A Uniform Framework for Regulating Service Access and Information Release on the Web

Piero A. Bonatti and Pierangela Samarati*
Dipartimento di Tecnologie dell'Informazione
Università di Milano
Via Bramante, 65, I-26013 - Crema, Italy
{bonatti,samarati}@dti.unimi.it

Abstract

The widespread use of Internet-based services is increasing the amount of information (such as user profiles) that clients are required to disclose. This information demand is necessary for regulating access to services, and functionally convenient (e.g., to support service customization), but it has raised privacy-related concerns which, if not addressed, may affect the users disposition to use network services. At the same time, servers need to regulate service access without disclosing entirely the details of their access control policy. There is therefore a pressing need for privacy-aware techniques to regulate access to services open to the network.

We propose an approach for regulating service access and information disclosure on the Web. The approach consists of a uniform formal framework to formulate—and reason about—both service access and information disclosure constraints. It also provides a means for parties to communicate their requirements while ensuring that no private information be disclosed and that the communicated requirements are correct with respect to the constraints.

*Corresponding author. Phone: +39(0)2-503.30061, Fax: +39(0)2-503.30010.

1 Introduction

We are moving towards a Globally Internetworked Infrastructure connecting remote parties through the use of large scale networks, such as the World Wide Web. Execution of activities at various levels is today based on the use of remote resources and services, and on the interaction between different, remotely located parties that may know little about each other. In such a scenario, traditional assumptions for establishing and enforcing access control regulations do not hold anymore. For instance, service requests may be presented by remote, previously unknown parties, not just by the local community of users. The assumption that a server be able to identify and authenticate each of these possible parties seems not applicable anymore. This paradigm shift is witnessed by a large body of work moving towards alternative approaches for establishing identities or properties of requesters (e.g., [2, 11, 12, 17]). These proposals are based on the use of digital certificates (or credentials), representing statements certified by given entities (e.g., certification authorities), which can be used in establishing properties of their holder (such as identity, accreditation, or authorizations). Besides users, also the traditional concept of access requests changes in this new scenario, and the separation between actions (access modes) and passive objects results limiting. A broader concept of service, encompassing actions and objects on which they are executed, seems to be needed to capture general applications, where traditional objects are just one of the parameters of the application. The overall interaction process between different parties also becomes more complex, moving towards a broader notion of negotiation [22], possibly carried out through software components such as browsers, agents, or wrappers [25, 26].

In a framework embracing this new paradigm, we can imagine parties interacting with each other in order to obtain or offer services and/or information. The process of completing a service often requires communicating information not related to the service itself, but related to additional restrictions on its execution. The certificates mentioned above are one type of such information. In addition, other uncertified declarations (i.e., not signed by any authority) are usually communicated during negotiations. For instance, we may be requested our credit card number in order to perform an electronic purchase; we may be requested to fill in a profile when using public or semipublic services (e.g., browsing for flight schedules) for the purpose of access control or service customization. Such information communication makes the picture even more complex since, on the client's side, releasing information to possibly unknown servers, and losing control on its further dissemination, is raising several concerns [10]. Debates about the need to regulate collection and dissemination of personal information is everyday news, and techniques and policies to satisfy it are being investigated [9, 20]. On the client side, too, there may therefore be the need to impose restrictions on the communication of credentials and declarations to other parties. For this purpose, a client may—like a server—require the counterpart to fulfill some requirements. For instance, a client may be willing to release a AAA membership number only to servers supplying a credential stating that the travel agent is approved by AAA.

In this paper we propose a uniform framework for regulating service access and information disclosure in an open, distributed network system like the Web. Like recent proposals (e.g., [22, 24, 28]) we depart from the classical authorization triple and from the assumption that access requesters be known at the server. We base regulations on the use of credentials—the traditional local authentication assumption being one specific case of this scenario. Going a step beyond previous proposals, we also include treatment of declarations (meaning by this any information required by the access control system that is not presented in the form of a certificate). Moreover, we provide a means for parties to regulate when and how credentials and declarations can be disclosed to other parties they may become involved with. Our framework, by providing the client with the ability to present counter-requests to servers and put restrictions on information disclosure, can be used to support privacy-related constraints of the sort advocated in recent proposals such as P3P [20] and OPS, as well as mechanized use of privacy notices publicized by servers [10]. Note that we are not interested in modeling negotiations and communications themselves [19, 22] but only in establishing regulations restricting the exchanges that can take place in negotiations, from the point of view of protecting service access and information disclosure. Issues related to term translation, involving, for instance, dictionaries, ontologies, and brokers [7, 26] are out of the scope of this paper. Also, our work addresses only information disclosure between the parties directly involved in a negotiation; the problem of controlling subsequent flow of information (to which techniques such as watermarking pertain) lies beyond the scope of this work.

The contributions of this paper can be summarized as follows. First, we present a uniform framework and model to reason about service access and information disclosure constraints. Second, we present a language for the specification of such constraints together with its formal semantics. Third, we study the problem of the communication between parties, covering both privacy protection (the access control policy being also considered private w.r.t. to all parties that do not need to know it) and enforcement of the established constraints, on both sides. For this purpose, we introduce a requirement communication mechanism based on a policy filtering and renaming process, that is proven to be correct w.r.t. the specifications. We also discuss privacy protection issues. A preliminary version of this paper appeared in [4], here we illustrate the negotiation framework and privacy issues in better details and provide proofs of the correctness and complexity results.

2 Reference scenario and basic concepts

2.1 Architecture and negotiation framework

We consider a network composed of different parties that interact with each other in order to offer services. Although interaction models are leaning toward a negotiation framework tending to depart from the traditional client-server architecture, ultimately interaction is intended for somebody to get a service, we can therefore distinguish between the *client*, which asks for the service, and the *server* which provides for the service.[1] While each party may behave interchangeably as either a client or a server at different times, with respect to a specific instance of a service request, roles are well defined and we can then distinguish, within the two parties involved, between a client and a server. The set of message exchanges (e.g., service requests and responses to them) between a client and a server is called *negotiation*. A negotiation always starts with a client sending an access/service request to a server, and it can comprise more than one service requests. Termination of a negotiation can be explicit (if a command like "done" or "logout" is considered) or implicit, e.g., assumed after a certain timeout period. A negotiation intuitively corresponds to the classical concept of session, although it may be broader and less structured than a session. For instance, in a negotiation a client may, similarly to a server, require the counterpart to fulfil some requirements (i.e., provide information) for it to proceed. Moreover, in a negotiation both parties can require the other party to provide them with certain information or digital certificates in order to proceed in their service request or fulfillment. The dialog between client and server can be carried out through software components (e.g., browsers, agents, or wrappers [26, 25]) which enforce possible restrictions on communication.

Each party has an associated *portfolio* of credentials (digital certificates) and declarations (unsigned data); in addition, each server has an associated set of *services* it provides. Release of

[1] For the sake of simplicity we restrict our scenario to interactions involving two parties, the approach can be extended and applied to cases where more parties are involved.

portfolio information and access to services is regulated by *rules* specified by the parties. More precisely, at the server side a set of rules regulate access to services, defining credentials and declarations that clients must submit. At each party (client and server), a set of rules regulates the release of declarations and credentials that may be requested from the party. Such rules, usually based on the evaluation of conditions, may also allow release of specific information only upon presentation by servers themselves of given credentials and declarations. For instance, a server may require clients to submit a credit card number in order to complete a purchase. In turn, a client, in order to release the credit card number, can require servers to provide a declaration that communication is encrypted or a certificate proving that the server is a seller accredited by a specific organization. We assume no automatic authentication to be in place. At the start of a negotiation, the server does not have any information about the client (user on behalf of which it is operating) with the exception, perhaps, of the client's IP address and analogous information usually included in the communication packets [6]. Identification and authentication requirements can be expressed through the language itself. To the purpose of regulating access and release, each party also maintains additional information, which we refer to as *state* information. State information can be interaction-specific or persistent, and can be viewed at the abstract level as a set of facts (e.g., user profiles).

The remainder of this section discusses *portfolio* and *services* in more details. Release and service access rules will be discussed in Section 4.

2.2 The portfolio

Each party's portfolio contains properties that the party can submit in order to obtain (or offer) services. Among these properties, we distinguish between *data declarations* (e.g., identity, address, and credit card number) and *credentials* (i.e., digital certificates). The difference between them lays in the fact that declarations are statements uttered by the party while credentials are statements uttered and signed (i.e., certified) by authorities trusted for making the statements [12]. As an example, the driver license number maintained as a data value at a party and communicated in a negotiation (e.g., by inserting it in a web interface form) is a declaration. A digital copy of the driver license, released by the DMV to the party, and that the party can submit to a server to prove that it has a driver license or that the DMV certifies some properties (e.g., address), is instead a credential. The portfolio may also represent views of certificates that are not actually stored at the party site but can be obtained if needed [25]. For instance, the portfolio of a student may include a description of the attendance certificate from the university she is enrolled in, regardless of whether the student has a copy of it at hand. This way, the model allows a party to refer to the set of its possible credentials without need of maintaining a copy of each of them.

Credentials must be unforgeable and verifiable. Both these goals are achieved through the use of public key encryption: each credential is signed with the *private digital signature key* K^- of the issuing authority [12], which can be successfully verified only with the corresponding *public digital signature verification key* K. Successful verification of a signature therefore proves both the fact that the credential has been released by the authority whose public key is K and that its content has not been altered. In some cases the credential content can also include the public key of the party to whom the credential was released. This key can be challenged by parties to which the credential is presented to make sure that the the credential holder is the same party to whom the credential was released. This feature is particularly important in cases where the party to which a credential is submitted cannot relate the presenter's identity (if any) stated in the credential with the one (if any) that it knows of. Examples of such cases are *accreditation certificates*, that certify properties (e.g., membership in a group) of a keyholder without necessarily identifying it, or *identity certificates*, that certify the identity of a keyholder [12]. Other examples are those those cases where client's anonymity is to be preserved and therefore no identity information is presented at the server nor is stated in the credential. The credential therefore works in a way similar to anonymous tickets bearing privileges in the paper world.

At a practical level, we view a credential as characterized by two elements: a *signed content*, and the *public digital signature verification key* to verify the signature. Note that it is important that all the information carried by a credential be included in the signed content to ensure its integrity. Any property of a credential (included the possible expiration date as well as the credential issuer) must therefore be signed with the private key of the issuer. Possibly, information can be encrypted. We can also imagine the existence of (meta)information associated with a credential, outside the signed content. Such information cannot however be trusted as being certified by the authority that signed the credential.

Although there is no specific structure for credentials, or more precisely their content, in the following we assume a *semi-structured* [1] organization of credentials that allows us to query for specific data, such as name or address in a driver license, number or expiration date in a credit card. Note that the fact that we allow reference to specific properties within a credential content does not mean that we allow to single them out in the submission process. A credential can only be submitted as a whole. Reference to specific properties within a credential is only needed for the purpose of querying its content. To refer to specific data in a credential we introduce the definition of credential term.

