# An Access Control System for SVG Documents

**E. Damiani**[1] **S. De Capitani di Vimercati**[2] **E. Fernández-Medina**[3] **P. Samarati**[1]

(1) Dip. di Tecnologie dell'Informazione - Università di Milano - 26013 Crema - Italy

{damiani,samarati}@dti.unimi.it

(2) Dip. di Elettronica per l'Automazione - Università di Brescia - 25123 Brescia - Italy

decapita@ing.unibs.it

(3) Escuela Superior de Informática - University of Castilla-La Mancha - 13071, Ciudad Real - Spain

Eduardo.FdezMedina@uclm.es

**Abstract**

Selectively controlling access to images' features is becoming more and more essential as modern computing and communications encourage the use of information in graphical form. The monolithic nature of traditional raster images makes controlled dissemination of their internal features a difficult task. Recently, however, XML-based graphics formats such as the Scalable Vector Graphics (SVG) standard, are becoming increasingly popular due to their recognized advantages in terms of application interoperability. In this paper we exploit the XML-based data model of SVG to present a model and a syntax aimed at selectively controlling access to graphic information on the Internet. Our model explicitly deals with the basic notions underlying representation of complex objects via vector graphics, while our enforcement technique efficiently exploits lower level DOM representation of XML data.

**Keywords:** Access Control, Multimedia Documents, SVG format, XML documents.

# 1 Introduction

Vector graphics is a time-honored technique that uses geometrical formulas to represent images, achieving more flexibility than usual raster graphics relying on bit maps. For instance, vector-oriented images can be resized and stretched without any loss of image quality; also, repetitive geometric elements can be defined once and used many times, so that high-quality vector images often require less memory than lower quality bit-mapped ones. While in the past vector graphics was confined to computationally intensive design applications, it is now spreading to new application fields. An increasing amount of the multimedia information being transmitted over the Internet is in the form of vector image data, encoded by means of new XML-based standards such as the World Wide Web Consortium's *Scalable Vector Graphics* (SVG) [13], which allows describing two dimensional vector graphics (specifically vector graphic shapes, images, and text) for

storage and distribution on the Web. In contrast to raster image format such as GIF, JPEG, and PNG, SVG has many advantages. Specifically:

- SVG documents are plain text, so they can be read and modified easily.
- Being SVG a vector format, SVG images can be resized without loss of quality and printed at any resolution. Also, graphical objects can be easily grouped, restyled and transformed.
- Sophisticated interactive and dynamic applications of SVG are made possible by the *Document Object Model* (DOM) [14] underlying all XML-based formats. User interaction is managed via a rich set of *event handlers* that can be assigned to any SVG graphical object.
- SVG offers all the advantages of XML, including interoperability, internationalization (via its support of the Unicode character sets), XSLT restructuring capability (`http://www.w3.org/Style/XSL/`) and easy manipulation through standard DOM APIs.

The current trend toward XML-based vector graphics is affecting different types of data, such as technical plans, organizational charts and diagrams, as well as medical images used in diagnosis and research. While controlling access to text-based documents has since long been a focus of research activities [12], raster graphic information has been seldom processed with much concern for access control, mainly because of its monolithic internal structure: either a user is allowed to see a bitmap image, or she is not. On the other hand, XML-based vector images present new and challenging *feature protection* problems, related to fine-grained access control to their internal structure. Of course, the feature protection problem could also be solved by storing graphical data in multiple copies at different levels of detail; but this solution is seldom practical. For instance, in a hospital, if some MRI-scan images are to be released for research purposes, they must be duplicated omitting any identifying information, making their distribution slow and costly [17, 18]. In this paper, we present a novel approach to fine-grained feature protection of Scalable Vector Graphics (SVG) data, one of the most successful XML standards for exchanging and representing graphics information. Our approach allows to selectively transform SVG graphical data according to the user's profile, releasing only the features that the user is entitled to see. While leveraging on our proposal for protecting XML sources [5, 6], the approach presented in this paper exploits the peculiar characteristics of SVG documents. Also, it provides a simple, yet expressive, solution for specifying authorization subjects. The paper is organized as follows. Section 2 provides a concise illustration of SVG and describes the example to which discussions in the paper will refer. Section 3 presents our authorization model for the specification of protection requirements on SVG documents together with a language for expressing them. Our language, called SVG-FPL (SVG Feature Protection Language), provides expressiveness and flexibility by exploiting XML-based profiles for describing authorization subjects and by exploiting the basic objects of vector graphics for fine-grained reference to individual elements in SVG documents. Section 4 illustrates the enforcement process executed server-side at every access request to obey the stated protection requirements. As a

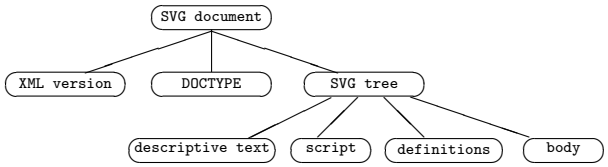



Figure 1: General Structure of an SVG Document

result of the enforcement, the requestor may be returned only a partial view of the original SVG document containing only the objects and functions that she is entitled to access. Finally, Section 5 presents our conclusions.

