A Fine-Grained Access Control System for XML Documents

ERNESTO DAMIANI
Università di Milano
SABRINA DE CAPITANI DI VIMERCATI
Università di Brescia
STEFANO PARABOSCHI
Politecnico di Milano
and
PIERANGELA SAMARATI
Università di Milano

Web-based applications greatly increase information availability and ease of access, which is optimal for public information. The distribution and sharing via the Web of information that must be accessed in a selective way, such as the one involved in Electronic Commerce transactions, requires the definition and enforcement of security controls, ensuring that information will be accessible only to authorized entities. Different approaches have been proposed that address the problem of protecting information in a Web system. However, these approaches typically operate at the file system level, independently from the data that have to be protected from unauthorized accesses. Part of this problem is due to the limitations of HTML, historically used to design Web documents.

The eXtensible Markup Language (XML), a markup language promoted by the World Wide Web Consortium (W3C), is de facto the standard language for the exchange of information in Internet and represents an important opportunity to provide fine grained access control. We present an access control model to protect information distributed on the Web that, by exploiting XML's own capabilities, allows the definition and enforcement of access restrictions directly on the structure and content of the documents. We present a language for the specification of access restrictions, which uses standard notations and concepts, together with a description of a system architecture for access control enforcement based on existing technology. The result is a flexible and powerful security system offering a simple integration with current solutions.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General—*Security, integrity, and protection*; H.2.7 [**Database Management**]: Database Administration—*Security,*

Authors' addresses: Ernesto Damiani and Pierangela Samarati, Dipartimento di Tecnologie dell'Informazione, Università di Milano, Via Bramante, 65, 26013 Crema (CR), email: {damiani,samarati}@dti.unimi.it; Sabrina De Capitani di Vimercati, Dipartimento di Elettronica per l'Automazione, Università di Brescia, Via Branze 38 - 25123 Brescia - Italy, e-mail: decapita@ing.unibs.it; Stefano Paraboschi, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza L. da Vinci 32 - 20133 Milano - Italy, email: parabosc@elet.polimi.it

integrity, and protection; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security, XML

Additional Key Words and Phrases: Access control, Authorizations specification and enforcement, World Wide Web, XML documents

1. INTRODUCTION

An ever-increasing amount of information is being made available in unstructured and semi-structured form via Web sites both on corporate Intranets and on the global Internet. Web-based information interchange is particularly important in Electronic Commerce (EC) applications, where basic transactions such as vendor registration, bidding submissions, requests for quotes and contracts are increasingly realized by exchanging the appropriate digital documents. The huge success of the Web as a platform for EC and information dissemination has brought an increasing awareness of the fact that document exchange on the Internet should meet precise security requirements such as fine-grained authenticity, secrecy, non-repudiation, and access control, involving data units at the level of granularity stipulated by communicating parties. However, fully meeting these requirements through HTML-based information processing turns out to be rather awkward, due to HTML's inherent limitations. HTML provides no clean separation between the structure and the layout of a document and some of its content is only used to specify the document layout. Moreover, site designers often prepare HTML pages according to the needs of a particular browser. Therefore, HTML markup has generally little to do with data semantics. In the last few years, this situation has improved dramatically, following the standardization effort by the World Wide Web Consortium (W3C) that gave birth to the *eXtensible Markup Language (XML)* [Bray et al. 2000]. XML is a markup meta-language providing *semantics-aware* markup without losing the formatting and rendering capabilities of HTML. XML's tags capability of self-description is shifting the focus of Web communication from conventional hypertext to data interchange. While HTML was defined using only a small and basic part of SGML (Standard Generalized Markup Language: ISO 8879), XML is a sophisticated subset of SGML, designed to describe data using arbitrary tags. As its name implies, extensibility is a key feature of XML; users and applications are free to declare and use their own tags and attributes. Therefore, XML ensures that both the logical structure and content of semantically rich information is retained. XML focuses on the description of information structure and content as opposed to its presentation. Presentation issues are addressed by a separate language, XSL (XML Style Language) [Adler et al. 2001], which is also a W3C standard for expressing how XML-based data should be rendered. In addition to XML and XSL, XLink (XML Linking Language), which is a specification language to define anchors and links within XML documents, is in the process of standardization [DeRose et al. 2001]. Due to its advantages, XML is now widely accepted in the Web community, and available applications exploiting this standard include OFX (Open Financial Exchange) [CheckFree Corp 2001] to describe

financial transactions, CDF (Channel Definition Format) [Ellerman 1997] for push technologies, and OSD (Open Software Distribution) [van Hoff et al. 1997] for software distribution on the Net. This wealth of applications suggests that XML has a great potential as an exchange format for semi-structured data. The application to XML data of the latest advancement of public-key cryptography has remedied most of the security problems in communication; commercial products are becoming available (such as AlphaWorks' XML Security Suite [AlphaWorks 2001]) providing fine-grained security features such as digital signatures and element-wise encryption to transactions involving XML data as a way to meet authenticity, secrecy, and non-repudiation requirements in XML-based transactions.

The objective of our work is to complete this picture, exploiting XML's own capabilities to define and implement an authorization model for regulating access to XML documents. The rationale for our approach is defining an XML markup for a set of *security elements* describing the protection requirements of XML documents. Our security markup can be used to provide both *instance level* and *schema level* authorizations at the granularity of XML elements. Taken together with a user's identification and associated group memberships, as well as with the support for both permissions and denials of access, our security markup allows to easily express different protection requirements with support of exceptions. The enforcement of the requirements stated by the authorizations produces a view on the documents for each requester; the view includes only the information that the requester is entitled to see. A main feature of our model is that it smoothly accommodates the needs of both organization-wide policy managers and single document authors, automatically taking both into account to define who can exercise which access privileges on a particular XML document. Our notion of subject comprises identity and location; identity can include information about group or organization membership. The granularity of objects may be as fine as single elements or even attributes within XML documents. Our model includes data-dependent conditions and is open and extendable so that other enforcement conditions, such as temporal ones, could be easily added. We also present an algorithm that ensures fast on-line computation of such a view on XML documents requested via an HTTP connection. The proposed approach, while powerful enough to define sophisticated access to XML data, makes the design of a server-side *security processor* for XML documents rather straightforward. We also describe the major aspects of our Java-based implementation of the system.

1.1 Related Work

Although several projects for supporting authorization-based access control in the Web have recently been carried out, authorizations and access control mechanisms available today are at a preliminary stage. For instance, the Apache server (`www.apache.org`) allows the specification of access control lists via a configuration file (`access.conf`) containing the list of users, hosts (IP addresses), or host/user pairs, which must be allowed/forbidden connection to the server. Users are identified by user- and group-names and passwords, to be specified via Unix-style password files. By specifying a different configuration file for each directory, it is possible to define authorizations on a directory basis; files belonging to the same directory are subject to the same authorizations. Although Apache 1.2 and later also allow

to protect individual files, it is not possible to specify authorizations on portions of files. This limitation may force protection requirements to affect data organization at the file system level. For instance, a file containing data with different protection requirements will have to be split in two different files. The proposal in [Samarati et al. 1996] overcome this limitation by allowing the specification of authorizations on portions of HTML document. However, again, no semantic context similar to that provided by XML can be supported and the model remains limited. Some approaches, such as the EIT SHTTP scheme [Rescorla and Schiffman 1999], explicitly represent authorizations within the documents by using security-related HTML tagging. Every document may have associated security (meta)tags describing the authorizations on the document. While this seems to be the right direction towards the construction of a more powerful access control mechanism, due to HTML fundamental limitations these proposals cannot take into full consideration the information structure and semantics.

The development of XML represents an important opportunity to solve this problem. Proposals are under development by both industry and academia, and commercial products are becoming available which provide security features around XML. However, some of these approaches focus on lower level features, such as encryption and digital signatures [AlphaWorks 2001; Eastlake et al. 2001], on query response authentication [Devanbu et al. 2001], or on privacy restrictions on the dissemination of information collected by the server [Reagle and Cranor 1999].

Work closest to ours is represented by proposals related to the specification and enforcement of security restrictions on XML documents or using XML. We first proposed the notion of a fine-grained access control model for XML documents in [Damiani et al. 2000a; Damiani et al. 2000b], were we introduced the use of an authorization sheet associated with each XML document/DTD expressing the authorizations on the document. The approach exploiting XML own capability as each authorization sheet is itself an XML document. In this paper we extend these proposals by enriching the authorization types supported by the model, proving a complete description of the specification and enforcement mechanism, and reporting on its implementation. Among comparable proposals, Bertino et al. [Bertino et al. 2000; Bertino et al. 2001] and Gabillon et al. [Gabillon and Bruno 2001] subsequently proposed an access control environment for XML documents and some techniques to deal with authorization priorities and conflict resolution issues. A distinct, though related, line of research has been pursued by Kudo et al. [Kudoh et al. 2000], that proposed a fine-grained authorization specification language where authorizations are always associated with single elements in a document. Other related work concerns exploiting XML as a security specification language, which include the current OASIS effort to define a standard XACML (eXtensible Access Control Markup Language) (`http://www.oasis.org`), and the XrML proposal [ContentGuard 2001], an eXtensible rights Markup Language (XrML) for describing usage restrictions on digital resources.

At the same time, the security community is proceeding towards the development of sophisticated access control models and mechanisms able to support different security requirements and multiple policies [Jajodia et al. 2001; Woo and Lam 1993]. These proposals have not been conceived for semi-structured data with their flexible and volatile organization. They are often based on the use of logic languages,

which are not immediately suited to the Internet context, where simplicity and easy integration with existing technology must be ensured. In the Web, the advantages brought by the use of a sophisticated language in terms of expressive power do not seem to be justified with respect to the added complexity. Our approach expresses security requirements in syntax, rather than in logic, leading to a simpler and more efficient evaluation engine. This characteristic ensures that our proposal can be smoothly integrated in an environment for XML information processing.

The use of authorization priorities with propagation and overriding, which is an important aspect of our proposal, may recall approaches made in the context of object-oriented databases, like [Fernandez et al. 1994; Jonscher et al. 1994; Rabitti et al. 1991]. However, the XML data model is not object-oriented [Bray et al. 2000] and the hierarchies it considers represent part-of relationships and textual containment, which require specific techniques different from those applicable to ISA hierarchies in the object-oriented context.

1.2 Outline of the Paper

The paper is organized as follows. Section 2 illustrates the basic characteristics of the XML proposal. Section 3 and 4 discuss the subjects and the objects, respectively, of our authorization model. Section 5 presents the authorizations supported by the access control model for expressing security requirements on XML documents. Section 6 introduces the process for computing, at access control time, the document view to be returned to the requester and presents an algorithm for efficiently computing such a view; it also discusses data modeling issues taking into account the schema changes due to the partial view. Section 7 discusses the support of write actions. Section 8 addresses design and implementation issues and illustrates the architecture of the Access Control Processor. Section 9 presents our concluding remarks.

