Chapter 4
Rule-Based Policy Representations and Reasoning

Piero Andrea Bonatti[1], Juri Luca De Coi[2], Daniel Olmedilla[2,3], and Luigi Sauro[1]

[1] Università di Napoli Federico II, Via Cinthia, 80126 Napoli, Italy
{bonatti,sauro}@na.infn.it
[2] Forschungszentrum L3S, Appelstr. 9a, 30167 Hannover, Germany
{decoi,olmedilla}@L3S.de
[3] Telefónica Research & Development, C/ Emilio Vargas 6, 28043 Madrid, Spain
danieloc@tid.es

Abstract. Trust and policies are going to play a crucial role in enabling the potential of many web applications. Policies are a well-known approach to protecting security and privacy of users in the context of the Semantic Web: in the last years a number of policy languages were proposed to address different application scenarios.

The first part of this chapter provides a broad overview of the research field by accounting for twelve relevant policy languages and comparing them on the strength of ten criteria which should be taken into account in designing every policy language. By comparing the choices designers made in addressing such criteria, useful conclusions can be drawn about strong points and weaknesses of each policy language.

The second part of this chapter is devoted to the description of the Protune framework, a system for specifying and cooperatively enforcing security and privacy policies on the Semantic Web developed within the network of excellence REWERSE. We describe the framework's functionalities, provide details about their implementation, and report the results of performance evaluation experiments.

4.1 Introduction

Trust is the top layer of the famous Semantic Web picture. It plays a crucial role in enabling the potential of the web. While security and privacy do not cover all the facets of trust, still they play a central role in raising the level of trust in web resources.

Security management is a foremost issue in large scale networks like the Semantic Web. In such a scenario, traditional assumptions for establishing and enforcing access control regulations do not hold anymore. In particular identity-based access control mechanisms have proved to be ineffective, since in decentralized and multicentric environments, the requester and the service provider are often unknown to each other.

F. Bry and J. Maluszynski (Eds.): Semantic Techniques for the Web, LNCS 5500, pp. 201–232, 2009.

Web Services obviously need some form of access control. Moreover, recent experiences with Facebook's "beacon" service[1] and Virgin's use of Flickr pictures[2] have shown that users are not willing to accept every possible use (or abuse) of their data.

Policies are a well-known approach to protecting security and privacy of users in the context of the Semantic Web: policies specify who is allowed to perform which action on which object depending on properties of the requester and of the object as well as parameters of the action and environmental factors (e.g., time). The application of suitable policies for protecting services and sensitive data may determine success or failure of a new service. In a near future, we might see Web Services compete with each other by improving and properly advertising their policies.

4.2 A Review of the State-of-the-Art in Policy Languages

The potential policies have proved to own is not fully exploited yet, since nowadays their usage is mainly restricted to specific application areas. On the one hand this depends on general lack of infrastructure services for such policies to truly function: for instance, there are no end user-oriented digital certification services (national digital ID providers are just appearing). On the other hand, lacking knowledge about currently available solutions is one of the main factors hindering widespread use of policies: in order to exploit a policy language the potential user needs to be provided with a clear picture of the advantages it provides in comparison with other solutions. Furthermore in the last years many policy languages were proposed, targeting different application scenarios and provided with different features and expressiveness: scope and properties of available languages have to be known to the user in order to help her in choosing the one most suitable to her needs.

In an attempt to help with these and other problems, comparisons among policy languages have been provided in the literature. However existing comparisons either do not consider a relevant number of available solutions or are mainly focused on the application scenarios the authors worked with (e.g., trust negotiation in [20] or ontology-based systems in [22]) Moreover policy-based security management is a rapidly evolving field and most of this comparison work is now out-of-date.

In this section we provide an extensive comparison covering twelve policy languages. Such a comparison will be carried out on the strength of ten criteria. Our analysis will hopefully have the side-effect of helping users in choosing the policy language mostly suiting their needs, as well as researchers currently investigating this area.

[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

This section is organized as follows. In section 4.2.1 related work is accounted for. Section 4.2.2 briefly sketches the evolution of the research field and introduces some concepts (e.g., role-based policy language as well as various kinds of policies) which will be massively exploited in the following. Sections 4.2.3 and 4.2.4 respectively introduce the languages which will be compared later on and the criteria according to which the comparison will be carried out. The actual comparison takes place in section 4.2.5, whereas section 4.2.6 presents overall results and draws some conclusions.

4.2.1 Related Work

The paper of Seamons et al. [20] is the basis of our comparison: some of the insights they suggested have proved to be still valuable right now and as such they are addressed in our work as well. Nevertheless in over six years the research field has considerably changed and nowadays many aspects of [20] are out of date: new languages have been developed and new design paradigms have been taken into account, what makes the comparison performed in [20] obsolete and many criteria according to which they were evaluated not suitable anymore.

The pioneer paper of Seamons et al. paved the way to future research on policy language comparisons like Tonti et al. [22], Anderson [2] and Duma et al. [14]: although [22] actually presents a comparison of two ontology-based languages (namely KAoS and Rei) with the object-oriented language Ponder, the work is rather an argument for ontology-based systems, since it clearly shows the advantages of ontologies.

Because of the impressive amount of details it provides, [2] restricts the comparison to only two (privacy) policy languages, namely EPAL and XACML, therefore a comprehensive overview of the research field is not provided, and features which neither EPAL nor XACML support are not taken into account at all among the comparison criteria.

Finally [14] provides a comparison specifically targeted to giving insights and suggestions to policy writers (*designers*): therefore the criteria, according to which the comparison is carried out, are mainly practical ones and scenario-oriented, whereas more abstract issues are considered out of scope and hence not addressed.

4.2.2 Background

In this section some concepts are introduced, which will help to smoothly understand the rest of the chapter. First an overall picture of the research field is provided by briefly outlining the historical evolution of policy languages, then the definitions of some policy types which will be used throughout the chapter are provided.

From uid/psw-based authentication to trust negotiation. Traditional access control mechanisms (like the ones exploited in traditional operating systems) make authorization decisions based on the identity of the requester: the user must provide a pair (*username*, *password*) and, if this pair matches with one of the entry in some static table kept by the system (e.g., the file `/etc/pwd` in Unix) the user is granted with some privileges. However, in decentralized or multicentric environments, peers are often unknown to each other, and access control based on identities may be ineffective. In order to address this scenario, role-based access control mechanisms were developed. In a role-based access control system a user is assigned with one or more roles, which are in turn exploited in order to take authorization decisions. Since the number of roles is typically much smaller than the number of users, role-based access control systems reduce the number of access control decisions. A thorough description of role-based access control can be found in [16].

In a role-based access control system the authorization process is split into two steps, namely assignment of one or more roles and check whether a member of the assigned role(s) is allowed to perform the requested action. The role-based languages we consider provide support only to one of the two steps: for instance, TPL (a role-assignment policy language) policies describe to which role the requester can be mapped; this role must then be fed as input to an existing role-based access control mechanism. A similar approach is taken by Cassandra and *RT*. On the other hand Ponder (authorization) policies are meant to support the second step, i.e., they allow to define which actions may be performed by a requester who has already been successfully authenticated.

Role-based authentication mechanisms require that the requester provides some information in order to map her to some role(s). In the easiest case this information can be once again a (uid, pwd) pair, but systems which need a stronger authentication usually exploit credentials, i.e., digital certificates representing statements certified by given entities (*certification authorities*) which can be used in establishing properties of their holder. More modern approaches (e.g., EPAL, WSPL and XACML) directly exploit the properties of the requester in order to make an authorization decision, i.e., they do not split the authorization process in two parts like role-based languages. Nevertheless they do not use credentials in order to certificate the properties of the requester.

Credentials, as well as declarations (i.e., not signed statements about properties of the holder) are however supported by PeerTrust, Protune and PSPL, which are languages designed to support the trust negotiation [24] vision. The notion of trust management was introduced by [7] as a new paradigm bringing together authentication and authorization in distributed systems. A scenario-based introduction to Trust Negotiation is provided in Section 4.3.2.

Policy types. Policies can be exploited in a number of fields and with different goals: security, management, conversation, quality-of-service, quality-of-protection, reliable messaging, reputation-based, provisional policies are just some examples of policies which are encountered in the literature. Here we focus on policy types which will be mentioned in the following, for instance because some language we consider has been explicitly designed to support that kind of policy.

Role-assignment policies. As the name suggests, role-assignment policies specify which conditions a requester must fulfill in order to belong to some server-defined role. Role-assignment policies are typically used in role-based policy languages like Cassandra, *RT* and TPL which postulate the existence of a back-end role-based access control mechanism to which the role will be fed in order to perform the actual authorization.

Access control policies. Access control is concerned with limiting the activities a user is allowed to perform. Consequently access control policies define the prerequisites the requester must fulfill in order to have the activity she asked for performed.

