Downstream Usage Control

Laurent Bussard[1], Gregory Neven[2], and Franz-Stefan Preiss[2]

[1] European Microsoft Innovation Center, Aachen, Germany
lbussard@microsoft.com
[2] IBM Research – Zürich, Switzerland
{nev,frp}@zurich.ibm.com

Abstract. Whereas access control describes the conditions that have to be fulfilled *before* data is released, usage control describes how the data has to be treated *after* it is released. Usage control can be applied to digital rights management, where the data are usually copyright-protected media, as well as in privacy, in which case the data are privacy-sensitive personal information. An important aspect of usage control for privacy, especially in light of the current trend towards composed web services (so-called *mash-ups*), is *downstream usage*, i.e., with whom and under which usage control restrictions data can be shared. In this work, we present a two-sided XML-based policy language: on the one hand, it allows users to express in their preferences in a fine-grained way the exact paths that their data is allowed to follow, and the usage restrictions that apply at each hop in the path. On the other hand, it allows data consumers to express in their policies how they intend to treat the data, with whom they intend to share it, and how the downstream consumers intend to treat the data. Downstream usage paths can be specified up to any desired depth, with the option to let the final usage control restrictions apply recursively to any further sharing of the data. Moreover, we describe a matching algorithm by which users can efficiently test whether all hops in a proposed policy match their own preferences, allowing them to decide autonomously and in an automated way whether it is safe to release the requested personal information. When a match occurs, a sticky policy is generated describing the precise rights and obligations that the consumers have to adhere to.

1 Introduction

Many web services today are so-called service *mash-ups*. A mash-up is a service that acts as a front-end for a composition of multiple subservices that are offered by different companies. For example, a travel booking mash-up may offer an integrated interface to book flights, hotels, and rental cars. In the background, however, it invokes the web service APIs of different specialized airline, hotel, and car rental subsidiaries to collect offers. The best offers are presented to the user, who selects an offer, enters her booking and payment information, and confirms to let the mash-up make all the bookings for her through the subsidiaries' APIs.

Service mash-ups are important for leveraging online service APIs to create new functionality, but pose significant privacy risks for their users. The users

cannot keep track of who stores what information about them, and often they do not even know the subsidiaries their data is shared with.

To overcome this problem, service providers publish their privacy policies to inform the users about how the gathered data is used. Human-readable privacy policies, e.g., have the disadvantage of mostly being written in complex language influenced by the legal profession and thus being ignored by users. Even if clear privacy policies are presented, they often remain vague about the sharing of information with third parties. For example, the privacy policy of Expedia.com[1], a popular online travel service, mentions the following with regard to sharing personal information with suppliers:

> *We do not place limitations on our suppliers' use or disclosure of your other personal information* [i.e., other than the email address]. *Therefore, we encourage you to review the privacy policies of any travel supplier whose products you purchase through this site.*

Switching to machine-interpretable privacy policy languages such as EPAL [1] and P3P [2] is a promising approach, in particular when used in combination with a privacy preference language such as APPEL [3]. In the latter language, users can express how they expect their data to be treated, so that an automated or semi-automated matching procedure can decide on the acceptability of a proposed P3P policy.

Unfortunately, EPAL and P3P are both rather constrained in expressivity regarding sharing personal information with third parties, or *downstream usage* as we call it here. EPAL leaves the definition of specific actions and obligations up to enterprise-defined vocabularies, and is hence silent about downstream usage. Support in P3P is limited to specifying which of six classes of third-party recipients the information will be shared with; it is up to the server to classify his recipients into one or more of the classes. SecPAL for Privacy (S4P) [4] proposes a logic-based language to specify human readable policies and preferences. S4P does not focus on downstream usage control and does not specify downstream in terms of access control.

In this work, we investigate how to structure a policy language suitable for downstream usage control. The difficulty here is that downstream usage control involves a mixture of what is typically considered access control (who is allowed to receive the data, e.g., by roles or owned credentials) and usage control (how is the recipient supposed to treat the data, e.g., usage purposes or retention period). We consider the most general setting here, where the user states in her privacy preferences, for each hop in a chain of downstream recipients, how they have to treat her data and to whom they can further forward it. At the same time, each recipient specifies in his privacy policy how he intends to treat the data and to whom he intends to forward it. We propose XML-based languages to express both the user's preferences and the servers' policies, and describe an automated matching algorithm to determine whether the proposed policies are allowed by the specified preferences. The user and the servers can specify

[1] cf. `http://www.expedia.com/daily/service/privacy.asp`

downstream usage restrictions up to any number of hops (not necessarily the same number). Optionally, they can either specify that the last restrictions in the chain are valid for all subsequent hops, or that after that hop no further downstream usage is allowed. Moreover, for situations where the server does not know at the time of data collection to whom he may forward the data, we propose an alternative matching algorithm at which a server declares his willingness to impose any restrictions on further downstream usage that the user may specify.