Definition 2.1 (Credential term) *A credential term is an expression of the form* credential_name(attribute_list)*, where* credential_name *is the name of the credential, and* attribute_list *is a possibly empty list of elements of the form "*attribute_name=value_term*", where value term is either a ground value or a variable.*

Example 2.1 *Some examples of credential terms, whose interpretation is immediate, are:* driver-license*(*name="John Doe"*),* driver-license*(*name=X*),* enrollment_certificate*(*issuer=I,student_id ="John Doe",university=U*).*

Declarations and credentials in a portfolio may be organized into a partial order $\sqsubseteq_{\mathsf{P}}$ by collecting declarations into classes representing named sets of properties (e.g., "demographic_data" or "personal_data"), and credentials into abstractions. For instance, photo_id can be seen as an abstraction for credentials driver-license, passport, and id-card; credit_card can be defined as an abstraction containing credentials visa_card, master_card, amex_card; class bank_info can be defined grouping all attributes related to a bank account. Figure 1 illustrates an example of credential and declaration abstractions regarding membership cards (for credentials) and credit card information (for declarations). [2] Classes and abstractions can be used to refer to a set of declarations or credentials as a whole. The definition of such a partial order, which can be orthogonal to the data model, can be done in different ways [13] (for instance, declarations can be organized into XML documents [1]) and is outside the scope of this paper.

2.3 Services

The functionalities offered by a server are defined by a set of services. Intuitively, each service can be seen as an application that clients can execute (e.g., reserve a flight). Services may be collected

[2] In some cases the value of certain data to be released may differ depending on given conditions, such as whom it is being disclosed to. As an example, the user-id passed by a given client to a server may be different depending on the server. To avoid the complication of dealing with different values for a same property, we assume in this case, different properties, declarations with different names, are defined at the client, one for each possible identifier.

into classes. Classes need not be disjoint and may be nested. Intuitively, classes are grouping of services (and classes thereof), which can be used to refer to a set of services as a whole. For instance, at a digital library server, a class "`browse`" may group services "`search_for`", "`view_abstract`", or "`view_toc`". Classes can be used to simplify the specification of security requirements. For instance, a single security constraint associated with service class "`browse`" can be used to restrict access to any such service. Classes are also convenient to model "gradual" access to applications. For instance, a client, guided by a web interface, can request access to class "`browse`" upon which, presented with the specific services to choose from, she can specialize her request. Note how this approach mimics, in some way, the process that clients often execute when accessing server through web interfaces.

In the following, we use the term service to indiscriminately refer to *basic* services (applications) and classes thereof. We will explicitly distinguish between basic services and classes when needed. The relationship between services and their classes introduces a partial order $\sqsubseteq_{\mathsf{SN}}$ on the set of service names SN. Basic services are minimal elements of the partial order. Figure 2(a) illustrates an example of services and classes thereof for a digital library server.

Each basic service may have attributes associated with it. Attributes play the role of formal parameters and specify inputs to the application implementing the service. Values in the domain of an attribute can be collected into ranges or abstractions partially ordered by relation $\sqsubseteq_{\mathsf{V}}$, where $x \sqsubseteq_{\mathsf{V}} y$ holds if y is a range/abstraction containing x. For instance, as illustrated in Figure 2(b), value `material` can be defined as an abstraction of the set of values `article`, `book`, `journal`, and `proceedings`, denoting the type of a publication. In the following, we use the term "value" to indiscriminately refer to both primitive values and their abstractions. Value abstractions can be used to specify security requirements applicable to different service calls. For instance, the security constraints associated with service buy with parameter `material`, will be applicable to all its specialized values. Note that this feature is particularly important in our model, where the traditional "authorization object", which can be hierarchically organized [13], is embedded as a parameter to the service.

To model partial specifications of services we introduce the concept of *service terms*, and extend to them the partial orders defined on service names and values. A specification is partial if it provides only a subset of the parameters of the services and/or provides only partial information on them (e.g., refer to abstract values).

Definition 2.2 (Service term) *A service term is an expression of the form* service_name(attribute_list), *where* service_name $\in \mathsf{SN}$ *and* attribute_list *is a, possibly empty, list of elements of the form "*attribute_name=value_term*", where* value_term *is either a ground (primitive or abstract) value or a variable.*

Definition 2.3 (Service term partial order)
Given two ground *service terms $s_1(L_1)$ and $s_2(L_2)$, we write $s_1(L_1) \sqsubseteq_{\mathsf{ST}} s_2(L_2)$ iff both the following conditions hold:* 1) *$s_1 \sqsubseteq_{\mathsf{SN}} s_2$, and* 2) *for each element "$a = V$" in L_2 there exists an element "$a = V'$" in L_1 such that $V' \sqsubseteq_{\mathsf{V}} V$.*

Example 2.2 *Some examples of service terms and relationships between them are:*
print(year=1990, pub_type=`proceedings`)$\sqsubseteq_{\mathsf{ST}}$print(pub_type=`proceedings`)
$\sqsubseteq_{\mathsf{ST}}$print(pub_type=`material`).

Each basic service may have *facets* associated with it. Facets can capture additional, or alternative (*polymorphic behavior*), functionalities of the service that can be enabled upon presentation of given credentials or declarations. For instance, at an airline server, a flight-reservation service may have facets seat_assignments or lunch_choices that provide additional features (choice of seat and food) enabled only for frequent flyer members, upon insertion of the membership number. As another example, at a digital library server, service buy may have a facet discount that applies a discount on the cover price of proceedings to all buyers who present an attendance certificate to conference.

3 Basic elements of the requirement specification language

We illustrate the basic constructs of the language that will be used to specify service access and information release rules. In the following, we use "term" to denote either a value in a given domain or a variable over it; we use $\vec{x}$ to denote a list $x_1, \ldots, x_n$ of parameters, where the number n of parameters and their type will be clear from the context. The basic predicates of the language are as follows.

1. A binary predicate symbol, credential. The first argument is a credential term (Definition 2.1). The second argument is a public key term. Intuitively, a ground atom credential(c, K) is true iff the current state contains a credential c verifiable with public key K.

2. A predicate symbol, declaration, whose argument is an element of the form "*attribute_name = value_term*". Intuitively, a ground atom declaration(d) is true iff the current state contains declaration d.[3]

3. A binary predicate symbol cert_authority, whose arguments are a certification authority and its public key. Intuitively, predicate cert_authority(CA, K_{CA}) states that the party trusts certificates signed by certification authority CA whose public key is K_{CA}. Note the importance of such a statement: a party usually accepts only certificates signed by authorities that it trusts (or chains of certificates eventually ending with them) [12].

4. A set of non predefined *state* inquiry predicates that evaluate information stored at the site (*persistent state*) or acquired during a negotiation (*negotiation state*). State predicates can also be built-ins or calls to external packages [25]; thus querying information that is not necessarily available at the site but must be retrieved through certain procedures. Negotiation specific information describes knowledge on a current negotiations, and is removed at the end of the negotiation. Example of negotiation specific information at a party are messages exchanged, and credentials and declarations that the party has received or released in the negotiation. The persistent state includes information that spans different negotiations, such as user profiles maintained at web sites. Examples of negotiation state predicates are predicates determining the current service or the credentials/declarations already submitted to the opponent party within a negotiation. A typical example of server's persistent knowledge is represented by user *profiles* containing information about different users registered at the server, such as demographic information or preferences. For instance, in negotiations in which the identity of the user represented by the client is known, the server can evaluate the profile to automatically retrieve demographic information of the user instead of requiring it every time.

5. A set of non predefined *abbreviation* predicates denoting abbreviation of requirements, e.g., declarations, credentials, and chains thereof. For instance, `photo-id` can be defined as an abbreviation for any of the elements in the set of credentials {`passport`,`driver-license`,`id-card`}

[3]For simplicity, in the following we will often use declaration with more than one argument. A declaration literal with n argument is to be interpreted as a shorthand for n declaration literals, one for each of the arguments.

(see Example 3.1). A state containing any of such credentials is then considered to contain `photo-id`.

6. A set of standard built-in mathematic predicates, including $=, \neq, <$.

The above predicates, which we refer to as *basic predicates*, constitute the basic literals that can be used in rules restricting services accessibility and portfolio disclosure. These rules will be discussed in the next section. Abbreviation rules, which have the same form at both the client and the server side, are defined as follows.

Definition 3.1 (Abbreviation rules) *An* abbreviation rule *has the form:* $p(\vec{x}) \leftarrow q_1(\vec{x_1}), \ldots, q_n(\vec{x_n})$, *where* p *is an abbreviation predicate, and* $q_i, i = 1, \ldots, n$, *is a basic predicate.*

Intuitively, abbreviation rules define "macros" that can be used as a shorthand for disjunctions or conjunctions of conditions (those in the body of abbreviation rules).

Example 3.1 *The following are examples of abbreviation rules.*

1. $\mathsf{photo\_id}(\text{name}{=}X) \leftarrow \mathsf{driver\_licence}(\text{name}{=}X)$
2. $\mathsf{photo\_id}(\text{name}{=}X) \leftarrow \mathsf{passport}(\text{name}{=}X)$
3. $\mathsf{info}(\text{hobby}{=}X) \leftarrow \mathsf{declaration}(\text{name}{=}Y),$
   $\mathsf{usr\_profile}(Y, \text{hobby}{=}X)$
4. $\mathsf{info}(\text{hobby}{=}X) \leftarrow \mathsf{declaration}(\text{preferences.hobby}{=}X)$
5. $\mathsf{principal}(P, K) \leftarrow \mathsf{cert\_authority}(P, K)$
6. $\mathsf{principal}(P, K) \leftarrow \mathsf{credential}(\mathsf{belongs\_to}(\text{issuer}{=}I,$
   $\text{principal}{=}P, \text{key}{=}K), K'), \mathsf{principal}(I, K')$

Rules 1 and 2 define photo_id *as an abbreviation denoting either* driver_licence *or* passport*. A request for a* photo_id *is then to be interpreted as a request for any of the two. Rules 3 and 4 introduce an abbreviation to refer to the* hobby *of a client, which can be obtained by querying the user's profile, if her identity is known (rule 3), or by explicit declaration by the user (rule 4). Rules 5 and 6 define the association between a principal* P *and a public key* K*, when this association is certified by a certification authority (*cert_authority *literals), which the party trusts either directly (rule 5) or through a chain of certificates (rule 6) [12].*

4 Service accessibility and portfolio disclosure rules

We illustrate the rules that regulate negotiation and client-server interaction. We distinguish two kinds of rules: *service accessibility rules*, used by servers to specify restrictions that clients must satisfy to access a service, and *portfolio disclosure rules*, used by both clients and servers to specify restrictions regulating disclosure of their portfolio. Examples refer to the rules reported in Figure 3, where predicate principal is defined as in Example 3.1, cert_authority information is assumed given, and the other predicates that do not occur in the head of any rules are state predicates, whose interpretation is immediate.

4.1 Service accessibility rules

Service accessibility rules specify under which conditions clients may access services. Conditions can be associated with any service term, including classes and partial specifications. Restrictions associated with an abstraction (e.g., print()) apply also to its more specialized terms (e.g., print(journal="CACM", year="1999")). Rules can also be specified with reference to specific facets of services, in which case their satisfaction (lack of, respectively) only affects the enabling (disabling, respectively) of the corresponding facet, not the access to the service itself. Requirements associated with facets are *optional*, and we treat them separately.