Privacy policies. Privacy policies are meant to protect the privacy of the user: they need to reflect current regulations and possibly promises made to the customers. Privacy policies arise further issues in comparison to access control policies, as they require a more sophisticated treatment of deny rules and conditions on context information; moreover privacy policy languages have to take into account the notion of "purpose", which is essential to privacy legislation. A subset of privacy policies are *enterprise* privacy policies which furthermore have to provide support to more restrictive enterprise-internal practices and may need to handle customer preferences. EPAL was especially designed in order to target enterprise privacy policies.

Obligation policies. Obligation policies specify the actions that must be performed when certain events occur, i.e., they are event-triggered condition-action rules. Obligation policies may be exploited, e.g., to specify which actions must be performed when security violations occur or under which circumstances auditing and logging activities have to be carried out. Obligation policies are supported, among others, by KAoS, Ponder and Rei.

4.2.3 Presentation of the Considered Policy Languages

To date a bunch of policy languages have been developed and are currently available: we have chosen those which at present seem to be the most popular ones, namely Cassandra [6], EPAL [3], [4], KAoS [23], PeerTrust [15], Ponder [13], Protune [8], [10], PSPL [9], Rei [17], *RT* [18], TPL [16], WSPL [1] and XACML [19], [21]. The information we will provide about the aforementioned languages is based on the referenced documents. Whenever a feature we are going to tackle is not addressed in the considered literature nor is it known to the authors in other way, the feature is supposed not to be provided by the language.

The number and variety of policy languages proposed so far is justified by the different requirements they had to accomplish and the different use cases they were designed to support. Ponder was meant to help local security policy specification and security management activities, therefore typical addressed application scenarios include registration of users or logging and audit events, whereas firewalls, operating systems and databases belong to the applications targeted by the language. WSPL's name itself (namely Web Services Policy Language) suggests its goal: supporting description and control of various aspects and features of a Web Service. Web Services are addressed by KAoS too, as well as general-purpose grid computing, although it was originally oriented to software agent applications (where dynamic runtime policy changes need to be supported). Rei's design was primarily concerned with support to pervasive computing applications (i.e. those in which people and devices are mobile and use wireless networking technologies to discover and access services and devices). EPAL (Enterprise Privacy Authorization Language) was proposed by IBM in order to support enterprise privacy policies. Some years before IBM had already introduced the pioneer role-based policy language TPL (Trust Policy Language), which paved the way to other role-assignment policy languages like Cassandra and *RT* (Role-based Trust-management framework), both of which aimed to address access control and authorization problems which arise in large-scale decentralized systems when independent organizations enter into coalitions whose membership and very existence change rapidly. The main goal of PSPL (Portfolio and Service Protection Language) was providing a uniform formal framework for regulating service access and information disclosure in an open, distributed network system like the web; support to negotiations and private policies were among the basic reasons which led to its definition. PeerTrust is a simple yet powerful language for trust negotiation on the Semantic Web based on a distributed query evaluation. Trust negotiation is addressed by Protune too, which supports a broad notion of "policy" and does not require shared knowledge besides evidences and a common vocabulary. Finally XACML (eXtensible Access Control Markup Language) was meant to be a standard general purpose access control policy language, ideally suitable to the needs of most authorization systems.

Given the multiplicity of available languages and the sometimes very specific contexts they fit into, one may argue that a meaningful comparison among them is impossible or, at least, meaningless. We claim that such a comparison is not only possible but even worth: to this aim we identified ten criteria which should be taken into account in designing every policy language. By comparing the choices designers made in addressing such criteria, useful conclusions can be drawn about strong points and weaknesses of each policy language.

4.2.4 Presentation of the Considered Criteria

We acknowledge the remark made by [14], according to which a comparison among policy languages on the basis of the criteria presented in [20] is only partially satisfactory for a designer, since general features do not help in understanding which kind of policies can be practically expressed with the constructs available in a language. Therefore in our comparison we selected a good deal of criteria having a concrete relevance (e.g., whether actions can be defined within a policy and executed during its evaluation, how the result of a request looks like, whether the language provides extensibility mechanisms and to which extent . . .). On the other hand, since we did not want to come short on theoretical issues, we selected four additional criteria, basically taken from [20] and somehow reworked and updated them. We called these more theoretical criteria *core policy properties* whereas more practical issues have been grouped under the common label *contextual properties*. In the following presentation core policy properties precede contextual properties.

Well-defined semantics. According to [20] we consider a policy language's semantics to be well-defined if the meaning of a policy written in that language is independent of the particular implementation of the language. Logic programs and Description logic knowledge bases have a mathematically defined semantics, therefore we assume policy languages based on either of the two formalisms to have well-defined semantics.

Monotonicity. In the sense of logic, a system is monotonic if the set of conclusions which can be drawn from the current knowledge base does not decrease by adding new information to the knowledge base. In the sense of [20] a policy language is considered to be monotonic if an accomplished request would also be accomplished if accompanied by additional disclosure of information by the peers: in other words, disclosure of additional evidences and policies should only result in the granting of additional privileges. Policy languages may be not monotonic in the sense of logic (as it happens with Logic programming-based languages) but still be monotonic in the sense of [20], like Protune.

Condition expressiveness. A policy language must allow to specify under which conditions the request of the user (e.g., for performing an action or for disclosing a credential) should be accomplished. Policy languages differ in the expressiveness of such conditions: some languages allow to set constraints on properties of the requester, but not on parameters of the requested action, moreover constraints on environmental factors (e.g., time) are not always supported. This criterion subsumes "credential combinations", "constraints on attribute values" and "inter-credential constraints" in [20].

Underlying formalism. A good deal of policy languages base on some well-known formalism. Knowledge about the formalism a language bases upon can be useful in order to understand some basic features of the language itself: e.g., the fact that a language is based on Logic programming with negation (as failure) entails consequences regarding the monotonicity of the language (in the sense of logic), whereas knowing that Description logic knowledge bases may contain contradictory statements could induce to infer that a Description logics-based language needs a way to deal with such contradictions.

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), to send a query to a database or to record some information in a log file.

It is worth noticing that this criterion evaluates whether a language allows the *policy writer* to specify actions within a policy: during the evaluation of a policy the engine may carry out non-trivial actions on its own (e.g., both *RT* and TPL engines provide automatic resolution of credential chains) but such actions are not considered in our investigation.

Delegation. Delegation is often used in access control systems to cater for temporary transfer of access rights to agents acting on behalf of other ones (e.g., passing write rights to a printer spooler in order to print a file). The right of delegating is a right as well and as such can be delegated, too. Some languages provide a means for cascaded delegations up to a certain length, whereas others allow unbounded delegation chains.

In order to support delegation many languages provide a specific built-in construct, whereas others exploit more fine-grained features of the language in order to simulate high-level constructs. The latter approach allows to support more flexible delegation policies and is hence more suited for expressing the subtle but significant semantic differences which appear in real-world applications.

Evidences. The result of a policy's evaluation may depend on the identities or other properties of the peer who requested for its evaluation: a means needs hence to be provided in order for the peers to communicate such properties to each other. Such information is usually sent in the form of digital certificates signed by trusted entities (*certification authorities*) and called *credentials.* Credentials are not supported, among else, by languages not targeting authentication policies. PeerTrust, Protune and PSPL provide another kind of evidence, namely *declarations* which are non-signed statements about properties of the holder (e.g., credit-card numbers).

Negotiation support. [1] adopts a broad notion of "negotiation", namely a negotiation is supposed to happen between two peers whenever (i) both peers are allowed to define a policy and (ii) both policies are taken into account when processing a request. According to this definition, WSPL supports negotiations as well. In this chapter we adopt a narrower definition of negotiation by adding a third prerequisite stating that (iii) the evaluation of the request must be distributed, i.e., both peers must locally evaluate the request and either decide to terminate the negotiation or send a partial result to the other peer who will go on with the evaluation.

Whether the evaluation is local or distributed may be considered an implementation issue, as long as policies are freely disclosable. Distributed evaluation is required under a conceptual point of view as soon as the need for keeping policies private arises: indeed if policies were not private, simply merging the peers' policies would reveal possible compatibilities between them.

Policy engine decision. The result of the evaluation of a policy must be notified to the requester. The result sent back by the policy engine may carry information to different extents: in the easiest case a boolean answer may be sent (allowed vs. denied). Some languages support error messages. Protune is the only language providing enough informative content to let the user understand how the result was computed (and thereby why the query succeeded/failed).

Extensibility. Since experience shows that each system needs to be updated and extended with new features, a good programming practice requires to keep things as general as possible in order to support future extensions. Almost every language provides some support to extensibility: in the following we will provide a description of the mechanisms languages adopt in order to support extensibility.

4.2.5 Comparison

In this section the considered policy languages will be compared according to the criteria outlined in section 4.2.4. The overall results of the comparison are summarized in Table 1. Notice that Table 1 does not contain criterion "condition expressiveness" which can be hardly accounted for in a table.