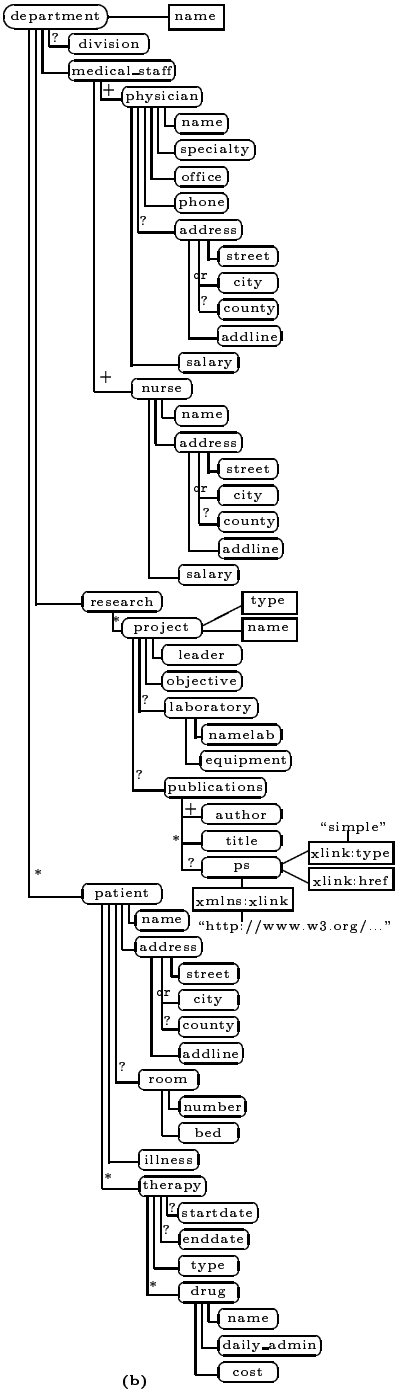
2. PRELIMINARY CONCEPTS

XML [Bray et al. 2000] is a markup language for describing semi-structured information. An XML document is composed of a sequence of nested elements, each delimited either by a pair of start and end tags (e.g., `<nurse>` and `</nurse>`) or by an empty tag (e.g., `<organization/>`). XML documents can be classified into two categories: *well-formed* and *valid*. An XML document is well-formed if it obeys the syntax of XML (e.g., non-empty tags must be properly nested, each non-empty start tag must correspond to an end tag). A well-formed document is valid if it conforms to a proper *Document Type Definition* (DTD). A DTD is a file (external, included directly in the XML document, or both) which contains a formal definition of a particular type of XML documents.

A DTD may include declarations for elements, attributes, entities, and notations. Elements are the most important components of an XML document. Element declarations in the DTD specify the names of elements and their content. The content specification may coincide with `Empty`, `Any`, or with a group of one or more subelements/groups. `Empty` means that the element has no content, where `Any` means that the element may have any content. Groups can be either sequence or choice of sub-elements and/or subgroups. Sequences of elements are separated by a comma ",", while choices are separated by the vertical bar "|". Declarations of sequence

```
<!DOCTYPE department[
  <!ELEMENT department (division?,medical_staff,research,patient*)>
  <!ELEMENT medical_staff (physician+,nurse+)>
  <!ELEMENT physician (name,specialty,office,phone,address?,salary)>
  <!ELEMENT nurse (name,address,salary)>
  <!ELEMENT research (project)*>
  <!ELEMENT project (leader,objective,laboratory?,publications?)>
  <!ELEMENT laboratory (namelab,equipment)>
  <!ELEMENT publications (author+,title,ps?)*>
  <!ELEMENT patient (name,address,room?,illness,therapy*)>
  <!ELEMENT room (number,bed)>
  <!ELEMENT therapy (startdate?,enddate?,type,drug*)>
  <!ELEMENT address (street,(city|county)?,addline)>
  <!ELEMENT drug (name,daily_admin,cost)>
  <!ELEMENT type (#PCDATA)>
  <!ELEMENT division (#PCDATA)>
  <!ELEMENT specialty (#PCDATA)>
  <!ELEMENT office (#PCDATA)>
  <!ELEMENT phone (#PCDATA)>
  <!ELEMENT salary (#PCDATA)>
  <!ELEMENT objective (#PCDATA)>
  <!ELEMENT street (#PCDATA)>
  <!ELEMENT city (#PCDATA)>
  <!ELEMENT addline (#PCDATA)>
  <!ELEMENT daily_admin (#PCDATA)>
  <!ELEMENT illness (#PCDATA)>
  <!ELEMENT equipment (#PCDATA)>
  <!ELEMENT ps ANY>
  <!ELEMENT cost (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT bed (#PCDATA)>
  <!ELEMENT startdate (#PCDATA)>
  <!ELEMENT enddate (#PCDATA)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT namelab (#PCDATA)>
  <!ELEMENT leader (#PCDATA)>
  <!ELEMENT county (#PCDATA)>
  <!ELEMENT number (#PCDATA)>
  <!ATTLIST department  name CDATA #REQUIRED>
  <!ATTLIST project   type CDATA #REQUIRED
    name CDATA #REQUIRED>
  <!ATTLIST ps  xmlns:xlink CDATA #FIXED
                "http://www.w3c.org/.../namespace"
    xlink:type (simple|extended|locator|arc) #FIXED "simple"
    xlink:href CDATA #REQUIRED>
]>
```

(a)



(b)

Fig. 1. An example of DTD (a) and the corresponding graphical representation (b).

and choices of sub-elements also describe the sub-elements' cardinality; with a notation inspired by extended BNF grammars, "`*`" indicates zero or more occurrences, "`+`" indicates one or more occurrences, "`?`" indicates zero or one occurrence, and no label indicates exactly one occurrence. XML also allows to declare elements with a *mixed* content, that is, elements containing parsable character data (`#PCDATA`), optionally interleaved with sub-elements. Attributes represent properties of elements. Attribute declarations specify the attributes of each element, indicating their name, type, and, possibly, default value. Attributes can be marked as `required`, `implied`, or `fixed`. Attributes marked as `required` must have an explicit value for each occurrence of the elements with which they are associated. Attributes marked as `implied` are optional. Attributes marked as `fixed` have a fixed value indicated at the time of their definition. Entities are used to include text and/or binary data into a document and can be *internal* or *external*. Internal entities are used to introduce special characters in the document or as an alias for some frequently used text. *External* entities are external files containing either text or binary data. Notation declarations specify how to manage the binary entities. Entities and notations are important in the description of the physical structure of an XML document, but are not considered in this paper, where we concentrate the analysis on the XML logical description. Our authorization model can be easily extended to cover these components.

XML documents may include *links* that express relationships between resources. A link is defined by an XLink (XML Linking Language) *linking element* [DeRose et al. 2001]. XLink allows two types of links: *simple links*, similar to the HTML links, and *extended links*, that express relationships among more than two resources. We refer the reader to the W3C proposal [DeRose et al. 2001] for a complete description of XLink elements and attributes.

Example 2.1. Figure 1(a) illustrates an example of DTD for XML documents describing a department of a hospital. Each department is composed of zero, one, or more divisions and is responsible to create an XML document for each division (if any) conforming to the considered DTD. According to this DTD, a `department` includes elements `division` (optional) and `medical_staff`, `research` activity, and `patients`. The medical staff is composed of physicians and nurses (`physician` and `nurse` elements). Each physician is characterized by `name`, `specialty`, `office`, `phone`, home `address`, and `salary`. Each nurse is characterized by `name`, home `address`, and `salary`. The research activity of a department (or division) is organized into `projects`, each with a designated `leader` and consisting of an `objective`, `laboratory`, and a set, possibly empty, of related `publications`. Each publication has a `title`, one or more `author` elements, and the corresponding postscript file (linking element `ps`). Information about the `patients` includes `name`, `address`, `room`, `illness`, and `therapy`. A therapy is characterized by its `startdate` and `enddate`, a `type`, and a list of `drugs`. Each drug prescribed to a patient has a `name`, a daily administration quantity (`daily_admin` element), and a `cost`. Properties of each element are defined in the attribute definition portion of the document. Elements `department` and `project` have a required attribute `name`, which is a string (character data). In addition, element `project` has an attribute `type` representing the project type (public vs private). Attribute `xmlns:xlink` of element `ps` is

used to define an XLink namespace;[1] `xlink:type` denotes the type of the link, and `xlink:href` is the locator attribute that defines where the resource is located. ◇

XML documents valid according to a DTD obey the structure defined by the DTD. Figure 2 illustrates an example of an XML document valid with respect to the DTD in Figure 1. Intuitively, each DTD is a *schema*, and XML documents valid according to that DTD are *instances* of that schema. Note that since elements and attributes defined in a DTD may appear in an XML document zero (optional elements), one, or multiple times, according to their cardinality constraints, the structure specified by the DTD is not rigid; two distinct documents of the same schema may differ in the number and structure of elements.

DTDs and XML documents can be modeled graphically as follows. A DTD is represented as a labeled tree containing a node for each element, attribute, and value associated with `fixed` attributes in the DTD. There is an arc between an element and an element/attribute belonging to it, labeled with the cardinality of the relationship, and between a `fixed` attribute and each of its value(s). Figure 1(b) illustrates the tree for the DTD in Figure 1(a). Elements are represented as ovals and attributes as rectangulars. Arcs labeled `or` and with multiple branching are used to represent a choice in an element declaration. For instance, choice (`city`|`county`)`?` in the `address` declaration is represented by an arc, labeled with '`?`' and '`or`', from `address` node to both `city` and `county` nodes. An arc with multiple branching is also used to represent a sequence with a cardinality constraint associated with the whole sequence. For instance, sequence (`author`+,`title`,`ps`?)* in the `publications` declaration is represented with an arc labeled '*' starting from node `publications` and ending to nodes `author`, `title`, and `ps`. To preserve the order between elements in an element declaration, for any two elements e_i and e_j, if e_j follows e_i in the element declaration, node e_j appears below node e_i in the tree. Each XML document is described by a tree with a node for each element, attribute, and value in the document, and with an arc between each element and each of its sub-elements/attributes/values and between each attribute and each of its value(s). Each arc in the DTD tree may correspond to zero, one, or multiple arcs in the XML document, depending on the cardinality of the corresponding containment relationship. Note that arcs in XML documents are not labeled. Figure 2(b) illustrates the tree representation of the XML document in Figure 2(a). In the remainder of this paper we will use the term *tree* to indiscriminately refer to the graphical representation of either a DTD or an XML document. Also, we will use the term *object* to refer to XML documents and DTDs indiscriminately. We will explicitly distinguish them when necessary.

Each object can have associated *metadata*. These are data representing information *on* the object, such as creator, creation date, expiration date, as well as any other properties that may have been defined on it (e.g., public document, internal document, and so on). Metadata can be conveniently managed through a

[1] A namespace is a way of distinguishing element types and attribute names to allow correct processing by a software module [Bray et al. 1999]. For instance, an XML document may include one or more occurrences of the element `title` to represent either a paper's title or an identifying appellation such as Mr. or Professor. The different semantics is captured by associating them with different namespaces.

