PROTUNE: A Rule-based PROvisional TrUst NEgotiation Framework

P. A. Bonatti, J. L. De Coi, D. Olmedilla, L. Sauro

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 such as the Web, 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 framework For the first time, we give an overall picture of the framework, describe its implementation, provide an evaluation along several dimensions, and extensively compare it with competing frameworks. Moreover, we point out which features of declarative rule-based systems play a crucial role in tackling the many issues raised by TN scenarios.

Index Terms—Protune, rule-driven negotiation, policy rules, logic programming, policy filtering, explanations, privacy, trust.

1 INTRODUCTION

The protection of sensitive information and other sensitive resources plays a crucial role in raising the level of trust in web resources 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[1] and Virgin's use of Flickr pictures[2] 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 [33], [35], [41] 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. Therefore, 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). In summary, all the interacting parties eventually enjoy more guarantees, and less sensitive information needs to be disclosed.

In this paper we give an overall picture of the PROvisional TrUst NEgotiation framework PROTUNE [13]. PROTUNE is rule-based; we are going to illustrate the advantages that arise from a rule-based approach in terms of deployment efforts, user friendship, communication efficiency, and interoperability. PROTUNE intersects the focus of this special issue in several ways: It makes use of meta-rules and meta-annotations to control its behavior, and as an extensibility mechanism. PROTUNE agents exchange rules to interoperate and make decisions related to security and privacy. Moreover, PROTUNE's policies can be regarded as semantic markup: logical axioms that constrain behavior and usage of web resources. As such, PROTUNE can be regarded as a semantic web technique.

This paper is not about proving sophisticated new theorems. It is rather meant to identify methods and techniques developed in the areas of logic programming and rule-based systems that can be profitably integrated to meet the needs of trust negotiation frameworks. For this purpose, a complete view of the system and the interplay of its different components is crucial. We give such a complete view for the first time in this paper. It will turn out that policies have to be treated like knowledge bases: a single policy specification must be used in a variety of ways, each solving a different reasoning problem. Another purpose of this paper is discussing pros and cons of the strategic choices made in PROTUNE w.r.t. alternative approaches.

The paper is organized as follows. We start with a standard motivating scenario (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. Then we

1. http://www.washingtonpost.com/wp-dyn/content/article/2007/11/29/AR2007112902503.html?hpid=topnews

2. http://www.smh.com.au/news/technology/virgin-sued-for-using-teens-photo/2007/09/21/1189881735928.html

devote two sections to an extensive comparison with other policy languages and frameworks, and to an evaluation of PROTUNE according to several 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

Bob's birthday is next week and Alice plans to use today's lunch break for buying on-line a 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 say, belonging to the Better Business Bureau), 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 some important aspects. 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 (see also the discussion on weak and strong evidence below). Third, the need of collecting information conflicts with usability: in the absence of any automated support, 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) the policies of the client and of the server, respectively. 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.

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 receive by an on-line community; reputation values may be used by a policy to determine whether a peer is trusted enough for a given type of transaction.

TN agents may request evidence according to different strategies. 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 requests that she considers preferrable 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). This approach, however, has an important implication on message size, because the list of all the alternative sets of information items that fulfil 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 multiple alternative and compound information requests more efficiently and succinctly, 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 explicit credential set listing, is that it gives no information about the policy's structure, thereby preventing accurate automated explanations.

With this approach, 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 $\texttt{allow}(buy(book123))$ 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 $\texttt{allow}(release(C_A))$ 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 $\texttt{allow}(release(C_B))$ from its local policy, therefore C_B is sent to A;
7) A can now prove $\texttt{allow}(release(C_A))$ from its local policy P_A *and* C_B, therefore C_A is sent to B;
8) B can now prove $\texttt{allow}(buy(book123))$ from its local policy *and* C_A, and the purchase transaction is successfully completed.

We will see later that PROTUNE supports the widest range of (weak and strong) evidence varieties, and that agents communicate by exchanging parts of their policies and local information as in the above example, with the proviso that these messages should not leak any confidential information. By releasing their policies, PROTUNE-enabled servers allow PROTUNE-enabled clients to construct detailed, human-readable explanations of the policy and of negotiation outcomes, for a variety of purposes (see Section 7) , relieving the server from an expensive task.

Remark 3.1: Summarizing, the policy exchange techniques adopted by PROTUNE have the following advantages: (i) enhance privacy by giving more options to users, (ii) potentially reduce negotiation length and message size, (iii) enable advanced documentation and feedback services without overloading servers. ∎

4 PROTUNE'S ARCHITECTURE

As illustrated in the above section, PROTUNE agents alternatively act as clients (that request a web resource or a piece of evidence) and servers (that release a web resource or a piece of evidence). In this sense, Figure 1 shows the internal structure of *all* the peers involved in a negotiation: the meaning of "server" and "client" in the figure should be relativized to the role played by any peer in a particular step of a negotiation.

The PROTUNE engine, the execution handler and the network interface components constitute the core of PROTUNE's framework; they are identical across all PROTUNE peers. We have been experimenting with two inference engines, based on different Prolog technology.

The action selection component implements the *negotiation strategy* of a peer. Different strategies may be adopted by different peers. Currently PROTUNE provides a cooperative default strategy, that discloses at each step all the releasable information that appears to be relevant to negotiation success. So, for example, even if Alice's policy allowed to freely release her ID certificates, Alice's agent would not disclose any of them unless they are needed in some proof of $\texttt{allow}(buy(book123))$.

The execution handler may interact with the rest of the system (e.g., relational DBMS, local file system, etc.) in order to integrate PROTUNE with legacy systems and let policies make use of external data.

Figure 1 mentions also explanations (why/why-not, how-to, what-if) that will be introduced and discussed in a later section. Basically, these kinds of queries are helpful for improving usability and supporting policy validation.

5 PROTUNE'S LANGUAGE

Rule-based languages are appealing policy languages because users spontaneously tend to formulate policies as rules. Moreover, rule-based languages are expressive and high-level, and enjoy simple declarative semantics (the advantages of these properties will be discussed later).