# 2 A concise overview of SVG

An SVG document has a flexible structure, composed of several optional elements placed in the document in an arbitrary order. Figure 1 shows the general structure used. Nodes `XML Version` and `DOCTYPE` are common for any XML-based document and specify the XML version used in the document and information about the type of the document (the public identifier and the system identifier for SVG 1.0), respectively. Node `SVG` contains all the elements specific to SVG documents and is composed of four parts: descriptive text, script, definitions, and body. The descriptive text includes textual information not rendered as part of the graphic and is represented by two elements: `<title>`, usually appearing only once, and `<desc>`, appearing several times to describe the content of each SVG fragment. The script portion contains function definitions. Each function is associated with an action that can be executed on SVG objects in the document. Functions have a global scope across the entire document. The definition portion contains global patterns and templates of graphical elements or graphical properties that can be reused in the body of the SVG document. Each definition is characterized by a name, which is used in the body of the document to reference the definition, and by a set of properties. The graphical elements to be rendered are listed after the `<defs>` node, according to the order of rendering. Each element can belong to any of the basic SVG graphics elements, such as `path`, `text`, `rect`, `circle`, `ellipse`, `line`, `polyline`, `polygon`, and `image`, whose names are self-explanatory. The body of an SVG document contains any number of container and graphics elements. A container element can have graphics elements and other container elements as child elements. Container `<g>` is used for *grouping together* related graphics elements. A graphics element can cause graphics to be drawn. For instance, the `use` graphics element references another element (usually a definition) and indicates that the graphical contents of that element must be drawn at that specific point in the document. Each SVG element may have its own properties, modeled by XML attributes. All elements in the document can be uniquely identified including the special attribute `id='identifier'`. It is also possible to include user-defined properties, which can be useful for SVG data processing.

