Moreover, there must be a grounding substitution $\theta$ such that $G_\Delta\theta$ is coherent and all the

6. In the following, in order to simplify notation, we assume that each piece of evidence is denoted by a constant $e$ which is unique across all peers. We assume also the uniqueness of all the constants $d$ occurring in the atoms $d.\text{attr}:val$ that describe the attributes of semistructured evidence. The real system does not have to satisfy the same requirement: it suffices to rename all such constants when new evidence is incorporated in $\text{Prov}_i(s)$.

7. Programs may contain negative literals that are never rewritten by the resolution rule and are accumulated during the derivation.
negative literals in $G_\Delta \theta$ are satisfied by the unique stable model of $P_i(s)$.

Clearly, the following proposition holds:

**Theorem 6.1**: Let $E$ be a set of ground atoms, containing only provisional predicates and semi-provisional attribute atoms. A ground goal $G$ is true in the unique stable model of $P_i(s) \cup E$ iff there exist a derivation of type 1 of $G$ in $(i,s)$ with final goal $G_\Delta$, and a substitution $\sigma$ such that the positive literals of $G_\Delta \sigma$ are contained in $E$.

**Proof sketch**: Let $M$ be the unique stable model of $P_i(s) \cup E$. If $M \models G$, then $G$ belongs to its Gelfond-Lifschitz reduct $\text{GL}(P_i(s) \cup E, M)$, therefore there exists a successful SLD derivation $\Delta$ of $G$ from $\text{GL}(P_i(s) \cup E, M)$. Obtain an SLD derivation $\Delta'$ from $\Delta$ by re-introducing in the applied rules the negative literals removed by the Gelfond-Lifschitz reduction, and by removing all resolutions with the facts in $E$. It is not hard to see that one obtains a derivation of type 1 of $G$ in context $(i,s)$.

Conversely, consider a derivation $\Delta$ of type 1 of $G$ in context $(i,s)$, with final goal $G_\Delta$ and substitution $\theta$. Let $E$ be the set of positive literals in $G_\Delta \theta$. It can be easily verified that by removing all negative literals from $\Delta$ we obtain an SLD derivation $\Delta'$ of $G$ from $\text{GL}(P_i(s) \cup E, M)$. This derivation can be extended to a successful derivation by unifying the atoms in its last goal with the atoms in $E$.

As a consequence of the above theorem, we know that a goal $G$ with a derivation of type 1 can be derived if enough additional evidence is given. Note that part of this evidence may consist of provisional predicates with meta-attribute actor:self, whose actions should be executed by $i$ itself. Then derivations of type 1 can be used to identify local actions that are potentially useful for the negotiation (a strategy can execute them immediately if their execution meta-attribute is immediate).

Next we turn to the issue of constructing a set of rules that encodes a request to another peer. Derivations of type 1 have to be restricted in order to prevent sensitive rules and predicates to be disclosed to other peers. We need two preliminary definitions:

Let $PF_i(s)$, the pre-filtered version of $P_i(s)$, be the program obtained from $P_i(s)$ by:

1) eliminating all so-called private rules $r$, i.e. such that $r$-sensitivity:private belongs to the unique stable model of $P_i(s)$;

2) for all ground instances $r\theta$ of a private rule $r$ such that body$(r\theta)$ is true in the unique stable model of $P_i(s)$, add the head of $r\theta$ to $PF_i(s)$.

In other words, sensitive rules are replaced with their immediate consequences.

Second, we say a literal $L$ is blurred iff either $L$-sensitivity:private or $L$-blurred:true belong to the unique stable model of $P_i(s)$, where $L'$ is either $L$ or its complement.

The following definition essentially filters a derivation of type 1 to remove all sensitive information.

**Definition 6.2**: A derivation of type 2 of $G$ in a context $(i,s)$ is an SLD derivation $\Delta$ of $G$ from $PF_i(s)$ satisfying the following conditions:

1) at each step of the derivation, the selected atom is not blurred;

2) the last goal $G_\Delta$ contains only (i) provisional predicates and semi-provisional atoms $t$.attr:u w.r.t. $G_\Delta$; (ii) negative literals; (iii) blurred literals.