This work relies on the trust model of P3P and EPAL: each service is willing to enforce its privacy policy. Authors assume that service providers are appropriately enforcing their policies to protect their reputation and/or that services are regularly audited and certified by trusted third parties. Proving the correctness of policy enforcement, for instance using a trusted stack (certified TPM, trusted OS, and trusted application), is out of the scope of this paper.

2 Related Work and Contributions

Our work is closely related to rights expression languages (RELs), privacy policy languages, and usage control. We give a brief overview of each of these lines of work and how they relate to the language we propose.

2.1 Rights Expression Languages

From a *protocol* point of view, there is a clear difference between privacy policies on the one hand, and digital right management (DRM) and enterprize right management (ERM) on the other hand. Indeed, in privacy, it is usually the consumer (data controller) who imposes the policy, while in right management, the author (or publisher) imposes the policy (license) to the consumer. The main purpose of *matching* is to help the end-user deciding whether he accepts the service and the policy. Such matching is interesting to make decisions related to privacy or to rights management and, in both case, is generally done by the user.

From a *trust model* point of view, privacy policies and enterprize right management are similar because they assume "honest consumers", data controller and consumer respectively, which are willing to enforce the policy (accept audit, use required client application, etc.). In digital right management, the threat model is different since consumers may mount attacks in order to violate the policy. As a result, trusted hardware and/or software are required in DRM.

There are three key differences between our approach and state of the art right expression languages used in ERM and DRM, e.g., MPEG-21 REL [5], XrML [6], and ODRL [7]:

- RELs focus on rights (e.g. print, play) and add some conditions and constrains (temporal, fees, device, etc.) but obligations remain underspecified.
- Matching is generally not specified even if automating the process of accepting an offer, i.e. a proposed license, would make sense.

– Downstream data sharing (transfer, i.e. sell, give, lease) is also defined as a right. However, it is not as expressive as our model and does not result in a matching algorithm.

The most important difference between right management and our work is the fact that in the former the data provider pushes sticky policies (i.e., licenses) onto the consumer without matching. In a privacy setting, the provider (i.e., user) usually does not have this power. Rather, it is the consumer who imposes a policy onto the provider.

Another major difference is that the domain-specific vocabulary is quite different for RELs and for privacy policies, even though overlaps exist, e.g., the obligation to delete data within a certain time makes sense in both domains. Apart from the vocabulary though, the same overall language structure should be usable for both.

2.2 Privacy Policy Languages

We already briefly discussed how the privacy policy languages such as EPAL [1], P3P [2], and APPEL [3] fall short in terms of expressing restrictions on the downstream usage. EPAL is mainly intended for writing enterprize-internal privacy policies to govern data handling practices. P3P, on the other hand, is mainly intended for websites to communicate their privacy practices to the outside world. APPEL is a preference language for P3P, i.e., enabling users to specify their privacy preferences and automatically match those against proposed P3P policies.

SecPAL for Privacy (S4P) [4] is a logic-based language to specify privacy policies and preferences. S4P specifies preferences as `may` assertions (i.e. authorizations) and `will` queries (i.e. obligation request). S4P specifies policies as `will` assertions (i.e. commitment on obligations) and `may` queries (i.e. authorization request). In S4P, matching is about evaluating queries with a set of assertions while, in the work presented in this document, matching is done by comparing statements.

Ardagna et al. [8] describe a data handling policy language that allows for specifying data recipients, usage purposes and obligations. In the proposed model, service providers present policy templates to users that the users may customize on the base of their own data handling policies. The outcome of a successful customization is a policy traveling with the data. The customization process is described as potentially automated, it remains unclear though how this automation can be achieved. In particular, because neither an obligation vocabulary nor matching semantics for purposes and obligations are defined. The authors do not distinguish between rights and obligations. Their *purpose* can be seen as dedicated right, however, there is no explicit right to share data downstream. The data recipient concept allows for specifying global rules on who may receive the data, and could therefore be seen as abstract downstream right. However, transitive downstream data disclosures are not explicitly discussed and specific paths along which data may be shared cannot be expressed.

2.3 Usage Control

Usage control [9, 10] is a generalization of access control that is also concerned with how data is treated *after* it has been given away. In contrast, pure access control only addresses how data is protected *before* it is released.