PROTUNE's language [13] 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:

$$\begin{aligned}&\texttt{allow(buy(Resource))} \leftarrow \\ &\quad \texttt{credential(C)},\ \texttt{valid\_credit\_card(C)},\\ &\quad \texttt{accepted\_credit\_card(C)}.\end{aligned} \tag{1}$$

$$\begin{aligned}&\texttt{valid\_credit\_card(C)} \leftarrow \\ &\quad \texttt{credit\_card(C)},\ \texttt{C.expiration} : \texttt{Exp},\\ &\quad \texttt{date(Today)},\ \texttt{Exp} > \texttt{Today}.\end{aligned} \tag{2}$$

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

$$\begin{aligned}&\texttt{allow(release(my\_credit\_card))} \leftarrow \\ &\quad \texttt{credential(C)},\ \texttt{C} : \texttt{issuer:'BBB'},\\ &\quad \texttt{C.expiration} : \texttt{Exp},\ \texttt{date(Today)},\ \texttt{Exp} > \texttt{Today}.\end{aligned} \tag{3}$$

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 $\texttt{allow(buy(Resource))}$ is treated like the function-free atom $\texttt{allow(buy, Resource)}$.[3]

Since digital credentials and unsigned declarations (that are frequently modeled as html forms) are semistructured objects, PROTUNE is enhanced with a FLORA-like object-oriented syntax. One can express by $X.\texttt{attr}{:}v$ the fact that X has an attribute $\texttt{attr}$ with value v, therefore in the previous example $C.\texttt{expiration}{:}Exp$ is an O.O. expression meaning that Exp is the value of C's attribute $\texttt{expiration}$. However, an O.O. expression is only an abbreviation for standard first-order syntax, actually $X.\texttt{attr}{:}v$ abbreviates the standard atom $\texttt{attr}(X, v)$. This representation allows multi-valued attributes, as in $X.\texttt{sibling}{:}\texttt{john}$, $X.\texttt{sibling}{:}\texttt{mary}$. This attribute semantics is compatible with semantic web standards such as RDF and OWL (in particular $X.\texttt{attr}{:}v$ corresponds to an RDF triple). We support attribute concatenation, like $X.\texttt{expiration.year}{:}2010$. In the body, this is expanded to $\texttt{expiration}(X, V), \texttt{year}(V, 2010)$, where V is a fresh variable. Concatenation is forbidden in the head, because of the technical difficulties related to the existential quantification of V (e.g., skolemization may lead to undecidability). Similarly, an atom $p(X.\texttt{attr})$ would be replaced

3. Function support might be further extended by analogy with ω-restricted programs [38], without affecting decidability.

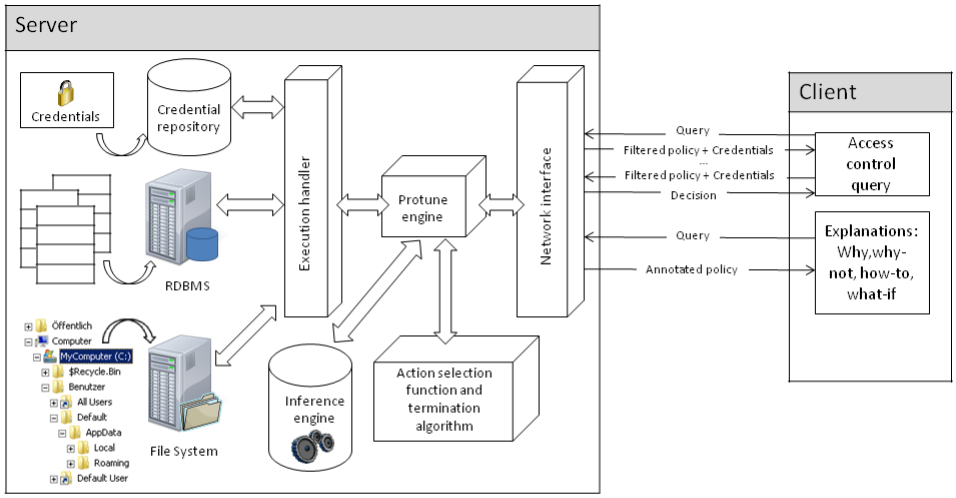

Fig. 1. Protune's architecture

by $p(Z)$ where Z is a fresh variable, and the rule body should be extended with $\texttt{attr}(X, Z)$.

Remark 5.1: By adopting a standard function-free logic programming language as internal format we obtain two benefits: (i) we can directly adopt well-established and understood semantics, and moreover (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 scenarios. ∎

Rules are interpreted according to the *stable model semantics* [24], a two-valued[4] semantics where negation faithfully models underivability. The stable model semantics is recalled in the Appendix.

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.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). The reason of this restriction can be traced back to [33] and caused this restriction to be systematically adopted thereafter. In that founding paper, it is convincingly argued that it is impossible to check reliably whether a peer does *not* have a certain digital token, therefore policies should not be allowed to make this kind of negative tests.

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

- **Decision Predicates**. These predicates represent policy outcomes, such as authorizations, and hence trigger appropriate behaviors in the framework when they can be derived. Currently, the supported decision predicates are `allow`, which is queried for access control decisions, and `sign`, which is used to issue statements signed by the principal owning the policy (in order to define dynamic certificate release policies).
- **Logical Predicates**. Comprise the *abbreviation predicates* that define auxiliary concepts, such as $\texttt{valid\_credit\_card}(\texttt{C})$, and *state-query* predicates that read the current state without modifying it, for example $\texttt{date}(\texttt{Today})$.[5]
- **Constraint Predicates**. Comprise the usual arithmetic predicates (e.g. $\texttt{Exp} > \texttt{Today}$).
- **Provisional Predicates**. These predicates may change their value during negotiation as a result of an action carried out by some of the peers. PROTUNE has a few built-in provisional predicates of common use, including $\texttt{credential}(\texttt{C})$ and $\texttt{declaration}(\texttt{D})$ 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 (that will be defined later).

Remark 5.2: A flexible support of state query predicates is of paramount importance in a policy language. Most policies make decisions based on the data encoded in the information system they protect, for example access to a digital library is typically regulated by taking into account the database of customers, their subscriptions, and other 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 introduced in the logic-based mediators of the Hermes project [37], and later inherited by Impact's logic-

4. Three-valued semantics (such as the well-founded semantics) are not popular in security. The problem is how to treat undefined authorizations properly. If access is granted, then obviously security may be violated (there is no uncontroversial proof that it should be granted); however, denials may cause denial of service problems (as there is no uncontroversial proof that the authorization should be denied, either). Moreover – roughly speaking – in the well-founded semantics all atoms that depend on an undefined atom are undefined, too, therefore it is impossible to define standard default policies such as *open* or *closed* policies (that transform undefined authorizations into grants or denials).

5. By *state* we conceptually mean a set of ground logical atoms that may change across negotiation steps, as a result of message exchanges and actions. More details are given when negotiations are defined.

based agents [36]. The built-in provisional predicate $in/2$ provides a uniform interface to external packages (possibly including databases and other data sources) that can be queried with atoms of the form $in([X_1, \ldots, X_n], packageName : function(Arg_1, \ldots, Arg_n))$ where the variable list $[X_1, \ldots, X_n]$ ranges over the fields of the set of objects returned by the code call $packageName : function(Arg_1, \ldots, Arg_n))$. 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 whose attribute $D = e$. In practice, the implementation needs suitable wrappers for the packages and appropriate caching techniques. Currently, PROTUNE has wrappers for file systems, RDF stores, LDAP servers, and relational database systems via the JDBC interface.