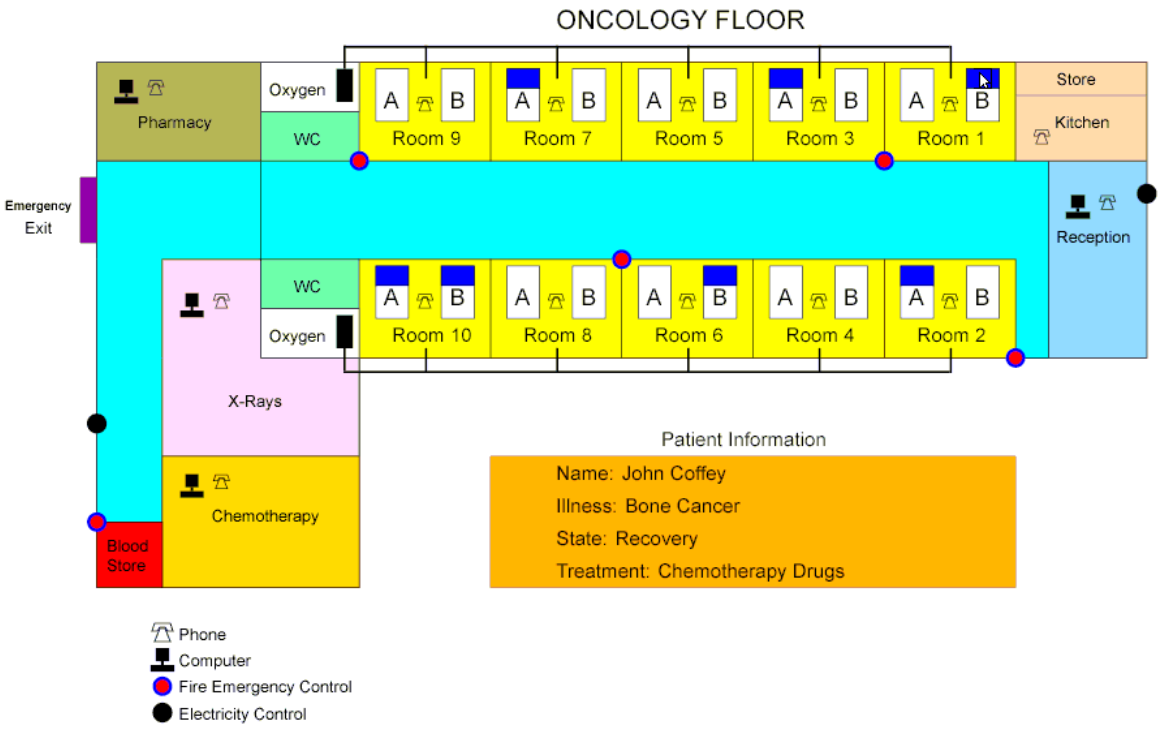


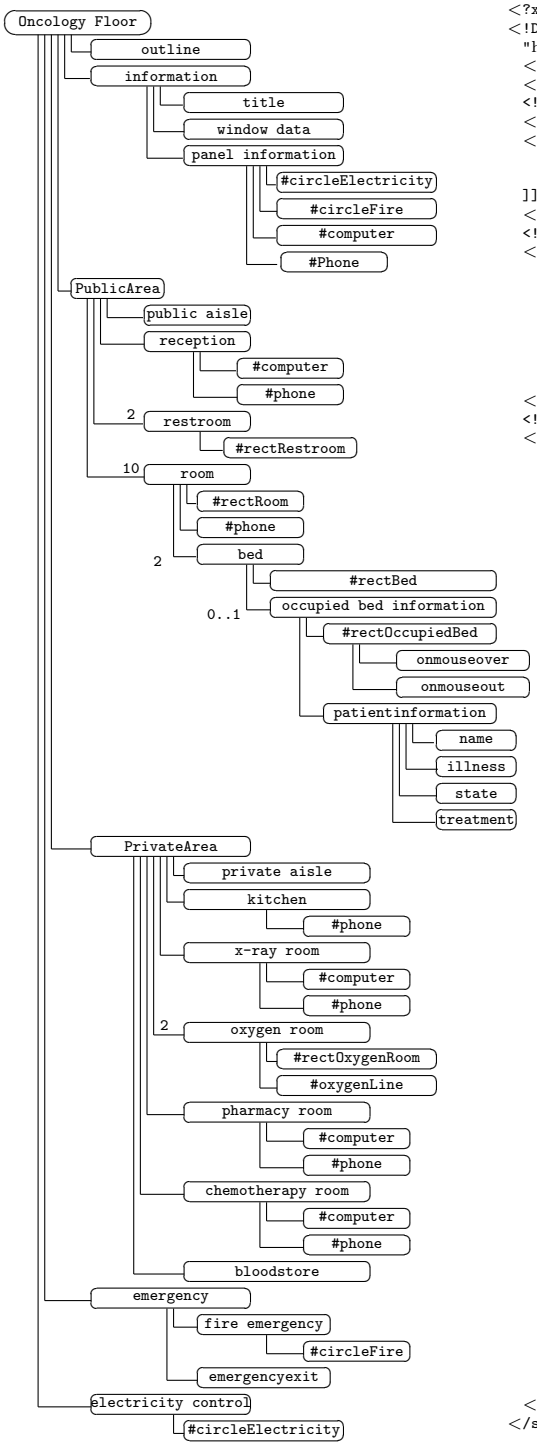

Figure 2: Example of an SVG document

## 2.1 Running example

Figure 2 illustrates the rendering of a sample SVG document, showing the oncology floor of a hospital, which will be used as a running example throughout the paper. The document, integrated in a web site, allows the hospital staff to know both details of the floor (e.g., rooms and equipments location) and recovered patient information.[1] In particular, the rectangular appearing at the bottom with the text provides the information of the patient of bed `1B` on which the mouse is currently positioned (moving the mouse on other beds the corresponding patient will be returned).

Figure 3(a) shows a tree-based representation of the document rendered in Figure 2, reporting the types associated with the group elements composing its body. In particular, the body is a group element with `oncologyfloor` as identifier and with sub-elements of type `outline`, `information`, `public area`, `private area`, `emergency` and `electricity control` (the document defines one group for each of them). Group `public area` includes `public aisle`, `reception`, two `restroom` instances, and ten `room` instances. Each room, in turn, is composed of a graphical representation (`rectRoom` definition), a name, and two beds. Each bed is composed of a graphical representation (`rectBed` definition) and a bed name. Occupied beds further include a group

[1] While obviously in a real case there would be much more details, the example already gives the ideas of graphically representing all details of a building, integrating both graphics and text in order to have a very powerful interactive query tool.




(a)

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
 "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
 <svg width="24cm" height="16cm" viewBox="900 400 3600 2200">
 <title> Small Fragment of a Oncology Floor </title>
 <!-- SCRIPT portion starts here -->
 <script type="text/ecmascript">
 <![CDATA[
  function display_information(evt) { ......... }
  function hide_information(evt) { ......... }
 ]]>
 </script>
 <!-- DEFINITION portion starts here -->
 <defs>
  <rect id="rectRoom" width="400" height="300" stroke="black" fill="beige"/>
  <rect id="rectBed" width="100" height="160" stroke="black" fill="white"/>
  <rect id="rectOccupiedBed" width="100" height="60" stroke="black" fill="blue"/>
  <symbol id="computer" viewBox="0 0 20 20"> ..... </symbol>
  <symbol id="phone" viewBox="0 0 20 20"> ..... </symbol>
  <linearGradient id="MyGradient"> ..... </linearGradient>
  .........
 </defs>
 <!-- BODY portion starts here -->
 <g id="oncologyfloor">
  <g id="information">
   <g id="title">
    <text x="1920" y="700" font-size="80"> ONCOLOGY FLOOR </text>
   </g>
   <g id="window-data">
    <rect id="data" fill="url(#MyGradient)" x="1500" y="1500"
     width="1600" height="400" stroke="black" />
    <text x="2020" y="1470" font-size="60"> Patient Information </text>
    <text x="1700" y="1570" font-size="60"> Name:  </text>
    <text x="1700" y="1670" font-size="60"> Illness:  </text>
    <text x="1700" y="1770" font-size="60"> State:  </text>
    <text x="1700" y="1870" font-size="60"> Treatment:  </text>
   </g>
   .........
  </g>
  .........
  <g id="room1" typeElement="room" >
   <use x="2700" y="300" xlink:href="#rectRoom" />
   <g typeElement='content'>
    <text x="2800" y="550" font-size="60"> Room 1 </text>
    <use x="2870" y="400" width="60" height="60" xlink:href="#phone" />
    <g id="A" typeElement="bed" >
     <use x="2750" y="320" xlink:href="#rectBed" />
     <text x="2775" y="440" font-size="70" > A </text>
    </g>
    <g id="B" typeElement="bed" >
     <use id="1B" x="2950" y="320" xlink:href="#rectBed" />
     <text x="2975" y="440" font-size="70" > B </text>
     <g id="bed1B" typeElement="occupiedBedInforamtion">
      <use id="1B" x="2950" y="320" xlink:href="#rectOccupiedBed"
      onmouseover="display_information(evt)" onmouseout="hide_information(evt)"/>
      <g id="patientBed1B" typeElement="infoPatient" visibility="hidden">
       <text typeElement="name" x="1900" y="1570" font-size="60">
        John Coffey
       </text>
       <text typeElement="illness" x="1910" y="1670" font-size="60">
        Bone Cancer
       </text>
       <text typeElement="state" x="1880" y="1770" font-size="60">
        Recovery
       </text>
       <text typeElement="treatment" x="2010" y="1870" font-size="60">
        Chemotherapy Drugs
       </text>
      </g>
     </g>
    </g>
   </g>
  </g>
  <g id="room2" typeElement="room" > ......... </g>
  .........
 </g>
</svg>
```

(b)

Figure 3: Tree-based graphical representation (a) of an SVG document (b)

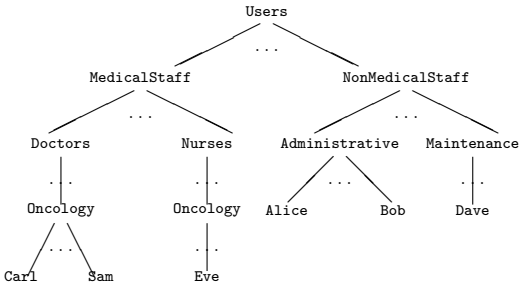



Figure 4: An example of user/group hierarchy

with a new graphic element (`rectOccupiedBed` definition) and information on the occupying patient. The graphical representation of an occupied bed has two procedural attributes, namely `onmouseover='display_information(evt)'` and `onmouseout='hide_information(evt)'`, which show and hide respectively the patient information as the mouse pointer is positioned over the bed or moved out.

Figure 3(b) gives a portion of the SVG document rendered in Figure 2 reporting its XML version, `DOCTYPE`, and part of its `SVG` tree with portions of its definitions, scripts, and body. The definition element (`<defs>`) includes the definition of several *abstract objects* (symbols, in the SVG terminology) like `computer` and `phone`, and different *auxiliary objects* like `rectRoom` and `rectBed`, which will be used as graphical interface for the objects of type 'room' and 'bed', respectively. Element `script` includes the definition of functions `display_information` and `hide_information`, which are triggered by the `onmouseover` or `onmouseout` events to show and hide information on a patient occupying a given bed. The body includes the definition of all the groups composing it (as illustrated in Figure 3(a)). The chunk reported in Figure 3(b) illustrates the definition of the `information` element and of room `room1`.

# 3 The feature protection model

We now present our approach to regulating access to SVG documents. Our approach is based on the use of authorization rules that are themselves expressed with an XML-based language. Each authorization rule specifies the *subject* to which the rule applies, the *object* to which the authorization refers, the *action* to which the rule refers, and the *sign* describing whether the rule states a permission (sign = '+') or a denial (sign = '−') for the access. The complete XML schema of authorization rules supported by SVG-FPL is reported in Appendix A. Here, for the sake of simplicity, we assume action to be the request to render the document (intuitively the `read` operation). This is not limiting as reference to specific actions defined on the document (e.g., `rotate`) can be regulated by allowing (or not allowing) access to the corresponding element. Given this, we can now focus on subjects and objects of our rules.

## 3.1 Authorization subjects

While subjects are not the main focus of our work, where we try to exploit the XML-based representation of SVG documents to provide fine-grained access control, for com-