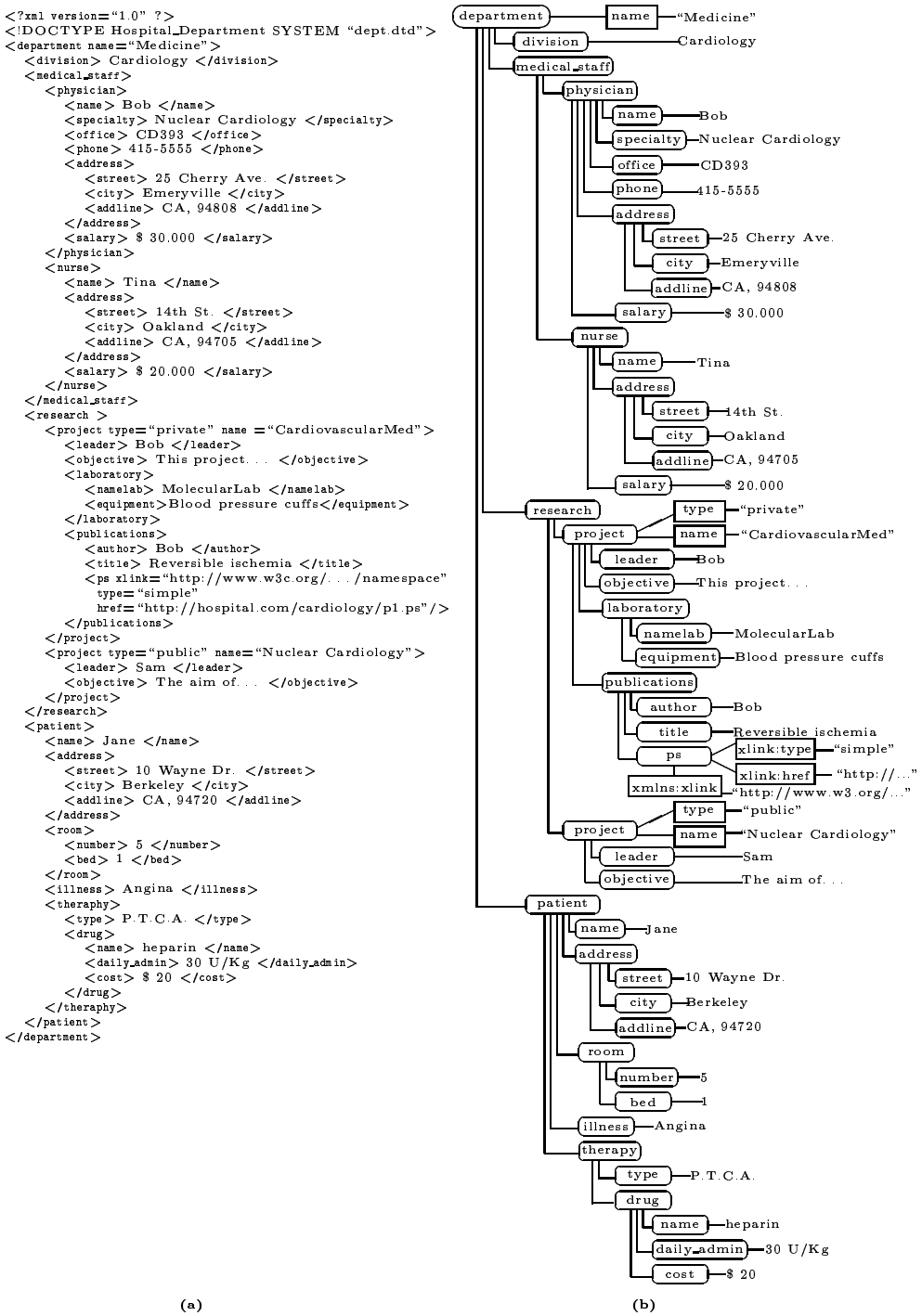


Fig. 2. An example of valid XML document (a) conforming to the DTD in Figure 1 and the corresponding graphical representation (b).

description in RDF (Resource Description Framework) [Brickley and Guha 2000], an application of XML resulting from a collaborative design effort among several W3C members. The RDF description associated with an object can be seen as a set of pairs of the form $\langle property, value \rangle$, where *property* is the name of a meta-property and *value* is its value with respect to the object. The value can be atomic (e.g., string or number), another resource, a collection of atomic values, or a meta-data instance. Meta-properties can be nested and their structure represented in a graphical framework similar to that of XML and DTD documents.

3. AUTHORIZATION SUBJECTS

The development of an access control system requires the definition of the *subjects* and *objects* against which authorizations must be specified and access control must be enforced. In this section we present the subjects; in Section 4 we describe the objects.

Usually, subjects can be referred to on the basis of their *identities* or on the *location* from which requests originate. Locations can be expressed with reference to either their numeric IP address (e.g., `150.100.30.8`) or their symbolic name (e.g., `tweety.cardiology.hospital.com`). Our model combines these features. Subjects requesting access are thus characterized by a triple ⟨`user-id`,`IP-address`,`sym-address`⟩, where `user-id` is the identity[2] with which the user connected to the server, and `IP-address` (`sym-address`, resp.) is the numeric (symbolic, resp.) identifier of the machine from which the user connected.

To allow the specification of authorizations applicable to sets of users and/or sets of machines, our model also supports user *groups* and *location patterns*. A group is a set of users defined at the server. Groups do not need to be disjoint and can be nested. A location pattern is an expression identifying a set of physical locations, with reference to either their symbolic or numerical identifiers. Patterns are specified by using the wild card character * instead of a specific name or number (or sequence of them). For instance, `151.100.*.*`, or equivalently `151.100.*`, denotes all the machines belonging to network `151.100`. Similarly, `*.mil`, `*.com`, and `*.it` denote all the machines in the Military, Company, and Italy domains, respectively. If multiple wild card characters appear in a pattern, their occurrence must be continuous (not interleaved by numbers or names). Also, consistently with the fact that specificity is left to right in IP addresses and right to left in symbolic names, wild card characters must appear always as right-most elements in IP patterns and as left-most elements in symbolic patterns. Intuitively, location patterns are to location addresses what groups are to users.

Users and groups together with their membership relationship, IP addresses with patterns, and symbolic names with their patterns, form partially ordered sets (hierarchies). To provide a uniform treatment for the different components of subjects, we first give the definition of hierarchy as follows.

[2]We assume user identities to be local, that is, established and authenticated by the server, because this is a solution relatively easy to implement securely. Obviously, in a context where remote identities cannot be forged and can therefore be trusted by the server (using a Certification Authority, a trusted third party, or any other secure infrastructure), remote identities could be considered as well.

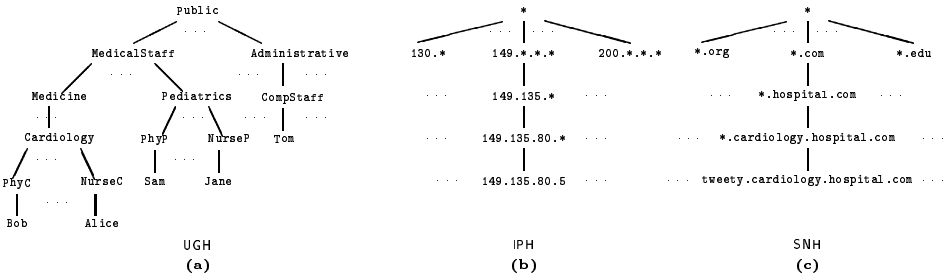


Fig. 3. An example of user-group (a), IP (b), and symbolic name (c) hierarchies.

Definition 3.1. *Hierarchy A* hierarchy *is a triple* $(\mathsf{X}, \mathsf{Y}, \leq)$*, where* $\leq$ *is a partial order on* Y*, called the* dominance relation*, and* $\mathsf{X} \subseteq \mathsf{Y}$ *is the set of minimal elements of* Y *with respect to the partial order.*

The model considers the following three hierarchies.

—A *user-group hierarchy* $\mathsf{UGH} = (\mathsf{U}, \mathsf{UG}, \leq_{\mathsf{UG}})$, where U is a set of user identifiers and $\mathsf{UG} = \mathsf{U} \cup \mathsf{G}$, with G a set of group names in which users are organized. Given two elements $x, y \in \mathsf{UG}$, $x \leq_{\mathsf{UG}} y$ if and only if x is a member of y. In most practical applications the hierarchy is rooted at a group, called `Public` or `Any`, to whom everybody (directly or indirectly) belongs.

—An *IP hierarchy* $\mathsf{IPH} = (\mathsf{I}, \mathsf{IP}, \leq_{\mathsf{IP}})$, where I is a set of completely specified numerical addresses and IP is a set of IP patterns. Given two elements $x, y \in \mathsf{IP}$, $x \leq_{\mathsf{IP}} y$ if and only if each component of y is either the wild card character or is equal to the corresponding (positionwise from left to right) component of x.

—A *symbolic name hierarchy* $\mathsf{SNH} = (\mathsf{S}, \mathsf{SN}, \leq_{\mathsf{SN}})$, where S is a set of completely specified symbolic names and SN is a set of symbolic name patterns. Given two elements $x, y \in \mathsf{SN}$, $x \leq_{\mathsf{SN}} y$ if and only if each component of y is either the wild card character or is equal to the corresponding (positionwise from right to left) component of x.

A hierarchy can be pictured as a directed graph containing a node for each element of the hierarchy and with an arc from an element x to an element y, if x directly dominates y in the hierarchy. Dominance relationships holding in the hierarchy correspond to paths in the graph. User-group hierarchies are arbitrary DAGs, while IP and symbolic name hierarchies are necessarily trees. Figure 3 illustrates an example of user-group, IP, and symbolic name hierarchies. We note that the hierarchies introduced serve for explanatory purposes and only the user-group hierarchy needs to be explicitly defined and stored at the server (or retrieved at access control time).

Instead of specifying authorizations with respect to only one of either the user/group identifier or location identifier, and having the problem of how different authorizations should be combined at access request time, we allow the specification of authorizations with reference to both user/group and location. This choice provides more expressiveness (it allows to express the same requirements as the alternative and more) and provides a natural treatment for different authorizations

applicable to the same request.[3] We define the *authorization subject hierarchy* as the Cartesian product of the three hierarchies previously introduced, where a subject s_j is dominated by another subject s_i if each of s_j's components is dominated by the corresponding component in s_i, as clarified by the following definition.

Definition 3.2. *Authorization subject hierarchy The* authorization subject hierarchy *is a hierarchy* $\mathsf{ASH} = (\mathsf{R}, \mathsf{AS}, \leq_{\mathsf{AS}})$, *where* $\mathsf{R} = \langle \mathsf{U} \times \mathsf{I} \times \mathsf{S} \rangle$, $\mathsf{AS} = \langle \mathsf{UG} \times \mathsf{IP} \times \mathsf{SN} \rangle$, *and* $\langle ug_i, ip_i, sn_i \rangle \leq_{\mathsf{AS}} \langle ug_j, ip_j, sn_j \rangle$, *if and only if* $ug_i \leq_{\mathsf{UG}} ug_j$, $ip_i \leq_{\mathsf{IP}} ip_j$, *and* $sn_i \leq_{\mathsf{SN}} sn_j$.

Authorizations can be defined with reference to any of the elements of ASH. In particular, authorizations can be specified for users/groups regardless of the physical location (e.g., $\langle$`Alice`$, *, *\rangle$), for physical locations regardless of the user identity (e.g., $\langle$`Public`$, $`150.100.30.8`$, *\rangle$), or for both (e.g., $\langle$`Sam`$, *, *$`.acme.com`$\rangle$). Intuitively, authorizations specified for subject $s_j \in \mathsf{ASH}$ are applicable to all subjects s_i such that $s_i \leq_{\mathsf{AS}} s_j$. Possible conflicts between authorizations applicable to a given requester will be investigated in Section 5.

4. AUTHORIZATION OBJECTS

A set Obj of Uniform Resource Identifiers (URI) [Berners-Lee et al. 1998] denotes the resources to be protected. For XML documents, URI's can be extended with *path expressions*, which are used to identify the elements and attributes within a document. In particular, we adopt a W3C proposal for the identification of internal components of an XML document, namely the XPath language [World Wide Web Consortium (W3C) 2001]. There are considerable advantages deriving from the adoption of a standard language. First, the syntax and semantics of the language are known by potential users and well-studied. Second, several tools are already available which can be easily reused to produce a functioning system.

We keep at a simplified level the description of the language which expresses patterns in XPath. The W3C proposal [World Wide Web Consortium (W3C) 2001] contains the complete specification of the language.

Definition 4.1. *Path expression A* path expression *on a document tree is a sequence of element names or predefined functions separated by character / (slash):* $l_1/l_2/\ldots/l_n$. *Path expressions may terminate with an attribute name as the last term of the sequence. Attribute names are syntactically distinguished by preceding them with special character* `@`.

A path expression $l_1/l_2/\ldots/l_n$ on a document tree represents all the attributes or elements named l_n that can be reached by descending the document tree along the sequence of nodes named $l_1, l_2, \ldots, l_{n-1}$. For instance, path expression `/department/medical_staff/physician` denotes the `physician` elements that are children of `medical_staff` elements, that are children of the `department` element. Path expressions may start from the root of the document (if the path expression starts with a slash, it is called *absolute*) or from a predefined starting point in the document (if the path expression starts directly with an element name, it is called

[3] We will elaborate on this in Section 5.