Well-defined semantics. We assume policy languages based on Logic programming or Description logics to have well-defined semantics. Since the formalisms underlying the considered policy languages will be accounted for in the following, so far we restrict ourselves to list the languages provided with a well-defined semantics, namely, Cassandra, EPAL, KAoS, PeerTrust, Protune, PSPL, Rei and RT.

Monotonicity. In the sense of [20] a policy language is considered to be monotonic if disclosure of additional evidences and policies only results in the granting of additional privileges, therefore the concept of "monotonicity" does not apply to languages which do not provide support for credentials, namely EPAL, Ponder, WSPL and XACML. All other languages are monotonic, with the exception of TPL, which explicitly chose to support *negative certificates*, stating that a user can be assigned a role if there exists no credential of some type claiming something about it.

The authors of TPL acknowledge that it is almost impossible proving that there does not exist such a credential somewhere, therefore they interpret their statement in a restrictive way, i.e., they assume that such a credential does not exist if it is not present in the local repository. Despite this restrictive definition the language is not monotonic since, as soon as such a credential is released and stored in the repository, consequences which could be previously drawn cannot be drawn anymore.

Condition expressiveness. A role-based policy language maps requesters to roles. The assigned role is afterwards exploited in order (not) to authorize the requester to execute some actions. The mapping to a role may in principle be performed according to the identity or other properties of the requester (to be stated by some evidence) and eventually environmental factors (e.g., current time). Cassandra (equipped with a suitable constraint domain) supports both scenarios.

Environmental factors are not taken into account by TPL, where the mapping to a role is just performed according to the properties of the requester; such properties can be combined by using boolean operators, moreover a set of built-in operators (e.g., greater than, equal to) is provided in order to set constraints on their values.

Environmental factors are not taken into account by RT_0 either, where role membership is identity-based, meaning that a role must explicitly list its members; nevertheless since (i) roles are allowed to express sets of entities having a certain property and (ii) conjunctions and disjunctions can be applied to existing roles in order to create new ones, then role membership is finally based on properties of the requester.

RT_1 goes a step beyond and, by adding the notion of *parametrized role*, allows to set constraints not only on properties of the requester but even on the ones of the object, the requested action should be performed upon; the last feature makes the second step traditional role-based policy languages consist of unnecessary, therefore RT_1, as well as the other RT flavors basing on it, may be considered to lay on the border between role-based and non role-based policy languages.

A non role-based policy language does not split the authentication process in two different steps but directly provides an answer to the problem whether the requester should be allowed to execute some action. In this case the authorization decision can be made in principle not only depending on properties of the requester or the environment, but also according to the ones of the object the action would be performed upon as well as parameters of the action itself. EPAL introduces the further notion of "purpose" for which a request was sent and allows to set conditions on it.

Some non role-based languages make a distinction between conditions which must be fulfilled in order for the request to be taken into consideration (which we call *prerequisites*, according to the terminology introduced by [9]) and conditions which must be fulfilled in order for the request to be satisfied (*requisites* according to [9]); not always both kinds of conditions have the same expressiveness.

Let start checking whether and to which extent the non role-based policy languages we considered support prerequisites: WSPL and XACML allow only to use a simple set of criteria to determine a policy's applicability to a request, whereas Ponder provides a complete solution which allows to set prerequisites involving properties of requester, object, environment and parameters of the action. Prerequisites can be set in EPAL and PSPL as well; the expressiveness of PSPL prerequisites is the same as the one of its requisites, which we will discuss later.

With the exception of Ponder, which allows restrictions on the environment just for delegation policies, each other language supports requisites (Rei is even redundant in this respect): KAoS allows to set constraints on properties of the requester and the environment, Rei also on action parameters and Protune, PSPL, WSPL and XACML also on properties of the object. EPAL supports conditions on the purpose for which a request was sent but not on environmental factors. Attributes must be typed in EPAL, WSPL, XACML and typing can be considered a constraint on the values the attribute can assume, anyway the definition of the semantics of such attributes is outside WSPL's scope. Finally, in PeerTrust conditions can be expressed by setting guards on policies: each policy consists of a guard and a body, the body is not evaluated until the guard is satisfied.

Underlying formalism. The most part of languages provided with a well-defined semantics rely on some kind of Logic programming or Description logics. Logic programming is the semantic foundation of Protune and PSPL, whereas a subset of it, namely Constraint DATALOG, is the basis for Cassandra, PeerTrust and RT. KAoS relies on Description logics, whereas Rei combines features of Description logics (ontologies are used in order to define domain classes and properties

Table 1. Policy language comparison ("–" = not applicable)

	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
Monotonicity	Yes	–	Yes	Yes	–	Yes	Yes	Yes	Yes	No	–	–
Underlying formalism	Constraint DATALOG	Predicate logic without quantifiers	Description logics	Constraint DATALOG	Object-oriented paradigm	Logic programming	Logic programming	Deontic logic, Logic programming, Description logics	Constraint DATALOG	–	–	–
Action execution	Yes (side-effect free)	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
Evidences	Credentials	No	No	Credentials, Declarations	–	Credentials, Declarations	Credentials, Declarations	–	Credentials	Credentials	No	No
Negotiation	Yes	No	No	Yes	No	Yes	Yes	No	No	Yes	No (policy matching supported)	No
Result format	A/D and a set of constraints	A/D, `scope error`, `policy error`	A/D	A/D	A/D	Explanations	A/D	A/D	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

associated with the classes), 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 only bases on the Object-oriented paradigm), TPL, WSPL and XACML.

Action execution. Ponder allows to access system properties (e.g., time) 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.

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.

The only actions which the policy writer may specify in PeerTrust and PSPL are related to the sending of evidences, whereas Protune supports whatever kind of actions, not necessarily side-effect free, as long as a basic assumption holds, namely that action results do not interfere with each other (i.e., that actions are independent).

Cassandra (equipped with a suitable constraint domain) allows to call side-effect free functions (e.g., to access the current time).

It is worth noticing that languages allowing to specify actions within policies can to some extent simulate obligation policies, as long as the triggering event is the reception of a request, although the flexibility provided by Ponder is not met in such languages.

Finally, KAoS, Rei, *RT* and TPL do not support execution of actions.

Delegation. Ponder defines a specific kind of policies in order to deal with delegation: the field `valid` allows *positive* delegation policies to specify constraints (e.g., time restrictions) to limit the validity of the delegated access rights. Rei allows not only to define policy delegating rights but even policy delegating the right to delegate (some other right). Delegation is supported by RT^D ("D" stands indeed for "delegation"): being *RT* 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 chains always have unbounded length. Cassandra and Protune provide a more flexible mechanism which allows to explicitly set the desired length of a delegation chain (as well as other properties of the delegation): in order to obtain such a flexibility the aforementioned languages do not provide high-level constructs to deal with delegation but simulate them by exploiting more fine-grained features of the language.

Delegation (of authority) can be expressed in PeerTrust by exploiting operator "@". Finally, EPAL, KAoS, PSPL, TPL, WSPL and XACML do not support delegation.

Type of evaluation. The most part of the considered languages require that all policies to be evaluated are collected in some place before starting the evaluation, which is hence performed locally: 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 in order 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 (possibly the requester's one) over the network.

Policies can be collected into a single place if they are freely disclosable (assuming that the place they are collected into is not a trusted one), therefore the languages mentioned so far do not address the possibility that policies themselves may have to be kept private. Protection of sensitive policies can be obtained only by providing support to distributed policy evaluation, like the one carried out by PeerTrust, Protune or PSPL.

Evidences. Credentials are a key element in Cassandra, *RT* and TPL, whereas they are unnecessary in Ponder, whose policies are concerned with limiting the activity of users who have already been successfully authenticated.

The authors of PSPL were the first ones advocating for the need of exchanging non-signed statements (e.g., credit card numbers), which they called *declarations*; declarations are supported by PeerTrust and Protune as well.

Finally, EPAL, KAoS, Rei, WSPL and XACML do not support evidences.

Negotiation support. As stated above, we use a narrower definition of negotiation than the one provided in [1], into which WSPL does not fit, therefore only pretty few languages support negotiation in the sense we specified above, namely Cassandra, PeerTrust, Protune and PSPL.

Policy engine decision. The evaluation of a policy should end up with a result to be sent back to the requester. In the easiest case such result is a boolean stating whether the request was (not) accepted (and thereby accomplished): KAoS, PeerTrust, Ponder, PSPL, *RT* and TPL conform to this pattern.

Besides `permit` and `deny` WSPL and XACML provide two other result values to cater for particular situations: `not_applicable` is returned whenever no applicable policies or rules could be found, whereas `indeterminate` accounts for some error which occurred during the processing; in the latter case optional information is available to explain the error.