3) there must be a grounding substitution $\theta$ such that $G_\Delta \theta$ is coherent and all the negative literals in $G_\Delta \theta$ are satisfied by the unique stable model of $P_i(s)$.

We denote by $P_\Delta$ the set of rules and facts applied in $\Delta$.

Now, roughly speaking, if another peer $j$ can prove $G$ using $P_\Delta$ and a subset $E$ of its portfolio $E_{v_j}(s)$, then the authorization encoded in $G$ can certainly be obtained by releasing $E$, as proved by the following theorem.

**Theorem 6.2**: For all peers $j \neq i$ and all sets of provisional atoms $E \subseteq E_{v_j}(s)$, if there exists a successful SLD derivation $\Delta$ of a ground $G$ from $P_\Delta \cup E$, then $G$ holds in the unique stable model of $P_i(s) \cup E$.

**Proof sketch**: Obtain $\Delta'$ from $\Delta$ by removing the applications of resolution to facts in $E$. Since $P_\Delta$ contains no private rules, nor any rule with a private predicate in the head, we have that $\Delta'$ is a derivation of type 2 of $G$ in $(i,s)$ (the coherency requirement on the last goal is trivially satisfied as $G_\Delta$ contains only (positive) provisional and semi-provisional literals). Now obtain another derivation $\Delta''$ from $\Delta$ as follows: replace the applications of resolution to the heads of “compiled” private rules, with (possibly more articulated) derivations, using the original rules of $P_i(s)$. This process may introduce negative literals in the last step of the derivation; however, due to the syntactic restrictions on negation, these literals cannot contain any complement of any of the (provisional) atoms in $E$. Then it is easy to check that $\Delta''$ is a derivation of type 1, and hence $G$ holds in the stable model of $P_i(s) \cup E$, by Theorem 6.1.

In general, however, $j$ has no guarantees, even if $G$ is actually derivable – e.g., if all the derivations of $G$ from $P_\Delta \cup E_{v_j}(s)$ end up in a nonempty set of blurred literals, then there is no certainty that the actual definition of those literals in $P_i(s)$ allows to prove $G$.

PROTUNE adopts the convention of inserting $\neg A \rightarrow \text{blurred}\:\text{false}$ in disclosed policies whenever $A$ matches no rules in $P_i(s)$. Then, if the policy is safe, Theorem 6.2 can be extended to all $\Delta$ whose last goal is a set of non-blurred (ground) negative literals. The proof relies on the uniqueness of the constants that denote semi-structured objects (cf. footnote 6), that together with safeness implies that the additional evidence $E$ cannot invalidate any negative literal used in any previous inference from $P_i(s)$ (i.e. policies are monotonic w.r.t. the new evidence).

Given a goal $G \in O_{G_i}(s)$, the negotiation strategy of $i$ may opt for releasing to a peer $j$ either the union of all $P_\Delta$, for all the type 2 derivations $\Delta$ of $G$, or a
subset thereof. Denote the released rule set with \( R_{j,i} \) and consider how the recipient \( j \) can use it to select in its portfolio \( Ev_j(s) \) the evidence to be returned to \( i \). Given a ranking of evidence sensitivity, the system may adopt a greedy strategy and minimize the sensitivity of disclosed evidence at each step. In general, this strategy minimizes the sensitivity of the information disclosed by the entire negotiation only in the absence of private policy rules and blurring, because the uncertainty they introduce may cause unnecessary information disclosure. The problem is a weighted abduction problem: Given \( R_{j,i}, Ev_j(s) \), a set of integrity constraints \( IC_j \) forbidding the disclosure of certain credential sets (because they are too sensitive), and a PTIME-computable, monotonic sensitivity aggregation function \( \text{sen} \) mapping sets of evidence onto a finite poset \((\Sigma, \preceq)\) of sensitivity values, compute a \( \preceq \)-minimal set \( E \subseteq Ev_j(s) \) such that \( R_{j,i} \cup E \models G \) and \( R_{j,i} \cup E \cup IC_j \) is consistent. We proved in [7] that this problem is \( \text{FNP}/\text{log-complete} \) [15], that is, a solution can be nondeterministically computed in polynomial time given an oracle for an optimization problem in \( \text{NP} \).