Last but not least, `in` can be used to gather reputation evidence by having a wrapper read scores from internal repositories or external web sites. Depending on the application context, the wrapper may invoke a suitable web service, if available, or "scrape" the required information, i.e. extract reputation values by parsing given areas of a web page. More details can be found in [11].

We can assume that `in` is the only state-query predicate; 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 can be modelled as the (possibly infinite) set of all true ground facts $\texttt{in}(\alpha, \beta)$, which are implicitly added to the policy – this requires no changes to the stable model semantics.

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 as prescribed by the X.509 standard; if χ passes the validity check, then it is converted into a set of facts $attr(c_\chi, value)$ that represent its attributes (as discussed before in relation to O.O. syntax), where c_χ is a constant that represents the entire credential. These facts and $\texttt{credential}(c_\chi)$ 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 declaration (which is a semi-structured object such as a web form) is encoded as a set of ground facts that are added to the negotiation state; however, validity check are not needed for unsigned declarations.

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

```
authenticated←declaration(D),
      valid_login_data(D.username, D.password)
```

where `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).

By means of another built-in provisional predicate, `challenge(K)`, the peer can be asked to prove to be the owner of the private key associated to the public key K, through a standard challenge procedure.[6]

This small set of primitive predicates let peers express and exchange the wide range of information needed for trust negotiation and check ownership of digitally encrypted data.

Remark 5.3: In order to interoperate, PROTUNE agents need to share only the procedural understanding of the few built-in predicates, such as `allow`, `credential` and `declaration`, and the meaning of rules. 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. ∎

meta-attribute	Domain	Range
`type`	atoms	*provisional*, *logical*
`actor`	provisional atoms	*self*, *peer*
`ontology`	provisional atoms	URI
`execution`	provisional atoms	*immediate*, *deferred*
`explanation`	atoms, literals	string expression
`sensitivity`	atoms, rules	*private*, *public*
`blurred`	literals	*true*, *false*

TABLE 1
Table of meta-attributes

The metalanguage

PROTUNE needs a metalanguage for several purposes: (i) as an extensibility mechanism that supports the definition of application-dependent provisional predicates and explanations; (ii) as a means of annotating the confidential parts of policies and auxiliary definitions. 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, check groundness, call an object-level goal G against the current state ($hold(G)$). Metarules can use a set of reserved attributes in order to describe properties of the atoms occurring in the object policy (see table 1).

Remark 5.4: The first formalization of the framework mentioned some reserved meta-attributes which are not supported in this way anymore. In particular, provisional related attributes such as `action`, `cost` or `expected_outcome` were used to link user defined actions to the code that implements them and to represent some meta-information (such as costs and expected outcomes of a certain action) useful for negotiation strategies. All this information is now assumed to be represented in the ontology associated to an action. ∎

For example a piece of metapolicy could be:

$log(X) \rightarrow \texttt{type} : provisional.$
$log(X) \rightarrow \texttt{ontology} :' http : //www.protune.com/logAction'.$
$log(X) \rightarrow \texttt{actor} : self.$

Here "→" connects a metaterm to its metaproperties (we use it in place of '.' in order to disambiguate the grammar

6. The challenge procedure consists in encoding a random number with a peer's public key and asking the peer to decrypt it. Only the owner of the corresponding private key can perform this operation.

and distinguish meta-atoms from complex terms). The meta-attribute `type` establishes the type of a user defined atom, i.e. logical or provisional. It is assumed that the type an atom depends only on its predicate symbol and arity.

For provisional atoms the meta-attribute `ontology` univocally determines the action to be performed by identifying it by means of a URI. Finally, if an action p must be executed locally we assert $p \rightarrow \texttt{actor} : self$, otherwise $p \rightarrow \texttt{actor} : peer$.

Another meta-attribute which applies to provisional predicates is `execution`: in general an action may require some input parameter(s), therefore the PROTUNE framework must make sure that an 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.

$$log(X) \rightarrow execution : immediate \leftarrow ground(X).$$

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

Furthermore, atoms or ever entire 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 detect 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.

Sensitivity is not the only reason to withdraw part of a predicate's definition. Size (hence efficiency) is another reason: 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 set of rules and facts that unify with a given literal might not be completely included in the policy sent to another peer (the meta-facts with the blurred attribute are included in the disclosed policy). 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 $(L' \rightarrow \texttt{blurred:true}) \leftarrow (L \rightarrow \texttt{blurred:true})$ and $(L \rightarrow \texttt{blurred:false}) \leftarrow (L' \rightarrow \texttt{blurred:false})$, for all literals L and L' such that L' is more general than L. Moreover we tacitly assume that metapolicies are coherent, in the sense that no pair of metafacts $L \rightarrow \texttt{blurred:true}$ and $L \rightarrow \texttt{blurred:false}$ can be simultaneously derived.

Finally, the meta-attribute `explanation` is used to specify how to render a specific atom or literal in natural language while generating documentation or explanations.

Sections 6.2 and 7 will discuss further details of meta-attributes `sensitivity`, `blurred` and `explanation`. In the following a metapolicy will be considered part of the policy KB_i of a peer. Object rules can always be distinguished from metarules because the latter always have heads of the form $A \rightarrow \texttt{meta\_attribute} : Val$.

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[7] i is associated 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 `in`;
- $Prov_i(s)$, 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; $Prov_i(s)$ may include an encoding of some credentials and declarations from other peers;
- $Ev_i(s)$, 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 $Prov_i$ and 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 contex 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: $Prov_i(s) \subseteq Prov_i(s')$ and $Ev_i(s) \subseteq Ev_i(s')$.

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 systematically omit the parameter s from the programs KB_i), 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 by 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. ∎

7. So far we have talked about 2-peer negotiations, however the framework can be extended to 3 or more peers.

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 example, in a negotiation between 2 peers, `release`(e) denotes the action of releasing a credential or declaration e to the other peer. All these 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 $P_i(s) = KB_i \cup Prov_i(s)$. 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 `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:[8]

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

The changes to the above logic programs are determined by the actions executed by each peer. In turn, these actions (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 to decide the next action.

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 `allow`(Req), where Req is the initial request received from a client. Moreover, each peer maintains in $OG_i(s)$ the set of ground atoms `allow`(`release`(e)) such that e is a piece of evidence in $Ev_i(s)$ that i's strategy has selected as a candidate for release, but at step s the action `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:

- Is it at all possible to derive G, given enough further evidence?
- Which are the local actions that may help in deriving G?
- 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.[9] We need an auxiliary definition first: an atom t.`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.`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 Δ of G from $P_i(s)$ whose last goal G_Δ contains only (i) provisional predicates and semi-provisional atoms $t.attr{:}u$ w.r.t. G_Δ; and (ii) negative literals. Moreover, there must be a grounding substitution θ such that $G_\Delta\theta$ is coherent[10] and all the 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_Δ, and a substitution σ such that $G_\Delta\sigma \subseteq E$.