A boolean value, stating whether the request was (not) fulfilled, does not make sense in the case of an obligation policy, which simply describes the actions which must be executed as soon as an event (e.g., the reception of a request) happens, therefore besides the so-called *rulings* `allow` and `deny` EPAL defines a third value (`don't care`) to be returned by obligation policies; one of the elements an EPAL policy consists of is a global condition which is checked at the very beginning of the policy evaluation: not fulfilling such a condition is considered an error and a corresponding error message (`policy_error`) is returned; a further message (`scope_error`) is returned in case no applicable policies were found.

Cassandra's request format contains (among others) a set of constraints c belonging to some constraint domain; the response consists of a subset c' of c which satisfies the policy; in case $c' = c$ (resp. c' is the empty set) `true` (resp. `false`) is returned.

Protune allows for more advanced explanation capabilities: not only is it possible to ask why (part of) a request was (not) fulfilled (`Why` and `Why-not` queries respectively), but the requester is even allowed to ask since the beginning which steps she has to perform in order for her request to be accomplished (`How-to` and `What-if` 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 the obligation (`NotMetEffects`).

Extensibility. Extensibility is a fuzzy concept: almost all languages provide some extension points to let the user adapt the language to her current needs, nevertheless the extension mechanism greatly varies from language to language: here we will briefly summarize the means the various languages provide in order to address extensibility.

Extensibility is described as one of the criteria taken into account in designing Ponder: in order to provide smoothly support to new types of policies that may arise in the future, inheritance was considered a suitable solution and Ponder itself was therefore implemented as an object-oriented language.

XACML's support to extensibility is two-fold

- on the one hand new datatypes, as well as functions for dealing with them, may be defined in addition to the ones already provided by XACML. Datatypes and functions must be specified in XACML requests, which indeed consists of typed attributes associated with the requesting subjects, the resource acted upon, the action being performed and the environment
- 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); use of non-standard algorithms would in turn mean that the policies could not be supported using a base standard policy engine. Being standardization the main goal of WSPL, no wonder that it comes short on the topic "extensibility", which is not necessarily a drawback, if the assertion of [1] holds: "most Web Services will probably use fairly simple policies in their service definitions".

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.

Extensibility was the main issue taken into account in the design of Cassandra: its authors realized that standard policy idioms (e.g., role hierarchy or role delegation) occur in real-world policies in many subtle variants: instead of embedding such variants in an *ad hoc* way, they decided to define a policy language able to express this variety of features smoothly; in order to achieve this goal, the key element is the notion of *constraint domain*, an independent module which is plugged into the policy evaluation engine in order to adjust the expressiveness of the language; the advantage of this approach is that the expressiveness (and hence the computational complexity) of the language can be chosen depending on the requirements of the application and can be easily changed without having to change the language semantics.

A standard interface to external packages is the means provided by Protune in order to support extensibility: functionalities of a component implementing such interface can be called from within a Protune policy.

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

4.2.6 Discussion

In this section we review the comparison performed in section 4.2.5 and provide some general comments.

By carrying out the task of comparing a considerable amount of policy languages, we came to believe that they may be classified in two big groups collecting, so to say, *standard-oriented* and *research-oriented* languages respectively. EPAL, WSPL and XACML can be considered standard-oriented languages since they provide a well-defined but restricted set of features: although it is likely that this set will be extended as long as the standardization process proceeds, so far the burden of providing advanced features is charged on the user who need them; standard-oriented languages are hence a good choice for users who do not need advanced features but for whom compatibility with standards is a foremost issue.

Ponder, *RT* and TPL are somehow placed in between: on the one hand Ponder provides a complete authorization solution, which however takes place after a previously overcome authentication step, therefore Ponder cannot be applied to contexts (like pervasive environments) were users cannot be accurately identified; on the other hand *RT* and TPL do not provide a complete authorization solution, since they can only map requesters to roles and need to rely on some external component to perform the actual authentication (although parametrized roles available in RT_1 and the other *RT* flavors basing on it make the previous statement no longer true).

Finally research-oriented languages strive toward generality and extensibility and provide a number of more advanced features in comparison with standard-oriented languages (e.g., conflict harmonization in KAoS and Rei, negotiations in Cassandra, PeerTrust and PSPL or explanations in Protune); they should be hence the preferred choice for users who do not mind about standardization issues but require the advanced functionalities that research-oriented languages provide.

4.3 A Framework for Semantic Web Policies

The languages presented in Section 4.2 can be expected to be used by security experts or other computer scientists. Common users cannot profit for them, since almost no policy framework offers facilities or tools to meet the needs of users without a strong background in computer science. Yet *usability* is a major issue in moving toward a policy-aware web. It is well known that as protection increases, usability is affected by the extra steps required for authentication and other operations related to access control. The information collected for security and privacy purposes extends the amount of sensitive information released by users while navigating the web. Moreover, it is frequently not clear to a common user which policy is actually applied by a system, and which are its consequences (cf. Virgin's case mentioned in Section 4.1). Similarly, common users may find it difficult to formulate their own privacy requirements and compare them with whatever privacy policy is advertised by a Web Service.

The work on policies carried out within the network of excellence REWERSE has tackled these aspects by regarding *policies as semantic markup*. By regarding policies as pieces of machine understandable knowledge:

- it is possible to assist some of the operations related to access control and information release, thereby improving a user's navigation experience;
- it is easier to support attribute-based access control, that increases the level of privacy in on-line transactions;
- it is possible to create policy documentation automatically; in this way alignment is guaranteed between the policy enforced by the system and the policy documented in natural language for end users; moreover it is possible to specialize explanations to specific contexts (such as a particular transaction); this may help users to understand why a transaction fails (policy violation or technical problems?), how to get the permissions for obtaining a service, and so on;
- it is possible to create tools for verifying policies and more generally supporting policy authoring; other tools may help users to compare privacy policies and make (semi-)automated policy-aware service selections.

In this section we describe the policy framework *Protune*, designed and implemented within REWERSE to incarnate the above ideas. Protune is meant to support policy creation and advanced policy enforcement, providing not only traditional access control but also trust negotiation (to automate security checks and privacy-aware information release) and second generation explanation facilities (to improve user awareness about—and control on—policies).

In the next section we summarize the different semantic techniques applied in Protune. Section 4.3.2 introduces a possible reference scenario that inspires most of the examples used in the following sections. Then Section 4.3.3 recalls the policy language and the core functionalities of Protune, followed by a section devoted to the explanation facility Protune-X. When needed, we point out differences from the previous theoretical papers that set the foundations of Protune and Protune-X [11,12]. Sections 4.3.5 and 4.3.6 describe the implementation and the existing facilities for integrating Protune in a web application such as policy-driven personalized presentation of web content. In Section 4.3.7 we report the results of a preliminary experimental performance evaluation. The chapter is concluded by a section on further research perspectives.

4.3.1 Policies as Semantic Markup in Protune

Policies are semantic markup because they specify declaratively part of the semantics (in terms of behavior constraints and admissible usage) of the static or dynamic resources that policies are attached to. Accordingly, semantic techniques have several roles in Protune:

- Policies are formulated as sets of axioms and meta-axioms with a formal, processing-independent semantics; this is the basis for consistent treatment of policies for different tasks: enforcement, negotiation, explanations, validation, etc.;
- The aforementioned tasks involve different automated reasoning mechanisms, such as deduction for enforcement, abduction and partial evaluation for negotiation, pruning and natural language generation for explanations, etc.;
- The auxiliary concepts needed to formulate policies (such as what is a public resource, what is an accepted credit card, ...) and the link between such concepts and the evidence needed to prove their truth (e.g. which X.509 credentials are needed to prove authentication, or what forms need to be filled in) are defined by means of lightweight ontologies that may be included in the policy itself or referred to by means of suitable URIs; therefore, unlike XACML *contexts*, Protune's auxiliary concepts are machine understandable and allow agent interoperability.

4.3.2 Negotiations

In response to a resource request, a server may return its policy for accessing the resource. The policy may contain (a reference to) an auxiliary ontology, as explained in the previous section. In the simplest case, user agents may use such machine understandable information to check automatically whether the policy can be fulfilled and how, thereby (partially) automating the operations needed for (traditional) access control and facilitating navigation in the presence of articulated policies. In advanced scenarios, a user agent may reply with a counter-request in order to enforce the user's privacy policy, as explained below.

An example 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 favourite writer. She finds out that an on-line bookshop she never heard about before sells the book at a very cheap price.

The bookshop groups its customers in different categories, according to personal data (country, age, profession ...) and purchase-related data (frequency, item, payment preferences ...). Different sale strategies are applied to different customer categories (e.g. prices discounted to different rates, no delivery fees, sending of promotional material, and so on).

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. This is just the bookshop's default policy, the custom-tailored policies described above are disclosed only after getting more information about the customer.