Usage control distinguishes between two kinds of requirements: *provisions* that state access control requirements and have to be fulfilled before access is granted, and *obligations* that state usage control requirements and are concerned with the future usage of data [10]. In obligations one further differentiates between *rights* (also called permissions) and *duties*, and specify constraints mainly related to time, cardinality, purpose, and events. What we call rights and obligations in this work can be seen as the equivalent of rights and duties in the usage control literature.

Usage control takes place in a distributed setting where a process acting in the role of a *data provider* sends sensitive data to a process acting in the role of a *data consumer* based on provisions and obligations. Access and usage control requirements are stated in *usage control policies*. For expressing such policies, several specification languages have been developed [8, 11, 12], however, the policies of Leumann [12] come closest to our approach. They are expressed by a number of rules whereby a *rule* specifies its applicability, provisional actions and obligations. They additionally contain *optional actions* and *contracts*, which reflect the obligations, and are expressed in XACML.

The concept of policy evolution for distributed usage control as proposed by Pretschner et al. [13] bears similarities to our concept of downstream usage control. Here, The roles of data provider and data consumer change dynamically with each transfer: a consumer becomes provider if data is forwarded that was received before. The Obligation Specification Language (OSL) proposed in [13] allows the data provider to specify which consumers (indicated by their roles) have to adhere to which rights and duties when receiving the data. The language is logic-based, so that a sequence of events can automatically be checked for compliance with the specified policy.

The language we propose differs from OSL in two important aspects. First, we envision a double-sided setting where both the data consumer and the data provider have policies (resp., preferences) describing how they will treat the data (resp., want the data to be treated). These policies and preferences are then automatically matched to yield a sticky policy that acts as the agreed-upon contract. In the OSL, it is the data provider who unilaterally describes the sticky policy that has to be adhered to, which we think is especially unrealistic in privacy, where the data provider is a private user. Second, our language is more expressive than OSL in that it can describe (and automatically match) the full path that the data is allowed to follow, including who is allowed to share the data with whom, and how many hops the data is allowed to take. In contrast, in OSL one can specify which role of consumers have to adhere to which usage control policy. For example, in our language a patient could impose one usage control policy when a health insurance company obtains her medical record through the

hospital, and another policy when it obtains from the patient directly. In OSL both cases would have to be treated the same.

3 Description of Solution

In this chapter, we present XML-based languages for the providers' preferences and the consumers' policies in which they can express their precise downstream usage control restrictions. We illustrate our languages with the following example scenario.

3.1 Example Scenario

Alice is a privacy-aware user who regularly shops online, but who is concerned about what happens to the data that she provides about herself. For example, she's willing to provide her postal address to online shops so that the goods can be delivered, but she realizes that most shops rely on external shipping companies. She wants to impose a detailed set of usage control preferences though, where the restrictions on the shipping company depend on who the front-end service is from which it obtained the address. Namely, when obtained through a book shop, she is fine with the shipping company using her address for statistics, but when obtained through any other store, which may include liquor or lingerie stores, she is not. More precisely, she wants to enforce the following preferences:

- Book shops can collect her address for the purpose of statistics, contact, and account administration. They must delete it after two years and are allowed to forward to shipping companies who can use it for shipping and statistics, have to delete it after two weeks, and can further forward it to shipping companies under the same restrictions.
- Any shop can collect her address for the purpose of account administration provided they delete it within one year. They can further forward it to shipping companies who can use it for shipping only, have to delete it after two weeks, and are not allowed to share the data with anyone.

Alice regularly buys books at the online book shop bookshop.com because its privacy policy matches her preferences. Namely, bookshop.com states that it will

- use her address for statistics and account administration, that it will delete the address after one year, and that it will forward the address only to shipping.com.

The policy of shipping.com states that

- the address will be used for shipping and statistical purposes, and will be deleted after one week.

When buying a book at bookshop.com, Alice can safely give her address away since bookshop.com's and shipping.com's privacy policies match her own preferences. However, when buying a bottle of wine at liquor.com, who also use shipping.com for shipping, the transaction will be refused, as Alice's preferences in this case do not allow shipping.com to use her address for statistical analysis.

3.2 Language Model

The abstract scenario we consider is one where two parties, typically a user and a server, engage in an interaction where one of the parties, typically the server, requests some personally identifiable information (PII) from the other party. We will from now on call the party that provides the data the *data provider* and the party that requests the data the *data consumer*. Moreover, we consider a scenario where at a later point in time, the data consumer may want to forward the PII to a third party, called the *downstream data consumer*.