Proof sketch: Let M be the unique stable model of $P_i(s) \cup E$. If $M \not\models G$, then G belongs to its Gelfond-Lifschitz reduct $\mathsf{GL}(P_i(s) \cup E, M)$, therefore there exists a successful SLD derivation Δ of G from $\mathsf{GL}(P_i(s) \cup E, M)$. Obtain an SLD derivation Δ' from Δ 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 Δ of type 1 of G in context (i, s), with final goal G_Δ and substitution θ. Let E be the set of positive literals in $G_\Delta\theta$. It can be easily verified that by removing all negative literals from Δ we obtain an SLD derivation Δ' of G from $\mathsf{GL}(P_i(s) \cup E, M)$. 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 ad predicates to be disclosed to other peers. We need two preliminary definitions:

8. 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 $d.attr{:}val$ 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 $Prov_i(s)$.

9. We assume the reader to be familiar with the basics of logic programming and SLD derivations in particular. In this context, programs may contain negative literals that will never be rewritten by the resolution rule and will be accumulated during the derivation.

10. That is, it contains no pair of complementary literals.

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

1) eliminating all rules r such that r.sensitivity:private belongs to the unique stable model of $P_i(s)$; call them *private rules*;
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 Δ of G from $PF_i(s)$ satisfying the following conditions:

1) at each step of the derivation, the selected atom[11] is not blurred;
2) the last goal G_Δ contains only (i) provisional predicates and semi-provisional atoms $t.attr{:}u$ w.r.t. G_Δ; (ii) negative literals; (iii) blurred literals.
3) there must be a grounding substitution θ 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_Δ the set of rules and facts applied in Δ.

Note that, by definition, P_Δ contains no private rules or predicates. Then P_Δ can be disclosed to other peers to (partially) describe which evidence is needed to fulfil the policy (and prove the auhorization G).

Now, roughly speaking, if another peer j can prove G using P_Δ and a subset E of its portfolio $Ev_j(s)$, 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 $E \subseteq Ev_j(s)$, if there exists a successful SLD derivation Δ 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 Δ' from Δ by removing the applications of resolution to facts in E. Since P_Δ contains no private rules, nor any rule with a private predicate in the head, we have that Δ' 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 Δ'' from Δ 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 Δ'' 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 $P_\Delta \cup Ev_j(s)$ end up in a nonempty set of blurred literals, 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 $\neg A \rightarrow$ blurred:false in disclosed policies when A matches no rules in $P_i(s)$. Then, if the policy is safe, Theorem 6.2 can be extended to all Δ 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 8), 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).

A negotiation strategy may opt for releasing the union of all P_Δ, for all the derivations of type 2 or a subset thereof. By releasing the union of all such P_Δ simultaneously, one obtains the full potential of privacy preservation of PROTUNE.

Note that the compilation of private policies (in the definition of $PF_i(s)$) may hide the dependencies between a goal and the evidence it depends on – in extreme cases, all the policy is private and P_Δ is empty, so it gives no clue on how to fulfil the policy. This is unavoidable with confidential policies. As a consequence, it may be useful to release evidence proactively, without any explicit evidence of its relevance.

A study of this problem lies beyond the scope of this paper. We refer the interested reader to the solution we gave in [7] where we defined simple guidelines on how to program strategies so as to guarantee that negotiations succeed whenever the policies allow it. We proved formally that this is always possible when the peers give top priority to negotiation success (e.g. servers really want to sell and clients really want to buy).

7 DOCUMENTATION AND EXPLANATIONS

As the machine-understandable 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.

11. Recall that the selected atom is the atom that is unified with a rule head and replaced with the rule's body.

For the reasons discussed above, we augmented PROTUNE with an explanation facility called PROTUNEX [14] in order to support the automated creation of high-quality documentation and contextualized explanations.[12] 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:

`auth(X)→verbalization:[X,' is authenticated'].`

Unlike other recent approaches in the semantic web arena, such as [31], [28] PROTUNEX does not simply output a single proof tree or a counterexample. PROTUNEX rather provides a hypertext (a set of interlinked web pages) 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 hypertexts constructed by PROTUNEX have a node for each atomic condition considered in some proof attempt. A single atom occurrence A in a policy 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 atom instance are collapsed.[13] 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 semistructured objects such as credentials and declarations (as opposed to using the internal identifier of these structures). Roughly speaking, for each atom `credential(C)` or `declaration(C)`, PROTUNEX collects all the attribute atoms `C.att:`V sharing the same term `C`, and outputs a combined description of all these atoms, as in:

- c012 is a credential whose issuer is UK_government and 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 occuring 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 PROTUNEX 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, PROTUNEX 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 *un*derivability 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, PROTUNEX has to combine successful subgoals and failed subgoals, as in:

- c012 is a credential whose issuer is Open_University and whose type is ID
  but
- Open_University is not a recognized certification authority for credentials of type ID.

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

12. A live demo is available on http://cs.na.infn.it/rewerse/demos/protune-x/demo-protune-x.html.

13. In XSB Prolog this mechanism is known as *tabling*.

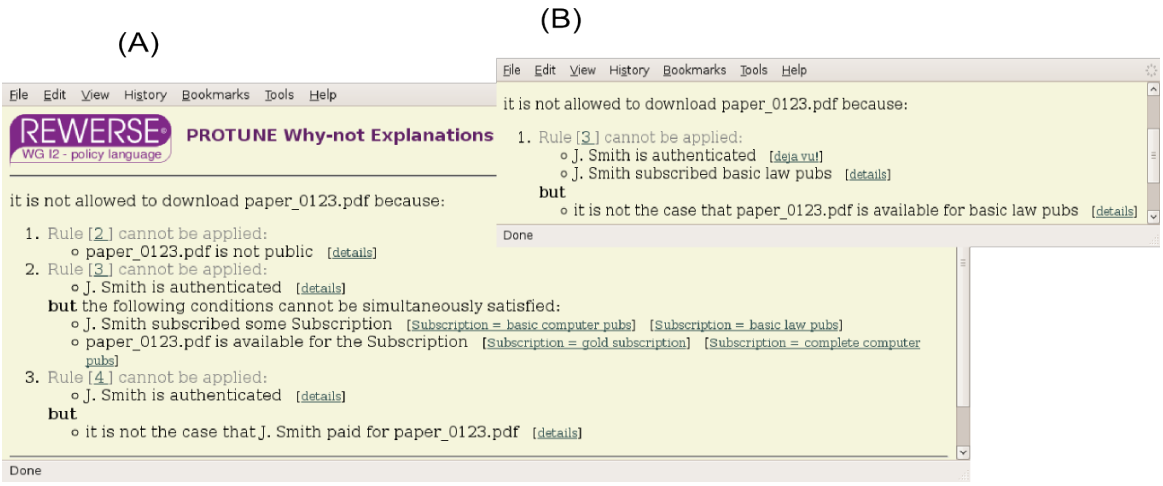

Fig. 2. Two PROTUNEX screenshots.

