Application Report: An extensible policy editing API for privacy and identity management policies

Giles Hogben,
European Commission,
Joint Research Centre,
Via Enrico Fermi 1,
21020 VA, Ispra, Italy,
+39 0332789187

giles.hogben@jrc.it

ABSTRACT
This paper describes an open source policy editing API, which has been developed for use with privacy policies including P3P1.1 policies, semantic web privacy policies and enterprise privacy policies. The API has been designed to be extensible to a wide range of policy editors for access privacy and identity management. It is also designed to support the use of ontologies to specify validated and updateable human readable translations of policy elements. It provides libraries for editing any kind of policy which is associated to URI resources and which describes behaviour in terms of discrete statements. The paper gives a brief overview of new features of the API which have allowed us to generalize its application.

Categories and Subject Descriptors
D.2.6 [**Software**]: Programming Environments, Graphical Environments

General Terms
Management, Design Security, Human Factors, Languages

Keywords
Policy Authoring, Applications, Semantic Web Groundings

This work was supported by the IST PRIME project; it represents the view of the authors only.

1. INTRODUCTION
This paper is a short application report on a policy editing framework produced by the Joint Research Centre as part of the PRIME project. Many policy editors already exist in the context of P3P 1.0 [1], so we concentrate in this paper on the innovations we have introduced in order to extend the policy editing API from a P3P editor to other types of policy editing such as enterprise access control (privacy layer). We also discuss the introduction of a legal hints mechanism.

2. Editor usage scenarios
The editor has been designed for the following use cases:

2.1 P3P 1.0 Policies
The editor is designed to be able to edit P3P 1.0 policies and to output Policy Reference Files specifying which P3P 1.0 policies apply to which sets of web resources. It is also designed to be able to validate policies, and to give legal hints to policy writers about points of interest in their jurisdiction.

2.2 P3P 1.1 Policies
The editor is designed to integrate the enhancements provided by P3P1.1.[2] These are mainly in the area of the human readable strings corresponding to policy concepts, but also include a new data schema format.

2.3 Semantic web P3P style policies
The editor API is also designed to be able to produce policies using P3P semantics translated into OWL (as described in [3]).

2.4 Enterprise access control (XACML style) policies
This is the most challenging adaptation of the editor. We decided that there are sufficient similarities in the model of P3P and experimental privacy enhanced access control policy languages such as EPAL [4], and [5] to be able to justify an adaptation of the API to support editing of this type of policy.

The specification we are using is the working specification for the Prime [6] project access control module. Although this is not currently available publicly, it is however close to the specification described in the publicly available document [4] in terms of policy editing requirements. The working specification of [6] conforms to the requirements stated in [7].

In general terms, the policy framework comprises Access Control Policies, Data Release Policies and Data Handling Policies. All these operate over RDF data stores and use prolog type semantics encapsulated in XML syntax for creating inferences over access-control rules.

Throughout this document, we refer to this type of policy as "XACML style" (XACML:Oasis standard – vide http://www.oasis-open.org) as this is the closest existing standard (apart from the W3C member submission, EPAL [4]). It is important to note that the API requires access control policies of this type to operate over RDF data with data typing via RDF/OWL ontologies or P3P data schema syntax.

3. Policy editing interface API Components

3.1 Common features
Any API design always plays off simplicity against general applicability. It is clearly not possible to build an API suitable for building any conceivable type of policy. However, we have managed to abstract the features of privacy and IDM policies, including enterprise access control policy languages for privacy in

order to maximize reuse. The following features are common to all types of policy and therefore represent the building blocks of the policy editor API. The API uses the MVC (Model View Controller) paradigm, which divides the management of the user interface and storage objects into Business (Model), Interface (View) and Events (Control). Before reading the following sections, the reader may wish to refer to the end-to-end walkthrough in section 4.

3.2 Resource-policy binding

A common feature of all the above policy types is the need to associate rules or practice statements with groups of resources. This defines which policy should be applied to which resources in the data space. P3P policies, for example, use Policy reference files to associate XML P3P policies with parts of web sites which are resources groups. We found however that this model can also be extended to semantic web and XACML style policies as defined in section 2.4

XACML style policies are of 2 kinds:

a. Access control policies, which associate subject, condition, action rules with *abstract* data types drawn from an ontology describing data types and credentials. A set of policies applies in a given context defined by the administrator. Such policies contain rules of the form:

   For data or credentials of type "prime:e-Healthcard", if the accessing subject is a doctor who is employed in hospital x, allow access with the following obligations….

b. Access control policies which represent user preferences on data collected. These apply rules and obligations to specific data instances.