7 Documentation and Explanations

As the formal representation of policies is typically understood by trained people only, explaining policies and related systems decisions to end users is essential to bring policy-aware systems to their full potential. An automated explanation facility is expected to reduce the costs associated to writing documentation, and guarantees that documentation be always up to date, since it is derived automatically from the executable rules. Explanation facilities are also a powerful tool for validation and debugging. In case of an unexpected decision about access control or information release, an explanation facility can help trace which inferences led to that decision and diagnose the causes of the problem. Moreover, if an explanation facility can explain policy decisions in what-if scenarios, then explanations can be used to validate policies before their deployment, by running a suite of test cases on the policy’s logic.

For these reasons, we augmented PROTUNE with the explanation facility PROTUNEX [12] in order to support the automated creation of high-quality documentation and contextualized explanations. PROTUNEX can answer how-to queries that explain the general structure of a policy, as well as context-dependent queries such as why/why-not, based on the last filtered policy of a negotiation, and what-if queries, that are based on a hypothetical set of credentials and declarations.

PROTUNEX has been designed to keep deployment costs low. Most of the text is produced automatically, by assembling automatically elementary pieces of text that describe atomic conditions or literals. Such building blocks can be specified with simple metafacts such as:

\[
\text{auth}(X) \rightarrow \text{explanation}:[X, \text{\textquoteleft is authenticated\textquoteright}].
\]

Unlike other recent approaches in the semantic web arena, such as [29], [27] PROTUNEX does not simply output a single proof tree or a counterexample. PROTUNEX rather provides an HTML hypertext that allows navigation across different proof attempts, including both successful and failed derivations. However, unlike prolog tracers, PROTUNEX does not follow the backtracking process step by step – a process that prolog programmers know to be often frustrating.

The hypertests constructed by PROTUNEX have a node for each atomic goal called in some proof attempt. A single atomic goal \( A \) in a rule may generate different nodes corresponding to different instances of \( A \), if the rule is called multiple times. Nodes corresponding to different variants of the same goal are collapsed. Figure 2(A) shows a hypertext node created by PROTUNEX. Note that each node provides both local information and global information, as it illustrates the rules that directly apply to the atom associated to the node, as well as the final outcome of the proofs that can be built for each of those rules. For example, from Figure 2(A) one can understand that: (i) an authorization for downloading a paper can be derived in three possible ways, by applying rules 2, 3, or 4 (local information); (ii) all of these proof attempts eventually fail (global information), even if (iii) some of the subgoals of rule 3 can actually be proved (possibly with the use of further rules) and eventually return answer substitutions such as Subscription = basic_law_pubs (global information). Note also that each node provides information about all the proofs for its associated atom, by listing all rules that directly match it, as well as all proof outcomes for each choice.

The [details] hyperlinks associated to each subgoal let users navigate to the node describing that subgoal. By selecting a link like [Subscription = basic_law_pubs], instead, one can move to a specialized version of the current node, where the selected substitution is applied. In our example, by following the aforementioned link one would reach the page illustrated in Figure 2(B). By reading the global information associated to each rule, users can quickly identify the proof attempts that do not match their expectations; by following detail links, users can visit the proof steps and try a diagnosis; by applying substitution links, users can simplify general, potentially complex descriptions and specialized them to the special case they are interested in, to enhance readability. To the best of our knowledge, PROTUNEX is the first and only explanation facility that supports a navigation mechanism with these characteristics.