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 – BBB), therefore she asks the bookshop to provide such information.

The bookshop belongs indeed to the BBB and is willing to disclose such credential to anyone. This satisfies Alice, who provides her credit card number.

After having interacted with the VISA server to check that the credit card is valid, the bookshop asks Alice for other information, in order to understand which customer category she belongs to and apply the corresponding sale strategy.

The lunch break is already over and Alice has no time left to provide the data requested, therefore she decides to abort the transaction.

Scenario revisited. Automated server *and client* policy processing may significantly speed up interactions like the above. As soon as Alice decides which book she wants to buy, a negotiation between her agent and the on-line bookshop's agent would be triggered. Since the bookshop is not willing to provide the book for free, it would answer by returning its (default) policy protecting the book. The returned policy would contain the description of the actions Alice has to perform (either sending a credit card number or providing log-in data): the use of shared ontologies to identify such actions would grant common understanding of their semantics. Alice's agent would then quickly check whether the server's policy can be fulfilled and how. Additionally, in the presence of a privacy policy, Alice's agent would reply with a counter-request asking the bookshop to provide a certificate. Again, the common vocabulary would allow the bookshop to understand the request, which would be accepted since the certificate is not protected by any policy, and as a consequence Alice would finally deliver her credit card number and have the book delivered.

The availability of a framework capable to enforce access control and negotiations automatically given the two policies has remarkable consequences on privacy as well as usability. On the one hand a direct intervention of the user in the decision process would be required less frequently, since the user's decision would be already embedded to some extent into the policies (s)he defines: therefore sensitive resources would (or would not) be disclosed without necessarily

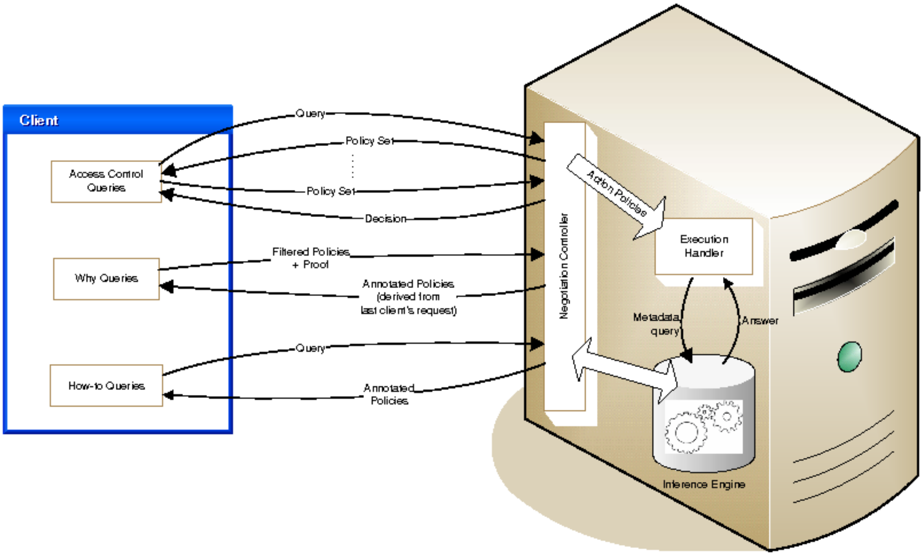


Fig. 1. Protune's architecture

asking the user each and every time. On the other hand, such usability improvement may encourage users to refine their policies by specifying articulated (eventually attribute-based) policies, thereby improving privacy guarantees.

4.3.3 Protune's Policy Language and Framework

In order to support assisted credential dislosure and handle negotiations (when needed), Protune's architecture comprises negotiation agents both on server side and on client side, as illustrated by Fig 1. Each agent reasons about access control or information disclosure policies to interpret the requests of the other peer and select possible negotiation actions.

Protune's policy language is a logic programming language enhanced with an object oriented syntax. For example, the rule that allows to buy a book by giving a credit card could be encoded with a set of rules including:

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

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

where $C.expiration : Exp$ is an O.O. expression meaning that Exp is the value of C's attribute *expiration*. Protune policies may use and define different categories of predicates:

- *decision* predicates, used to specify a policy's outcome, such as $allow()$ in the above example;[3]
- *provisional* predicates, that are meant to represent actions as described below;
- *abbreviation* predicates, defining useful abstractions such as $valid\_credit\_card$.

Protune supports two pre-defined provisional predicates: *credential* and *declaration.* An atom $credential(X)$ is true when an object X representing an X.509 credential is stored in the current negotiation state. A peer may make $credential(X)$ true on the other peer by sending the corresponding credential; this is the action attached to this particular provisional predicate. Predicate *declaration* is analogous but its argument is an unsigned semi-structured object similar to a web form that, for example, can be used to encode a traditional password-based authentication procedure as in:

$$\begin{array}{l} authenticated \leftarrow \\ \quad declaration(D),\ valid\_login\_data(D.username, D.password). \end{array}$$

When a set of rules like the above one is disclosed by a server in response to a client's request, the client—roughly speaking—works back from $allow(Request)$ looking for the credentials and declarations in its portfolio that match the conditions listed in the rules' bodies. In logical terms, the selected credentials and declarations (represented as logical atoms) plus the policy rules should entail $allow(Request)$: this is called an *abduction problem* by the automated reasoning community. After receiving credential and declarations from a client, a server checks whether its policy is fulfilled by trying to prove $allow(Request)$ using its own rules and the new atoms received from the client, as in a standard *deduction problem.*

When a client enforces a privacy policy and issues a counter-request as in Alice's scenario, the roles of the two peers are inverted: the client plays the role of the server and viceversa. For example, the client may publish rules governing credit card release such as:

$$\begin{array}{l} allow(release(C)) \leftarrow credit\_card(C),\ bbb\_member(Server), \ldots \\ bbb\_member(Server) \leftarrow\ credential(BBB),\ BBB.issuer ='' BBB\_CA'', \ldots \end{array}$$

Abbreviation predicates define in a machine understandable way the meaning of the conditions listed in rule bodies (unlike XACML *contexts*, which are black boxes). The rules defining such predicates constitute a lightweight, rule-based ontology. Abbreviation predicates are eventually defined on facts (e.g., listing the accepted credit cards, or the certification authorities recognized by a server) and/or on X.509 credentials and declarations, as in the rules for *authenticated* and *bbb_member*. Therefore, the ontologies associate each condition in a policy rule to the kind of *evidence* needed to fulfill the condition (specifying whether it should be signed or unsigned, issued by which certification authority, with what

[3] The specifications in [11] include also a predicate to sign and issue new credentials; this predicate is not yet implemented.

attributes, etc.) as well as the *actions* that need to be taken. In this way, negotiation agents can interoperate even if their policies use different abbreviation predicates.

So far, we have illustrated only information disclosure actions, such as those associated to *credential* and *declaration*. However, policies may require to execute actions that do not have negotiation purposes, such as logging some requests or notifying an administrator. New provisional predicates like these can be defined by means of *metapolicies* that specify the action associated to a predicate and the actor in charge of executing the action, for example:

$$\begin{aligned}&\mathit{log}(X) \rightarrow \mathit{type} : \mathit{provisional}.\\&\mathit{log}(X) \rightarrow \mathit{action} :' \mathit{echo}\ \$X > \mathit{logfile}'.\\&\mathit{log}(X) \rightarrow \mathit{actor} : \mathit{self}.\end{aligned}$$

where "→" connects a metaterm to its metaproperties.

Rules and ontologies may be *sensitive*. For example, a server may want to publish which credit cards it accepts, but not the list of username and passwords encoded by predicate *valid_login_data*. As another example, in a social network scenario a rule such as

$$\mathit{allow}(\mathit{download}(\mathit{pictures})) \leftarrow \mathit{best\_friend}$$

may have to be protected, because in case of a denial it may reveal to a friend that he or she is not considered as a *best* friend. The sensitivity level of predicates and rules is defined with metapolicies, e.g. by means of metafacts like

$$\mathit{valid\_login\_data}(X, Y) \rightarrow \mathit{sensitivity} : \mathit{private}.$$

Such metapolicies drive a *policy filtering* process that selects relevant rules (for efficiency), and removes sensitive parts if needed. The first definition of policy filtering [11] performed also partial evaluation w.r.t. the available facts. The current implementation does no partial evaluation anymore because (i) it may significantly increase the size of the messages exhanged during negotiations, and (ii) it destroys much of the structure of the policies thereby making the explanation facility (illustrated later on) much less effective.

Metapolicies are also used for other purposes, such as specifying atom verbalizations (see Section 4.3.4), controlling *when* actions are to be executed, and more generally driving negotiations in a declarative way. Metapolicies are an effective declarative way of adapting the framework to new application domains by means of activities much closer to configuration than general program encoding, thereby reducing deployment efforts and costs. For more details on the metapolicy language and its possible uses see [11] and REWERSE deliverable I2-D2 reachable from `http://rewerse.net`. Such documents illustrate also the facilities for integrating legacy software and data.