We distinguish two kinds of service accessibility rules: *prerequisites* and *requisites*. Prerequisites are conditions that must be satisfied for a service request to be taken into consideration (they do not guarantee that it will be granted); requisites are conditions that allow the service request to be successfully granted. The basic motivation for this separation is to avoid unnecessary disclosure of information from both parties, and can therefore be seen as twofold: 1) *server's privacy* and 2) *client's privacy*. Let us first discuss server's privacy. The credentials and declarations required from a client may need to satisfy certain relations with (protected) server's state information. For instance, the login and password supplied by the client must match those registered in the server state; clearly this part of the server state must not be disclosed to the client and the match can be evaluated only after the client itself has provided the information. Protecting the privacy of this information implies that such conditions (or their possible evaluation) cannot be disclosed to the client but should be controlled at the server's side, after credentials and declarations are received. Through prerequisites, it is also possible to limit the disclosure of the access control policy itself. To illustrate, suppose that a given service is to be made accessible only to users who satisfy all the following conditions: 1) are registered at the server, 2) are US residents, 3) are members of a partner association. Instead of communicating all such requirements to the client, and therefore unrestrictedly disclosing the whole policy, the server could first ask the counterpart for her login name (prerequisite); if she is not registered, there is no reason to proceed further. On the other hand—concerning the client's privacy—it is usually legitimate for the client, after submitting the requested credentials and declarations, to expect a successful completion of the service. If it weren't so, and the server could advance further requests that the client may not be willing or able to fulfill, the initial release of information from the client to the server would prove ineffective from the point of view of getting the service, resulting in an unnecessary disclosure of information (imagine the situation of a user submitting personal and financial information required for a purchase and then being denied it). It is therefore important that servers clearly indicate whether the required credentials and declarations are only a necessary (prerequisite), or also a sufficient (requisite), condition for service access (this assuming, of course, honesty of the server.)

Service prerequisite rules are composed of two parts. The first part states credentials and declarations that the client must submit to have its request considered. The second part states conditions that such credentials and declarations must satisfy for the rule's successful evaluation, and they will be checked by the server after reception of the required credentials and declarations.

Definition 4.1 (Service prerequisite rules) *A* service prerequisite *rule is a rule of the form* $\mathsf{service\_prereqs}(s(L)) \leftarrow q_1(\vec{x_1}),\ldots,q_n(\vec{x_n}) \mid p_1(\vec{y_1}),\ldots,p_1(\vec{y_m})$ *, where $s(L)$ is a service term, q_i, $i=1,\ldots,n$, is a basic predicate, and p_j, $j=1,\ldots,m$, is either a state predicate or a math built-in.*

Service requisite rules are simpler. Their body is composed of only one part, whose evaluation determines the requirements to be communicated to the client.

Definition 4.2 (Service requisite rules) *A* service requisite *rule is a rule of the form* $\mathsf{service\_reqs}(s(L)) \leftarrow q_1(\vec{x_1}), \ldots, q_n(\vec{x_n})$ *, where* $s(L)$ *is a service term and* q_i*,* $i = 1, \ldots, n$ *is a basic predicate.*

Consider the rules in Figure 3. Rules 1 and 2 define two (alternative) prerequisite conditions that must be satisfied to get access to a digital library service. Rule 1 applies to users registered as individuals, who must declare their login and password. Rule 2 applies to users affiliated with an organization that is registered as a corporate member, who must present a certificate of affiliation to one such organization.[4] Rules 3 through 7 are examples of service requisite rules. Rule 3 requires new users registering at a service to declare login, password, and affiliation, and to supply a credential issued by either ACM or IEEE stating the user's membership in the association. Rules 4 requires clients to accept a copyright agreement (operation that can be simply accomplished with a click on the "ok" button of a pop up window) in order to access the print service. Moreover, by rule 5, printing of journal articles appeared in old issues is allowed only to those users affiliated with an organization that has a subscription for the considered year. Finally, rules 6 and 7 require buyers to provide either a credit card number (purchase by an individual) or a certificate stating that the purchase is authorized by the buyer's organization subscribed to the service (corporate purchase).

Definition 4.3 (Facet requisite rules) *A* facet requisite *rule is a rule of the form* $\mathsf{facet\_reqs}(s(L), f) \leftarrow q_1(\vec{x_1}), \ldots, q_n(\vec{x_n})$ *, where* $s(L)$ *is a service term,* f *is the name of a facet associated with* s*, and* q_i*,* $i = 1, \ldots, n$*, is any of the basic predicates.*

Rule 8 in Figure 3 is an example of facet rule. It enables facet `discount` on service buy for `proceedings` material upon presentation of a certificate proving the buyer's attendance to a conference organized by an accredited organization.

Prerequisites, requisites, and facet rules, together with abbreviation rules and certification authority information discussed in the previous section, are all is needed to specify service accessibility restrictions at a server. It must be noted, however, that the support of abstractions with the semantics discussed implies that, to access a service, a client must satisfy, beside the accessibility restrictions associated with the service, also the possible restrictions specified at a more abstract level. For instance, a client requesting print(journal=`"CACM"`,year=`"2000"`) will be required both to accept the copyright agreement (rule 4) *and* to provide a certificate of affiliation with an organization that has a subscription to `CACM` for year 2000 (rule 5).

Additional predefined (meta)rules must then be considered by the access control system to enforce propagation of requisites. This propagation is made explicit by the following definition.

Definition 4.4 (Requisite propagation rule, Inh) *Let* APol *be a set of requisite rules specified at a server. The* requisite propagation *rules for a given service term* $s(L)$ *and* APol*, denoted by* $\mathsf{Inh}(s(L), \mathsf{APol})$*, are* predefined *rules of the form* $\mathsf{service\_reqs*}(s(L)) \leftarrow \mathsf{service\_reqs}(s_1(L_1)), \ldots,$ $\mathsf{service\_reqs}(s_n(L_n))$*, where* $s_1(L_1) \ldots, s_n(L_n)$ *are all and only the ground instances of the service terms that occur in* APol*, such that the variables are bounded to leaves of the value hierarchy and* $s(L) \sqsubseteq_{\mathsf{ST}} s_i(L_i)$*.*

The rules in the definition are the simplest description of the declarative semantics of inheritance, that will be needed for reasoning about the correctness of the access control process illustrated

[4]Note the importance of treating this as a prerequisite; the server can control that the organization is a registered member without communicating the list of registered members to the client [5].

in Section 5. We note however, that the actual implementation of inheritance does not rely on the rules in Definition 4.4, but on a more concise and flexible recursive formulation (see Section 5.3).

Having clarified the requisites that apply (either directly or through inheritance) to a service requests, we can now define requisite satisfaction for a given service.

Definition 4.5 (Requisite satisfaction) *A set of credentials/declarations* Info satisfies the requisites for *a service $s(L)$ w.r.t. a server state Σ iff* $\mathsf{APol} \cup \mathsf{Inh}(s(L), \mathsf{APol}) \cup \Sigma \cup \mathsf{Info} \models \mathsf{service\_reqs}{*}(s(L))$.

The formal definition (and the implementation) of prerequisite and facet requirement propagation and satisfaction is analogous.

To illustrate, with respect to the accessibility requirements in Figure 3, the server, upon reception of a direct (i.e., no previous communication) request to buy a proceedings volume, will require the counterpart to satisfy the requirements specified in order for it to be able to: enter the digital library service (rules 1- 2), access the buy service (rules 6- 7), and (optionally) be enabled for the discount facet (rule 8). This will happen assuming the negotiation does not walk down the service abstraction hierarchy. Note, however, that in case the access to a service is carried out "gradually" with the web interface access system navigating down the hierarchy (i.e., the user first accesses the digital library and then walks in the choices of (classes of) services), necessary credentials/declarations will also be requested gradually. When processing specific services, such credentials will be recognized as already in possession of the server and not requested anymore.

Note that Definition 4.5 yields the traditional closed policy (only accesses explicitly authorized are allowed). Such an option can be easily reversed (open policy) w.r.t. a service $s(L)$ by specifying a rule $\mathsf{service\_reqs}(s(L)) \leftarrow \mathtt{true}$. This provides a convenient way to combine the open and closed control policy paradigm [13].

By using the above illustrated rules therefore, servers as a counterpart for offering specific services, usually require clients to provide certain (possibly certified) information in the form of credentials and declarations, which, as discussed, are part of the client portfolio. Whether the client is willing to release such credentials and declarations is determined by portfolio release rules maintained at the client, discussed next.

4.2 Portfolio disclosure rules

At each party, release rules regulate the disclosure of declarations and credentials in the portfolio. Although it is natural to think of these rules at the client side, and for concreteness we refer our discussion to those, it is important to note that they can be enforced at any of the parties (server as well) that may release declarations/credentials. In particular, the server too will have a portfolio of declarations and credentials that it may need to release when offering services. For instance, the server of a travel agency can submit to its clients a credential that proves the fact that the travel agent is approved by AAA. For simplicity and concreteness, in the remainder of this section, we refer the discussion to client's release rules; with the note however that such rules can be enforced at any of the party when releasing information. Release rules are defined as follows.

Definition 4.6 (Release rule) *A* release *rule is a rule of the form* $\mathsf{release\_reqs}(o) \leftarrow q_1(\vec{x_1}), \ldots, q_n(\vec{x_n})$, *where o is either a credential term $c(L)$ or a declaration, and q_i, $i = 1, \ldots, n$, is a basic predicate.*

Consider the portfolio disclosure rules in Figure 3. Rule 12 restricts the release of declarations concerning credit card information (Figure 1(b)), allowing their disclosure only in the process of a buy operation and upon presentation of a nondisclosure agreement by the server.[5] Intuitively, rule 12 can be seen as an example of *need to know* principle enforcement: only at the time of a purchase does the server need to know the credit card number of the client. Rule 13 restricts the release of a card proving membership in an organization only to servers certified by that organization. This rule is an example of release rule dependent on the servers to which information is to be released. Another example of a rule of this type could be a rule specifying that the united frequent flyer number can be released only to united airlines or travel agents; where this latter requirement can be controlled, for instance, through a local classification of servers or by having the server itself submitting a credential stating that the server is a certified travel agent. As a matter of fact, a negotiation can be composed of different service requests.

By allowing release rules to explicitly refer to individual attributes, our language provides for fine grained control on the information to be released. At the same time, credential abstractions and classes of attributes (and consequent partial order) are a convenient shorthand for specifications. Similarly to requisites, pre-defined (meta)rules enforce propagation of restrictions to abstraction and class instances.

The support of release restrictions at different levels (actual certificates vs abstractions of them, and declarations vs collections of them) and the semantics that restrictions specified for an abstraction (class respectively) apply to its more specific credentials (declarations respectively) requires propagating restrictions from abstractions to more specific terms, in a way analogous to what done for services. This is made explicit by the following definition (Again, note that the implementation of inheritance does not rely on the above rules, but on a more concise and flexible recursive formulation, analogous to what seen for requisite propagation).