Function	Argument	Description
`ancestor`	*element-name*	returns all *element-name* ancestors of the context node
`ancestor-or-self`	*element-name*	returns all *element-name* ancestors of the context node and, if the context node is an *element-name* element, the context node as well
`descendant`	*element-name*	returns all *element-name* descendants of the context node
`descendant-or-self`	*element-name*	returns all *element-name* descendants of the context node and, if the context node is an *element-name* element, the context node as well
`following`	*element-name*	returns all the *element-name* nodes that are after the context node in the document order, excluding any descendants, attribute nodes, and namespace nodes
`following-sibling`	*element-name*	returns all the following *element-name* siblings of the context node
`preceding`	*element-name*	returns all the *element-name* nodes that are before the context node in the document order, excluding any ancestors, attribute nodes and namespace nodes
`preceding-sibling`	*element-name*	returns all the preceding *element-name* siblings of the context node
`parent`	*element-name*	returns the parent of the context node, if there is one and it is an *element-name* element, and otherwise returns nothing
`child`	*element-name*	returns all *element-name* elements children of the context node
`self`	*element-name*	returns the context node, if it is an *element-name* element, and otherwise returns nothing
`attribute`	*attribute-name*	returns attribute *attribute-name* of the context node
`namespace`	*namespace*	returns the *namespace* nodes of the context node

Table 1. XPath Predefined Functions

relative). The path expression may also contain the operators *dot*, which represents the current node; *double dot*, which represents the parent node; and *double slash*, which represents an arbitrary descending path. For instance, path expression `/department//leader` retrieves all the elements `leader` descendants (at any level) of the document root `department`.

Path expressions may also include functions. These functions serve various needs, like the extraction of the attributes of an element and the navigation in the document structure. Table 1 illustrates the XPath predefined functions, their arguments type, and a brief description of the function. The name of a function and its arguments are separated by a double colon '`::`'. For instance, expression `research/ancestor::department` returns the `department` nodes that appear as ancestors of the `research` node; expression `ps/attribute::xlink:href` returns attribute `xlink:href` of `ps` elements; expression `/department/child::medical_staff//city` returns all `city` nodes descendants of the `medical_staff` node child of the `department` node (with reference to the document in Figure 2, the expression identifies the cities "Emeryville" and "Oakland"). Note that the operators *dot*, *double dot*, and *double slash* previously listed can be used as abbreviations for functions `self`, `parent`, and `descendant`, respectively. Analogously, character `@` is used as an abbreviation for function `attribute`. For instance, expression `ps/@xlink:href` is short for path expression `ps/attribute::xlink:href`. The syntax for XPath expressions also permits to

associate conditions with the nodes of a path; in this case the path expression identifies the set of nodes that satisfy all the conditions. Conditions greatly enrich the power of the language, and are a fundamental component in the construction of a sophisticated authorization mechanism. The *conditional expressions* used to represent conditions may operate on the "text" of elements (i.e., the character data in the elements) or on names and values of attributes. Conditions are distinguished from navigation specifications by enclosing them within square brackets. Given a path expression $l_1/\ldots/l_n$ on the tree of an XML document, a condition may be defined on any label l_i, enclosing in square brackets a separate evaluation context containing a predicate that compares the result of the evaluation of the relative path expression with a constant or another expression. Conditional expressions may be combined via `and` and `or` operators to build boolean expressions. Multiple conditional expressions appearing in the same path expression are considered to be `and`ed (i.e., all the conditions must be satisfied). In addition, conditional expressions may include functions `last()` and `position()` that permit to extract the children of a node which are in given positions. Function `last()` evaluates to true on the last child of the current node. Function `position()` evaluates to true on the node in the evaluation context whose position is equal to the context position. For instance, expression `/department/research/project/publications/ps[position()=1]` returns the first `ps` child of the `publications` element (note that the conditional expression `[position()=1]` can be abbreviated as `[1]`); expression `/department[./@name = "Medicine" and ./division = "Cardiology"]/medical_staff/nurse` identifies all the nurses of the `"Cardiology"` division of the `"Medicine"` department; expression `/department/research/project[./@type="public"][1]` returns the first "public" projects child of the `research` element.

The proposed formalism can be also used to specify conditions on metadata, such as the RDF descriptions illustrated in Section 2. In this case, a new predefined function `meta` is used to access meta information on the XML document. For instance, expression `/[meta()./@creation_date = "2000-01-05"]`, evaluates to true for documents created on January 5, 2000.

5. ACCESS AUTHORIZATION SPECIFICATIONS

We first describe the basic features that access authorizations need to provide to regulate access to Web documents and then give their formal definition.

5.1 Basic Features of the Access Authorizations

The authorization model supports authorizations at *all levels of granularity*, including individual documents and elements within them. The object granularity for which authorizations can be specified spans from the DTD (meaning the set of its instances) to single elements/attributes within individual documents, where elements and attributes can be referenced by means of path expressions as illustrated in Section 4. Authorizations can be either *positive* (permissions) or *negative* (denials). The reason for having both positive and negative authorizations is to provide a simple and effective way to specify authorizations applicable to sets of subjects/objects with support for exceptions [Jajodia et al. 2001; Lunt 1989].

Authorizations specified on an element can be defined as applicable to the element's attributes only (*local* authorizations) or, in a recursive approach, to its

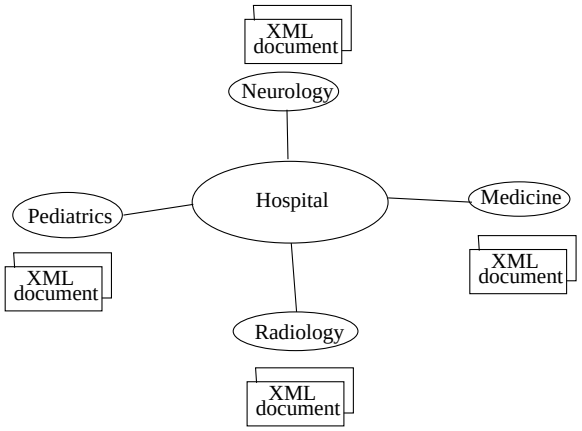


Fig. 4. An example of hospital organization.

sub-elements and their attributes (*recursive* authorizations). Local authorizations on an element apply to the direct attributes of the element but not to those of its sub-elements. As a complement, recursive authorizations, by propagating permissions/denials from nodes to their descendants in the tree, represent an easy way to specify authorizations holding for the whole structured content of an element (on the whole document if the element is the root). To support exceptions (e.g., the whole content *but* a specific element can be read), recursive propagation from a node applies until stopped by an explicit conflicting (i.e., of different sign) authorization on the descendants. Intuitively, authorizations propagate until overridden by an authorization on a *more specific object* [Jajodia et al. 2001].

Authorizations can be specified on single XML documents (*document* or *instance* level authorizations) or on DTDs (*DTD* or *schema* level authorizations). Authorizations specified on a DTD are applicable (i.e., are propagated) to all XML documents that are instances of the DTD. Since large enterprises are often organized into multiple domains, protection requirements may be specified both at the level of the enterprise, stating general regulations that should hold, and at the level of specific domains (part of the enterprise) where, according to a local policy, additional constraints may need to be specified or some constraints may need to be relaxed. Organizations specify authorizations with respect to DTDs; specific sites can specify authorizations with respect to individual documents (instance level authorizations) as well as with respect to DTDs. The two types of DTD-level authorizations have complementary roles in increasing access control flexibility. *Organization* DTD-level authorizations stated by a central authority can be effectively used to implement corporate-wide access control policies on document classes. *Site* DTD-level authorizations specified by departmental authorities allow for department-wide access control policies complementing the corporate ones. Moreover, they alleviate administration chores by allowing concise specification of site-wide authorizations. For instance, suppose that a hospital is composed of different departments each of which is responsible to manage specific XML documents (see Figure 4). In this scenario, general protection requirements that should be satisfied by all departments

Level/Strength	Propagation	
	Local	Recursive
Instance	L	R
Instance (soft statement)	LS	RS
DTD	LD	RD
DTD (hard statement)	LDH	RDH

Table 2. Authorization Types

of the hospital can be expressed through (organization) DTD-level authorizations stated at the hospital level. Specific protection requirements, applicable only within a single department, can be expressed by means of (site) DTD-level authorizations. Analogously, requirements applicable only to a specific document are expressed by means of instance level authorizations associated with the document. We anticipate that, in the access control processing, organization DTD-level authorizations and site DTD-level authorizations are, with respect to each DTD, merged by performing a *flat union*. In other words, organization-wide and site-specific authorizations are treated in the same way (although, remember, organization-wide authorizations apply to all the documents in the network while site-specific authorizations apply only to documents stored at the site). Given this, in the following we will simply refer to DTD authorizations without making any distinction of where they have been specified. The reason for merging the two sets of authorizations with a simple flat union is simplicity. We observe that, in principle, even at this level some notion of "specificity" could be applied. This reasoning could also be possibly extended by considering any number of intermediate organizational levels which could be reflected in priorities associated with the authorizations [Jonscher et al. 1994]. Authorizations at the DTD level, together with path expressions with conditions, provide an effective way for specifying authorizations on elements of different documents, possibly in a content-dependent way. Again, according to the "most specific takes precedence" principle, DTD level authorizations being propagated to an instance are overridden by possible authorizations specified for the instance. To address situations where these precedence criteria should not be applied (e.g., cases where an authorization on a document should be applicable unless otherwise stated at the DTD level, or cases where an authorization on a DTD *must* be applied to all instances of the DTD), we allow users to specify instance level authorizations as *soft* and DTD level authorizations as *hard*. Non-soft and non-hard authorizations have the behavior sketched above (i.e., non-soft instance-level authorizations have priority over non-hard DTD-level authorizations). Soft authorizations are authorizations that apply to the document unless otherwise stated at the DTD level (intuitively, a department can state that its documents can/cannot be accessed unless the organization states otherwise). In a dual way, hard authorizations allow an organization to specify authorizations that must be enforced to all instances of a DTD, no exceptions. The combination of the options above introduces the eight authorization types summarized in Table 2. Their semantics dictates a priority order among the authorization types. The priority order from the highest to the lowest is: `LDH` (local hard authorization), `RDH` (recursive hard authorization), `L` (local authorization), `R` (recursive authorization), `LD` (local authorization specified at the

	Subject user/group,IP,domain	Object (path expression)	Action	Sign	Type
a	Public,*,*	/department/@name	read	+	L
b	Public,*,*	/department/division	read	+	L
c	Administrative,*,*.hospital.com	/department//name	read	+	LDH
d	Administrative,*,*.hospital.com	/department//address	read	+	RDH
e	Administrative,159.101.80.5,*	/department/medical_staff//salary	read	+	LDH
f	Administrative,159.101.80.5,*	/department/patient//cost	read	+	LDH
g	Public,*,*	/department/medical_staff//salary	read	−	LDH
h	Public,*,*	/department/patient//cost	read	−	LDH
i	Public,*,*	/department[./@name="medicine"]/medical_staff	read	+	R
j	Public,*,*	/department[./@name="medicine"]/medical_staff//address	read	−	R
k	Public,*,*	/department[./@name="medicine"]/medical_staff//salary	read	−	L
m	PhyC,*,*	/department[./@name="medicine" and ./division="cardiology"]/patient	read	+	R
n	Public,*,*	/department[./@name="medicine" and ./division="cardiology"]/patient	read	−	R
o	MedicalStaff,*,*	/department[./@name="medicine"]/research	read	+	R
p	Public,*,*	/department[./@name="medicine"]/research	read	−	R
q	PhyC,159.*,*	/department/research/project[./@type="private"]	read	+	R
r	*,*,*	/department/research/project[./@type="private"]	read	−	R
s	NurseC,*,*	/department/patient//illness	read	+	LS
t	NurseC,*,*	/department/patient//name	read	+	L
u	NurseC,*,*	/department/patient//drug	read	+	R
v	NurseC,*,*	/department/patient/room	read	+	R

Fig. 5. Example of access authorizations.

schema level), RD (recursive authorization specified at the schema level), LS (local soft authorization), and RS (recursive soft authorization). For instance, if there are a positive local hard authorization and a negative local authorization both applicable to the same object and subject, the positive local hard authorization overrides the negative one. However, it may be also the case that several authorizations, possibly of different sign, apply to a given request with reference to a given authorization type and element/attribute. As we will see in the following section, conflicts between such authorizations are solved by applying a conflict resolution policy [Jajodia et al. 2001; Lunt 1989].