4.3.4 Explanations: Protune-X

Even with a policy with relatively few rules it could be hard for a common user—with neither a general training in Computer Science nor a specific knowledge of mechanisms and formats of the system—to understand what is actually required to access a certain service. Even more, a denial that results simply in a *no* does not help a user to see what has gone wrong during an acknowledgment process and hence may discourage new users from using a system. Therefore, a policy framework such as Protune would be effective only if it provides some explanation facilities that increase the user's awareness and control over a policy and provide a means to ask the system why a certain acknowledgment has been denied or granted.

Protune-X, the explanation facility of Protune, plays an essential role in improving user awareness about—and possibly control over—the policy enforced by a system. Protune-X is also a major element of Protune's *cooperative enforcement* strategy: the explanation system is meant to enrich the denials with information about how to obtain the permissions (if possible) for the requested service or resource.

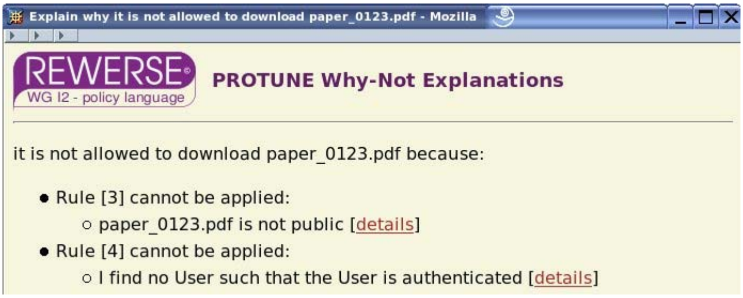
For this purpose four kinds of queries are supported: *How-to* queries provide a description of a policy and may help a user in identifying the prerequisites needed for fulfilling the policy. How-to queries may also be used to verify a complex policy. *What-if* queries are meant to help users *foresee* the results of a hypothetical situation, which may be useful for validating a policy before its deployment. Finally, *why* and *why-not* queries explain the outcome of a concrete negotiation (i.e. provide a *context-specific* help). Why/why-not queries can be used both by end users who want to understand an unexpected response, and by policy administrators who want to diagnose a policy.

Some of the major desiderata that guided the design of Protune-X are:

- *Explanations should not increase significantly the computational load of the servers.* The explanation-related processes have not to be interwoven with the reasoning process of Protune. On the contrary, it would be desirable that the server simply provides the relevant piece of information (rules and facts) whenever an explanation is demanded and the client has the burden to produce it. For this reason explanations are produced in our approach by a distinct module, ProtuneX, which operates client-side.
- *Almost no further effort should be added to the policy instantiation phase.* This is achieved by exploiting *generic heuristics* as much as possible. For example *how-to* explanations exploit the *actor* meta-attributes defined in the metapolicy to distinguish automatically the prerequisites that should be satisfied by users from the conditions that are locally checked by the server. In most cases, the only extra effort needed for enabling explanations consists in writing verbalization metarules in order to specify how single, domain-specific atoms have to be rendered, e.g.:

$$passwd(X, Y) \rightarrow verbalization : Y \ \& \ \text{“}is\ the\ correct\ password\ of\text{''} \ \& \ X.$$

- *Explanation have to be presented in manageable pieces.* An acknowledgment process is essentially an attempt to show that some (state or provisional) facts satisfy/not satisfy a policy. This proof generally consists of an *AND-OR* tree where each node is a goal, *OR*-alternatives represent the different


Explain why it is not allowed to download paper_0123.pdf - Mozilla
REWERSE
WG I2 - policy language

PROTUNE Why-Not Explanations

it is not allowed to download paper_0123.pdf because:

- Rule [3] cannot be applied:
  - paper_0123.pdf is not public [details]
- Rule [4] cannot be applied:
  - I find no User such that the User is authenticated [details]

Fig. 2. A ProtuneX screenshot

rules that apply to that goal. Finally, *AND*-edges are the subgoals in the body of each rule. This structure cannot be easily captured all in a single view, therefore ProtuneX represents it by means of linked web-pages. Each web-page represents a view on a single goal and the rules that apply to it. Web-pages are linked in order to form a tree that reproduces the structure of the proof.

– *Explanations should be presented in a user-friendly format.* ProtuneX is meant to present explanations in natural language with the help of verbalization metarules.
– *Explanations should support so-called second generation features.* Such features include methods to highlight relevant information while pruning irrelevant parts and make easier to focus on the paths that do not match the expectations of the user.

In the following we illustrate some of these second-generation features by means of examples taken or adapted from the on-line demo. For a deeper discussion and a more complete description of Protune-X the reader is referred to [12].

A typical why-not explanation for a failed negotiation is an HTML hypertext whose first page may look like the screenshot in Fig. 2. The explanation may look different depending on the causes of failure. In Fig. 2 the negotiation fails because the paper is not public and the user released an invalid ID credential; if the ID credential were valid, then other conditions in the body of the rule corresponding to the second item would become relevant to explain the failure and would appear in the explanation, as in

```
– Rule [4] cannot be applied:
   • J. Smith is authenticated [details]
  but
   • There is no Subscription such that J. Smith subscribed the
     Subscription [details].
```

Note that the same rule can be explained in a completely different way depending on the context. This is an example of irrelevant information pruning, that results from another generic heuristic adopted by Protune-X. It exploits the metarules identifying so-called *blurred* predicates, that is, predicates whose definition is not communicated because the predicate is either sensitive or too large to be transmitted efficiently over the network. Such predicates are not (completely) evaluated, therefore in selected cases they cannot be responsible for success or failure. A few more features can be illustrated via the following explanation item:

```
– Rule [6] cannot be applied:
    • c012 is an id whose name is J. Smith and issuer is myCA
  but
    • myCA is not a recognized certification authority [details].
```

Here the first bullet covers *several* atoms in the body of Rule [6], whose internal format looks like

$$\ldots,\ \mathit{credential}(C),\ C.\mathit{name}{:}\mathit{User},\ C.\mathit{issuer}{:}\mathit{CA},\ \ldots$$

In this case the variables are bound to constants `c012`, '`J. Smith`', `myCA` because there is a unique answer substitution for this group of atoms; this heuristics is called *unique answer propagation.* Moreover the group of atoms is verbalized in one phrase because the group constitutes a so-called *cluster.* These heuristics enhance readability by producing text about concrete entities (as opposed to variables with unspecified values) and referring to structured objects through their properties (`name` and `issuer`) rather than internal handlers (`c012`) that are meaningless to users.

However, if the user had provided a credential that had not been recognized as an identifier, the resulting explanation would have been

```
– Rule [6] cannot be applied:
    • I find no Credential such that the Credential is an id [details]
```

This is another example of pruning irrelevant information, if an object of a certain type is not found, as an id credential in this case, it is not relevant to report its properties.

The explanation hypertext can be navigated by clicking on the `[details]` links, that give more details about why the corresponding condition succeeds or fails. Note that this presentation technique combines local information (the rules that directly apply to a specific condition) with global information (which conditions eventually succeed, which of them fail, which answer substitutions are returned) that together describe a *set* of alternative (possibly incomplete or failed) proof *attempts.* For example, in case the types of subscription that allow to download the paper did not match the ones owned by J. Smith, we can obtain the following explanation.

```
– Rule [3 ] cannot be applied:
    • J. Smith is authenticated [details]
  but the following conditions cannot be simultaneously satisfied:
    • J. Smith subscribed some Subscription [Subscription = basic
      computer pubs] [Subscription = basic law pubs]
    • paper_0123.pdf is available for the Subscription [Subscription =
      gold subscription] [Subscription = complete computer pubs]
```

As you can see, the conditions *J. Smith subscribed some Subscription* and *paper_0123.pdf is available for the Subscription* do not have a common answer, therefore ProtuneX states that they cannot simultaneously satisfied. But, as they singularly succeed, ProtuneX provides a global view on the their possible results. So the user can more easily follow the paths that do not match the user expectations and focus more rapidly on the pages of interest.

4.3.5 The Engine

Protune can be entirely compiled onto Java bytecode. Network communications and the main flow of control for negotiations are implemented directly in Java, while reasoning (including filtering) is implemented in TuProlog, a standard Prolog that can be compiled onto Java bytecode.

Figure 3 shows the overall algorithm for a single negotiation step implemented within the Protune system.