PROTUNEX supports a simple heuristics called clustering for referring in a human-understandable way to a semistructured object \( C \) (e.g. a credential or a declaration) rather than using the internal identifier of \( C \). Roughly speaking, PROTUNEX collects all the attribute atoms \( \text{att}_V \) and outputs a combined description, as in:

- \( c012 \) is a credential whose \text{issuer} is \text{UK government and}

9. The implementation is analogous to XSB’s tabling mechanism.
whose type is ID.

The set of attributes helps users to identify the credential denoted by the internal identifier c012. Since attributes are selected among those occurring in the verbalized rule, only the attributes relevant to the current context are displayed. This heuristic has been working well in our experiments, and – unlike classical approaches – it does not require a knowledge engineer to give any explicit definition of “key attributes” a priori.

Another important feature of PROTUNE is that it can frequently focus explanations by removing irrelevant information. This is relatively simple in the explanation of why an atom \( A \) can be derived: by looking at global information, PROTUNE can remove from \( A \)’s node all the rules that cannot be applied in any proof of \( A \). Recognizing irrelevant information in the proof of underviability is more difficult. One may be tempted to remove from rule bodies all the literals that can be derived. The verbalization of rule 3 in Figure 2(A) shows why this would not be appropriate, in general, as the failure may concern a conjunction of literals, each of which may have some answer substitution. Another reason is that the explanatory power of clusters would be affected. For example, in order to describe a credential that fails to satisfy some property, PROTUNE has to combine successful subgoals and failed subgoals, as in:

- c012 is a credential whose issuer is Open University and whose type is ID
- Open University is not a recognized certification authority

What can be removed is blurred information. Recall that the complete definition of blurred literals is generally not included in a filtered policy, which prevents to conclude the negation of an atom \( A \) from the lack of any proof from \( A \). Then blurred literals – roughly speaking – cannot be blamed for failure and can be omitted from a why-not explanation. This mechanism works well in conjunction with variable binding propagation. For example, if the subgoal authenticated(User) of the rule 3 verbalized in Figure 2 fails, then variable User remains free; as a consequences the other subgoals of rule 3, namely subscribed(User, Subscription) and available_for(Resource, User) are blurred (because a complete definition of these atoms would disclose large and partly sensitive parts of the bookstore’s local data) and hence removed from the explanation. As a consequence, when authentication fails, the verbalization of rule 3 collapses to:

- no User is authenticated

like the natural answer that a human would give. More generally, in order to explain correctly blurred policies, PROTUNE needs to distinguish atoms that are surely derivable, those that are certainly failed, and those that might be derivable (what-if queries sometimes contain verbalizations such as “...might be true”). The underlying definitions are the same illustrated in the previous sections. The full formal details lie beyond the scope of this paper; the interested reader is referred to [12], that contains also an extensive discussion of related work beyond the realm of policies.

Remark 7.1: Sophisticated explanation systems, as PROTUNE, may be computationally too expensive for a server, but since servers send policies to their clients, explanations can be constructed on the latter, thereby making this approach feasible. This is yet another advantage of the approach based on policy exchange. Moreover, the filtering and blurring processes automatically prevent explanations from leaking confidential information.

8 IMPLEMENTATION AND INTEGRATION

PROTUNE can be entirely compiled on Java bytecode in order to facilitate installation and encourage users to play with the framework. PROTUNE negotiators can be downloaded as applets from a trusted third party. For this purpose, the inference engine has been implemented in tuProlog [20], a prolog engine that can be compiled on Java bytecode. Network communications and the main flow of negotiation control have been implemented directly in Java. We have integrated PROTUNE into a
Web scenario: our demo\textsuperscript{10} shows the protection of static content as well as parts of dynamic documents by annotating tags with embedded policies (this approach can be regarded as policy-based personalization). Policies are evaluated during the generation of the dynamic page, so that sensitive parts are automatically removed before the page is sent to the client. Moreover, we developed an extension to the web design tool Macromedia Dreamweaver that helps web designers in assigning policies to their web pages by means of a visual interface\textsuperscript{11}. A drawback of the tuProlog implementation is that it is complete only for acyclic policies. We have developed also a tabled XSB implementation that supports positive cycles (as XSB is complete for DATALOG) and improves performance at the cost of adding XSB’s virtual machine and its licensing to the system’s prerequisites.