Definition 4.7 (Release propagation rule) *Let* RPol *be a set of release rules and let o be an information object, representing either a credential term or a declaration attribute/class. The* release propagation *rule for o and* RPol*, denoted by* $\mathsf{Inh}(o, \mathsf{RPol})$*, is a predefined rule of the form* $\mathsf{releasable}{*}(o) \leftarrow \mathsf{release\_reqs}(o_1), \ldots, \mathsf{release\_reqs}(o_n))$, *where $o_1, \ldots, o_n$ are all and only the ground instances of the terms that occur in* RPol*, such that the variables are bounded to leaves of the value hierarchy and $o \sqsubseteq_{\mathsf{P}} o_i$, for $i = 1, \ldots, n$.*

Given this, we can now state the condition under which credentials/declarations can be released.

Definition 4.8 (Releasable term) *A credential/declaration term o is* releasable *w.r.t. a party's state Σ and a set* Info *of received credentials/declarations iff* $\mathsf{RPol} \cup \mathsf{Inh}(o, \mathsf{RPol}) \cup \Sigma \cup \mathsf{Info} \models \mathsf{releasable}{*}(o)$.

5 Client-Server Interplay

Service execution is regulated by the rules enforced by the two parties. The client-server interplay, depicted in Figure 4, can be summarized as follows. Upon reception of a service request, the server reacts by asking the client for prerequisite information (determined by evaluating the service prerequisite rules), if any. If the client sends back the required prerequisites, and if they satisfy the conditions after the bars, then the server determines (based on its service rules and state) the credentials/declarations that the client must submit to get the service, and communicates them to the client. The client, in turn, evaluates the required credentials and declarations with respect to its own portfolio disclosure rules and possibly presents a counter request R' for credentials/declarations

[5] Note that the release of such an agreement by the server, might be conditioned to the enforcement of corresponding release rules at the server side.

to the server. Upon satisfaction of its request[6], the client will present the server with the requested credentials/declarations, thus gaining access to the service.

Now the question is: *how should the credential/declaration requests involved in the negotiation be formulated?* To fix the ideas, let us see the problem from the point of view of the server (the client's point of view is symmetrical). The naive way to formulate a declaration/credential request—that is, giving the client a list with all the possible sets of credentials that would enable the service—is not feasible, due to the large number of possible alternatives. In particular, *i)* the precise nature of the credentials might not be known in advance (as it happens with chains of credentials), and *ii)* in the presence of compound credential requests such as "*one* ID *and one* membership certificate from a federated association", there may be a combinatorial explosion of alternatives, as each individual request can potentially be fulfilled in many possible ways. This can be avoided by presenting the client with a compact requirement description, based on the abbreviations (such as 'ID' and 'membership card') defined by abbreviation rules, which are not fully expanded. The client evaluates these rules to find a set Info of credentials and declarations from its portfolio such that Info plus the abbreviation rules entail the abbreviated requirements—then it is guaranteed that Info suffices to enable the desired service. Similar considerations apply to the requirements formulated for *classes* of services and inherited by their subclasses and instances; combinatorial explosion of inherited alternative requirements should be avoided, and the mechanism of rule attachment is a way of achieving this goal.

However, the server cannot simply send its rules to the client. Some rules may query the server's private state information (e.g., to check whether the client is a `good_customer`). Clearly, the server should not send its private information to the client; it should then evaluate state predicates at its side. Moreover, since requisite satisfaction must ensure service access, state predicates must be evaluated *before* sending the requisite request to the client. In principle, other parts of the state, such as the client's preferences need not be hidden from the client, but they should be evaluated anyway before sending the rules to the client. The reason is that the client is not expected to bother with profile information submitted during previous interactions; profiles are maintained by the server precisely for the purpose of making client-server interactions more concise and less redundant.

On the client's side symmetrical problems arise. Since the client may ask the server for declarations or credentials before releasing parts of its portfolio, we face the problem of expressing the client's requirements in a compact way. Again, this is accomplished by exploting the abbreviation rules, which will not be fully expanded. Such abbreviations belong to the client—in general, they have nothing to do with the abbreviations used by the server. Therefore, in order to let the server understand the client's requests, the client's abbreviation rules are attached to the requests. Again, lLike for the server, each reference to the client's state is evaluated before sending the rules to the server.

In this scenario, a crucial property of the requirement request construction is that the *rule manipulation process should not change the original policy*. After selecting the "right" abbreviations to be attached to the requirements, and after the pre-evaluation of those rules, the sets of credentials/declarations Info that satisfy the requirements according to the rules sent to the other party should coincide with the sets Info that satisfy the requirements according to the original policy. To ensure this, a formal account of the policy manipulation process is needed, which we examine next. Such a formal account will also be needed to prove privacy protection results.

[6] Although in principle different request/counter-request steps can be imagined, for the sake of simplicity, and practical considerations, we assume the server not to present counter-requests but either satisfy or not satisfy the client requirement.

For brevity, in the next subsection we introduce formally the technical details of policy manipulation on the server's side, which encompasses—by analogy—also the technical aspects related to the client's side.

5.1 Server's policy filtering mechanism

The process of selecting the rules that should be sent to the client and their partial pre-evaluation yields a form of *filtering*, which is described in detail in the following. We shall prove formally that the original policy is not distorted by filtering, that is, the sets of requirements that enable a service under the original policy coincide with the requirements specified by the filtered rules (Theorem 6.1).

We first consider requisite rules. In order to define the filtering process for them, we introduce an operator Relevant that collects rules and abbreviations that—after partial evaluation—will be returned to the client. Roughly speaking, Relevant selects all the requisite rules for services $s'(L')$ more general than the given request $s(L)$, as well as all rules that define an abbreviation needed by those requisite rules or—recursively—by other needed abbreviations.

Definition 5.1 (Relevant) *Let* $s(L)$ *be a ground service term, and let* APol *be a set of rules, including both requisite and abbreviation rules. Define* $\mathsf{Relevant}(\mathsf{APol}, s(L))$ *to be the least set satisfying the following closure conditions:*

1. *The hierarchy propagation rule for* $s(L)$ *(cf. Definition 4.4) is in* $\mathsf{Relevant}(\mathsf{APol}, s(L))$.
2. *For all rules* $R = [\mathsf{service\_reqs}(s'(L')) \leftarrow Body]$ *in* APol, *and all substitutions* θ *binding all and only the variables in* L' *to leaves of the value hierarchy, such that* $s(L) \sqsubseteq_{\mathsf{ST}} s'(L')\theta$, $R\theta$ *must be in* $\mathsf{Relevant}(\mathsf{APol}, s(L))$.[7]
3. *If some rule in* $\mathsf{Relevant}(\mathsf{APol}, s(L))$ *contains an abbreviation predicate* p *in the body, then every rule in* APol *with* p *in the head is in* $\mathsf{Relevant}(\mathsf{APol}, s(L))$.

Example 5.1 *Let* APol *be the set of server rules in Figure 3, and let*
$s(L) = \mathsf{print}(\text{journal}=\textit{"CACM"}, \text{year}=\textit{"2000"})$ *be a service request.*
By condition 1, $\mathsf{Relevant}(\mathsf{APol}, s(L))$ *contains the rule*

R1) $\mathsf{service\_reqs*}(\mathsf{print}(\text{journal}=\textit{"CACM"}, \text{year}=\textit{"2000"})) \leftarrow$
$\mathsf{service\_reqs}(\mathsf{print}(\text{journal}=\textit{"CACM"}, \text{year}=\textit{"2000"}))$, $\mathsf{service\_reqs}(\mathsf{print}())$.

By condition 2, $\mathsf{Relevant}(\mathsf{APol}, s(L))$ *contains the rules*

R2) $\mathsf{service\_reqs}(\mathsf{print}(\text{journal}=\textit{"CACM"}, \text{year}=\textit{"2000"})) \leftarrow$
$\mathsf{customer\_affiliation}(A), \mathsf{subscription}(\text{subscriber}=A, \text{year}=\textit{"2000"}, \text{journal}=\textit{"CACM"})$,
$\mathsf{current\_year}(Y)$, $\textit{2000} < Y$.

R3) $\mathsf{service\_reqs}(\mathsf{print}()) \leftarrow \mathsf{declaration}(\text{copyright}=\textit{"accept"})$.

Consider now the request $s(L) = \mathsf{new\_user}()$.
$\mathsf{Relevant}(\mathsf{APol}, s(L))$ *consists of the rules:*

$\mathsf{service\_reqs*}(\mathsf{new\_user}()) \leftarrow \mathsf{service\_reqs}(\mathsf{new\_user}())$.
$\mathsf{service\_reqs}(\mathsf{new\_user}()) \leftarrow \mathsf{declaration}(\text{login}=X, \text{passwd}=Y, \text{name}=Z, \text{affiliation}=U)$,
$\mathsf{membership}(\text{name}=Z)$.
$\mathsf{membership}(\text{name}=Z) \leftarrow \mathsf{credential}(\mathsf{acm\_membership}(\text{issuer}=\textit{"CACM"}, \text{member}=Z), K)$.

[7] Operationally, θ is obtained by unifying each variable X in an equality $a = X$ of L' to the value v in the corresponding equality $a = v$ of L. If the latter equality does not exist, then $s(L) \sqsubseteq_{\mathsf{ST}} s'(L')\theta$ cannot possibly hold, and rule R should not be included in $\mathsf{Relevant}(\mathsf{APol}, s(L))$.

$\mathsf{membership}(\text{name}{=}Z) \leftarrow \mathsf{credential}(\mathsf{ieee\_membership}(\text{issuer}{=}\texttt{"IEEE"}, \text{member}{=}Z), K)$.
where the last two rules (defining membership*) are abbreviations.*

It is easy to prove—by means of a standard least fixpoint construction [16]—that the minimality requirement in Definition 5.1 can always be fulfilled, and hence Relevant is well-defined.

Proposition 5.1 *For all* APol *and* $s(L)$*, there exists a unique minimal set* $\mathsf{Relevant}(\mathsf{APol}, s(L))$ *satisfying conditions 1 and 2 of Definition 5.1.*

As we anticipated, the rules selected by Relevant must be simplified by evaluating state inquiry predicates and built-in math predicates (when possible). Relation $\rightsquigarrow_\Sigma$ defined below captures a single evaluation step, presented as rule transformation. Σ models the server's state information, that here is viewed as a set of ground atoms. Intuitively, we shall write $R_1 \rightsquigarrow_\Sigma R_2$ if R_2 is obtained from R_1 by simplifying one of the conditions in R_1's body.

Definition 5.2 ($\rightsquigarrow_\Sigma$, $\rightsquigarrow^*_\Sigma$, PartEval) *Let Σ be a set of ground state inquiry atoms, and let $H \leftarrow B, q(\vec{x}), C$ be a logic program rule, where B and C are sequences of atoms.*

We write $(H \leftarrow B, q(\vec{x}), C) \rightsquigarrow_\Sigma (H \leftarrow B, C)\theta$, if $q(\vec{x})$ satisfies one of the following conditions:

1. *$q(\vec{y})$ is a* "standardized apart variant of an atom in Σ",[8] *and θ is the* "most general unifier" *of $q(\vec{x})$ and $q(\vec{y})$;*

2. *q is a built-in mathematical atom, and $q(\vec{x})$ is ground and true.*

The reflexive and transitive closure of $\rightsquigarrow_\Sigma$ will be denoted by $\rightsquigarrow^*_\Sigma$.