5.2 Access Authorizations

At each server, a set Auth of access authorizations specifies the actions that subjects are allowed (or forbidden) to exercise on the objects stored at the server. Access authorizations are formally defined as follows.

Definition 5.1. *Access authorization An* access authorization $a \in$ Auth *is a 5-tuple of the form:* ⟨subject, object, action, sign, type⟩, *where:*

—subject $\in$ AS *is the subject to whom the authorization is granted;*

—object *is either a URI in* Obj *or is of the form URI:PE, where URI* $\in$ Obj *and PE is a path expression on the tree of document URI;*

—action = `read` *is the action being authorized or forbidden;*[4]

—sign $\in \{+, -\}$ *is the sign of the authorization, which can be positive (allow access) or negative (forbid access);*

—type $\in \{$LDH, RDH, L, R, LD, RD, LS, RS$\}$ *is the type of the authorization (*Local D*TD* Hard, Recursive D*TD* Hard, Local, Recursive, Local D*TD*, Recursive D*TD*, Local Soft, *and* Recursive Soft, *respectively).*

[4]We limit our consideration to read authorizations. The support of write actions like insert, update, and delete does not complicate the authorization model (see Section 7). However, full support for such actions in the framework of XML has yet to be defined.

Example 5.1. Consider the XML document in Figure 2. This document is an instance of the DTD in Figure 1 that includes information regarding the `Cardiology` division of the `Medicine` department of a given hospital. We now illustrate some examples of protection requirements that may need to be expressed and their representation as authorizations in our model. The bold-face letters between square brackets at the end of each requirement identify the authorizations in Figure 5 expressing the requirement. Figure 5 lists the resulting authorizations. The horizontal line between authorizations **p** and **q** separates DTD-level authorizations (**a** through **p**) from instance level authorizations (**q** through **v**). Note that for simplicity, in the object field we report only the path expression and omit the URI.

Hospital's policy. (Organization DTD-level authorizations applicable to all the departments of the hospital). Requirements expressed as "must" specify statements that do not allow exceptions (which translate to hard authorizations)

(1) Department and division names are publicly accessible. [**a**]-[**b**]

(2) Information about the name and home address of medical staff and of patients *must* be accessible to the members of `Administrative` group connected from domain `*.hospital.com`.[**c**]-[**d**]

(3) Information about the salary of the medical staff and the cost of the therapy of all patients of the hospital *must* be accessible to the members of group `Administrative` connected from host `159.101.80.5`. [**e**]-[**f**]
Everybody else *must* be explicitly forbidden access to this information. [**g**]-[**h**]

`Medicine` *department's policy.* (Site DTD-level authorizations to complement or override the organization DTD-level authorizations)

(4) Information about medical staff working at the `Medicine` department with exception of their salary and home address, is publicly accessible. [**i**]-[**j**]-[**k**]

(5) Information about patients hospitalized in a given division is accessible only to the physicians working in the same division. [**m**]-[**n**]

(6) Information about the research activity of the `Medicine`'s divisions is accessible only to the medical staff of the hospital. [**o**]-[**p**]

`Cardiology` *division's policy.* (Specified at the instance level to complement or override the hospital's and department's policy)

(7) Information about "private" projects of the `Cardiology` division is accessible to the physicians working in the `Cardiology` division when connected from network `159.*`. [**q**]
Everybody else cannot access information about "private" projects. [**r**]

(8) Information about patients illnesses is accessible to nurses of the `Cardiology` division unless otherwise stated at the DTD level. [**s**]

(9) Information about name, drug, and room of patients hospitalized in the `Cardiology` division is accessible to the members of `NurseC` group. [**t**]-[**u**]-[**v**]

◇

The following section discusses the interpretation of authorizations to produce the view of a requesting subject on the document requested.

6. REQUESTER'S VIEW ON DOCUMENTS

The view of a subject on each document depends on the access permissions and denials specified by the authorizations and their priorities. Such a view can be computed through a *tree labeling process*, described next. In the following, we use the term node (of a document tree) to refer to either an element or an attribute in the document indiscriminately.

6.1 Document Tree Labeling

Each access authorization states whether a subject can (or cannot) access an element/attribute (or set of them). The type associated with each authorization on a given object (at the instance or schema level) determines the "behavior" of the authorization with respect to the object structure, that is, whether it propagates down the tree and whether it is overridden or it overrides other authorizations. The enforcement of the authorizations on a document according to the principles discussed in Section 5.1 essentially requires the indication of whether, for an element/attribute in a document, a positive authorization (+), a negative authorization (−), or no authorization (ε) applies. Since authorizations can be of different level (instance vs schema), strength (hard vs soft) and propagation (local vs recursive), we associate with each node n an array, $n.veclabel$, of eight components[5], one for each authorization type, which is a record including three fields: *sign*, *Allowed*, and *Denied*. Let $t \in \{\texttt{LDH},\texttt{RDH},\texttt{L},\texttt{R},\texttt{LD},\texttt{RD},\texttt{LS},\texttt{RS}\}$ be an authorization type. The value of $n.veclabel[t].sign$ can be '+' for permission, '−' for denials, and 'ε' for no authorization, and it indicates the sign associated with the node according to the authorizations and the conflict resolution policy. The determination of the sign is preceded by the computation of $n.veclabel[t].Allowed$ and $n.veclabel[t].Denied$, that are two lists storing all the subjects for which there is a positive (negative, respectively) authorization of type t that applies to n. Figure 6 illustrates an algorithm, **Compute-view**, enforcing the labeling process. Given a requester *rq* and an XML document *URI*, the algorithm first initializes variable T to the tree representing the document and r to the root of T. After initialization, the algorithm invokes procedure **InitialLabel**(T,*rq*). The purpose of **InitialLabel** is to associate authorizations with the corresponding elements/attributes. Since not all the authorizations defined on a document are applicable to all requesters, the set of authorizations on the document' elements, and the authorizations behavior along the tree, can vary for different requesters. Thus, the first step of **InitialLabel** consists in the determination of the set A of authorizations defined for the document *URI* at the instance and schema level and applicable to the requester *rq*. For each authorization $a = \langle subject, object, action, sign, type\rangle$ in A, the method determines the set N of nodes in T that are identified by $a.object$. After that, for each node n in N, **InitialLabel** adds $a.subject$ to $n.veclabel[a.type].Allowed$ if $a.sign$ is '+'; it adds the subject to $n.veclabel[a.type].Denied$ if $a.sign$ is '−'. Since several authorizations, possibly of different sign, may exist for each authorization type t (i.e., $n.veclabel[t].Allowed$ and $n.veclabel[t].Denied$ can be both not empty), the determination of the $n.veclabel[t].sign$ value requires the application of a conflict resolution

[5]We use a Java-like notation where *obj.att* (*obj.meth*, resp.) denotes the attribute (method, resp.) associated with object *obj*.

ALGORITHM 6.1. *Compute-view algorithm*
Input: A requester rq and an XML document URI
Output: The document's to be returned to rq

```
void main()
{   SecureDocument T(URI) /* Constructor creates tree from the XML document URI */
    r = T.Root;
    InitialLabel(T,rq);
    r.SetLabel();
    r.GetFinalLabel(empty);     /* empty is a labeling vector whose components have
    r.Prune();                     all sign fields equal to ε */
}

void InitialLabel(T,rq)
{ A = {a = ⟨subject,object,action,sign,type⟩ | a ∈ Auth, rq ≤AS subject, uri(object)==URI
        OR uri(object) == dtd(URI)};
  For a in A do
  { N = {n | n ∈ T, n ∈ a.object};
    Case a.sign of
      '+': For n in N do n.veclabel[a.type].Allowed.Add(a.subject);
      '−': For n in N do n.veclabel[a.type].Denied.Add(a.subject);
  }
}

void SetLabel()
/* Evaluates the set of authorizations of each type on the node */
{ For t in [LDH,RDH,L,R,LD,RD,LS,RS]
  { s = this.veclabel[t].Denied.Head(); keepSubject = TRUE;
    While s != null do
    { s' = this.veclabel[t].Allowed.Head();
      While s' != null AND keepSubject do
      { If (s ≤AS s' AND ¬(s' ≤AS s))          /* s' dominated and "most specific takes precedence" */
        then this.veclabel[t].Allowed.Remove(s');
        else If (s' ≤AS s AND ¬(s ≤AS s'))
        then keepSubject = FALSE; s' = this.veclabel[t].Allowed.Next();
      }
      If ¬keepSubject            /* s dominated and "most specific takes precedence" */
      then this.veclabel[t].Denied.Remove(s);
      s = this.veclabel[t].Denied.Next();
    }
    If this.veclabel[t].Allowed.IsEmpty()
    then If this.veclabel[t].Denied.IsEmpty()
           then veclabel[t].sign = 'ε';
           else veclabel[t].sign = '−';
    else If this.veclabel[t].Denied.IsEmpty()
           then veclabel[t].sign = '+';
           else veclabel[t].sign = ConflictPolicy;     /* '−' for the "denials take precedence" policy */
  }
  For c in this.Children() do c.SetLabel()
}

void GetFinalLabel(pveclabel)
{ this.finlabel = 'ε';
  For t in [LDH,RDH,L,R,LD,RD,LS,RS]
  { this.veclabel[t].sign = this.veclabel[t].sign ⊕ pveclabel[t].sign;
    this.finlabel = this.finlabel ⊕ this.veclabel[t].sign;
  }
  For a in this.Attribute() do a.GetFinalLabel(this.veclabel);
  For e in this.ElemChildren() do e.GetFinalLabel(masklocal(this.veclabel));
}

void Prune()
{   For c in this.Children() do c.Prune();
    If this.Children() == ∅ and this.finlabel ≠ '+' then remove the current node from T;
}
```

Fig. 6. Compute-view algorithm.

policy. Different approaches can be used to solve these conflicts [Samarati and De Capitani di Vimercati 2001]. One solution is to consider the authorization with the most specific subject (*"most specific subject takes precedence"* principle), where specificity is dictated by the partial order defined over ASH; other solutions can consider the negative authorization (*"denials take precedence"*), or the positive authorization (*"permissions take precedence"*), or no authorizations (*"nothing takes precedence"*). Other approaches could also be envisioned, such as, for example, considering the sign of the authorizations that are in larger number. For simplicity, in the examples and discussion in the remainder of this paper we refer to a specific policy and solve conflicts with respect to the "most specific subject takes precedence" principle and, in cases where conflicts remain unsolved (the conflicting authorizations have incomparable subjects), we stay on the safe side and apply the *"denials take precedence"* principle. We refer to the combination of these two conflict resolution policies as *"most specific subject/denials take precedence"* principle. The reason for this specific choice is that the two principles so combined naturally cover the intuitive interpretation that one would expect from the specifications [Lunt 1989]. This behavior is realized by method **SetLabel**, applied on all the document nodes in a preorder visit starting from the root r. **SetLabel** combines, for each type t, the two lists *veclabel*[t].*Allowed* and *veclabel*[t].*Denied* of subjects according to the *"most specific subject/denials take precedence"* principle. Intuitively, a given subject s belonging to list *veclabel*[t].*Denied* is compared with each subject s' in the list *veclabel*[t].*Allowed*. If s is more specific than s', s' can be removed from *veclabel*[t].*Allowed* as it is dominated by s. If, on the contrary, s' is more specific than s, it is s that is dominated and can be removed from the list *veclabel*[t].*Denied*. When all the subjects appearing in *veclabel*[t].*Denied* have been compared with the subjects in *veclabel*[t].*Allowed*, the content of the two lists is considered: if the two lists are empty, this means that no authorization was originally defined on the node and the value 'ε' is assigned to *veclabel*[t].*sign*; if the list *veclabel*[t].*Allowed* is empty and the list *veclabel*[t].*Denied* is not empty, only negative authorizations are applicable on the node and '$-$' is assigned to *veclabel*[t].*sign*; if the list *veclabel*[t].*Allowed* is not empty and the list *veclabel*[t].*Denied* is empty, only positive authorizations are applicable on the node and '+' is assigned to *veclabel*[t].*sign*; finally, if both the lists are not empty, this means that there is a conflict where authorizations with unordered subjects have been defined on the same node and the sign specified by the conflict resolution policy must be assigned to *veclabel*[t].*sign* (in our case, '$-$'). It is important to note, however, that our model can support any of the conflict resolution policies discussed. Indeed, a different policy requires only a change in method **SetLabel**. Also, different policies could be applied to the same server, towards the definition of multiple policy systems [Jajodia et al. 2001]. The only obvious restriction we impose is that no more than one policy applies for each document.