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 we would expect a human to give.

More generally, in order to explain correctly blurred policies, PROTUNEX 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 – including the mathematical model underlying PROTUNEX – lie beyond the scope of this paper; the interested reader is referred to [14], that contains also an extensive discussion of related work beyond the realm of policies. For the purposes of this paper, we only point out that:

Remark 7.1: Sophisticated explanation systems, as PROTUNEX, make use of reasoning tasks that involve both deduction and abduction-like reasoning. This way, the computational cost of explanations might be too high for a server, but since servers can send policies to their clients, the computational burden can be shifted on the latter, thereby making this approach feasible. This is yet another advantage of the negotiation approach based on policy exchange. The filtering and blurring processes remove all sensitive information from the policy, so the explanation process (that applies to the filtered policy) cannot possibly leak any confidential information. ∎

8 IMPLEMENTATION AND INTEGRATION

PROTUNE can be entirely complied on Java bytecode in order to facilitate installation and encourage users to play with the framework. For example, PROTUNE negotiators can be downloaded as applets from a trusted third party, without requiring any uncommon prerequisite on the clients. For this purpose, the inference engine has been implemented in tuProlog [21], a prolog engine that can be compiled on Java bytecode. A more articulated discussion of the advantages of this technological solution can be found in [21]. Network communications and the main flow of control for negotiations have been implemented directly in Java.

We have integrated PROTUNE into a Web scenario: our demo[14] shows that the PROTUNE engine can automatically perform non-trivial actions (like credential negotiation and reasoning) and make decisions on the basis of expressive policies. In addition to the protection of static content, it is also possible to protect parts of dynamic documents by annotating tags with embedded policies (this approach can be regarded as policy-based personalization). 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[15]. In this way, web developers can specify which policies (possibly involving negotiations) the requester must satisfy in order to access (parts of) a document. The policy is evaluated during the generation of the dynamic page, so that sensitive parts are automatically removed before the page is sent to the client.

9 RELATED WORK

In this section we compare PROTUNE with other popular policy languages and TN frameworks, namely Cassandra [8], EPAL [4], [5], KAoS [39], PeerTrust [23], Ponder [18], PSPL [10], Rei [27], *RT* [29], TPL [26], WSPL [3] and XACML [30], [34].

In our comparison we focus both on criteria having a practical relevance (e.g., action support, extensibility

14. http://policy.l3s.uni-hannover.de/
15. Cf. http://skydev.l3s.uni-hannover.de/gf/project/protune/wiki/admin/?pagename=Integration+with+Dreamweaver

mechanisms etc.) and on more theoretical criteria, derived from the requirements in [33].

Well-defined semantics. Formal semantics makes the meaning of a policy independent from the particular implementation of a policy framework (a necessary prerequisite for interoperability). Among the formal languages, PROTUNE and PSPL are based on standard function-free Logic Programming, whereas Cassandra, PeerTrust and *RT* are based on Constraint DATALOG. KAoS relies on Description Logics, whereas Rei combines features of Description Logics (ontologies are used in order to define domain classes and properties associated with them), Logic Programming (Rei policies are actually particular Logic programs) and Deontic logic (in order to express concepts like rights, prohibitions, obligations and dispensations). EPAL exploits Predicate logic without quantifiers. Finally, no formalisms underly Ponder (which essentially belongs to the Object-oriented paradigm), TPL, WSPL and XACML.

Action execution. During the evaluation of a policy some actions may have to be performed: one may want to retrieve the current system time (e.g., in case authorization should be allowed only in a specific time frame), send a query to a database or record some information in a log file. PROTUNE supports all of these actions, as long as a basic assumption holds, namely that action results do not interfere with each other (i.e., that actions are independent). KAoS, Rei, *RT* and TPL do not support action execution, and the only actions supported in PeerTrust and PSPL are related to sending evidence.

XACML allows to specify actions within a policy: these actions are collected during the policy evaluation and executed before sending a response back to the requester. A similar mechanism is provided by EPAL and of course by WSPL, which is indeed a specific profile of XACML. Cassandra, if equipped with a suitable constraint domain, allows to specify side-effect free actions (e.g., to access the current time). Ponder allows to access system properties from within a policy, moreover it supports obligation policies, asserting which actions should be executed if some event happens: examples of such actions are printing a file, tracking some data in a log file and enabling/disabling user accounts. 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). Some languages provide a means for cascaded delegations up to a certain length, whereas others allow unbounded delegation chains. PROTUNE and Cassandra exploit the expressiveness of rules in order to simulate high-level constructs. This approach allows expert users to express more flexible delegation policies.

Ponder defines a specific kind of policies in order to deal with delegation: *positive* delegation policies can specify constraints (e.g., time restrictions) to limit the validity of the delegated access rights. Rei allows not only to define policies delegating rights but even policies delegating the right to delegate (some other right). Delegation is supported by RT^D ("D" stands precisely for "delegation")Since *RT* is a role-based language, the right which can be delegated is the one of activating a role, i.e., the possibility of acting as a member of such a role. Ponder delegation chains have length 1, whereas in *RT* delegation chain length is not bounded. Delegation of authority can be expressed in PeerTrust by exploiting the operator `@`. Finally, EPAL, KAoS, PSPL, TPL, WSPL and XACML do not support delegation.

Type of evaluation. Most of the languages we consider here require that all policy rules be collected in some place before starting their evaluation: this is the way EPAL, KAoS, Ponder, *RT* and TPL work. Other languages, namely Cassandra, Rei, WSPL and XACML, perform policy evaluation locally, nevertheless they provide some facility to collect policies (or policy fragments) which are spread over the net: e.g., in XACML combining algorithms define how to take results from multiple policies and derive a single result, whereas Cassandra allows policies to refer to policies of other entities, so that policy evaluation may trigger queries of remote policies. Policies can be collected into a single place if they are freely disclosable (unless the collection place is trusted by all parties), therefore the languages mentioned so far do not address the possibility that policies themselves may have to be kept private. Protection of sensitive policies is instead a key feature in PROTUNE which can be obtained only by providing support to distributed policy evaluation. PROTUNE, PeerTrust and PSPL are the only frameworks which provide distributed policy evaluation.

Evidence. Only PSPL, PeerTrust and PROTUNE support unsigned declarations. Credentials are the only evidence available in Cassandra, *RT* and TPL, whereas they are unnecessary in Ponder, whose policies concern users who have already been successfully authenticated. Finally, EPAL, KAoS, Rei, WSPL and XACML support neither credentials nor unsigned declarations.