```
<user_profile id='Carl'>
  <name value='Carl'/>
  <address value='California Ave.'/>
  <citizenship value='US'/>
  <job value='doctor'/>
  <specialization value='oncology'/>
</user_profile>
```

(a)

```
<user_profile id='Dave'>
  <name value='Dave'/>
  <address value='Forest Ave.'/>
  <job value='maintenance worker'/>
  <level value='senior'/>
  <building value='ADM125'/>
  <office value='33'/>
</user_profile>
```

(b)

```
<user_profile id='Sam'>
  <name value='Sam'/>
  <address value='Manchester Rd'/>
  <citizenship value='EU'/>
  <job value='doctor'/>
  <specialization value='oncology'/>
</user_profile>
```

(c)

Figure 5: Examples of XML user profiles

pleteness we also provide treatment for subjects. The specification of subjects in access control rules has often two apparently contrasting requirements [12]. On the one side subject reference must be simple, to allow for efficient access control and for exploiting possible relationships between subjects in resolving conflicts between the authorizations (e.g., most specific relationships between groups and sub-groups). On the other side, one would like to see more expressiveness than the simple reference to user identities and groups, providing support of profile-dependent authorizations whose validity depend on properties associated with users (e.g., age, citizenships, or field-of-specialization) [2, 3]. Our solution nicely encounters both requirements by supporting both user groups and user profiles. As usual, *groups* are sets of users hierarchically organized: groups can be nested and need not be disjoint [9]. Figure 4 reports an example of a user-group hierarchy. In addition, for each user, a profile may be maintained specifying the values of properties such as the name, address, and nationality that our model can exploit to characterize them. The profile is modeled as a semi-structured document and can then be referenced by means of XPath expressions [15]. Figure 5 illustrates three XML documents defining the profiles for three users (all documents will be instances of an XML schema where all the used properties have been defined as optional). A *path expression* is a sequence of element names or predefined functions separated by character / (slash): $l_1/l_2/\ldots/l_n$, and is used to identify the elements and attributes within a document. For instance, XPath expression `user_profile//[./citizenship[@value = 'EU'] AND [./job[@value='doctor']]` returns element `user_profile` of profiles of EU citizens who work as doctors. In particular, such an expression, evaluated on the profiles in Figure 5 would return the profile of user `Sam`.

Therefore, the subject component of our authorization rules includes two parts:

- an *identity*, whose value can be a user or a group identifier;
- a *subject expression* which is an XPath expression on users' profiles.

Note that the purpose of using path expressions in our context is not to retrieve elements and attributes satisfying certain criteria, but to determine whether a given profile (that of the requestor) satisfies the criteria. Intuitively, the authorization rule should apply only if the profile of the requestor satisfies the constraints expressed in the XPath expression. Given this, we ignore the result of a path expression and simply consider whether it is satisfied (meaning its result is not empty) or not.

Authorization subjects are then defined as XML-elements of the form:

```
<subject>
  <id value='user/group-id'/>
  <subj-expr>xpath-expr</subj-expr>
</subject>
```

For instance, `subject` element:

```
<subject>
  <id value='MedicalStaff'/>
  <subj-expr>user_profile//[./citizenship[@value='EU']</subj-expr>
</subject>
```

denotes European users belonging to group `MedicalStaff`.

Again, with reference to the the group hierarchy in Figure 4 and the profiles in Figure 5, the subject expression will evaluate to true for `Sam` and therefore an authorization rule using this subject will be considered applicable to him. By contrast, the authorization rule will not be applicable to `Dave` (who does not belong to `MedicalStaff`), or to `Carl` (who does not satisfy the constraint on citizenship).

## 3.2 Authorization objects

According to the description in Section 2, we identify three kinds of protection objects: *definitions* (`<defs>`), *groups* (`<g>`), and *SVG elements*. SVG elements can be graphical or textual elements, such as `rect` or `circle`, or can be element referencing the definitions (e.g., element `use` in Figure 3(b)). As our authorization model is fine-grained, we allow the association of authorizations with any of such specific elements within an SVG document. As SVG is XML-based, generic XPath expressions on the SVG document can be used to specify the elements to which an authorization applies [6]. For instance, with respect to the document in Figure 3(a), the path expression `defs/rect[position() = 1]` identifies the first `rect` child of the `defs` element (corresponding to the definition of the perimeter of a room). Such an expression can then be used in an authorization rule to grant or deny access to that specific element. Although generic XPath expressions are sufficient to provide fine-grained authorization specification, their only support result limiting from the point of view of the authorization administration. While verbose, these path expressions refer to the syntax and are detached from the semantics of the elements in the SVG document. As a result, the translation of high-level protection requirements into corresponding path expressions on the document is far from being trivial. It is therefore important to provide a higher level support for the specification of authorization objects. Providing higher level support for the definition of authorization objects translates into solving two problems:

- *object identification*: how we identify the portion (element) of the SVG document to which an authorization refers;
- *condition support*: how we specify conditions that the identified element/s have to satisfy.

**Object identification** To provide an expressive authorization language, we exploit the free format of SVG documents by assuming that *semantics* aware tags and good

design techniques can defined and exploited. In particular, as illustrated in Section 2, each SVG element can have an identifier (attribute `id`). Identifiers provide useful as they permit explicit reference to specific elements (e.g., `room1`) based on their name. However, identifiers are not sufficient as listing explicit elements may in some situation result inconvenient (e.g., to protect all rooms of the floor we will have to specify one authorization for each room identifiers). Also, support for distinguishing the shape of an object from its content seems to be needed (e.g., to support cases where a user can see the existence of a room - and then its shape - but cannot see what is inside the room). We address these two requirements as follows. First, in addition to the identifier, we allow each element to have an attribute `typeElement`, that defines the *conceptual type* of the element (e.g., room, bed, telephone, computer). The type element can be exploited to allow reference to all objects of a given type (e.g., all rooms) in a single expression. Second, if the *shape* of an element is conceptually meaningful we assume a good design of the element where the shape (i.e., the *drawing instructions*) appears at the first level and the content appear in a nested element group.[2] Our predefined function `perimeter()` identifies the shape (i.e., the drawing instructions) of an element (referenced via its identifier or type).

Summarizing an object can then be referenced via any of the following:

- a *path expression* resolving in the object;
- its *object identifier* (value of its attribute `id`);
- its *type* (value of its attribute `typeElement`);
- the application of function `perimeter` to any of the above.

To distinguish which of the above means is used in the specification of an authorization object, we use a dot notation prefixing the object with either `id.`, `type.`, or path.. For instance, value '`type.room`' indicates objects with `typeElement` equal to room while value '`id.room1`' denotes the room with `id`'s value `room1`. Analogously, `perimeter(type.room)` identifies the perimeter of all the rooms, and `perimeter(id.room1)` identifies the perimeter of room `room1`.

**Condition support** To provide a way for referencing all elements satisfying specific semantically rich conditions, we allow the specification of *object conditions* that identify a set of objects satisfying specific properties. For instance, we may need to define an access rule stating that "a doctor can see computers only if they are in the same room as (i.e., *together with*) diagnostic machines". In our model, conditions are boolean expressions that can make use of the following *predicates*:

- `inside(`*obj*`)`. It returns the object in the authorization rule if it is inside an element whose identifier, type, or name is *obj*.

[2]Note that we are not forcing documents to obey this structure: if the structure is obeyed it can be exploited for the specification of authorizations.

- `together_with(`*obj*`)`. It returns the object in the authorization rule if it is a child of an element together with an object whose identifier, type, or name is *obj*.
- `number_of(`*obj*,*n*`)`. It returns the object in the authorization rule if there are $n$ instances of the object whose identifier, type, or name is *obj*.

Authorization objects in our model are then defined as:

```
<object>
  <refer value='object-id'/>
  <cond>pred-expr</cond>
</object>
```

where element `refer` provides the object identification and element `cond` specifies additional conditions.

Some examples of object expressions are as follows:

- `<refer value='type.phone'/> <cond>together_with(type.computer)</cond>`

  denotes all 'phones' that are in the same room as (together with) a 'computer'. With respect to our example in Figure 2, it denotes the phones in the 'Pharmacy', 'X-Rays', 'Chemotherapy', 'Kitchen', and 'Reception' rooms.

- `<refer value='perimeter(type.room)'/> <cond>inside(type.oncologyfloor)</cond>`

  denotes all the graphical elements (the `use` elements referencing the `rectRoom` definition) that draw the perimeter of the rooms of the oncology floor.

Summarizing our authorization rules (whose schema is reported in the Appendix) allow the specification that specific users or groups thereof satisfying specific properties can (if sign='+') or cannot (if sign='−') access given objects (referred by means of their, identity, type, or path expressions as well as of conditions that they satisfy). With this expressive way of referring to subjects and objects, it worth noticing how the combined use of positive and negative authorizations result convenient for the specification of different constraints, providing an additive or subtractive way of defining authorized views. In particular, one could start from the empty map and add positive authorizations specifying the objects that may be released, or specifying a global positive authorization and further (more specific) negative authorizations for the objects that cannot be released. Of course, the two approaches can be combined as they best fit the application.

## 3.3 An example

We present some examples of protection requirements and corresponding authorizations to regulate access to the graphic introduced in Section 2. The user groups used and the properties used in the authorizations refer to the user group hierarchy in Figure 4 and the users' profiles in Figure 5.

**Rule 1** Everybody can see the emergency exits