9 Related Work

This section compares PROTUNE with other policy languages and TN frameworks: Cassandra [5], EPAL [2], KAoS [36], PeerTrust [22], Ponder [17], PSPL [8], Rei [26], RT [28], TPL [25], WSPL [1] and XACML [33]. We focus both on practical issues (e.g., action support, extensibility mechanisms etc.) and on more theoretical criteria, derived from the requirements in [32].

Well-defined semantics. Formal semantics makes the meaning of a policy independent from any particular implementation (a necessary prerequisite for interoperability). PROTUNE and PSPL are based on standard DATALOG, whereas Cassandra, PeerTrust and RT are based on Constraint DATALOG. KAoS relies on Description Logics, whereas Rei combines features of Description Logics, Logic Programming, and Deontic logic. EPAL exploits Predicate logic without quantifiers. Finally, no formalisms underlie Ponder, TPL, WSPL and XACML.

Actions and obligations. During the evaluation of a policy some actions may have to be performed, such as recording information for audit purposes. PROTUNE supports independent actions, i.e. action results should not interfere with each other. Actions are not supported in KAoS, Rei, RT, TPL, PeerTrust, Cassandra, and PSPL. XACML and EPAL allow to specify actions that are collected during policy evaluation and executed before sending a response back to the requester. Ponder supports obligation policies, asserting which actions should be executed if some event happens. It is worth noting that by specifying actions within policies one can to some extent simulate obligation policies, although the flexibility provided by Ponder is of course not met.

Delegation. Delegation is used to cater for temporary transfer of access rights to agents acting on behalf of the grantor (e.g., passing write rights to a printer spooler in order to print a file). Delegation is supported (either natively or by writing suitable rules) by PROTUNE, Cassandra, Ponder, Rei, the extension RT\textsuperscript{D} of RT, and PeerTrust. EPAL, KAoS, PSPL, TPL, WSPL and XACML do not support delegation.

Distributed evaluation. Most of the languages we consider here collect all policy rules in a single place before starting their evaluation: this is how EPAL, KAoS, Ponder, RT and TPL work. Cassandra, Rei, WSPL and XACML, locally evaluate policies, nevertheless they provide some facility to collect or query remote policies. Policies can be collected into a single place if they are freely disclosable or the collecting peer is trusted by all parties, therefore the languages mentioned so far are not adequate for private policies. Protection of sensitive policies requires distributed policy evaluation, which is supported only by PROTUNE, PeerTrust and PSPL.

Evidence and requests. Only PSPL, PeerTrust and PROTUNE support unsigned declarations. Only PROTUNE supports reputation measures. Only PSPL and PROTUNE formulate evidence requests as policies.

Explanations. WSPL and XACML may return error messages such as not\_applicable and indeterminate; optional information is available to explain the error. The other languages—with the exception of PROTUNE and Rei—only return the policy’s decision (either “grant” or “deny”). Only PROTUNE supports second-generation explanation capabilities and how-to queries. A rudimentary form of what-if queries is supported also by Rei obligation policies: the requester can decide whether to complete the obligation by comparing the effects of meeting the obligation (MetEffects) and the effects of not meeting it (NotMetEffects).

Extensibility. Extensibility is a fuzzy concept: almost all languages provide some extension points to let users adapt the framework to their current needs.

As described in Section 5, a standard interface to external packages is one of PROTUNE’s extensibility mechanisms. Second, metapolicies allow to define new provisional predicates. The third powerful extensibility mechanism of PROTUNE is based on libraries of rules (or rule-based ontologies). For example, PROTUNE can simulate RT\textsubscript{0} this way.\textsuperscript{12}

Ponder achieves extensibility through standard object-oriented inheritance. XACML can be extended by (i) defining new datatypes and functions, and (ii) defining new policy combining algorithms. Ontologies are the means to adapt KAoS and Rei to a new application environment. In Cassandra only the constraint domain (and solver) can be modified. Finally, PeerTrust, PSPL, RT and TPL do not provide any extension mechanisms. Note that logic-based language extensions (such a PROTUNE’s) make it possible to share local extensions with other peers without any previous agreement on a common standard, unlike the procedural approach of XACML.