Given a rule R, $\mathsf{PartEval}(R, \Sigma)$ *denotes the set of rules R' such that $R \rightsquigarrow^*_\Sigma R'$ and the body of R' contains no state inquiry predicate and no ground math atoms.*

With abuse of notation, in the following we will often use $\mathsf{PartEval}(\mathsf{APol}, \Sigma)$ as a shorthand for $\bigcup\{\mathsf{PartEval}(R, \Sigma) \mid R \in \mathsf{APol}\}$, where APol is a set of rules.

Example 5.2 *Suppose that Σ contains the facts:* $\mathsf{customer\_affiliation}(\texttt{"ACME"})$,
$\mathsf{subscription}(\text{subscriber}{=}\texttt{"ACME"}, \text{year}{=}\texttt{"2000"}, \text{journal}{=}\texttt{"CACM"})$*, and* $\mathsf{current\_year}(2000)$.
From the second rule of Example 5.1 we have
$R2 \rightsquigarrow^*_\Sigma \mathsf{service\_reqs}(\mathsf{print}(\text{journal}{=}\texttt{"CACM"}, \text{year}{=}\texttt{"2000"})) \leftarrow \mathit{2000} < \mathit{2000}$.

As another example, consider the rules for service new_user() *and abbreviation* membership *in Example 5.1. None of these rules contains a state predicate or any mathematical built-in, and we have only $R \rightsquigarrow^*_\Sigma R$, for each of such rules R.*

Note that the evaluation steps are iterated until all the atoms $q(\vec{x})$ with the specified properties have been evaluated. If some of such $q(\vec{x})$ cannot be simplified away, then $q(\vec{x})$ fails, and hence the rule must be deleted, because it is not applicable in the current state Σ.

The filtered policy consists of all the selected and partially evaluated rules of the policy.

Definition 5.3 (Policy filtering) *Let $s(L)$ be a service term,* APol *be a set of rules, and Σ be a set of ground state inquiry atoms. Define:* $\mathsf{Filter}(\mathsf{APol}, s(L), \Sigma) = \mathsf{PartEval}(\mathsf{Relevant}(\mathsf{APol}, s(L)), \Sigma)$.

[8] This means that $q(\vec{y})$ is obtained from an atom of Σ by uniformly renaming all variables to fresh variables, not occurring in $(H \leftarrow B, q(\vec{x}), C)$.

Example 5.3 *In the scenario of the last examples, if* $s(L) = \mathsf{print}(\text{journal}=\textit{"CACM"}, \text{year}=\textit{"2000"})$ *then* $\mathsf{Filter}(\mathsf{APol}, s(L), \Sigma)$ *consists of the two rules:*

$\mathsf{service\_reqs}{*}(\mathsf{print}(\text{journal}=\textit{"CACM"}, \text{year}=\textit{"2000"})) \leftarrow$
$\qquad \mathsf{service\_reqs}(\mathsf{print}(\text{journal}=\textit{"CACM"}, \text{year}=\textit{"2000"}))\ ,\ \mathsf{service\_reqs}(\mathsf{print}())\ .$
$\mathsf{service\_reqs}(\mathsf{print}()) \leftarrow \mathsf{declaration}(\text{copyright}=\textit{"accept"})\ .$

If $s(L) = \mathsf{new\_user}()$ *then* $\mathsf{Filter}(\mathsf{APol}, s(L), \Sigma)$ *coincides with* $\mathsf{Relevant}(\mathsf{APol}, \mathsf{new\_user}())$ *(cf. Example 5.1), because those rules contain neither state nor mathematical built-in predicates.*

The partially evaluated service requisite rules in $\mathsf{Filter}(\mathsf{APol}, s(L), \Sigma)$, show the relation between each service term in the head and the associated requirements. In cases where servers wish to hide the details of such correspondence, a renaming mechanism can be applied that transforms each head of the form $\mathsf{service\_reqs}(s_i(L_i))$ into an anonymous abbreviation p_i.

Definition 5.4 (Service renaming) *Let* ρ *be an injective function, called renaming function, mapping each atom* $\mathsf{service\_reqs}(s_i(L_i))$ *occurring in* $\mathsf{Relevant}(\mathsf{APol}, s(L))$ *onto some propositional symbol* p_i *not occurring in* APol*. The result of substituting* $\rho(\mathsf{service\_reqs}(s_i(L_i)))$ *for* $\mathsf{service\_reqs}(s_i(L_i))$ *in* $\mathsf{Filter}(\mathsf{APol}, s(L), \Sigma)$ $(1 \leq i \leq n)$ *is called a* service renaming of $\mathsf{Filter}(\mathsf{APol}, s(L), \Sigma)$ based on ρ, *and will be denoted by* $\mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho)$.

Example 5.4 *With reference to Example 5.3, if* ρ *renames the rules in* $\mathsf{Filter}(\mathsf{APol}, \mathsf{new\_user}(), \Sigma)$ *by mapping* $\mathsf{service\_reqs}(\mathsf{print}(\text{journal}=\textit{"CACM"}, \text{year}=2000))$ *to* $\mathsf{p0464}$ *and* $\mathsf{service\_reqs}(\mathsf{print}())$ *to* $\mathsf{p0461}$*, then* $\mathsf{service\_reqs}(\mathsf{new\_user}())$ *to* $\mathsf{p0461}$*, then* $\mathsf{RenFilter}(\mathsf{APol}, \mathsf{new\_user}(), \Sigma, \rho)$ *consists of:*

$\mathsf{service\_reqs}{*}(\mathsf{new\_user}()) \leftarrow \mathsf{p0461}\ .$
$\mathsf{p0461} \leftarrow \mathsf{declaration}(\text{login}=X, \text{passwd}=Y, \text{name}=Z, \text{affiliation}=U, \text{address}=T),$
$\qquad \mathsf{membership}(\text{name}=Z)\ .$
$\mathsf{membership}(\text{name}=Z) \leftarrow \mathsf{credential}(\mathsf{acm\_membership}(\text{issuer}=\text{“}\texttt{ACM}\text{''}, \text{member}=Z), K).$
$\mathsf{membership}(\text{name}=Z) \leftarrow \mathsf{credential}(\mathsf{ieee\_membership}(\text{issuer}=\text{“}\texttt{IEEE}\text{''}, \text{member}=Z), K).$

Now the above rules can be sent to the client as a description of the alternative requirements that enable $\mathsf{new\_user}()$*. The (infinitely many) alternative expanded requirements corresponding to the above concise formulation are all the possible ground instances of the information* $\{\text{login}=X,$ $\text{passwd}=Y, \text{name}=Z, \text{affiliation}=U, \text{address}=T, \mathsf{acm\_membership}(\text{issuer}=\text{"}\texttt{ACM}\text{''}, \text{member}=Z)\}$ *and* $\{\text{login}=X, \text{passwd}=Y, \text{name}=Z, \text{affiliation}=U, \text{address}=T, \mathsf{ieee\_membership}(\text{issuer}=\text{“}\texttt{IEEE}\text{''}, \text{member}=Z)\}$.

The process above applies to the requisites associated to a given service. The technicalities related to the filtering and renaming of the other server rules—for facet requisites and prerequisites—are very similar, with small differences which we sketch below.

Facet rules are filtered exactly as requisite rules. However, the renaming phase does not hide the facet name. Technically, each atom $\mathsf{facet\_reqs}(s_i(L_i), f)$ is renamed to $p_i(f)$, where p_i is a new predicate. Facet names are not hidden because, in a typical commercial setting, some knowledge about the facet (e.g., the fact that the facet grants a discount) may motivate the client's decision to release optional credentials. For those applications where the client should not be told which extra services are associated to the optional credentials, the previous renaming mechanism should be applied.

Prerequisite rules are treated in a slightly different way. The selection operator (corresponding to Relevant) must drop the part of the rules following the bar '|'. After that, filtering and renaming proceed in the same way. Then, as an immediate corollary of the correctness theorem for requisites (Theorem 6.1), we obtain the equivalence of the filtered/renamed rules and the "cut-off" version of the original rules, where the conditions after '|' have been dropped. As a second corollary, we have that if the entire original rules are considered, then the equivalence in the correctness theorem must be replaced by an "only if" condition, in accordance with the intuition that after prerequisite credential submission, the requested service may be denied.

5.2 Client's policy evaluation

Given the server's requirements $\mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho)$, the client searches its portfolio for a set of credentials/declarations Info such that $\mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho) \cup \mathsf{Info} \models \mathsf{service\_reqs}{*}(s(L))$, where $s(L)$ is the service requested by the client. This can be done either by constructing a top-down proof of $\mathsf{service\_reqs}{*}(s(L))$ and collecting credential and declaration atoms as they are needed, or by collecting the credential and declaration atoms after the proof trees have been constructed [23]. (see the hints on implementation in Section 5.3) Once the alternative *sets* of credentials that would grant $s(L)$, say $C_1, \ldots, C_n$, have been collected, the client prunes them by using its release rules. Recall that this is done by checking for each credential/declaration o in a C_i whether $\mathsf{releasable}{*}(o)$ can be derived using the client's release rules and state (the client may send C_i to the server only if $\mathsf{releasable}{*}(o)$ holds for all $o \in C_i$).[9] In general, the releasability proof may require credentials and/or declarations that are not already available to the client and should therefore be asked to the server (e.g., rules 12 and 13 in Figure 3). As we anticipated, to formulate the client's requests concisely without disclosing private information, the client's release rules and related abbreviations should be filtered and renamed, and the result should be sent to the server as a counter-request. Release rules filtering and renaming are thoroughly analogous to filtering and renaming of service requirement rules. The main differences are: *i)* The analogous of operator Relevant applies to release rules, and makes use of the client's credential/declaration hierarchy, rather than the service term hierarchy; *ii)* Relation $\rightsquigarrow_\Sigma$ and $\mathsf{PartEval}(\mathsf{APol}, \Sigma)$ use as Σ the state information of the client. Since the technical definitions are perfectly analogous to those for service rules, we omit them here.

Note that for how the protocol has been defined, the set of credentials/declarations requested from a client as requisites (as opposed to prerequisites) are sufficient for the client to acquire access to the service. Hence, the client is ensured (of course assuming honesty of the server) that the delivery of the requested credentials/declarations will grant him the service.