Explanations. The result sent back by the policy engine, in the easiest case, is a boolean answer (`allowed` vs. `denied`). KAoS, PeerTrust, Ponder, PSPL, *RT* and TPL conform to this pattern. WSPL and XACML go just a little further by providing two more result values: `not_applicable` is returned whenever no applicable policies or rules could be found, whereas `indeterminate` accounts for processing errors; in the latter case optional information is available to explain the error.

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.

	Cassandra	EPAL	KAoS	PeerTrust	Ponder	PROTUNE	PSPL	Rei	RT	TPL	WSPL	XACML
Well-defined semantics	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No	No
Action execution	Yes	Yes	No	Yes (only sending evidences)	Yes (access to system properties)	Yes	Yes (only sending evidences)	No	No	No	Yes	Yes
Delegation	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes (RT^D)	No	No	No
Type of evaluation	Distributed policies, Local evaluation	Local	Local	Distributed	Local	Distributed	Distributed	Distributed policies, Local evaluation	Local	Local	Distributed policies, Local evaluation	Distributed policies, Local evaluation
Evidence	Credentials	No	No	Credentials, Declarations	–	Credentials, Declarations, Reputation (via `in`)	Credentials, Declarations	–	Credentials	Credentials	No	No
Explanations	A/D and a set of constraints	A/D, `scope error`, `policy error`	A/D	A/D	A/D	Explanations	A/D	A/D [16]	A/D	A/D	A/D, `not applicable`, `indeterminate`	A/D, `not applicable`, `indeterminate`
Extensibility	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	No	Yes

TABLE 2
Policy language comparison ("–" = not applicable)

As described in Section 5, a standard interface to external packages is one of the means provided by PROTUNE in order to support extensibility. 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_0 with a small set of rules.[17]

Extensibility is described as one of the criteria taken into account in designing Ponder: the strategy used there is using inheritance as in object-oriented languages.

XACML's support to extensibility is two-fold: (i) on the one hand, new datatypes and functions may be defined; (ii) As we mentioned above, XACML policies can consist of any number of distributed rules; XACML already provides a number of combining algorithms which define how to take results from multiple policies and derive a single result, nevertheless a standard extension mechanism is available to define new algorithms. Using non-standard user-defined datatypes would lead to wasting one of the strong points of WSPL, namely the standard algorithm for merging two policies, resulting in a single policy that satisfies the requirements of both (assuming that such a policy exists), since there can be no standard algorithm for merging policies exploiting user-defined attributes (except where the values of the attributes are exactly equal).

Ontologies are the means to cater for extensibility in KAoS and Rei: the use of ontologies facilitates a dynamic adaptation of the policy framework by specifying the ontology of a given environment and linking it with the generic framework ontology; both KAoS and Rei define basic built-in ontologies, which are supposed to be further extended for a given application.

In Cassandra the key extensibility element is the *constraint domain* (and solver) plugged into the policy evaluation engine.

Finally, PeerTrust, PSPL, *RT* and TPL do not provide any extension mechanisms.

10 EVALUATION

A complex framework like PROTUNE can be evaluated from different perspectives, including policy language expressiveness, performance, and usability. The following sections analyze PROTUNE from these viewpoints.

10.1 Language expressiveness

PROTUNE's rule language can be evaluated by analyzing its formal expressiveness. 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, the expressiveness results for rule-based query languages apply to PROTUNE's policy language.

It is known that stratified function-free logic programs with negation can 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 constants occurring in the database; the same holds even if negation is applied only to database (extensional) relations (see [19] 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 perdicates 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 [33] 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 $\wedge$ and $\vee$).

The requirements over privacy policy languages can be used to evaluate PROTUNE from a slightly different perspective. A recent paper [22] shows that PROTUNE is currently one of the most complete and advanced approaches. The analysis of [22] is summarized in Table 3.

17. See report I2-D2, http://rewerse.net/deliverables/m12/i2-d2.pdf

Criterion	Protune	Rei	Trust-X	KeyNote	Ponder	Appel
Resource classification	with ontology			none	taxonomy	P3P
Access control	yes					
Minimal disclosure	yes	no	partial	no	yes	no
Policy protection	yes	yes	partial	no	yes	no
Push control	yes	yes	no	yes	yes	no
Usage control	no(*)	yes	no	no	yes	yes

(*) work in progress; first results in [42]

TABLE 3
A comparison of some major privacy languages

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 [42] for more details. In Table 3, 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 Performance evaluation

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`[18] 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 can be typically encoded with a small number of rules. However, the increasing availability of sophisticated policy frameworks 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.[19]

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 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). Tab. 3 reports the results of these experiments, some of which are interrupted after 150sec. 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. This is interesting because the availability of frameworks like PROTUNE may encourage the adoptions of policies more articulated and sophisticated than those deployed 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 artifical 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 PROTUNEX 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 imple-

18. http://logging.apache.org/log4j/

19. The components described so far can be freely downloaded from http://skydev.l3s.uni-hannover.de/gf/project/protune/wiki/?pagename=Evaluation.

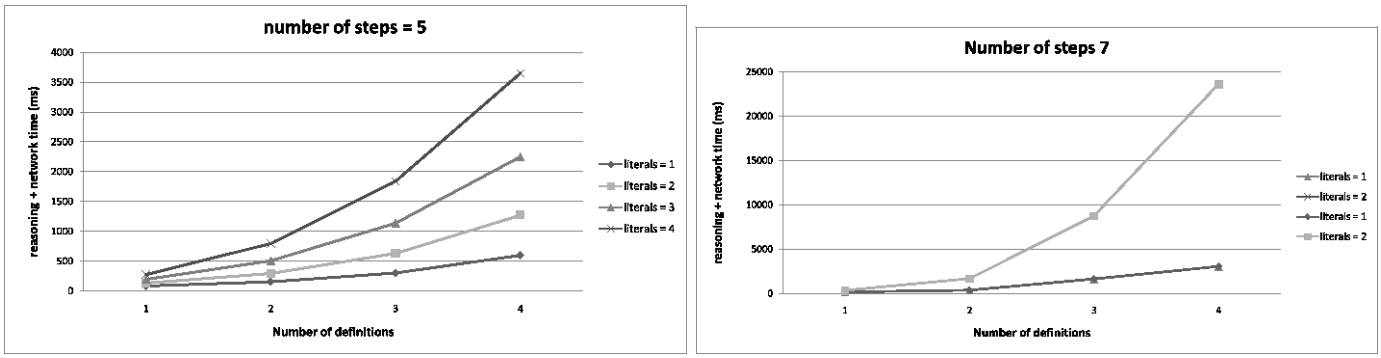

Fig. 3. Overall reasoning and network time (msec)

mentation based on tuProlog [21], the same Java-based Prolog engine adopted for implementing PROTUNE's core. We mention that there exists also a stand-alone implementation of PROTUNEX, available at http://cs.na.infn.it/rewerse/demos/protune-x/demo-protune-x.html, that runs on XSB-Prolog, a Prolog engine written in C equipped with memoizing methods (tabling) to improve performances and provide a more declarative semantics than standard Prolog. Tabling significantly improves the performance of PROTUNEX; 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.