Remark 9.1: Logic-based negotiation and contracting have been considered also in other contexts like service composition [31] and interoperability verification [3].

\textsuperscript{10} http://policy.iss.uni-hannover.de/
\textsuperscript{11} http://skydev.iss.uni-hannover.de/gf/project/protune/wiki/admin/?pagename=Integration+with+Dreamweaver

10 Evaluation

The comparison with related works shows that Protune excels w.r.t. all the adopted criteria, with the exception of actions, that are supported with some limitations for a better tradeoff with declarativeness. Here we deepen the discussion on Protune’s language restrictions, and discuss the feasibility of Protune’s approach and its usability.

10.1 Language expressiveness

To what extent do Protune's syntactic restrictions affect its expressive power? Note that access control and credential release policies are essentially mappings from a relational context (e.g., state predicates and incoming evidence) to a set of decisions, that may be represented in a tabular form, therefore policies are relational queries. Accordingly, we can apply the expressiveness results for rule-based query languages to Protune.

It is known that stratified function-free logic programs with negation express all the queries in PTIME, provided that the database is equipped with suitable relations first, last, and succ that define a total ordering over the database’s constants; the same holds even if negation is applied only to database (extensional) relations (see [18] for more details and an extensive survey of relevant results). The three predicates defining the total ordering are very close to the standard term comparison predicates of Prolog (such as @<); by supporting these predicates in Protune we cover all the PTIME-computable policies that depend only on the state.

However, we deliberately not cover all possible PTIME policies: as we pointed out in a previous section, we comply with the requirements laid out in [32] and restrict Protune’s language so that negation cannot be applied to incoming credentials and declarations (as a result, Protune is allowed to express only policies that are monotonic w.r.t. the available evidence). An interesting open question is whether Protune’s language expresses all the PTIME-computable policies that are monotonic w.r.t. evidence. Clearly, Protune can express all positive boolean combinations of credentials and declarations (i.e. all compound evidence requests that can be built with connectives ∧ and ∨).

Specific requirements for privacy policy language constructs can be used to evaluate Protune from a slightly different perspective. A recent paper [21] shows that Protune is currently one of the most complete and advanced approaches. The analysis of [21] is summarized in Table 2. According to that analysis, the only currently missing feature is a direct support to usage control, that is, constraints on the use of information after its disclosure. There is however some preliminary work on how to encode such constraints in a rule-based language like Protune, see [39] for more details. In Table 2, the classification type refers to the means for defining data semantics and relate them to policies; “taxonomy” denotes elementary inclusions of atomic classes. Minimal disclosure refers to the ability of minimizing the set of disclosed information and its sensitivity.

10.2 Technological feasibility (w.r.t. performance)

In order to evaluate the performance of Protune we first focus on its efficiency in carrying out negotiations. To this aim we measured the duration of each step of the negotiation algorithm with a profiling tool we built on top of the log4j utility. The experiments are all based on the default negotiation strategy of Protune that maximizes the release of provably relevant evidence.

The experimental evaluation of a TN frameworks has to overcome two obstacles. First, there are no large bodies of publicly accessible, formalized access-control policies. Since TN frameworks are not yet largely adopted, the list of available release policies is even shorter.

Second, the policies currently enforced by web applications are typically small. However, the increasing availability of sophisticated policy frameworks like those mentioned in the previous section may encourage the adoption of more and more complex policies in the future, therefore it is interesting to estimate the scalability of TN frameworks to large policies, consisting of hundreds or thousands of rules.