5.3 A note on implementation

We are currently developing a prototype implementation of our system. The mechanism for checking whether a given set of declarations and validated credentials enables service execution or information release is already implemented, and we are currently extending it with filtering and renaming. The implementation of inheritance and relevant rules selection uses an indexing structure corresponding to a sub-order of the term hierarchies. The index contains solely the terms that occur in the policy rules to keep it of a manageable size. To illustrate, assuming the abstraction propagation rules refer to a sub-hierarchy of service terms, consisting of all service terms for which requirements have been specified. The predicate parents is true if its first argument is the ordered list of all parents of the second argument in the sub-hierarchy. The hierarchy propagation rules are then implemented as follows:

$$\mathsf{service\_prereqs}{*}(s(L)) \quad \leftarrow \quad \mathsf{parents}([\,], s(L))\,, \mathsf{service\_prereqs}(s(L))\,.$$

[9] Any set of credentials with this property will do. Among the different criteria [27, 30] that can guide the choice, we are investigating *maximally parsimonious information release*. Intuitively, the client may simply rate the releasable sets of credentials $C_1, \ldots, C_k$, and then submit the corresponding requests to the server in order of decreasing parsimony, until the server fulfills a request. The assumption that no counter-counter requests are made by the server guarantees that this simple method yields maximal protection.

$$
\begin{array}{rcl}
\mathsf{service\_prereqs}{*}(s(L)) & \leftarrow & \mathsf{parents}([P \mid Ps], s(L)) \,, \mathsf{service\_prereqs}(s(L)) \,, \mathsf{service\_prereqs}{*}([P \mid Ps]) . \\
\mathsf{service\_prereqs}{*}([\,]) & \leftarrow & . \\
\mathsf{service\_prereqs}{*}([P \mid Ps]) & \leftarrow & \mathsf{service\_prereqs}{*}(P) \,, \mathsf{service\_prereqs}{*}(Ps) \,. \\
\\
\mathsf{service\_reqs}{*}(s(L)) & \leftarrow & \mathsf{parents}([\,], s(L)) \,, \mathsf{service\_reqs}(s(L)) \,. \\
\mathsf{service\_reqs}{*}(s(L)) & \leftarrow & \mathsf{parents}([P \mid Ps], s(L)) \,, \mathsf{service\_reqs}(s(L)) \,, \mathsf{service\_reqs}{*}([P \mid Ps]) . \\
\mathsf{service\_reqs}{*}([\,]) & \leftarrow & . \\
\mathsf{service\_reqs}{*}([P \mid Ps]) & \leftarrow & \mathsf{service\_reqs}{*}(P) \,, \mathsf{service\_reqs}{*}(Ps) \,. \\
\\
\mathsf{facet\_reqs}{*}(s(L), f) & \leftarrow & \mathsf{parents}([\,], s(L)) \,, \mathsf{facet\_reqs}(s(L), f) \,. \\
\mathsf{facet\_reqs}{*}(s(L), f) & \leftarrow & \mathsf{parents}([P \mid Ps], s(L)) \,, \mathsf{facet\_reqs}(s(L), f) \,, \mathsf{facet\_reqs}{*}([P \mid Ps], f) . \\
\mathsf{facet\_reqs}{*}([\,], f) & \leftarrow & . \\
\mathsf{facet\_reqs}{*}([P \mid Ps], f) & \leftarrow & \mathsf{facet\_reqs}{*}(P, f) \,, \mathsf{facet\_reqs}{*}(Ps, f) \,.
\end{array}
$$

We are developing our prototype with XSB [23], a Prolog system supporting goal tabulation and delay. Our choice is motivated by the following properties of XSB: *i)* It combines the advantages of top-down derivations (that are focussed on the goal to be proved and can deal with function symbols and infinite domains) and bottom-up derivations (that can be computed in polynomial time). *ii)* The tabling mechanism of XSB caches the results of previous derivations. If used appropriately, this can be very useful as one derivation can be used several times within a single session. *iii)* Some of the built-ins of XSB—originally designed for dealing with non-stratified negation—can be easily adapted to collect the sets of required credentials/declarations. This is an interesting alternative to the grammar-based approach by Winslett et al. [24], as in some cases it should reduce the amount of list cell allocations, thereby improving time and space efficiency.

6 Correctness and complexity

The correctness of the techniques introduced comprises of several aspects. A first aspect concerns the *correctness of the policy translation and filtering process.* Filtering and renaming are supposed to protect the server's privacy without changing its policy, so the equivalence of the original policy and the filtered requisites states the *correctness* of the filtering mechanism.

Intuitively, at the server the condition that must be satisfied for the service $s(L)$ to be accessible is verified if given a state Σ and a set of credentials/declarations Info, the user's policy APol, closed under inheritance by $\mathsf{Inh}(s(L), \mathsf{APol})$, entails that $s(L)$ is accessible ($\mathsf{service\_reqs}{*}(s(L))$). Formally, $\mathsf{APol} \cup \mathsf{Inh}(s(L), \mathsf{APol}) \cup \Sigma \cup \mathsf{Info} \models \mathsf{service\_reqs}{*}(s(L))$. The correctness of the approach then requires the satisfaction of this condition at the server,, must be equivalent to that performed by the client. The client, given $\mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho)$, should look for a set of credentials/declarations Info such that $\mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho) \cup \mathsf{Info} \models \mathsf{service\_reqs}{*}(s(L))$ holds. Then the client would know that by sending Info to the server, it can obtain service $s(L)$. This is stated by the following theorem.

Theorem 6.1 (Correctness of filtering/renaming) *For all* APol, $s(L)$, Σ, *renaming* $\mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho)$, *and sets* Info *of credential and declaration atoms,*

$$
\begin{aligned}
&\mathsf{APol} \cup \mathsf{Inh}(s(L), \mathsf{APol}) \cup \Sigma \cup \mathsf{Info} \models \mathsf{service\_reqs}{*}(s(L)) \\
&\qquad\qquad \iff \quad \mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho) \cup \mathsf{Info} \models \mathsf{service\_reqs}{*}(s(L)) \,.
\end{aligned}
$$

PROOF.(Sketch) We prove the equivalent thesis:

$$\mathsf{APol} \cup \mathsf{Inh}(s(L), \mathsf{APol}) \cup \Sigma \cup \mathsf{Info} \not\models \mathsf{service\_reqs}{*}(s(L)) \iff$$
$$\mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho) \cup \mathsf{Info} \not\models \mathsf{service\_reqs}{*}(s(L))\,.$$

($\Longrightarrow$) Assume $\mathsf{APol} \cup \mathsf{Inh}(s(L), \mathsf{APol}) \cup \Sigma \cup \mathsf{Info} \not\models \mathsf{service\_reqs}{*}(s(L))$. Then there exists an Herbrand model M of $\mathsf{APol} \cup \mathsf{Inh}(s(L), \mathsf{APol}) \cup \Sigma \cup \mathsf{Info}$ such that $\mathsf{service\_reqs}{*}(s(L)) \notin M$. Define another Herbrand model M' from M by renaming all atoms $\mathsf{service\_reqs}(s_i(L_i))$ in M to $\rho(\mathsf{service\_reqs}(s_i(L_i)))$, and by leaving the rest unchanged. It can be easily verified that M' is a model of $\mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho) \cup \mathsf{Info}$. Moreover, since $\mathsf{service\_reqs}{*}(s(L)) \notin M$, also $\mathsf{service\_reqs}{*}(s(L)) \notin M'$ holds, and hence $\mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho) \cup \mathsf{Info} \not\models \mathsf{service\_reqs}{*}(s(L))$.

($\Longleftarrow$) Assume $\mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho) \cup \mathsf{Info} \not\models \mathsf{service\_reqs}{*}(s(L))$. Then there exists an Herbrand model M' of $\mathsf{RenFilter}(\mathsf{APol}, s(L), \Sigma, \rho) \cup \mathsf{Info}$ such that $\mathsf{service\_reqs}{*}(s(L)) \notin M'$. Obtain M from M' as follows:

1. replace each atom $\rho(\mathsf{service\_reqs}(s_i(L_i)))$ in M' with $\mathsf{service\_reqs}(s_i(L_i))$;
2. remove all atoms with a state predicate occurring in Σ;
3. add all the atoms in Σ.

It can be easily verified that M is a model of $\mathsf{APol} \cup \mathsf{Inh}(s(L), \mathsf{APol}) \cup \Sigma \cup \mathsf{Info}$. Moreover, since $\mathsf{service\_reqs}{*}(s(L)) \notin M'$, also $\mathsf{service\_reqs}{*}(s(L)) \notin M$ holds, and hence $\mathsf{APol} \cup \mathsf{Inh}(s(L), \mathsf{APol}) \cup \Sigma \cup \mathsf{Info} \not\models \mathsf{service\_reqs}{*}(s(L))$.

∎

A second aspect of correctness concerns the fact that the policy must be respected during client-server interactions. This correctness criterion is the counterpart of access control process correctness in traditional authorization systems [13]. In our case, a correct implementation should ensure that: *1)* a party's declaration/credential term o is disclosed only if it is *releasable* as formalized by Definition 4.8; and *2)* a service request $s(L)$ is granted only if the service prerequisites and requisites are *satisfied* (cf. Definition 4.5).

A further, very important, correctness aspect concerns the protection of a party's private state. The requisites (filtered rules) communicated to a counterpart may reflect a party's current state (e.g., the filtering process may bind some rule variables to terms occurring in the state that then become visible to the other party). In general, this information flow is inevitable (independently of our approach) as information exchange is necessary for communication to take place, and some state information is supposed to be communicated. As an example, it should be possible for a server to convey the information (stored in its state) that it only accepts `Visa` and `Mastercard`. Given this, it is clear that the protection of state privacy should not be concerned with hiding the whole state, but with clearly identifying the state information conveyed in the communication. There are different possible notions of information protection, with different degrees of paranoia. A possible approach we have considered consists in determining whether it is possible for the client to rationally recostruct certain server's state predicates from the requisites it is requested to provide.[10]

[10]Another possible notion of protection we explored is based on flow analysis. Roughly speaking, if a state atom, say $p(t_1, t_2)$, is used to partially evaluate a rule during the filtering process, and if only t_1 eventually occurs in the rule communicated to the other party then the information conveyed includes $p(t_1, *)$. This represents the information that the state contains an atom whose predicate is p and whose first argument is t_2. To be more precise, to determine whether t_2 is conveyed, we check that the *occurrences* of t_2 created by unifying the rule with $p(t_1, t_2)$ are all eliminated by filtering.

We adopted the classical technique to prove protection: if the requisite may have been generated by a variety of different values for a state predicate, then the client cannot infer the actual value. For simplicity, we consider here the case where the whole state is to be protected. Intuitively, in the absence of certain variable patterns, the whole state is protected. We start by characterizing "dangerous" patterns. Intuitively, the syntactic condition checks for certain variables that may constitute a "bridge" from state atoms to filtered rules. Such variables are called *sensitive*.

Definition 6.1 *A variable x is* sensitive *in a rule R if x has at least two occurrences in R, one of which lies within an argument of a state predicate, while the other is within an argument of a non-state predicate.*

It is not hard to see that, in the absence of sensitive variables, it is always possible to rename uniformly all the atom arguments in a given state Σ without affecting the resulting set of filtered rules (because under this assumption, the variables that occur in a state predicate argument may occur *only* in state predicate atoms, which are completely removed from the rules that eventually belong to the filtered set). By this kind of argument, one gets the following result.