The labels (signs) associated with nodes are then propagated to their subelements and attributes according to the following criteria: (1) authorizations on a node take precedence over those on its ancestors, and (2) authorizations at the instance level, unless declared as soft, take precedence over authorizations at the schema level, unless declared as hard. The complete labeling of the document tree is thus obtained by calling method **GetFinalLabel**(*veclabel*) on root node r

A	B	A∧B
ε	ε	ε
ε	−	ε
ε	+	ε
−	ε	ε
−	−	−
−	+	−
+	ε	ε
+	−	−
+	+	+

A	B	A∨B
ε	ε	ε
ε	−	−
ε	+	+
−	ε	−
−	−	−
−	+	+
+	ε	+
+	−	+
+	+	+

A	¬ A
ε	+
−	−
+	ε

A	B	A⊕B
ε	ε	ε
ε	−	−
ε	+	+
−	ε	−
−	−	−
−	+	−
+	ε	+
+	−	+
+	+	+

Fig. 7. Truth tables of the propositional connectives ∧, ∨, and ¬, and operator ⊕.

of T. Method **GetFinalLabel** considers the nodes of T according to a preorder visit of the tree and propagates permissions/denials associated with a node to its descendants. Propagation of the value, for each type t, of $p.veclabel[t].sign$ of a node p parent of a node n is obtained by assigning to $n.veclabel[t].sign$ the value of $p.veclabel[t].sign$ if and only if $n.veclabel[t].sign$ is equal to 'ε'. This propagation can be performed by interpreting the three values '+', '−', and 'ε' as values of a 3-valued logic. To this end, we first need to map '+', '−', 'ε' in the logic. The only condition that such mapping must satisfy is that 'ε' must be mapped to 0 (`false`). To understand the reason for this, think of `false` as "no statement" has been made. Signs '+' and '−' must then be mapped to the other two values, namely 1 (`true`), and $\frac{1}{2}$ (`indeterminate`); whatever choice would do. Here, we map '+' to 1 and '−' to $\frac{1}{2}$. It is easy now to see that, with the defined mapping, the propagation is obtained by assigning to $n.veclabel[t].sign$ the result of the formula $n.veclabel[t].sign \vee (\neg n.veclabel[t].sign \wedge p.veclabel[t].sign)$, where the truth tables of the propositional connectives ∨, ¬, and ∧ coincide with the truth tables defined in the 3-valued logic of Lukasiewicz [Rescher 1969] (see Figure 7). We denote such formula as $n.veclabel[t].sign \oplus p.veclabel[t].sign$ in the following. The truth table for ⊕ is reported in Figure 7. In the case where n is an element, propagation follows the same principle but $n.veclabel$ is combined with a masked version of the parent array $p.veclabel$ obtained by means of function **masklocal** that sets to 'ε' the *sign* field of components `LDH`, `L`, `LD`, and `LS`. The reason for this is that local authorizations applicable to a node p can be propagated only to attributes of p. After this propagation step, according to the defined priorities, **GetFinalLabel** determines the sign *finlabel* that must hold for the specific node n. In particular, the final sign *finlabel* of each node n is determined as the result of operation ⊕ between the *sign* field of components of array $n.veclabel$ considered in their priority order: `LDH` (local hard), `RDH` (recursive hard), `L` (local), `R` (recursive), `LD` (local, schema level), `RD` (recursive, schema level), `LS` (local soft), and `RS` (recursive soft).

6.2 Transformation Process

As a result of the labeling process, the value of *finlabel* for each node n contains the sign, if any, reflecting whether the node can be accessed ('+') or not ('−'). The value of *finlabel* is equal to 'ε' in the case where no authorizations have been specified nor can be derived for n. Value 'ε' can be interpreted either as a negation or as a permission, corresponding to the enforcement of the *closed* and the *open* policy, respectively [Jajodia et al. 2001]. In the following, we assume the closed policy. Accordingly, the requester is allowed to access all the elements and attributes whose

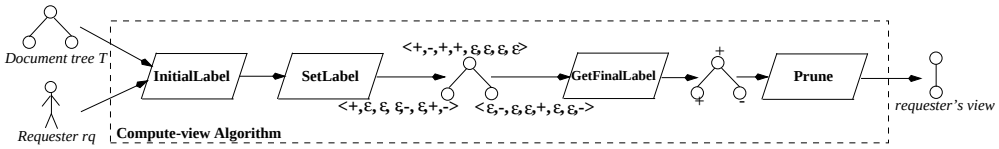


Fig. 8. Execution steps of the **Compute-view** algorithm.

label is positive. To preserve the structure of the document, the portion of the document visible to the requester will also include start and end tags of elements with a negative or undefined label that have a descendant with a positive label. The view on the document can be obtained by pruning from the original document tree all the subtrees containing only nodes labeled negative or undefined. This pruning is enforced by method **Prune** in Figure 6, which executes a postorder visit on the document tree and removes any leaf labeled '$-$' or 'ε'. The pruned document may be not valid with respect to the DTD referenced by the original XML document. This may happen, for instance, when required attributes are deleted. To avoid this problem, a *loosening* transformation is applied to the DTD. Loosening a DTD simply means to define as optional all the elements and attributes marked as required in the original DTD. DTD loosening prevents users from detecting whether information was hidden by the security enforcement or simply missing in the original document.

Figure 8 summarizes all the execution steps of the **Compute-view** algorithm.

Example 6.1. Consider the set of authorizations defined in Figure 5, and the user-group hierarchy in Figure 3(a). Consider a request to read the XML document in Figure 2 submitted by user `Alice` connected from host `159.101.80.10` with symbolic name `tweety.cardiology.hospital.com`. According to the authorizations stated at the DTD and instance level, since `Alice` is a member of the medical staff of the hospital, she can only access medical information like the name of patients, their room, the drug name, and the corresponding daily administration quantity given to patients. Consider now the same request submitted by user `Tom` connected from host `159.101.80.5` with symbolic name `hole.admin.hospital.com`. Since `Tom` is a member of group `Administrative`, he can access administrative information such as the name, home address and salary of the medical staff. By contrast, not belonging to the medical staff he cannot access medical information on patients (e.g., illness, type of therapy and drug name). Figures 9(a) and 9(b) show the resulting view on the document returned to `Alice` and `Tom`, respectively. These views reflect a general principle according to which each user can access only information needed to complete his activity (*need-to-know* principle [Samarati and De Capitani di Vimercati 2001]): `Alice`'s view contains all the information that she might need as a nurse, while `Tom`'s view contains information related to administrative tasks. Neither `Alice` nor `Tom` have a full view on the document. ◇

We conclude this section with a mention at the performance characteristics of our system. The two tasks that have the greatest impact on performance are the identification of the authorization objects and the evaluation of node labels.

Authorization objects are identified by XPath expressions, which are evaluated

Fig. 9. The view of user `Alice` (a) and the view of user `Tom` (b).

inside the **InitialLabel** procedure. This step dominates the complexity in the system, as XPath is a rich programming language that permits the definition of search expressions requiring exponential time for their evaluation [Mendelzon and Wood 1995]. Several works identify restrictions on XPath that reduce its complexity characteristics [Buneman et al. 2000; Deutsch and Tannen 2001; Mendelzon and Wood 1995], keeping a level of expressivity adequate for most situations.

The evaluation of node labels, while expressive and flexible, bears limited computational cost. This can be noticed by quickly evaluating Algorithm 6. In fact, the tree labeling initialization (**InitialLabel**) is linear in the number of authorizations associated with the document, while the label computation (**GetFinalLabel**) and the pruning process **Prune**) are linear in the number of nodes in the document. The most expensive operation seems therefore the set label computation (**SetLabel**), that, for each node in the tree enforces the conflict resolution policy for solving inconsistencies among authorizations of the same type. In principle such an operation could have a worst-case cost, for each node, quadratic in the highest number of authorizations of a given type that are associated with the node (subject comparison for specificity being assumed constant [Raynaud and Thierry 2001]). It is however legitimate to assume the number of such authorizations to be very small, and limited by a reasonable constant, making the **SetLabel** method also linear in

the number of nodes in the document.

7. SUPPORTING WRITE ACTIONS

In previous sections, we treated only read authorizations. This is justified by practical considerations, as currently XML applications are mostly read-only. Read authorizations permit also a simpler description of the approach, since the emergence of conflicts is easier to understand if only read privileges are involved. Finally, whereas read operations can be immediately modeled, no consensus has emerged up to now in the research community on a model for XML updates. In this section, we introduce a basic model for XML writes that permits to introduce write authorizations. Richer models have already been defined for the representation of XML writes (e.g., [Abiteboul et al. 1999; Goldman et al. 1999; Liefke and Davidson 2000] and write authorizations could be specialized for a specific model, with direct support for the complex operations that the model offers (e.g., movement of nodes in the Lorel model [Goldman et al. 1999] or merge of XML trees in the WHAX model [Liefke and Davidson 2000]), but these are specific customizations that we do not treat here. Like read authorizations, write authorizations can be local or recursive, hard and soft, and can be specified on elements/attributes within either single XML documents or DTDs. The semantics of local and recursive authorizations remains unchanged; local authorizations on elements/attributes apply only to the considered elements/attributes while recursive authorizations apply also to their sub-elements. Write authorizations specified at DTD level apply to all the DTD instances and write authorizations specified at instance level apply only to the document on which they are defined. Conflict resolution is applied in the same way as for read operations, with a complete separation among authorizations on different actions.[6]

We define write operations with a basic model where only operations on single nodes are considered. The operations on the node can be `insert` the node, `delete` the node, and `update` the node (i.e., change the value for an attribute or a change of the text for an element). The three operations correspond to distinct write privileges: `insert`, `delete` and `update`. Note that insert privileges allow the insertion of new elements and attributes in a document, which, although not existing before the insert operation itself, can be specified with reference to the schema (DTD). For instance, an insert authorization on element `therapy` of a document allows the insertion of a new therapy within the document. The consideration of the three write privileges above offers most of the services required for an access control mechanism on write operations. Being at low level, it is also compatible with many models for the representation of XML operations (e.g., graph-based [Goldman et al. 1999], object-based [Abiteboul et al. 1999], tree-based [Liefke and Davidson 2000], or object-relational [Oracle Corp. 2000]), which can all be mapped in terms of simple operations.

Insert operations are evaluated by executing the labeling process on the document with the new node inserted. If the labeling produces a positive label on

[6]The model could be extended to the support of action hierarchies in the form of authorization implication (e.g., a write access implies a read access) or pre-condition (e.g., a write access on an element requires at least a read access on the element's ancestors) [Sandhu and Samarati 1997].

the new node the insert complete successfully, otherwise the document remains invaried. Note that possible conditions associated with authorizations can be exploited to specify restrictions on the document where the insertion can be executed as well as on the values that can be inserted. For instance, authorization ⟨⟨PhyC,*,*.hospital.com⟩,/department[./@name='medicine']/patient//therapy [./cost < 10,000],insert,+,R⟩ states that members of group PhyC can insert a new therapy for a patient in the Medicine department, and that the privilege is limited to insertion of therapies whose cost does not exceed $10,000.