   Such policies apply rules and obligations to specific data instances. For example

   Delete data item X, after 10 years

The API applies the same model to all of the use case policies. In each case, the editor is required to apply rules or statements to *groups of resources*. In the case of the XACML style policies, the groups of resources are either OWL concepts (defined by a URI – scenario a. above) or RDF triples in a datastore (defined by reification ids, or an RDF query – scenario b. above). We have therefore abstracted the policy-resource association function in the API as follows.

Every editor has 3 sub-windows (see figure 1 and 2) which are managed by a set of extensible classes according to the MVC model.

1. The resource grouping window (top left):

Shows a list of resource groups organized by namespace or site domain. The underlying business object is the same for all types of policy (an XML object stores the resource groups as named patterns according to namespace), but these business objects can then be transformed into customized mapping objects. In other words, the business object abstracts Policy Reference Files for P3P1.0 and allows it to be mapped into other format (e.g. XACML targets).

The user interfaces used for capturing patterns may differ from the default implementations but can customize API implementationsn by extending the PatternInterface class, which captures the specification of the content groups from the user. Each resource group group defines a space of resources which can be either web URI's (P3P and Semantic Web P3P), Ontology concepts (XACML style a.) or RDF triple sets (XACML style b.). In P3P, this corresponds to an area of a web domain or set of domains. In semantic web based access control, this corresponds to a space of resources.

2. The policies window:

Shows the policies available. This is just a list of policy names associated to their logical identifiers (file system paths), which can be dragged onto resource groups and can be double clicked for editing the content of the policy, using a class conforming to the policyeditor interface. This interface is completely independent of the format and content of the policy and it is therefore not foreseen that this would need to be extended.

3. The mappings window:

By dragging a policy onto a resource group, the user can associate policies to resources. This association is then automatically displayed in a third window, the mappings window. The storage format for mappings is abstracted from the particular format it will eventually be output in. For example in the case of P3P, this abstraction will be mapped to a Policy Reference File. In the case of semantic web based access control, it may for example be mapped to a target statement within a policy. The API implements this abstraction using the "publish" method of the mappings tree, which currently only implements the transformation to a P3P Policy Reference File, but can be overridden to provide other transformations for example using XSLT to provide target statements within XACML style policies.

Figure 1. P3P scenario

Figure 2. Semantic Web Scenario

3.3 Statement handler

Upon opening a policy for editing, the user is presented with a list of statements. Statements are derived directly from the XML policy document in memory defining the policy being edited. So in terms of the MVC architecture model, the XML document is the model (Business object) and there is no further abstraction.

The API provides a statement management package, which includes a class which abstracts the visualization of statements. The class StatementType defines how human readable strings are extracted from the XML document by means of a query string. It also defines not only the content of the strings, but also how they will be organized for display to the user.

In order to achieve this, the StatementType class defines the list of attributes into which the statement is broken down. These may, but are not required to correspond to XML attribute or tag names. For example P3P StatementType definitions define how to extract CONSEQUENCE, DATA, PURPOSE, RECIPIENT and RETENTION attributes of the statement by means of XPATH queries. This is done as follows:

Each Attribute object specifies XML or RDF queries and/or procedural code which define its relationship to the user interface. This allows the editor builder to define new types of statements and attributes and their display to the user simply by defining their attributes and queries which extract the display text.

Each attribute in a StatementViewer's AttributeList Array has a getHRQueryString method, which returns the results an RDF or XML query over the policy document (and may transform this using Java code for display). This method returns the text to display to the user to summarize the value of that attribute.

For example for P3P statements, getHRQueryString() for each attribute returns a conversion to string of the node names returned by the XPath queries:

".//*[local-name()='CONSEQUENCE']/*",

".//*[local-name()='DATA']/*"

".//*[local-name()='PURPOSE']/*"

Etc…

Separate XML and RDF flavours of these classes have been defined in order that the query language is flexible.

Once the StatementViewer object for the policy editor is defined, the API automatically creates a table displaying all non-hidden attributes. It is assumed that statements are logically independent objects i.e. that no inter-statement data (e.g. OR and AND) needs to be displayed. These kind of booleans may be included in a language but statements should be defined on a level whereby the booleans are contained within each statement but do not connect statements. StatementTypes can also be created dynamically if the number of attributes is variable.

Attributes can be assigned visible or non visible status. For example a P3P editor would not want to display the consequence attribute of a statement in the statement summary table, so this would be assigned hidden status.