Proposition 6.1 *Suppose that no variable x is sensitive in any rule of a given set* APol. *Then for all service terms $s(L)$, for all states Σ and all ground atoms $p(t_1,\ldots,t_n) \in \Sigma$ there exists a state Σ' such that* $\mathsf{Filter}(\mathsf{APol}, s(L), \Sigma) = \mathsf{Filter}(\mathsf{APol}, s(L), \Sigma')$ *and for all $p(u_1,\ldots,u_n) \in \Sigma'$, $t_i \neq u_i$ $(i = 1,\ldots,n)$.*

Briefly, the information flow is characterized by collecting *i)* the state atoms used in the filtering process, *ii)* the corresponding *mgus*; then *iii)* some values in the mgu composition are replaced with wildcards (because those values eventually disappear from the filtered rule), and *iv)* the modified *mgus* are applied to the collected state atoms. The resulting set of expressions represents the information about the current state that could be rationally reconstructed if the original policy rules and the filtering process were known to the other party.

In general, however, it may be profitable or even necessary to adopt rules with sensitive variables. Such variables may be harmless (e.g. when they carry information known to the other party, such as its own profile). Then a more general approach to security enforcement proofs consists in characterizing the information flow from the current state to the filtered rules and prove that information disclosure through that flow is confined to harmless portions of the state. The details of this approach, however, are quite articulated and cannot be presented here due to space limitations.[11]

In the simplest cases, requirement and disclosure rules may be designed so as to satisfy an easy-to-check syntactic condition, which ensures full state protection, even when the other party knows the original rules in detail.

We close this section with a remark on the complexity of the evaluation process.

Theorem 6.2 *For fixed service and value hierarchies, and under the assumption that the size of $s(L)$ and the service terms in* APol *are bounded by a fixed constant, the set* $\mathsf{Filter}(\mathsf{APol}, s(L), \Sigma)$ *can be computed in time $O(n \log n)$, where $n = |\mathsf{APol}| + |\Sigma|$.*

[11] To put it briefly, there are different possible notions of disclosed information, with different degrees of paranoia. For example, according to one of the notions we studied, the information flow can be characterized by collecting i) the state atoms used in the filtering process, ii) the corresponding *mgus*; then iii) some values in the mgu composition are replaced with wildcards (because those values eventually disappear from the filtered rule), and iv) the modified *mgus* are applied to the collected state atoms. The resulting set of expressions represents the information about the current state that could be rationally reconstructed if the original policy rules and the filtering process were known to the other party.

Proof.(Sketch) The operator $\mathsf{Relevant}$, in the worst case, must consider all the rules in APol; for each such rule $R = [\mathsf{service\_reqs}(s'(L')) \leftarrow Body]$, it must find all substitutions θ over the variables of L' such that $s(L) \sqsubseteq_{\mathsf{ST}} s'(L')\theta$ (cf. Definition 5.1.(2)). Since variables can be bound only to the leaves of the value hierarchy, it can be shown that if such θ exists then it is unique, and can be obtained by unifying each variable X in an equality $a = X$ of L' to the value v in the corresponding equality $a = v$ of L. If the latter equality does not exist, then $s(L) \sqsubseteq_{\mathsf{ST}} s'(L')\theta$ cannot possibly hold.

Then the relevance check can be done as follows: i) sort the attribute lists L and L' (in time $O(|L| \log |L| + |L'| \log |L'|)$); ii) for each $a = t$ in L' check for a corresponding $a = v$ in L (since the two lists are sorted this requires only one visit to each member of L and L'); if a does not occur in L then fail; otherwise, if t is a variable then add the binding $t \mapsto v$ to θ; otherwise check whether $v \sqsubseteq_{\mathsf{V}} t$ (in constant time since the hierarchies are fixed); (overall time of the second step: $O(|L| + |L'|)$). The service terms $s'(L')$ so selected can be used also to assemble the hierarchy propagation rule (Definition 5.1.(1)). The relevant abbreviation rules can be collected in time $|\mathsf{APol}|$, by using as an index a suitable *dependency graph* that can be built in time $|\mathsf{APol}| \log |\mathsf{APol}|$. Therefore, the global cost of $\mathsf{Relevant}$ is dominated by

$$O(|\mathsf{APol}| \cdot [|s(L)| \log |s(L)| + \max\{|s'(L')| \log |s'(L')| : s'(L') \text{ occurs in } \mathsf{APol}\} + \log |\mathsf{APol}|]) .$$

By hypothesis, the size of $s(L)$ and $s'(L')$ is bounded by a constant, so the cost of $\mathsf{Relevant}$ is $O(|\mathsf{APol}| \log |\mathsf{APol}|)$.

The partial evaluation steps modeled by $\rightsquigarrow_{\Sigma}$ are *unit resolution* steps, that can be computed in time $n \log n$, where n is the size of the given set of clauses [18]. Secondly, an ordered table of state predicate names can be built in time $m \log m$, where m is the number of state predicates; the table can then be queried in time $\log m$. It follows that the rules containing state predicates or ground math atoms in the body can be removed in time $n \log m$. The global cost of partial evaluation is then dominated by $n \log n$. In turn, this dominates the cost of $\mathsf{Relevant}$, and the theorem immediately follows. ∎

7 Related work

By assuming the possibility of distributing and verifying credentials, our work builds on a large body of proposals on trust management (e.g., [2, 14]), public key infrastructures and key management systems (e.g., [17, 21]), certificate management [12], and certificate-based authorizations enforcement [8, 15]. Our work is complementary to all these proposals as it focusses on the use of such techniques rather than redefining them. Other related work lies in proposals addressing negotiation and communication aspects [22, 19]. Among them, Roscheinsen and Winograd [22] present a framework for participants to negotiate conditions establishing boundaries in their relationships. The model is based on the use of communication pacts by which parties communicate each other their expectations. However, their focus is on establishing a negotitation framework, and no control model is actually proposed. Similarly, the proposal by Minsky [19] is focussed on message exchanges and e-commerce specific constraints. The work by Blaze et al. [3] is probably the first proposal moving away from the traditional certificate-based security systems binding identities to keys and adopting authorization certificates allowing requesters to directly prove to servers that they hold credentials that authorize them to use given services. Although the work represent an important contribution, the only consideration of authorization certificates may result limiting in a complex structure such as the World Wide Web, where often credentials can be used to provide the parties with given certified information, but ultimately the decision of whether access can be granted is more complex and may depend on additional credentials, uncertified declarations, or satisfaction of conditions that can be enforced only locally. Even in subsequent proposals [8] and [15] the focus is mostly on trust management. New certification models such as SPKI [11], providing certificates for access control applications, represent a contribution towards a flexible access control system. However, this is not sufficient and, from a control language and modeling point of view, a more complete approach need to be thought of. Our proposal embraces a broader context allowing the specification and enforcement of regulations based on different types of certificates (e.g., identity, accreditation, authorization) [12], and allowing to reason about declarations and state conditions. The result is a more flexible and powerful control framework, which also allows the use of certificates for purposes broader than the one they may have been originally released for (in a way similar to how driver's license are used in paper world transactions to prove identity or address information of a party).

The work closest to ours is the work by Winslett et al. [24, 28] who was probably the first to address the problem from a broader view, investigating the application of credential-based access control regulating access at a server. However, although the use of a logic language was proposed to express regulations, no specific model or language was presented. Also, their work focussed only on protecting service access and did not investigate privacy protection (that we address by means of release rules and filtering). Upon receiving a request, a server communicates the client the involved access control rules, upon reception of which the client determines the necessary credentials that it must submit to get the service. From the server's side no filtering process is executed and access rules themselves are communicated. From the client's side, no restriction is assumed to be in place with respect to the credentials can be released. All the proposals discussed focussed on access controls, i.e., the server regulations. No considerations were made concerning possible restrictions to disclosure of information. Such a feature, possible in our model, seems instead to be needed in real applications. Subsequent work [27, 30] investigated trust negotiation issues and strategies, more or less protective of a party's privacy, that a party can apply to select credentials to submit to the opponent party in a negotiation. This work is complementary to the work presented in this paper, focussed on expressing, communicating, and enforcing, in a privacy-preserving way, service access and credential submission policies. Our work presents a model and a language for which both the client and server can establish their regulations, considering the possibility of requesting and submitting uncertified information (often used in real applications, e.g., filling in web interface forms), and investigating the policy filtering and communication process.

Finally, like recent models (e.g., [29, 13]) we adopt a logic language for the specification of access restrictions. Beside this, there is very little in common with previous logic-based proposals, applying the traditional ⟨subject,action,object⟩ paradigm.

8 Conclusions

We introduced a uniform framework and model to regulate service access and information release in large scale networks. The framework comprises a language (with formal semantics) for expressing access and release policies, and a policy filtering mechanism to let the parties exchange their requirements in a compact and privacy preserving way. We investigated the correctness and privacy preserving properties of the filtering process, whose execution has been proved polynomial in the size of the policies. We are currently developing a prototype implementation of our framework in XSB [23], a Prolog system supporting goal tabulation and delay, which is particularly efficient on the computations carried out in our framework. Many aspects have to be investigated in more detail (from negotiations and maximally parsimonious release to the different forms of privacy protection); they will be the subject of further work.

Acknowledgements

The work reported in this paper was partially supported by the European Community within the Fifth (EC) Framework Programme under contract IST-1999-11791 – FASTER project. A preliminary version of this paper appeared under the title "Regulating Service Access and Information Release on the Web" in *Proc. of the 7th ACM Computer and Communication Security Conference,* Athens, Greece, Nov. 2001 [4].

References

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Academic Press/Morgan Kaufmann, 1999.

[2] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems.* Springer Verlag – LNCS State-of-the-Art series, 1998.

[3] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proc. of 1996 IEEE Symposium on Security and Privacy*, pages 164–173, Oakland, CA, May 1996.

[4] P. Bonatti and P. Samarati. Regulating service access and information release on the web. In *Proc. of the Seventh ACM Conference on Computer and Communications Security*, Athens, Greece, 2000.

[5] Piero Bonatti and Pierangela Samarati. *Regulating Service Access and Information Release on the Web (*extended version*).* In preparation.

[6] N. Speciner C. Kaufman, R. Perlman. *Network Security.* Prentice Hall, 1995.

[7] Chen-Chuan K. Chang and Hector Garcia-Molina. Mind your vocabulary: Query mapping across heterogeneous information sources. In *Proc. of the 1999 ACM-SIGMOD*, pages 335–346, 1999.

[8] Y-H. Chu, Joan Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. Referee: trust management for Web applications. *World Wide Web Journal*, 2(3):706–734, 1997.

[9] Communication of the ACM. *Special Issue on Internet Privacy*, February 1999.

[10] Electronic Privacy Information Center. http://www.epic.org.

[11] Carl Ellison. SPKI certificate documentation. http://www.pobox.com/ cme/html/spki.html.

[12] Brian Gladman, Carl Ellison, and Nicholas Bohm. Digital signatures, certificates and electronic commerce. http://www.clark.net/pub/cme/html/spki.html.

[13] Sushil Jajodia, Pierangela Samarati, V.S. Subrahmanian, and Elisa Bertino. A unified framework for enforcing multiple access control policies. In *Proc. of the 1997 ACM-SIGMOD*, Tucson, AZ, May 1997.

[14] Rohit Khare and Adam Rifkin. Weaving a web of trust. *World Wide Web Journal*, 2(3), 1997.

[15] Ninghui Li, Joan Feigenbaum, and Benjamin Grosof. A logic-based knowledge representation for authorization with delegation. In *Proc. of the 12th IEEE Computer Security Foundations Workshop*, pages 162–174, July 1999.