Deletion of a node is permitted only if the labeling of the document produces a final positive label for the node to be deleted. If so the node is eliminated and will not be present in the new document. For instance, authorization ⟨⟨Administrative,159.101.80.77,secws.hospital.com⟩,/department/medical_staff// bonus,delete,+,L⟩ states that users of the Administrative group connected from the specified location have the privilege to delete element bonus of the members of the medical staff.

Updates are evaluated by executing the labeling process on both the existing document and on the document that results from the execution of the update. If the final label associated with the node being updated is positive in both versions the operation is completed successfully, otherwise it is rejected. For instance, authorization ⟨⟨NurseC,*,*.hospital.com⟩,/department[./@name='medicine']/patient/room/bed[number(value())>=100 and number(value())<=150],update,+,L⟩ specifies that members of group NurseC can update element bed of patients, only if the bed belongs to the block of beds 100 through 150. They are thus responsible for updating the distribution of patients in the block, but they are not allowed to transfer patients across blocks. From the example it is then clear that the reason for considering both the original and the updated version of the document is to require the satisfaction of the the condition in the authorizations in both the old and the new document's state [Atzeni et al. 1999].[7]

When the XML document is modified, the system must also check the correctness of the document with respect to the DTD and, if the document is not valid, the write must not be accepted. Incidentally, if the document is characterized by a loosened DTD, this check can have an impact only on insertions and updates: insertions can introduce elements with a tag incompatible with the DTD, and updates can specify a value in contrast with what the DTD specifies for the node; deletions are always accepted. With a generic DTD and possibly with semantically richer specifications, like those proposed by XML Schema [Thompson et al. 2001; Biron and Malhotra 2001], the constraints that must be verified on the document can become quite complex and checks are required for every action.

The model above considers write operations on single nodes. However, write requests often refer to sets of nodes (typically a subtree in the document). In this case it may be convenient to introduce a transactional mechanism based on the

[7]We note that alternative approaches could interpret the conditions in the authorizations only as conditions on the values being updated, in which case only the original document should be labeled (like for the delete operation) or only as conditions on the new values being introduced, in which case only the new document should be labeled (like for the insert operation).

"deferral" of controls, analogous to the SQL command `set constraints deferred` that relational systems offer for the management of constraints. The idea is that writes are collected in an atomic sequence, and all the checks on the correctness of the updates are deferred at the end of the sequence, when each single write is considered for permissions and correctness. If a single write is not permitted, the sequence is invalid and the XML data is rolled back to its original state. The transactional mechanism also allows the support of coordinated write operations which are individually incorrect but globally correct. We can consider this example. Suppose a user is given the authorization to manage the description of patients that are in the `Medicine` department and are in a critical condition. If a control is executed on every single write of a node, the user may be forbidden to insert new patients: the insertion of a new patient node is not allowed because the patient node has no other node and the user is not permitted to modify patients who are not in a critical condition. By contrast, deferring the control at the end of the update sequence, the write operation can be successfully completed.

8. SOFTWARE ARCHITECTURE

We briefly present the architecture of the *XML Access Control Processor* (ACP), the component which wraps up entirely the computation of access permissions to individual elements and performs the corresponding transformation on the XML document.

8.1 Architectural Requirements

From the architectural point of view, a first point to note is that our design is fully *server side*. The choice between *server-side* and *client-side* processing is a typical one when XML is used in the Web context [Park et al. 2001]. As the access control process should clearly be trusted to operate correctly and restrict each requestor to the data he is entitled to access, it is highly preferable to use server-side processing, otherwise protected information should be transmitted to the client and a complex infrastructure should be implemented to guarantee clients' trust in properly enforcing the access restrictions. The satisfaction of the requirements specific to the protection of information, is not sufficient to guarantee a successful adoption of this technology in real applications. Indeed, the following requirements must also be considered.

—**Seamless integration**: XML access control should be provided as seamlessly as possible, without interfering with the operation of other presentation or data-processing services. Moreover, the access control service should be introduced on existing servers with minimal interruption of their operation.

—**Quality of service**: The emergence of the World Wide Web as a mainstream technology has highlighted the problem of providing a high quality of service (*QoS*) to application users. This factor alone should caution about the risk of increasing substantially the processing load of Web servers.

With respect to the first requirement, the key technology used for the integration of access control with other services is the *Document Object Management* (DOM) specification [World Wide Web Consortium (W3C) 1998], an API defined by the W3C to process XML information. Several systems implement the DOM interface,

Technique	Advantages	Disadvantages
Single threaded	No context switch overhead.	Not scalable on multiprocessors
Process-per-request	Portability	Resource intensive
Process pool	No process creation costs	Not available on every OS
Thread-per-request	Speed	Requires mutual exclusion
Thread pool	Speed	Mutual exclusion on some OS

Table 3. Advantages and disadvantages of concurrency control techniques

in various languages; each of these systems offers services for the bidirectional transformation between the textual representation of an XML file and an internal proprietary representation, on which the methods of the API operate. The use of the DOM API offers a great potential in the integration of different components managing XML information, because each component can be designed as a DOM transformer independent from the others. For instance, our access control processor can operate on the DOM representation produced by a cache manager and the result of its simplification may be passed to an XML query engine computing a new document.

The system has been designed following the principles of object-orientation and is based on the specification of a set of classes. The classes have been defined in an abstract way using the *Interface Definition Language* (IDL) [Mowbray and Malveau 1997]. The classes are organized into two families. The first family is an extension of the DOM class hierarchy and enriches the description of the nodes of a document with the required security attributes. This extension makes use of the inheritance mechanism that permits an immediate integration with existing DOM implementations. The second family of classes is strictly related to the processing of the access control model and describes all the concepts that are part of the model, like authorization signs, subjects, path expressions, and complete authorizations.

The *quality of service* issue centers on performance. Several techniques are available to implement Web-based, high-concurrency systems, each having its positive and negative aspects, as illustrated in Table 3. Multi-threaded designs are currently the preferred choice for Web servers, as the cost of spawning a thread is usually much lower than that of a process. This is also our design choice for our ACP implementation. Beside being usable in a single-thread execution, our processor can be easily interfaced to a *Dispatcher* registered with an *Event Handler*. Dispatcher-based multi-threading can be managed *synchronously*, according to the *Reactor/Proactor* design pattern [Lavender and Schmidt 1995], or *asynchronously*, as in the *Active Object* pattern. We adopted the former choice, as it further facilitates the integration of our XML access control code in the framework of existing general-purpose server-side transformers based on the same design pattern (like *Cocoon* [Apache Software Foundation 2000]). To obtain a multi-threaded system, the classes must be implemented in a thread-safe way, making all the parameters of each request isolated in the context of a specific method invocation.

An architectural solution that further enhances the performance is the separation of the threads managing the user hierarchy. The services that compute if an authorization is applicable to a given user require to evaluate if a user is a member, directly or indirectly, of the group specified in the authorization's subject. The efficiency of this computation can be greatly increased building auxiliary in-

```
<!ELEMENT set_of_authorizations (authorization)+>
<!ELEMENT authorization (subject,object,action,sign,type)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT object (#PCDATA)>
<!ELEMENT action empty>
<!ELEMENT sign empty>
<!ELEMENT type empty>

<!ATTLIST set_of_authorizations  about CDATA #REQUIRED>
<!ATTLIST type  value (L|R|LDH|RDH|LS|RS|LD|RD) #REQUIRED>
<!ATTLIST sign  value (+ | -) #REQUIRED>
<!ATTLIST action  value (read) #REQUIRED>
```

Fig. 10. XAS syntax.

dexes [Agrawal et al. 1989] that synthetically describe the memberships in groups. These structures are expensive to build and describe relatively static information, thus it is best to build them once in a thread that stays always active and offers its services to the threads managing the document transformation.

Explicit synchronization mechanisms must be used to ensure that conflicting write requests for shared resources are correctly managed. In the ACP object, the only shared resource where writes can occur is the user hierarchy, but requests for user/group changes are infrequent, thus the synchronization does not have an impact on system performance.

8.2 Execution Phases

We are now ready to describe how the ACP works. Our processor takes as input a valid XML document requested by the user, together with its *XML Access Sheet* (XAS) listing the associated access authorizations at the instance level. The processor operation also involves the document's DTD and the associated XAS specifying schema level authorizations. The processor output is a valid XML document including only the information the user is allowed to access. To provide a uniform representation of XASs and other XML-based information, the syntax of XASs is given by the XML DTD depicted in Figure 10. The XASs associated with an XML document and its DTD are located by relying on the abstract nature of XML *XLink* specification [DeRose et al. 2001] to define *out-of-line* links that reside *outside* the documents they connect, making links themselves a viable and manageable resource. The repertoire of out-of-line links defining access control mappings is itself an XML document, easily managed and updated by the system manager; nonetheless it is easily secured by standard file-system level access control.

Our security processor computes an *on line transformation* on XML documents. Its execution cycle, illustrated in Figure 11, consists of three basic steps:

(1) **Parsing** The parsing step consists in the syntax check of the requested document with respect to the associated DTD and its compilation to obtain an *object-oriented document graph* according to the DOM format. Since the parsing is performed externally when the ACP is used as a transformer in the framework of a complete architecture complying to the well-know *Pipes and Filters* design pattern [Buschmann et al. 1996], here we do not deal with parsing issues in detail.

(2) **Compute View** The compute view step determines the requester's view, by applying the algorithm presented in Section 6 and according to the authorizations listed in the XASs associated with the document and its DTD. As already discussed, the resulting view preserves the validity of the document with respect to the *loosened version* of its original DTD.

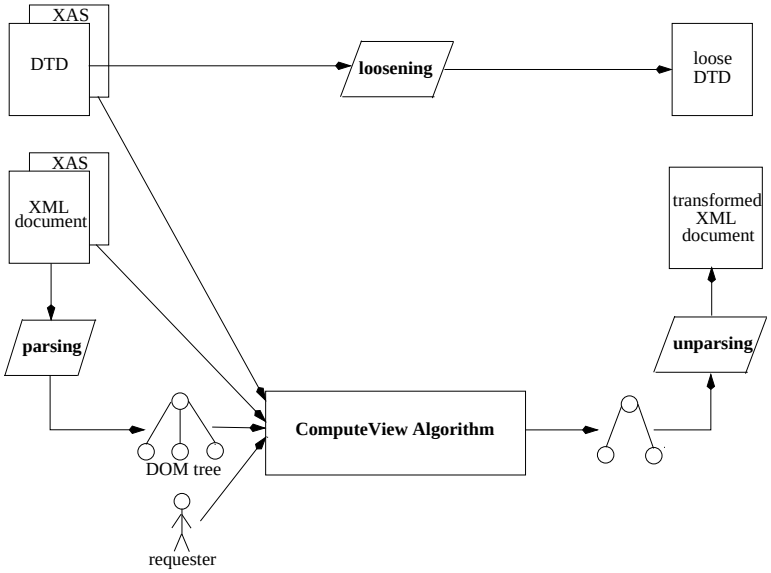


Fig. 11. Execution steps of the security processor.

(3) **Unparsing** Finally, the third step is the generation of a valid XML document in text format, simply by unparsing (again, by means of a standard component) the DOM tree computed by the previous step. Once again, this step is performed externally when the ACP is executed as a transformer in the framework of a *Pipes and Filters* system.

8.3 The Java Implementation

We designed the prototype of the access control model (`http://seclab.dti.unimi.it/~xml-sec`) in Java, using the services of a Java implementation of the DOM API (we used the IBM's XML4J processor [AlphaWorks 2001], which has now evolved into the Apache's Xalan tool [Foundation 2001]). This choice had several consequences on the behavior of our prototype. Here we make a few observations that strictly depend on this choice.