- $rfp \equiv$ Received filtered policy
- $s \equiv$ Negotiation state
- $rn \equiv$ Received notifications
- $g \equiv$ Overall goal
- $op \equiv$ Other peer
- $ta \equiv$ Termination Algorithm
- $ass \equiv$ Action Selection Strategy

```
add(rfp, s)
add(rn, s)
Action[] la = extractLocalActions(g, s)
while(la.length != 0)
    Notification[] ln = execute(la)
    add(ln, s)
    la = extractLocalActions(g, s)
if(prove(g, s))
    send(SUCCESS, op)
    return
if(isNegotiationFinished(s, ta))
    send(FAILURE, op)
    return
Action[] ua = extractUnlockedExternalActions(g, s)
Action[] aa = selectActions(ass, ua, s)
Notification[] sn = execute(aa)
FilteredPolicy sfp = filter(g, s)
add(sfp, s)
add(sn, s)
send(sfp, op)
send(sn, op)
```

Fig. 3. Negotiation algorithm pseudocode

At each negotiation step a peer P_1 sends another peer P_2 a (potentially empty) filtered policy rfp and a (potentially empty) set of notifications rn, respectively stating the conditions to be fulfilled by P_2, and notifying the execution by P_1 of any actions it was asked for. As soon as P_2 receives this information, it adds it to its negotiation state.

Then P_2 processes its local policy in order to identify the local actions that can be performed taking into account the new information received. When such local actions are performed, other local actions may become ready for execution: this is the case e.g., if the instantiation of a variable is a prerequisite for the execution of an action and the instantiation of this variable is (part of) the result of another action's execution like in the following example, where the execution of *action*1 makes *action*2 ready for execution.

$$\ldots \leftarrow action1(X), action2(X).$$

$$action1(\_) \rightarrow actor : self.$$
$$action1(\_) \rightarrow execution : immediate.$$

$$action2(\_) \rightarrow actor : self.$$
$$action2(X) \rightarrow execution : immediate \leftarrow ground(X).$$

For this reason local action selection and execution are performed in a loop, until no more actions are ready to be executed. The need for iteration was overlooked in [11] and is documented in this chapter for the first time.

After having performed all possible local actions the local policy is processed in order to check whether the overall goal of the negotiation is fulfilled. If this is the case, a message is sent to P_1 telling that the negotiation can be successfully terminated. Otherwise the Termination Algorithm is consulted in order to decide whether the negotiation should continue or fail.

If the negotiation is not yet finished, then two processes have to be performed

- It is P_2's turn to filter its local policy and collect all items that have to be sent back to P_1;
- P_2 has to decide which of the actions whose execution has been requested by P_1 will be performed. Therefore, it processes its local policy and the (last) filtered policy received from P_1 in order to identify such actions. Notice that only actions such that the policies protecting them are fulfilled (*unlocked actions*) are collected.

Unlocked actions represent potential candidates to execution, i.e., those actions which can be performed according to P_2's local policy and its current negotiation state. However, just a subset of them will be actually performed, namely the one selected by the Action Selection Function. At each step of the negotiation, Protune builds an *AND-OR* tree with all the actions (e.g., information disclosure) that must be performed in order to advance the negotiation. This *AND-OR* tree is passed to a class implementing an Action Selection Function. Such a class can be custom and it just needs to follow an open API [4]. Protune provides out-of-the-box a "relevant" strategy that performs in parallel those actions required to advance the negotiation. We are also working on strategies based on preferences defined by the user between pairs of actions (e.g., *it is preferred to provide information related to my credit card than to my bank account*) and use them at run-time. "Good" negotiation strategies are discussed in [25,5].

Finally, the filtered policy and the notifications of the performed external actions are added to the negotiation state and sent to P_1.

[4] Cf. `http://www.l3s.de/~olmedilla/policy/doc/javadoc/org/policy/strategy/ActionSelectionStrategy.html`

4.3.6 Demo: Policy-Driven Protection and Personalization of Web Content

Open distributed environments like the World Wide Web offer easy sharing of information, but provide few options for the protection of sensitive information and other sensitive resources. Furthermore, many of the protected resources are not static, but rather generated dynamically, and sometimes the content of a dynamically generated web page might depend on the security level of the requester. Currently these scenarios are implemented directly in the scripts that build the dynamic web page. This typically means that the access control decisions that can be performed are either simple and inflexible, or rather expensive to develop and maintain. Moreover it is commonly accepted that access control and application logic should be kept separate, as witnessed by the design of policy standards such as XACML and the WS-* suite. Frameworks like Protune provide a flexible and expressive way of specifying access control requirements.

We have integrated Protune in a Web scenario capable of advanced decisions based on expressive conditions, including credential negotiation to establish enough trust to complete a transaction while obtaining some privacy guarantees on the information released [11]. We have developed a component that is easily deployable in web servers supporting servlet technology (we currently support Apache Tomcat), which adds support for negotiations and policy reasoning. It allows web developers to protect static resources by assigning policies to them. In addition to protection of static content, it also allows web developers to generate parts of dynamic documents based on the satisfaction of policies (possibly involving negotiations). We provide an extension to the web design tool Macromedia Dreamweaver in order to help web designers to easily and visually assign policies to their dynamic web pages[5].

A live demo is publicly available[6] as well as a screencast[7].

4.3.7 Experimental Evaluation

In order to evaluate the performance of Protune we first focused on its efficiency in carrying out negotiations. To this aim we measured the duration of each step of the negotiation algorithm described in Section 4.3.5 with a profiling tool we built exploiting the `log4j` [8] utility by the Apache foundation.

In the absence of large bodies of complex formalized policies, we further 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.

[5] As described in `http://skydev.l3s.uni-hannover.de/gf/project/protune/wiki/admin/?pagename=Integration+with+Dreamweaver`

[6] `http://policy.l3s.uni-hannover.de/`

[7] `http://www.viddler.com/olmedilla/videos/1/`. We recommend viewing it in full screen.

[8] `http://logging.apache.org/log4j/`

Table 2. Overall reasoning and network time (msec)

			Reasoning time				Network time			
3 steps			Definitions				Definitions			
			1	2	3	4	1	2	3	4
	Literals	1	8 + 6 + 6	5 + 4 + 4	5 + 4 + 4	5 + 10 + 4	10	7	7	7
		2	5 + 4 + 5	5 + 4 + 4	5 + 4 + 10	5 + 3 + 4	9	7	7	6
		3	5 + 4 + 4	5 + 3 + 4	5 + 4 + 4	5 + 4 + 4	7	7	6	7
		4	5 + 10 + 5	5 + 3 + 5	5 + 3 + 4	5 + 4 + 4	7	7	7	7
5 steps			Definitions				Definitions			
			1	2	3	4	1	2	3	4
	Literals	1	16 + 20 + 34	34 + 54 + 50	81 + 93 + 111	199 + 173 + 208	14	17	18	18
		2	35 + 40 + 41	66 + 120 + 91	165 + 241 + 206	397 + 445 + 409	16	16	18	19
		3	42 + 78 + 56	116 + 212 + 161	291 + 481 + 347	646 + 867 + 719	17	17	19	19
		4	75 + 105 + 77	189 + 365 + 222	470 + 789 + 560	1012 + 1445 + 1173	17	17	20	21
7 steps			Definitions				Definitions			
			1	2	3	4	1	2	3	4
	Literals	1	65 + 37 + 47	196 + 100 + 63	1059 + 394 + 160	1922 + 893 + 230	19	12	20	14
		2	187 + 91 + 53	771 + 736 + 147	4423 + 3701 + 617	8526 + 12065 + 3030	12	12	14	14

Table 3. Realistic experiments for Protune core (left), and Protune-X performance (right)

Reasoning time	Network time
6 + 35 + 21	11
1 + 2 + 4	8
5 + 23 + 17	8
1 + 2 + 5	7
1 + 2 + 4	7

pol. size	output size	processing time	page rate	page squared rate
18	10	400 ± 70	40	4
35	20	1710 ± 60	85	4.3
71	22	2100 ± 50	95	4.3
42	31	3095 ± 31	99	3.2
40	32	3760 ± 40	117	3.7
42	35	3100 ± 40	88	2.5
40	41	6540 ± 130	159	3.9
39	41	6130 ± 30	150	3.6
59	42	5100 ± 60	121	2.9
83	46	6000 ± 60	130	2.8
57	50	8030 ± 90	160	3.2
109	63	20140 ± 110	319	5.0

Finally we assembled the components described above in a package which is freely available at `http://skydev.l3s.uni-hannover.de/gf/project/protune/wiki/?pagename=Evaluation`.

We ran a first set of experiments with realistic policies inspired by our reference scenarios. The results, reported in the leftmost part of Table 3, show that in these cases the system's performance is 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 1st level of the tree) are the credentials needed to get the resource; the 2nd level is the set of counter-requests of the client that are needed to unlock the credentials in the 1st 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 realistic scenarios). Table 2 reports the results of these experiments, some of which are interrupted after 150sec. The frontier of terminating runs touches examples with hundreds or thousands of interrelated credentials, which explains the high values for 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.