For these reasons, we developed a module to automatically generate policies according to the following input parameters: number of negotiation steps, number of rules per predicate, number of literals per rule body.14 Tested on policies inspired by a realistic scenario (a digital library whose private resources can be accessed either by subscription or by an ample range of payment modalities; authentication modalities cover both traditional logins and credential-based access, including

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Protune</th>
<th>Rei</th>
<th>Trust-X</th>
<th>KeyNote</th>
<th>Ponder</th>
<th>Appel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource classification</td>
<td>with ontology</td>
<td>none</td>
<td>taxonomy</td>
<td>P3P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access control</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimal disclosure</td>
<td>yes</td>
<td>no</td>
<td>partial</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Policy protection</td>
<td>yes</td>
<td>yes</td>
<td>partial</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Push control</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Usage control</td>
<td>no(*)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

(*) work in progress; first results in [39]

TABLE 2

A comparison of some major privacy languages

14. The components described so far can be freely downloaded from http://skydev.l3s.uni-hannover.de/gf/project/protune/wiki/?pagename=Evaluation.
credentials released by business partners) the system’s performance has been in the order of 10-100 ms and hence fully satisfactory. Then we tried the system on artificial policies that create large trees of dependencies: the root is the requested resource; its children (i.e., the first level of the tree) are the credentials needed to get the resource; the second level is the set of counter-requests of the client that are needed to unlock the credentials in the first level, and so on. The artificial aspects in such examples consist in the exponential number of credentials involved (corresponding to tree nodes) and the chains of dependencies between them (usually shorter and sparser in the scenarios inspired by real-world applications). Fig. 3 reports the results of these experiments, some of which are interrupted after 150 sec. The frontier of terminating runs touches examples with thousands of interrelated credentials, which explains the high reasoning time. Given the size of the examples involved, we conclude that this technology can scale up to policies and portfolios of credentials and declarations significantly larger than those applied today. Consider that there is space for improvements, as none of the possible optimization techniques (such as caching repeated requests and their filtered policies) has been applied in the above experiments. We do not have experimental figures for the approaches based on requesting the explicit list of all alternative sets of credentials (rather than a policy that represents them). However we can note that with the artificial policy with 4 definitions per predicate, 4 literals per body, and 3 negotiation steps, a single message should contain about 16 billion sets, each composed by 64 credentials, and that the network time for transmitting a message of this size is expected to take significantly longer than 3.5 seconds.

A performance evaluation of the explanation facility PROTUNE has been done on a sample of 12 tests, including both policies inspired by application scenarios and artificial policies.

Essentially, the processing time per page grows up linearly w.r.t. the the number of rules in the policy, in particular for a policy with 20 rules the processing time is 40 ms per page whereas for a policy with 110 rules it is 310 ms per page. These results refer to an implementation based on tuProlog [20], the same Java-based Prolog engine adopted for implementing PROTUNE’s core. We mention that there exists also a stand-alone implementation of PROTUNE, available at http://cs.na.infn.it/rewerse/demos/protune-x/demo-protune-x.html, that runs on XSB-Prolog, a Prolog engine equipped with memoizing methods (tabling). Tabling significantly improves the performance of PROTUNE; one of the main reasons is the ability of avoiding repeated proofs of the same literal, that are quite frequent in this application. The performance of the XSB implementation is typically over 10 time faster than the tuProlog implementation. Moreover, the XSB implementation supports cyclic policies.

### 10.3 Tackling usability issues

In many application contexts (such as social networks and location-sharing facilities), the effectiveness of security and privacy enhancing techniques is strongly influenced by the ability of end users to specify their policies and understand the policies of the systems they interact with. Such users cannot be assumed to have any special competence or training. The explanation facility PROTUNEx is meant to address the usability issue by automating the creation of context-specific documentation, and by providing a diagnosis and test tool for policy verification, as explained in a previous section.

A possible way of evaluating the quality of the explanations produced by PROTUNEx is by matching its features against the principles of second-generation explanation facilities [37], [24]. A second generation explanation system should provide:

1. methods for asking for explanations;
2. methods for breaking up proofs into manageable pieces;
3. methods and user interfaces for proof/explanation navigation tailored to the user’s problem solving method, rather than to the engine’s;
4. methods for removing irrelevant information;
5. methods for justifying conflicting answers.