First, it must be noted that no Java-based design of multi-threading components has full control on thread management: when running on an operating system that supports threads, the *Java Virtual Machine* (JVM) automatically maps Java threads to native threads [Lea 1996], while when no native thread support is available, the JVM has to emulate threads. In the latter case, the emulation technique chosen by the JVM implementors can have a significant impact on performance.

Another interesting point to discuss is the mechanism used for the integration of the ACP with the HTTP server. A simple technique consists in the adoption of the *Common Gateway Interface* (CGI), which allows a Web server to run a generic executable when managing a request; this solution is available independently from the language used to implement the IDL specification. A Java implementation offers specific solutions and indeed our prototype is invoked by the HTTP server

using Java *servlets*. The servlet specification [Sun Microsystems 1999b] defines a protocol for the exchange of information between the server and the JVM, where a request to the server for a resource identified by a URL generates the invocation of the JVM from the server, passing all the parameters part of the request. Overall, servlets constitute an interesting solution for the implementation of Web services, offering a higher level interface than CGI, with a relatively easy integration between the HTTP server and the Java environment. Other solutions, specific for Java, are available for the prototype implementation. In particular, for the next version of the prototype, we are investigating the use of *Java Server Pages* (JSP) [Sun Microsystems 1999a], a technology built upon servlets offering template-based invocation of Java services.

9. CONCLUSIONS

The definition of an authorization mechanism for protecting data offered on Web sites is an important research direction and a practical pressing need. Existing proposals, specifying protection requirements at the file system level or with reference to the HTML constructs, turn out to be very limited. By exploiting the opportunities offered by XML, we have defined an access control model for restricting access to Web documents that takes into consideration the semi-structured organization of data and their semantics. The result is an access control system that, while powerful and able to easily represent different protection requirements, proves simple and of easy integration with existing applications. Our proposal leaves space for further work. Issues to be investigated include: the consideration of requests in form of generic queries, extension of the model to role-based and credential-based authorizations, and the investigation of performance optimizations.

ACKNOWLEDGMENTS

The work reported in this paper was partially supported by the Italian MURST DATA-X project and by the European Community within the Fifth (EC) Framework Programme under contract IST-1999-11791 – FASTER project.

REFERENCES

Abiteboul, S., Amann, B., Cluet, S., Eyal, A., Mignet, L., and Milo, T. 1999. Active views for electronic commerce. In *Proceedings of 25th International Conference on Very Large Data Bases* (Edinburgh, Scotland, UK, September 1999), pp. 138–149.

Adler, S., Berglund, A., Caruso, J., Deach, S., Graham, T., Grosso, P., Gutentag, E., Milowski, A., Parnell, S., Richman, J., and Zilles, S. 2001. *Extensible Stylesheet Language (XSL) Version 1.0*. World Wide Web Consortium (W3C). http://www.w3.org/TR/xsl.

Agrawal, R., Borgida, A., and Jagadish, H. V. 1989. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the ACM SIGMOD'89 Conference* (Portland, Oregon, May-June 1989), pp. 253–262. ACM Press.

AlphaWorks. 2001. XML Security Suite. http://www.alphaWorks.ibm.com/tech/xmlsecuritysuite.

Apache Software Foundation. 2000. Cocoon, a Java publishing framework. http://xml.apache.org/cocoon.

Atzeni, P., Ceri, S., Paraboschi, S., and Torlone, R. 1999. *Database Systems - Concepts, Languages and Architectures*. McGraw-Hill.

Berners-Lee, T., Fielding, R., Irvine, U., and Masinter, L. 1998. Uniform Resource Identifiers (URI): Generic syntax. http://www.isi.edu/in-notes/rfc2396.txt.

Bertino, E., Castano, S., and Ferrari, E. 2001. Securing XML documents with author-x. *IEEE Internet Computing 5*, 3 (May/June), 21–31.

Bertino, E., Castano, S., Ferrari, E., and Mesiti, M. 2000. Specifying and enforcing access control policies for XML document sources. *World Wide Web Journal 3*, 3.

Biron, P. and Malhotra, A. 2001. *XML Schema Part 2: Datatypes*. World Wide Web Consortium (W3C). http://www.w3.org/TR/xmlschema-2.

Bray, T., Hollander, D., and Layman, A. 1999. *Namespaces in XML*. World Wide Web Consortium (W3C). http://www.w3.org/TR/REC-xml-names.

Bray, T., Paoli, J., Sperberg-McQueen, C., and Maler, E. 2000. *Extensible Markup Language (XML) 1.0 (Second Edition)*. World Wide Web Consortium (W3C). http://www.w3.org/TR/REC-xml.

Brickley, D. and Guha, R. 2000. *Resource Description Framework (RDF) Schema Specification 1.0*. World Wide Web Consortium (W3C). http://www.w3.org/TR/rdf-schema.

Buneman, P., Fan, W., and Weinstein, S. 2000. Path constraints in semistructured databases. *JCSS 61*, 2, 146–193.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons Ltd.

CheckFree Corp. 2001. *Open Financial Exchange Specification 2.0.1*. CheckFree Corp. http://www.ofx.net/.

ContentGuard. 2001. *eXtensible rights Markup Language (XrML) 2.0*. ContentGuard. http://www.xrml.org/.

Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., and Samarati, P. 2000a. Design and implementation of an access control processor for XML documents. *Computer Networks 33*, 1–6 (June), 59–75.

Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., and Samarati, P. 2000b. Securing XML documents. In *Proceedings of EDBT 2000*, Volume 1777 of *Lecture Notes in Computer Science* (Konstanz, Germany, March 2000), pp. 121–135. Springer.

DeRose, S., Maler, E., and Orchard, D. 2001. *XML Linking Language (XLink) Version 1.0*. World Wide Web Consortium (W3C). http://www.w3.org/TR/xlink.

Deutsch, A. and Tannen, V. 2001. Containment and integrity constraints for xpath. In *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases* (Rome, Italy, September 2001).

Devanbu, P., Gertz, M., Kwong, A., Martel, C., Nuckolls, G., and Stubblebine, S. 2001. Flexible authentication of XML documents. In *Proceedings of the 8th ACM Conference on Computer and Communications Security* (Philadelphia, PA, USA, November 2001).

Eastlake, D., Reagle, J., and Solo, D. 2001. *XML-Signature Syntax and Processing*. ftp://ftp.rfc-editor.org/in-notes/rfc3075.txt.

Ellerman, C. 1997. Channel definition format (CDF). http://www.w3.org/TR/NOTE-CDFsubmit.html.

Fernandez, E., Gudes, E., and Song, H. 1994. A model of evaluation and administration of security in object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering 6*, 2 (April), 275–292.

Foundation, A. S. 2001. Xalan-J version 2.2.d14. http://xml.apache.org/xalan-j/.

Gabillon, A. and Bruno, E. 2001. Regulating access to XML documents. In *Proceedings of the Fifteenth Annual IFIP WG 11.3 Conference on Database Security* (Niagara on the Lake, Ontario, Canada, July 2001).

Goldman, R., McHugh, J., and Widom, J. 1999. From semistructured data to XML: Migrating the lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)* (Philadelphia, Pennsylvania, June 1999).

Jajodia, S., Samarati, P., Sapino, M., and Subrahmanian, V. 2001. Flexible support for multiple access control policies. *ACM Transactions on Database Systems 26*, 2 (June), 214–260.

JONSCHER, D., MOFFETT, J., AND DITTRICH, K. 1994. Complex objects or: The striving for complexity is ruling our world. In T. KEEFE AND C. LANDWEHR Eds., *Database Security, VII: Status and Prospects* (1994), pp. 19–37. Elsevier Science Publishers B.V. (North-Holland).

KUDOH, M., HIRAYAMA, Y., HADA, S., AND VOLLSCHWITZ, A. 2000. Access control specification based on policy evaluation and enforcement model and specification language. In *Symposium on Cryptograpy and Information Security, SCIS'2000* (2000).

LAVENDER, R. G. AND SCHMIDT, D. 1995. Reactor: A object behavioral pattern for concurrent programming. In J. VLISSIDES, D. COPLIEN, AND M. KERTH Eds., *Pattern Languages of Program Design 2*. Addison Wesley.

LEA, D. 1996. *Concurrent Programming in Java*. Addison Wesley.

LIEFKE, H. AND DAVIDSON, S. 2000. View maintenance for hierarchical semistructured data. In *Proceedings of Data Warehousing and Knowledge Discovery (DaWaK 2000)* (London Greenwich, UK, Sept. 2000).

LUNT, T. 1989. Access control policies for database systems. In C. LANDWEHR Ed., *Database Security, II: Status and Prospects* (1989), pp. 41–52. North-Holland, Amsterdam.

MENDELZON, A. AND WOOD, P. 1995. Finding regular simple paths in graph databases. *SIAM J. Comput 24*, 6, 1235–1258.

MOWBRAY, T. AND MALVEAU, R. 1997. *CORBA Design Patterns*. John Wiley & Sons.

ORACLE CORP. 2000. Oracle and XML. http://www.oracle.com/xml.

PARK, J., SANDHU, R., AND AHN, G. 2001. Role-based access control on the web. *ACM Transactions on Information and Systems Security 4*, 1 (February), 37–71.

RABITTI, F., BERTINO, E., KIM, W., AND WOELK, D. 1991. A model of authorization for next-generation database systems. *ACM TODS 16*, 1 (March), 89–131.

RAYNAUD, O. AND THIERRY, E. 2001. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)* (Budapest, Hungary, June 2001).

REAGLE, J. AND CRANOR, L. 1999. The platform for privacy preferences. *Communications of the ACM 42*, 2 (February), 48–55.

RESCHER, N. 1969. *Many Valued Logics*. Mc Graw-Hill, New York.

RESCORLA, E. AND SCHIFFMAN, A. 1999. The secure hypertext transfer protocol. http://www.ietf.org/rfc/rfc2660.txt.

SAMARATI, P., BERTINO, E., AND JAJODIA, S. 1996. An authorization model for a distributed hypertext system. *IEEE Transactions on Knowledge and Data Engineering 8*, 4 (August), 555–562.

SAMARATI, P. AND DE CAPITANI DI VIMERCATI, S. 2001. Access control: Policies, models, and mechanisms. In R. FOCARDI AND R. GORRIERI Eds., *Foundations of Security Analysis and Design*, LNCS 2171. Springer-Verlag.

SANDHU, R. AND SAMARATI, P. 1997. Authentication, access control and intrusion detection. In A. TUCKER Ed., *CRC Handbook of Computer Science and Engineering*, pp. 1929–1948. CRC Press Inc.

SUN MICROSYSTEMS. 1999a. Java Server Pages (JSP) specification, release 1.1. http://www.javasoft.com/.

SUN MICROSYSTEMS. 1999b. Java servlet API specification, release 2.2. http://www.javasoft.com/.

THOMPSON, H., BEECH, D., MALONEY, M., AND MENDELSOHN, N. 2001. *XML Schema Part 1: Structures*. World Wide Web Consortium (W3C). http://www.w3.org/TR/xmlschema-1.

VAN HOFF, A., PARTOVI, H., AND THAI, T. 1997. The open software description format (OSD). http://www.w3.org/TR/NOTE-OSD.html.

WOO, T. AND LAM, S. 1993. Authorizations in distributed systems: A new approach. *Journal of Computer Security 2*, 2,3 (February), 107–136.

World Wide Web Consortium (W3C). 1998. *Document Object Model (DOM) Level 1 Specification Version 1.0*. World Wide Web Consortium (W3C). http://www.w3.org/TR/REC-DOM-Level-1.

World Wide Web Consortium (W3C). 2001. *XML Path Language (XPath) 2.0*. World Wide Web Consortium (W3C). http://www.w3.org/TR/xpath20.