```
<subject><id value='Users'/></subject>
<object><refer ='type.emergencyexit'/></object>
<sign value='+'/>
```

**Rule 2** Everybody can see the content of any room in the public area

```
<subject><id value='Users'/></subject>
<object><refer ='id.PublicArea'/></object>
<sign value='+'/>
```

**Rule 3** Everybody can see the perimeter of any room in the private area

```
<subject><id value='Users'/></subject>
<object>
  <refer ='g[@id='oncologyfloor']//g'/> <cond>inside(id.PrivateArea)</cond>
</object>
<sign value='+'/>
```

**Rule 4** Only members of the NonMedicalStaff whose job is 'maintenance worker' can see the fire emergency and electricity controls

```
<subject>
 <id value='NonMedicalStaff'/><cond>job[@value='maintenance worker']</cond>
</subject>
<object><refer ='type.electricitycontrol'/></object>
<sign value='+'/>

<subject>
 <id value='NonMedicalStaff'/><cond>job[@value='maintenance worker']</cond>
</subject>
<object><refer ='type.fire emergency'/></object>
<sign value='+'/>
```

**Rule 5** Medical staff can see the content of any room in the private area

```
<subject><id value='MedicalStaff'/></subject>
<object><refer ='id.PrivateArea'/></object>
<sign value='+'/>
```

**Rule 6** Doctors with specialty 'oncology' can read patient information; everybody else is explicitly forbidden.

```
<subject>
 <id value='Doctors'/><cond>specialty[@value='oncology']</cond>
</subject>
<object><refer ='type.patientinformation'/></object>
<sign value='+'/>

<subject><id value='Users'/>
</subject>
<object><refer ='type.patientinformation'/></object>
<sign value='−'/>
```

Figure 6 illustrates the views that will be returned to oncology doctors and maintenance workers respectively based on the specified authorizations. Notice in particular,

that the doctors' view does not include fire emergency and electricity control information (as the authorizations in Rule 4 are not applicable to them). The maintenance workers' view, while containing fire emergency and electricity control information is clearly missing many details, such as hospital's machines and patient details, whose access is restricted to physicians. Next Section illustrates the process for obtaining such views.

# 4 Policy enforcement

The enforcement algorithm consists of two main phases: *node labeling* (steps 1-4) and *tree transformation* (steps 5-6). Node labeling takes as input a user request and the DOM tree of the SVG document. Then, it evaluates the authorization rules to be enforced and selectively assigns a plus or minus sign to the nodes of the DOM tree. Subsequently, the tree transformation phase takes the labeled DOM tree as input and transforms it into another valid DOM tree. The result is a view of the SVG document containing only the elements that the requestor is entitled to access. The main steps of the algorithm are listed below:[3]

1. Determine the set *Applicable_authorizations* of authorizations applicable to the requestor. These are all the authorizations for which the requestor is a member (possibly proper) of the subject identity (`<id>`) and for which requestor's profile satisfies the subject expression (`<subj-expr>`).

2. Evaluate the object expressions in every authorization in *Applicable_authorizations* (e.g., resolving them into suitable XPath queries), and label the corresponding SVG elements with the authorization subject identity (`<id>`) and the sign of the authorization.

3. If an element has more than one label eliminate all labels whose subject identity is a super-group of the subject identity in another label (most-specific take precedence).

4. If an element remains with more than one label, if all labels are of the same sign assume that sign for the element. If labels are of different sign assume '−' (denials take precedence on remaining conflicts).

5. Starting from the root, propagate each label on the DOM tree as follows:

   (a) one step upward to the father node, provided the father node is a <g> element while the current node is not.

   (b) downward to descendants. Unlabeled descendants take the label being propagated and propagate it to their children, while unlabeled ones discard the label being propagated and propagate their own to their children (most specific take precedence).

[3]For the sake of simplicity, here we assume that conflicts between subjects (i.e., conflicts of authorization referred to a same element and with disjoint subjects) are solved by applying a most specific take precedence principle (i.e., the authorization with the subject more specific with respect to the user group hierarchy prevails) and the denials take precedence principle for remaining conflicts (i.e., conflicts between authorizations whose subjects are incomparable).

6. Discard from the document all subtrees rooted at a node with a negative label.

7. Discard from the document all subtrees whose nodes are all unlabeled nodes.

8. Render the resulting document.

While steps **1-3** solve the problem of determining labels to be given to nodes according to applicable authorizations, step **4** of the enforcement algorithm deals with completing the labeling phase by propagating initial labels on the SVG document's DOM tree; such step is specific to the SVG data model and deserves some comments. In a well-behaved SVG document (Section 3), `<g>` groups includes a `typeElement` attribute. Also, they contain only two children nodes: a definition of their perimeter and a subgroup for the rest of their content. If the perimeter of a well-behaved group node gets a minus label, our upward propagation ensures that the whole group subtree is pruned; if the minus label affects the subgroup node, the empty perimeter is preserved. On the other hand, if the `<g>` node content is a flat list of SVG elements (a situation which is discouraged but not prevented by current SVG specifications), upward propagation ensures that a minus on each contained node will make the group disappear completely, including the perimeter.[4] Suppose for instance a protection to deal with an ill-behaved SVG object like the following:

```
<svg>
<g typeElement="room">
<desc>Simplified room:  a perimeter rectangle with two rectangles inside</desc>
<rect style="&st0;" x="81.081" y="149.514" width="162.681" height="144.195"/>
<rect style="&st0;" x="109.427" y="175.395" width="29.578" height="23.416"/>
<rect style="&st0;"x="109.427" y="175.395" width="29.578" height="23.416"/>
</g>
</svg>
```

In this case, pruning one of the `<rect>` children of the group `<g>` could result in deleting the room's "walls" while leaving its content, thus conveying an incorrect view of the graphical data. In order to stay on the safe side, our single upward propagation step would take away the whole group. This way *consistency* of the view is preserved (i.e., our algorithm will not produce non-sense views, where for instance a door shows up in an empty area).

Of course, intelligent coordinates-based checking could solve the problem by distinguishing objects lying "inside" and "outside" other objects, but this would require our enforcement engine to provide complex query computing capabilities. While such capabilities may well turn out to be important in the long run, we chose not to include them in the present version of our system. Note also that non-group nodes (satisfying the FPL language predicates, if any) will be simply pruned together with their subtrees; no guarantee is offered in this case that the image semantics will be preserved.

[4] Alternatively, a log event could be generated instead to alert the application enforcing the policy.

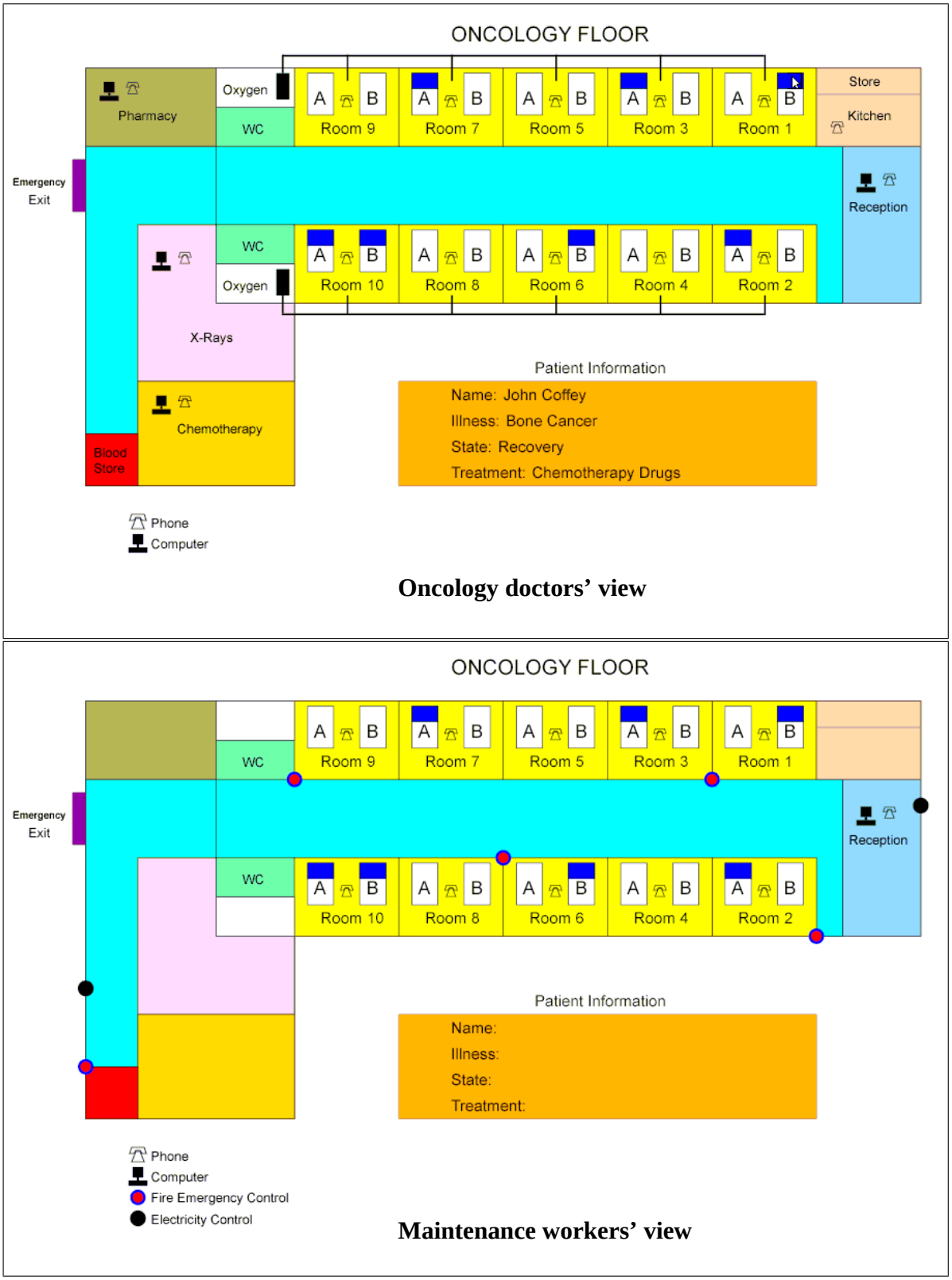



Figure 6: Example of a views on the SVG document in Figure 2

### 4.1 Implementation Guidelines

From the implementation point of view, SVG tree labeling may be performed equally well pragmatically (e.g. writing Java code) or using the XPath query engine of a server-side XSL transformer before releasing the SVG data to the client. For instance, all the `room` groups of our example can be easily selected via a single XPath, namely `root/.../g[@type='room']`. The same holds true for named definitions: labeling all instances of a given SVG definition `def` would cost a single XPath query. It should however be noted that `inside`, `number_of` and `together_with` predicates of FPL limit the scope of labeling and may introduce a performance burden on the enforcement algorithm, as they require linear scanning the XPath result node-set. Their execution goes as follows: for each node in the node set retrieved by the main XPath query identifying potential objects, an auxiliary XPath query is executed to extract all the node's siblings and check whether they satisfy the predicate; if this is the case, the current node gets labeled, otherwise it does not. On the other hand, single unnamed objects must be labeled by following the unique XPath leading to each of them. After labeling has been completed, the transformation phase will be performed on all nodes having a `sign` attribute; again, XSLT or Java code can be used.

## 5 Concluding remarks

We have presented a technique for fine-grained feature protection of XML-based formats. While we developed this technique mainly for controlled dissemination of graphical information representing confidential or sensitive data (e.g. about industrial plants or transportation and utility networks), other interesting potential applications of feature protection techniques are currently under discussion. XML-based standard graphical formats are considered by many the natural successors of current proprietary formats for high volume distribution of graphical content, such as Macromedia Flash. High volume distribution of graphics over the Internet presents a host of problems of its own, mainly related to digital rights management. Interestingly, however, some intellectual property protection problems can be straightforwardly mapped into feature protection ones. For instance, it is customary that distribution graphics does not include information about layers, scenes, and other authoring-specific information, leaving only essential content for fast download and, more importantly, for theft prevention. Experience has shown that regardless of its low-level format, graphics whose semantic information has been stripped out is less easily hacked and modified by unauthorized persons. While there is no way to make graphics intellectual content completely safe from theft, feature protection is a promising tool in order to eliminate the need of keeping "smart" images used in creation and editing (i.e., graphics fully equipped with semantics-aware information) separate from their dumb copied used for distribution. This issue promise to be an interesting direction in which to extend the proposal presented in this paper.

# References


[1] E. Bertino, S. Castano, and E. Ferrari. Securing XML documents with Author-X. *IEEE Internet Computing*, 5(3):21–31, May/June 2001.

[2] P. Bonatti, E. Damiani, S. De Capitani di Vimercati, and P. Samarati. An access control system for data archives. In *16th International Conference on Information Security*, Paris, France, June 2001.

[3] P. Bonatti, E. Damiani, S. De Capitani di Vimercati, and P. Samarati. A component-based architecture for secure data publication. In *17th Annual Computer Security Applications Conference*, New Orleans, Louisiana, December 2001.

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

[5] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Controlling access to xml documents. *IEEE Internet Computing*, 5(6):18–28, November/December 2002.

[6] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security (TISSEC)*, 2002. (to appear).

[7] Eduardo B. Fernández, Ehud Gudes, and Haiyan Song. A model for evaluation and administration of security in object-oriented databases. *TKDE*, 6(2):275–292, 1994.

[8] A. Gabillon and E. Bruno. Regulating access to XML documents. In *Proc. of the 15th Annual IFIP WG 11.3 Working Conference on Database and Application Security*, Niagara on the Lake, Ontario, Canada, July 2001.

[9] S. Jajodia, P. Samarati, M.L. Sapino, and V.S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):18–28, June 2001.

[10] M. Kudo and S. Hada. XML document security based on provisional authorization. In *Proc. of the 7th ACM Conference on Computer and Communication Security (CCS 2000)*, Athens, Greece, November 2000.

[11] S. Osborn, B. Thuraisingham, and P. Samarati. Panel on XML and security. In M. Olivier and D. Spooner, editors, *Database and Application Security XV*. Kluwer, 2002.

[12] P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, LNCS 2171. Springer-Verlag, 2001.

[13] World Wide Web Consortium. *Scalable Vector Graphics (SVG) 1.0 Specification*, September 2001.

[14] World Wide Web Consortium (W3C). *Document Object Model (DOM)*. http://www.w3.org/DOM/.

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

[16] Xml schema. http://www.w3.org/XML/Schema.

[17] J. Ze Wang, M. Bilello, and G. Wiederhold. Textual information detection and elimination system for secure medical image distribution. In *Proc. of the 1997 American Medical Informatics Association (AMIA'97) Annual Fall Symposium (formerly SCAMC)*, Nashville, Tennessee, October 1997. http://www-db.stanford.edu/ wangz/project/meditext/AMIA97/.

[18] J. Ze Wang and G. Wiederhold. System for efficient and secure distribution of medical images on the internet. In *Proc. of the 1998 American Medical Informatics Association (AMIA'98) Annual Fall Symposium*, Orlando, Florida, November 1998. http://www-db.stanford.edu/ wangz/project/meditext/AMIA98/.

# A Authorization rule syntax: SVG-FPL Schema