A performance evaluation of the explanation facility ProtuneX has been done on a sample of 12 tests, including both realistic and artificial policies. We have used a ProtuneX implementation designed to be run through TuProlog, the Java-based Prolog adopted in the Protune framework. Each test has been run 20 times on a computer equipped with an 2GHz Intel dual-core duo and 2GB ram. Table 3 (on the right) shows the obtained results: the first column reports the size of the policy, that is, the number of its rules and metarules; the second column reports the number of generated web pages; the third column shows for each test the mean time (in msec) occurred to generate all the web pages and the relative mean squared error.

Tests are ordered according to the size of their policy and the reader can note that it is not easy to find out regularities between size and processing time. For example, tests 2 and 3 refer to policies of approximatively the same size, but the processing time of the latter is about 10 times longer than the former's one. However, if we consider test 7, whose policy size is notably larger than 3, the number of generated web-pages is a bit bigger and accordingly the processing time is. For this reason we have reported in column 4 the page rate, that is, the ratio between processing time and the output size, this value grows up linearly with respect to the output size, showing that the processing time is approximatively quadratic in the output size (cf. column 5).

Finally, 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 to improve performances and provide a more declarative semantics than standard Prolog. Even if a precise performance evaluation has not yet been carried out, its performance is remarkably better (>10 time faster) than the TuProlog counterpart.

The Java-based implementation is still appealing due to deployment ease (it is even possible to download the user agents as signed applets). However, the above performance estimates suggest that the explanation hypertext should rather be generated incrementally during navigation. Note that the computational load for the hypertext generation is essentially confined on the client; the server only needs to disclose verbalization metarules.

4.3.8 Discussion and Conclusions

We have illustrated the policy framework Protune and its implementation, reporting some positive, preliminary performance evaluation experiments. Protune is one of the most complete frameworks according to the desiderata laid out in the literature. It makes an essential use of semantic techniques to achieve its goals. More information about Protune and the vision behind it can be found on the web site of REWERSE's working group on Policies: `http://cs.na.infn.it/rewerse/`. There, on the software page, the interested reader may find links to Protune's software and some on-line demos and videos.

Unlike other applications of Semantic Web ideas, the main challenges for Protune are related to usability rather than tuple-crunching. Protune currently tackles usability issues by (partially or totally) automating the information exchange operations related to access control and information release control, and by supporting advanced, second generation explanation facilities for policies and negotiations.

We are planning to continue the development of Protune by adding new features and improving the prototype. In particular we plan to explore variants and enhancements of what-if queries to improve policy quality. Another interesting line of research concerns support for reliable forms of evidence not based on standard certification authorities, e.g. exploiting services such as OpenId and supporting user-centric credential creation (we can already support reputation-based policies via the external call predicates [11]). Support to obligation policies is another foreseen extension. Finally, we point the interested reader to Chapter 2, where the ACE front-end for Protune is discussed. Such front-end enables to exploit the controlled natural language ACE in order to define policies. As soon as (controlled) natural language is made Protune's standard user interface, usability evaluations will be carried out as well.

Another important line of research concerns standardization. We are investigating how Protune's policies and messages can be encoded by adapting and combining existing standards such as XACML (for decision rules), RuleML or RIF (for rule-based ontologies), WS-Security (for message exchange), and so on. Concerning W3C RIF initiative, our working group has contributed with a use case about policy and ontology sharing in trust negotiation.

References

1. Anderson, A.H.: An introduction to the web services policy language (wspl). In: 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY), pp. 189–192. IEEE Computer Society, Los Alamitos (2004)
2. Anderson, A.H.: A comparison of two privacy policy languages: Epal and xacml. In: Proceedings of the 3rd ACM workshop on Secure web services, pp. 53–60. ACM Press, New York (2006)
3. Ashley, P., Hada, S., Karjoth, G., Powers, C., Schunter, M.: Enterprise privacy authorization language (epal 1.2). Technical report, IBM (November 2003)
4. Backes, M., Karjoth, G., Bagga, W., Schunter, M.: Efficient comparison of enterprise privacy policies. In: Proceedings of the 2004 ACM symposium on Applied computing, pp. 375–382. ACM Press, New York (2004)
5. Baselice, S., Bonatti, P., Faella, M.: On interoperable trust negotiation strategies. In: IEEE POLICY 2007, pp. 39–50. IEEE Computer Society, Los Alamitos (2007)
6. Becker, M.Y., Sewell, P.: Cassandra: Distributed access control policies with tunable expressiveness. In: 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004), Yorktown Heights, NY, USA, pp. 159–168. IEEE Computer Society, Los Alamitos (2004)
7. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. In: IEEE Symposium on Security and Privacy, pp. 164–173 (1996)
8. Bonatti, P., Olmedilla, D., Peer, J.: Advanced policy explanations. In: 17th European Conference on Artificial Intelligence (ECAI 2006), Riva del Garda, Italy. IOS Press, Amsterdam (2006)

9. Bonatti, P., Samarati, P.: Regulating service access and information release on the web. In: Proceedings of the 7th ACM conference on Computer and communications security, pp. 134–143. ACM Press, New York (2000)
10. Bonatti, P.A., Olmedilla, D.: Driving and monitoring provisional trust negotiation with metapolicies. In: 6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005), Stockholm, Sweden, pp. 14–23. IEEE Computer Society, Los Alamitos (2005)
11. Bonatti, P.A., Olmedilla, D.: Driving and monitoring provisional trust negotiation with metapolicies. In: 6th IEEE Policies for Distributed Systems and Networks (POLICY 2005), Stockholm, Sweden, pp. 14–23. IEEE Computer Society, Los Alamitos (2005)
12. Bonatti, P.A., Olmedilla, D., Peer, J.: Advanced policy explanations on the web. In: 17th European Conference on Artificial Intelligence (ECAI 2006), Riva del Garda, Italy, pp. 200–204. IOS Press, Amsterdam (2006)
13. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In: 2nd IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY), pp. 18–38. Springer, Heidelberg (2004)
14. Duma, C., Herzog, A., Shahmehri, N.: Privacy in the semantic web: What policy languages have to offer. In: Eighth IEEE International Workshop on Policies for Distributed Systems and Networks-TOC (POLICY), pp. 5–8. IEEE Computer Society, Los Alamitos (2007)
15. Gavriloaie, R., Nejdl, W., Olmedilla, D., Seamons, K.E., Winslett, M.: No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In: Bussler, C.J., Davies, J., Fensel, D., Studer, R. (eds.) ESWS 2004. LNCS, vol. 3053, pp. 342–356. Springer, Heidelberg (2004)
16. Herzberg, A., Mass, Y., Michaeli, J., Ravid, Y., Naor, D.: Access control meets public key infrastructure, or: Assigning roles to strangers. In: 2000 IEEE Symposium on Security and Privacy, pp. 2–14. IEEE Computer Society, Los Alamitos (2000)
17. Kagal, L., Finin, T.W., Joshi, A.: A policy language for a pervasive computing environment. In: 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY), Lake Como, Italy, pp. 63–74. IEEE Computer Society, Los Alamitos (2003)
18. Li, N., Mitchell, J.C.: Rt: A role-based trust-management framework. In: Third DARPA Information Survivability Conference and Exposition (DISCEX III). IEEE Computer Society, Los Alamitos (2003)
19. Lorch, M., Proctor, S., Lepro, R., Kafura, D., Shah, S.: First experiences using xacml for access control in distributed systems. In: Proceedings of the 2003 ACM workshop on XML security, pp. 25–37. ACM Press, New York (2003)
20. Seamons, K.E., Winslett, M., Yu, T., Smith, B., Child, E., Jacobson, J., Mills, H., Yu, L.: Requirements for policy languages for trust negotiation. In: 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY), Monterey, CA, USA, pp. 68–79. IEEE Computer Society, Los Alamitos (2002)
21. Simon Godik, T.M.: Oasis extensible access control markup language (xacml) version 1.0. Technical report, OASIS (February 2003)
22. Tonti, G., Bradshaw, J.M., Jeffers, R., Montanari, R., Suri, N., Uszok, A.: Semantic web languages for policy representation and reasoning: A comparison of kaos, rei, and ponder. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 419–437. Springer, Heidelberg (2003)

23. Uszok, A., Bradshaw, J.M., Jeffers, R., Suri, N., Hayes, P.J., Breedy, M.R., Bunch, L., Johnson, M., Kulkarni, S., Lott, J.: Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In: 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY), Lake Como, Italy, pp. 93–96. IEEE Computer Society, Los Alamitos (2003)
24. Winsborough, W., Seamons, K., Jones, V.: Automated trust negotiation. In: DARPA Information Survivability Conference and Exposition, DISCEX 2000. Proceedings, pp. 88–102. IEEE Computer Society, Los Alamitos (2000)
25. Yu, T., Winslett, M., Seamons, K.E.: Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. ACM Trans. Inf. Syst. Secur. 6(1), 1–42 (2003)