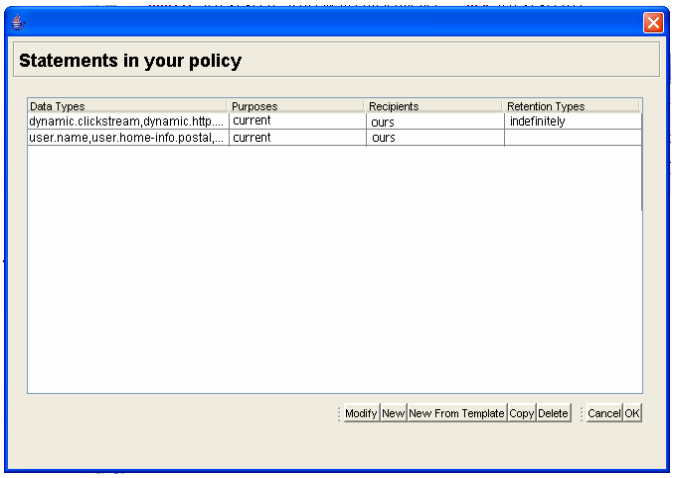


Figure 3. StatementViewer

3.4 Statement wizard

StatementType attribute arrays (see 3.3) also define the stages of the Statement wizard. The statement wizard proceeds through a series of windows on a per attribute basis. The attribute array of the StatementType therefore defines the stages of the statement wizard. Statement wizards can also be defined using a swing card layout to increase efficeincy.

Each attribute has a getViewer method, which uses classes extending the abstract class AttributeEditorWindow to specify the editing window to be displayed for that attribute.

The API provides 3 implementations of AttributeEditorWndow.

1. Typically a statement attribute editing window is a flat set of possible values displayed as a set of strings with checkboxes next to them. This uses a ConstrainedValueWindow The human readable strings for this type of attribute editing window may be defined according to an XML document or OWL ontology (See section 4.5)

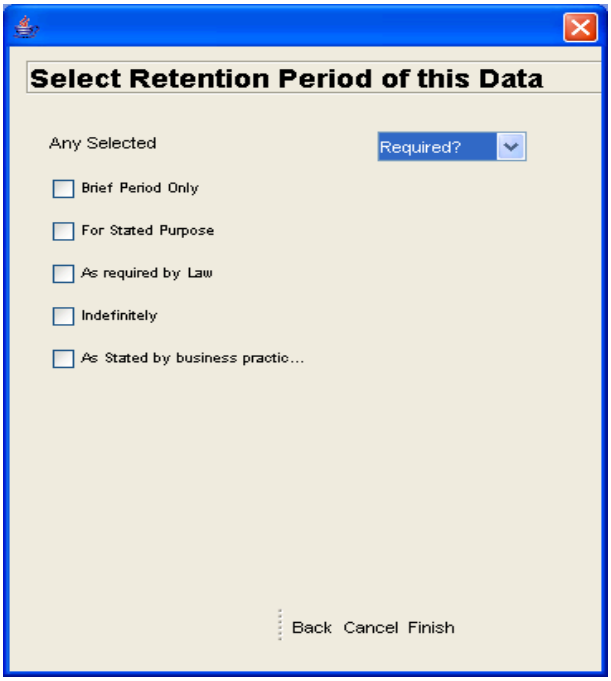


Figure 4. ConstrainedValueWindow

2. A datatype editing window (See 3.5)

3. A plaintext editing window (e.g. for P3P consequence). The type of window required is specified. Special editing windows can be created to replace the default implementations.

The above default implementations can be used to define attribute viewers.

3.5 Data typing schemes

Privacy and access control policies typically have to present the user with an ontology hierarchy of increasingly detailed data types to select from (an XML document is also understood here as an informal ontology). The editor API abstracts this process so that different data schemas can be used within the same view window as long as they have a structure representable by a JTree.

The data type editing window displays the data type tree on the left hand side and a list of selected types on the right hand side. The user simply moves types from the tree into the list on the right hand side. The elements in the list of types selected combine to make a custom type. The list element objects store the tree path as well as the leaf node selected so that they can be edited later.

The user can dynamically select different source files for the tree representation.

The API provides the abstract DataSchemaTreeViewer class which has the abstract LoadTree() method. This defines how the data typing schema is mapped onto the JTree. We will provide 3 implementations of this method - for P3P 1.0 [1], 1.1 [2] and OWL [8] versions of the P3P data schema. Once this mapping has been made, the chosen types are be inserted directly into the policy without further reference to the schema. New schemas of the given type can be loaded dynamically.