Concerning point 1, PROTUNEx supports a rich set of queries (how-to, why/why-not, what-if) and in PROTUNEx explanations it is possible to navigate from one type of query to another (e.g., when traversing a detail link from a failed node to a successful node).

Points 2 and 3 are tackled simultaneously by the sophisticated hypertext structure of PROTUNEx, that allows to jump across different proofs, see nonlocal information such as the outcome of a proof, etc., in a way that is oriented to users and does not follow the engine’s behavior. The clustering heuristics addresses readability. Point 4 is addressed by exploiting blurring as described in Sec. 7. Point 5 is currently not concerning PROTUNEx because PROTEUE access control and release policies cannot express any contradictions. To the best of our knowledge, PROTUNEx is the only second-generation explanation facility for TN frameworks.

Another fundamental evaluation method is by systematic user studies, where users are confronted with policy usage and comprehension tasks; different users groups are provided with different tools, in order to compare their relative effectiveness in helping users to carry out their tasks quickly and correctly. This is a research line in itself that will be the subject of future work.

### 11 Discussion and conclusions

Rule-based policy languages have been regarded as appealing policy languages for a long time (cf. [13]); in the new area of TN they show their advantages even more clearly, and more sophisticated rule-based techniques profitably come into play. A rule-based policy can be treated like a knowledge base: it can be shared and used
for a variety of reasoning tasks. For example, by having peers exchange (a filtered version of) their policies, PROTUNE can express in a compact way complex requests so that: (i) user agents can see at once multiple alternative ways of fulfilling a policy and select those that more closely match the user’s privacy preferences; (ii) this can be done without paying the price of combinatorial explosions of compound requests, which may decrease performance by increasing message size and/or number; (iii) the disclosed part of a policy can be used to construct sophisticated contextualized explanations on the clients, without overloading servers.

Without a simple formal and declarative language like function-free logic programming, it would be extremely difficult to reach the above goals that require a coherent integration of several reasoning tasks:

- deduction, for access control;
- abduction, to select relevant evidence from a portfolio, using the policy as a request;
- filtering, to protect the sensitive parts of policies and avoid the transmission of irrelevant information;
- explanations, that help users in understanding and verifying policies and negotiation outcomes.

Formal semantics is essential to guarantee both coherent outputs across the above tasks and interoperability via rule exchange. Moreover, logical approaches enable interesting functionalities, such as policy comparison tools for validation and privacy compliance checking [10].

Among logical approaches, rule technology is currently more mature than description logics technology with respect to the particular combination of functionalities and reasoning problems needed in TN. More specifically, abduction and explanations for description logics are relatively less developed. There are only a few, mostly recent works on abduction [16], [30], [6] and explanations [14], [19], and even less implementations.

PROTUNE has specific advanced features that address some fundamental requirements on policy languages and make it one of the most complete TN frameworks. Such features include an interface for legacy systems and data, a metalanguage that can be used to instantiate the framework in new application domains and drive its dynamic behavior with a moderate deployment effort, a powerful extension mechanism based on the metalanguage and on rule-based ontologies, and a unique second generation explanation facility. The explanation facility addresses both usability and deployment costs reduction (by facilitating policy diagnosis and testing with context-dependent and what-if queries). The extensive analysis reported in this paper shows that PROTUNE is currently one of the most complete and general TN frameworks. The practical feasibility of PROTUNE’s technological choices is supported by experimental evidence.

PROTUNE has also some limitations that call for extensions such as usage control constructs that might be based on deontic extensions like those adopted in [35]. Our technologies could be extended to support the well-founded semantics and negated evidence; however, such extensions should be very carefully motivated to address the objections raised in the area of security. Finally, it would be interesting to extend PROTUNE with policy negotiation methods to increase the set of successful transaction in a principled way.

REFERENCES