10.3 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 [40], [25]. 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 described in Sec. 7. Point 5 is currently not concerning PROTUNEX because PROTUNE access control and release policies cannot express any contradictions.

To the best of our knowledge, PROTUNE is currently the only second-generation explanation facility for a TN framework (i.e., the only such facility satisfying the above criteria).

Another fundamental evaluation method is by systematic user studies. In these studies, users are confronted with policy usage and policy understanding 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 constitutes a research line in itself that will be the subject of future work.

We are also exploring the use of controlled natural language to formulate policies. About 90% of the rules in the digital library policy can be formulated in a natural way using Attempto Controlled English;[20] the remaining 10% can be expressed in a more technical way, that requires deeper knowledge about the disambiguation conventions of Attempto. The evaluation of a controlled natural language interface can be based on systematic user studies, too, focussed on policy authoring. Another interesting direction for further research is the investigation of different disambiguation strategies, for the purpose of reducing the percentage of rules that require a "technical" formulation.

11 DISCUSSION AND CONCLUSIONS

Rule-based policy languages have been regarded as appealing policy languages for a long time (cf. [15]); in the new area of TN they show their advantages even more clearly. A rule-based policy can be treated like a

20. http://attempto.ifi.uzh.ch/.

knowledge base: it can be shared, and it can be used for a variety of reasoning tasks.

For example, by having peers exchange (a filtered version of) their policies, one can espress in a compact way complex requests consisting of boolean combinations of heterogeneous evidence. This brings several advantages: (i) users can see at once multiple alternative ways of fulfilling a policy and select those that more closely match their privacy preferences; (ii) this can be done without paying the price of enumerating all the possible alternative combinations of credentials and declarations, which may decrease performance and significantly increase message size and/or number by combinatorial effects; (iii) moreover, the disclosed part of a policy can be used to construct sophisticated contextualized explanations on the clients, without overloading the 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, to enforce policies and check authorizations;
- abduction, to select relevant evidence from a portfolio, using the policy as a request;
- filtering, to protect the sensitive parts of the policies and avoid sending irrelevant parts of policies and local states across the network;
- explanations, that help users in understanding and verifying policies and negotiation outcomes.

More precisely, a formal semantics is essential to guarantee coherent outputs across the activities associated to the above reasoning tasks.

We observe that rule technology is more mature than description logics technology with respect to the particular combination of functionalities and reasoning problems needed in TN. Automated deduction techniques are very advanced for both families of logics, while abduction and explanation in description logics are relatively less developed. There are only a few, mostly recent works on abduction [17], [32], [9] and explanations [16], [20] in the literature on description logic, and even less implementations.

The adoption of a logical policy language enables the realization of further interesting tools. For example, it is interesting to compare two policies P_1 and P_2 to see whether in all possible contexts, the authorizations granted by P_1 are also granted by P_2. Among the main concrete applications of such a tool we mention (i) checking whether the disclosure policy of a web site complies with a user's privacy preferences (a form of usage control), and (ii) verifying whether the set of authorizations is increased or decreased by a policy update (a verification task). Since policies can be regarded as particular relational queries, there is an opportunity of cross-fertilization from database theory; a first step in this direction can be found in [12].

So far we have been talking about features that are compatible with a variety of rule-based frameworks. Besides those properties, PROTUNE has specific features that address practical issues and make it one of the most complete TN frameworks. Such features include an interface with 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 effort, an extension mechanism based on the metalanguage and on rule-based ontologies, and a unique second generation explanation facility.

There are still many interesting open issues for future work. The study of policy comparison problems has just started; the decidability threshold, the complexity for different policy languages, and efficient implementation techniques are far from understood. Second, usage control is becoming more and more important in addressing the privacy issues of modern applications, and it is not yet clear how to incorporate it in rule-based languages. Last but not least, usability is a major issue that will require significant efforts and extensive user studies.

REFERENCES

[1] *8th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2007), 13-15 June 2007, Bologna, Italy*. IEEE Computer Society, 2007.

[2] *9th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2008), 2-4 June 2008, Palisades, New York, USA*. IEEE Computer Society, 2008.

[3] A. H. Anderson, "An introduction to the web services policy language (wspl)," in *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*. IEEE Computer Society, June 2004, pp. 189–192.

[4] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter, "Enterprise privacy authorization language (epal 1.2)." IBM, Tech. Rep., November 2003.

[5] M. Backes, G. Karjoth, W. Bagga, and M. Schunter, "Efficient comparison of enterprise privacy policies." in *Proceedings of the 2004 ACM symposium on Applied computing*. ACM Press, 2004, pp. 375–382.

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

[7] S. Baselice, P. A. Bonatti, and M. Faella, "On interoperable trust negotiation strategies," in *POLICY*. IEEE Computer Society, 2007, pp. 39–50.

[8] M. Becker and P. Sewell, "Cassandra: Distributed access control policies with tunable expressiveness," in *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*. IEEE Computer Society, June 2004, pp. 159–168.

[9] M. Bienvenu, "Complexity of abduction in the el family of lightweight description logics," in *KR*, G. Brewka and J. Lang, Eds. AAAI Press, 2008, pp. 220–230.

[10] P. Bonatti and P. Samarati, "Regulating service access and information release on the web," in *Proceedings of the 7th ACM conference on Computer and communications security*. ACM Press, pp. 134–143.

[11] P. A. Bonatti, C. Duma, D. Olmedilla, and N. Shahmehri, "An integration of reputation-based and policy-based trust management," in *Semantic Web Policy Workshop in conjunction with 4th International Semantic Web Conference*, Galway, Ireland, nov 2005.

[12] P. A. Bonatti and F. Mogavero, "Comparing rule-based policies," in *POLICY*. IEEE Computer Society, 2008, pp. 11–18.

[13] P. A. Bonatti and D. Olmedilla, "Driving and monitoring provisional trust negotiation with metapolicies," in *6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*. Stockholm, Sweden: IEEE Computer Society, jun 2005, pp. 14–23.

[14] P. A. Bonatti, D. Olmedilla, and J. Peer, "Advanced policy explanations on the web." in *17th European Conference on Artificial Intelligence (ECAI 2006)*. Riva del Garda, Italy: IOS Press, Aug-Sep 2006, pp. 200–204. [Online]. Available: 2006/2006_ECAI_explanations.pdf

[15] P. A. Bonatti and P. Samarati, "Logics for authorization and security," in *Logics for Emerging Applications of Databases*, J. Chomicki, R. van der Meyden, and G. Saake, Eds. Springer, 2003, pp. 277–323.