Future work would include an editor for creating custom data schemas. Figure 6 shows the datatype editing window with the P3P base data schema loaded in the left pane and the types selected in the right pane. Above the schema tree is a button for loading a new schema tree.

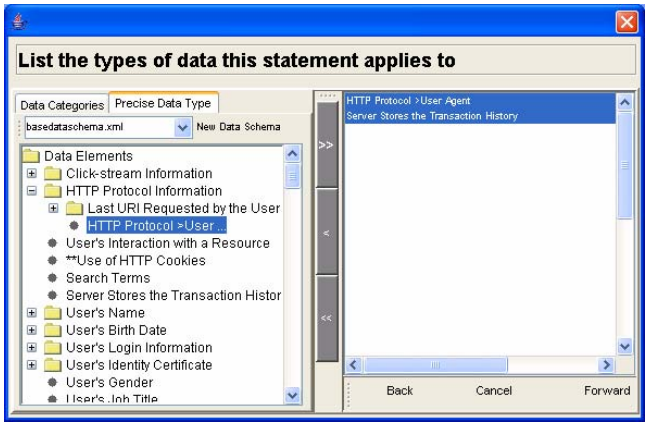


Figure 6. Datatype editing window

3.6 Linking of option handling and human readable strings to ontologies.

Because of the importance of displaying human readable translations of attributes in a consistent way [see 9], label strings for ConstrainedWindow [See Section 3.4.1] implementations are taken from an XML specification document which may be either an RDF ontology, or an XML document.

In the case of P3P, the latter is just a translation into XML of the Human readable translations in the draft P3P 1.1 specification [9]. The checkboxes and their labels are created dynamically from this document by the getAlternatives method of the Attribute object.

The exact method of associating human readable strings to checkboxes depends on whether an XML specification document is used, or an RDF ontology.

1. For an XML specification: each Alternative in the Attribute's alternative array is an object which can return an XML fragment from its getXML() method. This is the XML fragment to be inserted into the statement being edited if the choice is selected. It may also be derived from a query over an XML schema in order to minimize programming work in case of changes to the specification schema.

   Each alternative also has a getHumanReadable() method which performs a query over a human readable equivalences document in well-defined format, in order to return the human readable string for that alternative. In our implementation, the equivalences are stored as fragments of the document with sibling CDATA text nodes containing the human readable equivalent.

   For example the PURPOSE translations are stored as follows:

   <equivalence>

   <node><ours/></node>

   <hrstring>Only parties related to this site</hrstring>

   <equivalence>

   The user's choices are then automatically saved to the Statement's base document when the user click's OK by inserting the node associated to the alternative into the statement.

2. For an OWL ontology (parsed by the Jena [10] API): The procedure for extracting and displaying the alternatives is

the same as 1. except that the query extracting the equivalence will be an RDF query rather than an XPATH query.

3.7 Use of XSLT transforms for policy views.

The base window of the policy editor shows a set of views of the policy being edited. These views are produced by XSLT transforms which define views such as for example Human readable, statement summary and To Do (a list of incomplete parts of a policy). Another important view is the legal hints view (See next section).

The policy views can also be produced using prolog style rules running over RDF (e.g. using Jena rules). This then outputs a set of statements inferred from the policy, with a transformation to natural language. (See also 3.8).

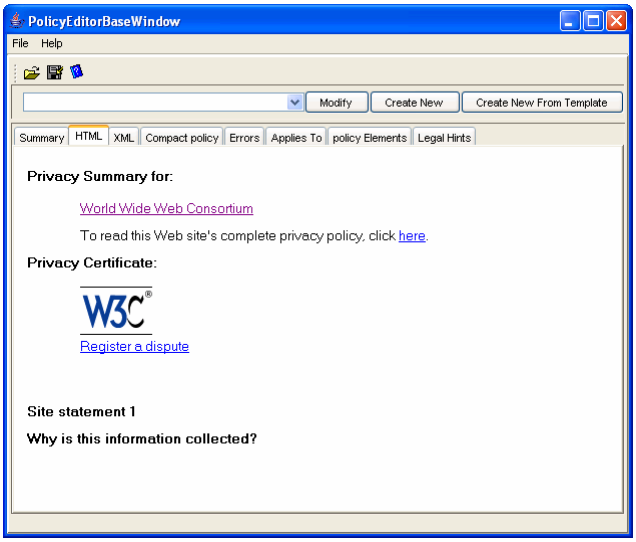


Figure 7. Policy Transformation View (Mirrors view in MS Internet Explorer Privacy Report)