[16] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, 1984.

[17] Ueli Maurer. Modeling a public key infrastructure. In *Proc. of the Fourth European Symposium on Research in Security and Privacy*, volume LNCS 1146, pages 325–350, Rome, Italy, September 1996.

[18] M. Minoux. LTUR: a simplified linear-time unit resolution algorithm for horn formulae and computer implementation. *Information Processing Letters*, 29:1–12, 1998.

[19] Naftaly Minsky and Victoria Ungureanu. A mechanism for establishing policies for electronic commerce. In *Proc. of the 18th Int. Conf. on Distributed Computing Systems (ICDCS)*, May 1998.

[20] Joseph Reagle and Lorrie Faith Cranor. The platform for privacy preferences. *Communications of the ACM*, 42(2):48–55, February 1999.

[21] Michael Reiter, Matthew Franklin, John Lacy, and Rebecca Wright. The Ω key management service. *Journal of Computer Security*, 4(4):267–287, 1996.

[22] Martin Roscheisen and Terry Winograd. A communication agreement framework for access/action control. In *Proc. of 1996 IEEE Symposium on Security and Privacy*, pages 154–163, Oakland, CA, May 1996.

[23] K. Sagonas, T. Swift, D.S. Warren, J. Freire, and P. Rao. The XSB programmer's manual, version 2.2. http://xsb.sourceforge.net, April 2000.

[24] K. E. Seamons, W. Winsborough, and M. Winslett. Internet credential acceptance policies. In *Proceedings of the Workshop on Logic Programming for Internet Applications*, Leuven, Belgium, July 1997.

[25] V.S. Subrahmanian, Sibel Adali, Anne Brink, James J. Lu, Adil Rajput, Timothy J. Rogers, Robert Ross, and Charles Ward. Hermes: Heterogeneous reasoning and mediator system. http://www.cs.umd.edu/projects/hermes.

[26] Gio Wiederhold and Michael Genesereth. The conceptual basis for mediation services. *IEEE Expert*, 12(5):38–47, Sept.-Oct. 1997.

[27] W. Winsborough, K. E. Seamons, and V. Jones. Automated trust negotiation. In *Proc. of the DARPA Information Survivability Conf. & Exposition*, Hilton Head Island, SC, USA, January 25-27 2000. IEEE-CS.

[28] M. Winslett, N. Ching, V. Jones, and I. Slepchin. Assuring security and privacy for digital library transactions on the web: Client and server security policies. In *Proceedings of ADL '97 — Forum on Research and Tech. Advances in Digital Libraries*, Washington, DC, May 1997.

[29] Thomas Y. C. Woo and Simon S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2,3):107–136, 1993.

[30] T. Yu, X. Ma, and M. Winslett. An efficient complete strategy for automated trust negotiation over the internet. In *Proceedings of 7th ACM Computer and Communication Security*, Athens, Greece, November 2000.

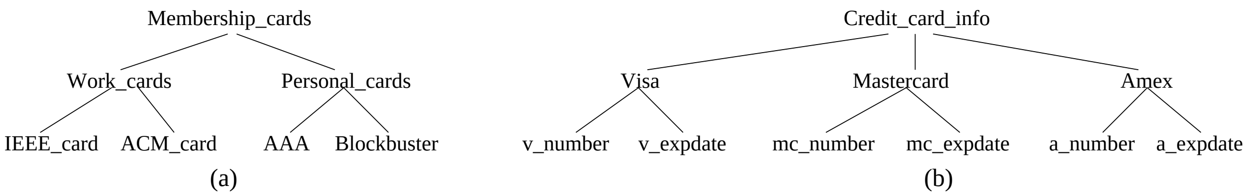

Figure 1: An example of credential (a) and declaration (b) abstractions

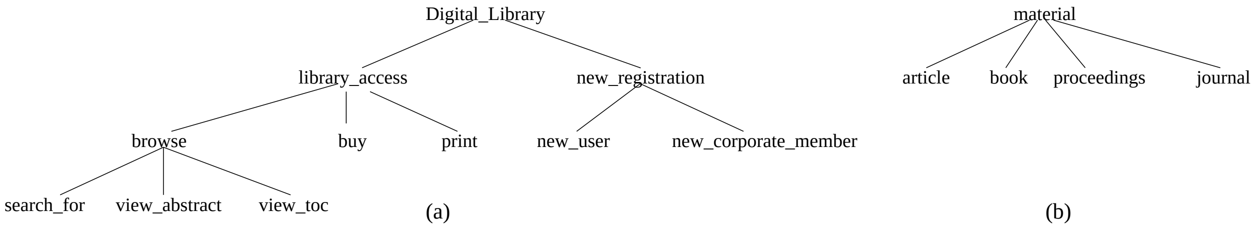

Figure 2: An example of service (a) and value (b) abstractions for a digital library server

---

Service accessibility rules (at the digital library server)

1. $\mathsf{service\_prereqs}(\mathsf{library\_access}()) \leftarrow \mathsf{declaration}(\text{login}{=}X, \text{passwd}{=}Y) \mid \mathsf{usr\_profile}(\text{login}{=}X, \text{passwd}{=}Y)$.
2. $\mathsf{service\_prereqs}(\mathsf{library\_access}()) \leftarrow \mathsf{credential}(\mathsf{affiliation}(\text{issuer}{=}I, \text{user}{=}U, \text{user\_key}{=}K_U), K_I)$,
   $\mathsf{principal}(I, K_I) \mid \mathsf{registrations}(\text{subscriber}{=}I)$.
3. $\mathsf{service\_reqs}(\mathsf{new\_user}()) \leftarrow \mathsf{declaration}(\text{login}{=}X, \text{passwd}{=}Y, \text{name}{=}Z, \text{affiliation}{=}U), \mathsf{membership}(\text{name}{=}Z)$.
4. $\mathsf{service\_reqs}(\mathsf{print}()) \leftarrow \mathsf{declaration}(\text{copyright}{=}\text{``}\mathtt{accept}\text{''})$.
5. $\mathsf{service\_reqs}(\mathsf{print}(\text{journal}{=}J, \text{year}{=}X)) \leftarrow \mathsf{customer\_affiliation}(A), \mathsf{subscription}(\text{subscriber}{=}A, \text{year}{=}X, \text{journal}{=}J)$,
   $\mathsf{current\_year}(Y), X < Y$.
6. $\mathsf{service\_reqs}(\mathsf{buy}()) \leftarrow \mathsf{declaration}(\text{credit\_card\_number}{=}X)$.
7. $\mathsf{service\_reqs}(\mathsf{buy}()) \leftarrow \mathsf{credential}(\mathsf{authorized\_to\_buy}(\text{issuer}{=}I, \text{user}{=}U, \text{user\_key}{=}K_U), K_I)$,
   $\mathsf{current\_customer}(U), \mathsf{customer\_affiliation}(I)$.
8. $\mathsf{facet\_reqs}(\mathsf{buy}(\text{material}{=}\mathtt{proceedings}, \text{conf}{=}C, \text{ass}{=}A), \text{discount}) \leftarrow \mathsf{current\_customer}(U)$
   $\mathsf{credential}(\mathsf{attendance\_certificate}(\text{issuer}{=}O, \text{attendant}{=}U, \text{conference}{=}C)K_O)$,
   $\mathsf{accredited\_organizer}(\text{company}{=}O, \text{association}{=}A), \mathsf{principal}(O, K_O)$.
9. $\mathsf{accredited\_organizer}(\text{company}{=}O, \text{association}{=}A) \leftarrow$
   $\mathsf{credential}(\mathsf{accredited}(\text{issuer}{=}A, \text{organizer}{=}O), K_A), \mathsf{principal}(A, K_A)$.
10. $\mathsf{membership}(\text{name}{=}Z) \leftarrow \mathsf{credential}(\mathsf{acm\_membership}(\text{issuer}{=}\texttt{"ACM"}, \text{member}{=}Z), K)$.
11. $\mathsf{membership}(\text{name}{=}Z) \leftarrow \mathsf{credential}(\mathsf{ieee\_membership}(\text{issuer}{=}\texttt{"IEEE"}, \text{member}{=}Z), K)$.

Portfolio disclosure rules (at the client)

12. $\mathsf{release\_reqs}(\text{credit\_card\_info}) \leftarrow \mathsf{current\_service}(\mathsf{buy}()), \mathsf{declaration}(\text{no\_disclosure}{=}\texttt{"accept"})$.
13. $\mathsf{release\_reqs}(\mathsf{membership\_card}(\text{issuer}{=}X)) \leftarrow \mathsf{credential}(\mathsf{certified\_server}(\text{issuer}{=}X, \text{server}{=}M), K)$,
    $\mathsf{current\_server}(M), \mathsf{principal}(X, K)$.

Figure 3: An example of service access and portfolio disclosure rules

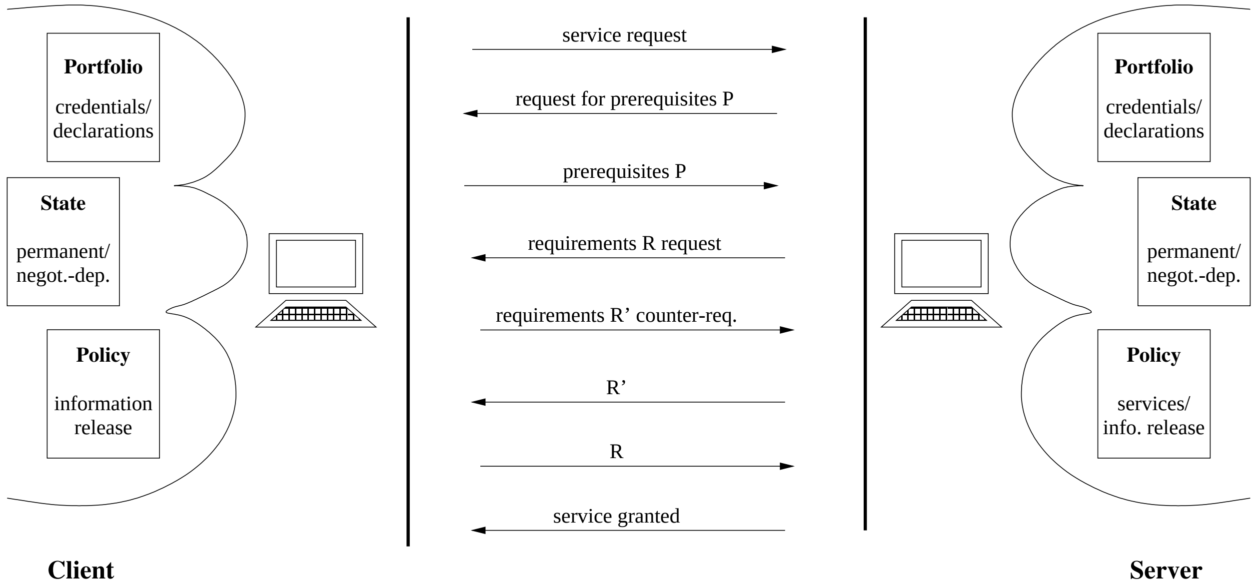

Figure 4: Client/server interplay