```
<schema
xmlns='http://www.w3.org/2000/10/XMLSchema'
targetNamespace='http://www.w3.org/namespace/'>
 <element name='rules'>
  <complexType>
   <sequence maxOccurs='unbounded'>
    <element ref='rule'/>
   </sequence>
   <attribute name='about' type='string' use='required'/>
  </complexType>
 </element>
 <element name='rule'>
  <complexType>
   <sequence>
    <element ref='subject'/>
    <element ref='object'/> <element ref='sign'/>
   </sequence>
  <complexType>
   <attribute name='value' type='string' use='required'/>
  </complexType>
 </element>
 <element name='subject'>
  <complexType>
   <sequence>
    <element ref='id' minOccurs='0' maxOccurs='1'/>
    <element ref='subj-expr' minOccurs='0' maxOccurs='1'/>
   </sequence>
  </complexType>
 </element>
 </element>
 <element name='object'>
  <complexType>
   <sequence>
      <element ref='refer' minOccurs='0' maxOccurs='1'/>
    <element ref='cond' minOccurs='0' maxOccurs='1'/>
   </sequence>
  </complexType>
 </element>
 <element name='sign'>
  <complexType>
   <attribute name='value' use='required'>
   <simpleType>
    <restriction base='string'>
     <enumeration value='+'/>
     <enumeration value='-'/>
    </restriction>
   </simpleType>
   </attribute>
  </complexType>
 </element>
 <element name='refer'>
  <complexType>
```

```
    <attribute name='value' type='string' use='required'/>
   </complexType>
  </element>
  <element name='subj-expr' type='string'>
  </element>
</schema>
```