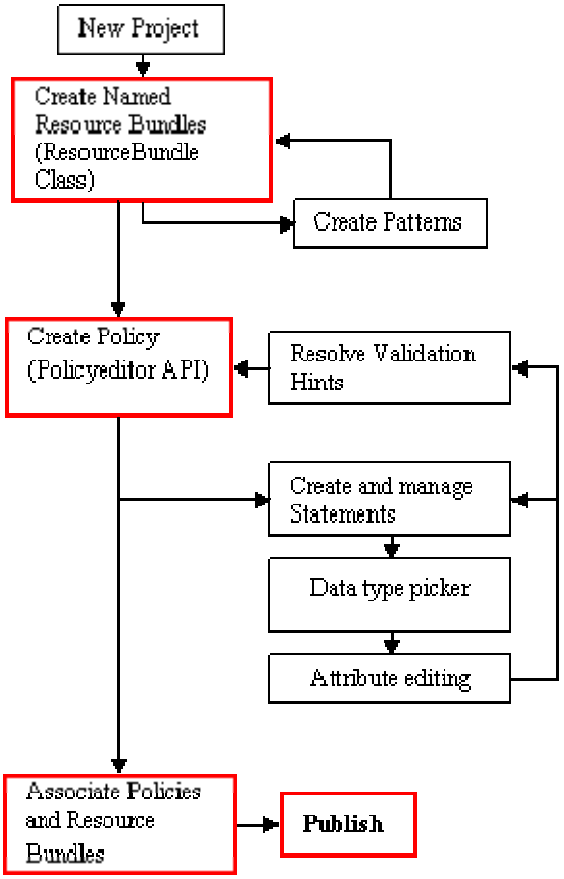
3.8 Legal hints mechanism

In Europe particularly, regulatory bodies have been concerned about the possibility of privacy languages which enable policy authors to write policies which specify data processing practices which are illegal in the author's jurisdiction.

One important policy view provided by the API is the legal hints view. This is based on XSLT transformation rules or RDF based rules which provide users with comments on the policy they have created based on legal knowledge encoded in the rules. It is envisaged that XSLT transforms or other inference rules will be imported based on jurisdiction.

For example if a user creates a P3P policy which says that they will use email data to contact the user without an opt-out (which would be illegal in Europe according to [11]), the legal hints can inform the user that this is an illegal practice in Europe. It is possible in future versions that these rules could also offer a set of corrections to the user.

4. Process walkthrough



5. Conclusion

The API described above provides a useful tool for policy authoring in many scenarios in the field of policies for the web. It provides an extensible framework for policy-resource association, statement management and statement composition. It also provides a framework for providing legal hints and different policy views.

6. REFERENCES

[1] Platform for Privacy Preferences Specification, Cranor et al. ,Platform for Privacy Preferences, W3C Recommendation, http://www.w3.org/tr/p3p

[2] Cranor, Dobbs, Egelman, Hogben et al., The Platform for Privacy Preferences 1.1 (P3P1.1) Specification W3C Working Draft 4 January 2005 http://www.w3.org/TR/2005/WD-P3P11-20050104/

[3] Hogben, G. *P3P Using the Semantic Web (OWL Ontology, RDF Policy and RDQL Rules)*, W3C Working Group Note 3 September 2000, http://www.w3.org/P3P/2004/040920_p3p-sw.html

[4] Powers, C., Schunter, M., Enterprise Privacy Authorization Language (EPAL 1.2), W3C Member Submission 10 November 2003, http://www.w3.org/Submission/EPAL/

[5] Piero A. Bonatti Ernesto Damiani Sabrina De Capitani di Vimercati Pierangela Samarati, A Component-based Architecture for Secure Data Publication http://www.acsac.org/2001/papers/114.pdf

[6] Privacy and Identity Management in Europe, European Research Project, see http://www.prime-project.eu.org

[7] Wilikens, M. et al., PRIME Requirements - Part 3: Application requirements, http://www.prime-project.eu.org/public/prime_products/deliverables/pub_del_D01.1.a.part3_ec_wp03.1_V5_final.pdf

[8] Hogben, G., Describing the P3P base data schema using OWL, Proceedings of PM4W, WWW2005 workshop.

[9] See Section on User Agent Guidelines, P3P 1.1 Draft Specification, http://www.w3.org/TR/2005/WD-P3P11-20050104/#ua

[10] Jena semantic web API, HP Labs, see http://jena.sourceforge.net

[11] EU Directive 2002/58/EC on Privacy and Electronic Communications