[16] A. Borgida, E. Franconi, I. Horrocks, D. L. McGuinness, and P. F. Patel-Schneider, "Explaining alc subsumption," in *Description Logics*, ser. CEUR Workshop Proceedings, P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. F. Patel-Schneider, Eds., vol. 22. CEUR-WS.org, 1999.

[17] S. Colucci, T. D. Noia, E. D. Sciascio, F. M. Donini, and M. Mongiello, "Concept abduction and contraction for semantic-based discovery of matches and negotiation spaces in an e-marketplace," *Electronic Commerce Research and Applications*, vol. 4, no. 4, pp. 345–361, 2005.

[18] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *Policies for Distributed Systems and Networks: International Workshop, POLICY 2001, Bristol, UK, January 2001. Proceedings*. Springer, February 2004, pp. 18–38.

[19] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, "Complexity and expressive power of logic programming," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 374–425, 2001.

[20] X. Deng, V. Haarslev, and N. Shiri, "Resolution based explanations for reasoning in the description logic *lc*," in *CSWWS*, ser. Semantic Web And Beyond Computing for Human Experience, M. T. Kone and D. Lemire, Eds., vol. 2. Springer, 2006, pp. 189–204.

[21] E. Denti, A. Omicini, and A. Ricci, "tu prolog: A light-weight prolog for internet applications and infrastructures," in *PADL*, ser. Lecture Notes in Computer Science, I. V. Ramakrishnan, Ed., vol. 1990. Springer, 2001, pp. 184–198.

[22] C. Duma, A. Herzog, and N. Shahmehri, "Privacy in the semantic web: What policy languages have to offer," in *POLICY'07*. IEEE Computer Society, 2007, pp. 109–118.

[23] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. E. Seamons, and M. Winslett, "No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web," in *1st European Semantic Web Symposium (ESWS 2004)*. Springer, May 2004, pp. 342–356.

[24] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *ICLP/SLP*, 1988, pp. 1070–1080.

[25] S. R. Haynes, "Explanation in information systems: A design rationale approach," Ph.D. dissertation, London School of Economics and Political Science, Dept. of Information Systems and Dept. of Social Psychology, 2001.

[26] A. Herzberg, Y. Mass, J. Michaeli, Y. Ravid, and D. Naor, "Access control meets public key infrastructure, or: Assigning roles to strangers," in *Security and Privacy, 2000. Proceedings. 2000 IEEE Symposium on*. IEEE Computer Society, May 2000, pp. 2–14.

[27] L. Kagal, T. W. Finin, and A. Joshi, "A policy language for a pervasive computing environment," in *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*. IEEE Computer Society, June 2003, pp. 63–74.

[28] L. Kagal, C. Hanson, and D. J. Weitzner, "Using dependency tracking to provide explanations for policy management," in *POLICY*. IEEE Computer Society, 2008, pp. 54–61.

[29] N. Li and J. C. Mitchell, "Rt: A role-based trust-management framework," in *Third DARPA Information Survivability Conference and Exposition (DISCEX III)*.

[30] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah, "First experiences using xacml for access control in distributed systems." in *Proceedings of the 2003 ACM workshop on XML security*. ACM Press, 2003, pp. 25–37.

[31] D. L. McGuinness and P. P. da Silva, "Explaining answers from the semantic web: The inference web approach," *Journal of Web Semantics*, vol. 1, no. 4, pp. 397–413, 2004.

[32] A. Ragone, T. D. Noia, E. D. Sciascio, F. M. Donini, S. Colucci, and F. Colasuonno, "Fully automated web services discovery and composition through concept covering and concept abduction," *Int. J. Web Service Res.*, vol. 4, no. 3, pp. 85–112, 2007.

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

[34] T. M. Simon Godik, "Oasis extensible access control markup language (xacml) version 1.0." OASIS, Tech. Rep., February 2003.

[35] S. Staab, B. K. Bhargava, L. Lilien, A. Rosenthal, M. Winslett, M. Sloman, T. S. Dillon, E. Chang, F. K. Hussain, W. Nejdl, D. Olmedilla, and V. Kashyap, "The pudding of trust," *IEEE Intelligent Systems*, vol. 19, no. 5, pp. 74–88, 2004.

[36] V. S. Subrahmanian, P. A. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross, *Heterogenous Active Agents*. MIT Press, 2000.

[37] V. Subrahmanian, S. Adali, A. Brink, R. Emery, J. Lu, A. Rajput, T. Rogers, R. Ross, and C. Ward, "Hermes: Heterogeneous reasoning and mediator system," http://www.cs.umd.edu/projects/publications/abstracts/hermes. html.

[38] T. Syrjänen, "Omega-restricted logic programs," in *LPNMR*, ser. Lecture Notes in Computer Science, T. Eiter, W. Faber, and M. Truszczynski, Eds., vol. 2173. Springer, 2001, pp. 267–279.

[39] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott, "Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement," in *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*. ACM Press, June 2003, pp. 93–96.

[40] M. R. Wick, "Second generation expert system explanation," in *Second Generation Expert Systems*, J.-M. David, J.-P. Krivine, and R. Simmons, Eds. Springer Verlag, 1993, pp. 614–640.

[41] W. Winsborough, K. Seamons, and V. Jones, "Automated trust negotiation," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*. IEEE Computer Society, 24 2000, pp. 88–102.

[42] C. Zhang, P. Bonatti, and M. Winslett, "Peeraccess: A logic for distributed authorization," in *12th ACM Conference on Computer and Communication Security (CCS 2005)*. Alexandria, VA, USA: ACM.

APPENDIX

The stable model semantics

The stable model semantics [24] is based on the notion of *Gelfond-Lifschitz reduct* of a normal logic program w.r.t. a Herbrand model.

The Gelfond-Lifschitz reduct of P with respect to a Herbrand model M, denoted bt $\mathsf{GL}(P, M)$, is obtained from the ground instantiation of P by

- removing all rules containing a literal $\neg B$ with $B \in M$;
- removing all negative literals from the remaining rules.

A Herbrand model M of P is a *stable model of* P iff M is the least model of the (positive) program $\mathsf{GL}(P, M)$.

In general a logic program may have zero, one, or more stable models. Checking the existence of stable models is NP-hard (even in the propositional case), and computing the atoms that belong to all models is co-NP-hard.

However, if a program is *stratified*, then it has exactly one stable model that in the propositional case can be computed in quadratic time. We recall that a program is stratified iff there exists a function λ from predicate symbols to integers, such that for all rules $r \in P$, if p is the predicate occurring in the head and q is any predicate occurring in the body, then

1) $\lambda(p) \geq \lambda(q)$;
2) if q occurs in the scope of negation (in r), then $\lambda(p) > \lambda(q)$.

This property can be checked in linear time using suitable graph algorithms.

A further nice property of stratified programs is that the major semantics for negation as failure agree: the unique stable model of a stratified program is also its well-founded model and its perfect model.

Further details and pointers can be found in [6].