Both the data provider and the data consumer have their own policies expressing the required and proposed treatment of the PII, respectively. These policies contain access control and usage control policies. A piece of PII is only sent to a data consumer after (1) the access control requirements have been met, and (2) a suitable usage control policy has been agreed upon.

We distinguish three kinds of policies:

Preferences: In his *preferences* the data provider describes, for specific pieces of PII, which access control requirements a data consumer has to satisfy in order to obtain the PII, as well as the usage control requirements according to which the PII has to be treated after transmission. These requirements may include *downstream usage requirements*, meaning the requirements that a downstream data consumer has to fulfill in order to obtain the PII from the (primary) data consumer.

Policy: The *policy* is the data consumer's counterpart of the data controller's preferences. In a policy the data consumer contains, for specific pieces of PII to be obtained, his certified properties (roles, certificates, etc.) that can be used to fulfill access control requirements, and a usage control policy describing how he intends to use the PII.

Sticky policy: The *sticky policy* describes the mutual agreement concerning the usage of a transmitted piece of PII. This agreement is the result of a matching process between a data providers's preferences and a data consumer's policy. Technically a sticky policy is quite similar to preferences as described above, but it describes a mutual agreement between the data provider and the data consumer that cannot be changed. After receiving the PII, the data consumer is responsible for storing and enforcing the sticky policy.

To illustrate our ideas we employ a simple XML-based language to express preferences, policies, and sticky policies. In the following we first introduce our language and then focus on how to express downstream usage requirements. Note that we provide the full language schemas in Appendix B.

Preferences model Figure 1 shows Alice's preferences for book shops expressed in our policy language. Figure 2 gives a graphical representation of the language schema for the preferences ; the full schema is given in Appendix B.The `Preferences` root element contains multiple `Preference` elements, each describing to which PII it applies and what the respective access and usage controls are. An attribute `sticky` indicates whether these preferences are in fact a sticky policy for the PII (cf. 3.2), acting as an explicit reminder that they cannot be changed. Alternatively, one could keep all sticky policies separately in a read-only policy store.

A `Preference` can refer to the applicable PII by their data type, meaning that the `Preference` applies to all PII of this type, or by a unique identifier pointing to a single instance of PII. We assume that a typing mechanism and unique naming scheme for PII are in place. A complete language would probably offer more powerful mechanisms to specify applicability, allowing for example attribute expressions or temporal constraints.

Our language is strictly limited to positive statements, in the sense that it explicitly lists the permitted information exchanges, and assumes that all other exchanges are forbidden. Apart from this being a safe privacy-conservative choice, it also simplifies the matching procedure. However, it means that one cannot express conditions of the form "do not forward to X" or "do not use for purpose Y".

If multiple `Preference` elements apply to a single piece of PII, then satisfying the conditions in either of them results in a match. In other words, `Preference`s are combined according to "or" semantics. This makes it possible to define more permissive exceptions to general preferences.

Within a `Preference`, access and usage control requirements occur in a pair enclosed in an `ACUC` element. A pair $(\mathrm{AC}, \mathrm{UC})$ means that any data consumer satisfying access requirements AC can obtain the PII when adhering to usage requirements UC. To allow multiple AC/UC combinations for a single piece of PII, one can use multiple `Preference` elements with the same `Applicability`.

An optional attribute `id` assigns a unique identifier to an `ACUC` pair. This identifier can be referred to from another `ACUC` element via a `reference` attribute. The referring element is then interpreted as if it was substituted with the referred element.

Access control requirements are expressed in terms of `Rule`s where each rule specifies a *property* that a data consumer must have in order to be granted access. Properties are stated in terms of attributes as certified by some certification authority (CA). Empty access control requirements mean that anybody who commits to fulfilling the usage control requirements is granted access. The simple access control language that we use here could in a real system, e.g., be substituted with a complete role-based [14], attribute-based [15,16], or logic-based [17] access control language.

Usage control requirements are expressed by distinguishing between `Rights` and `Obligations`. A right states an action that the data consumer is allowed to perform on the data, but doesn't have to perform to comply with the policy.

```
<Preferences>
 <Preference>
  <Applicability>
   <DataType> Address </DataType>
  </Applicability>
  <ACUC>
   <AccessControl>
    <Rule>CertifiedAsBy{bookshop, CAx}</Rule>
   </AccessControl>
   <UsageControl>
    <Rights>
     <UseDownstream allowLazy="false">
      <ACUC id="ACUCshipping@alice">
       <AccessControl>
        <Rule>CertifiedAsBy{shipping, CAy}</Rule>
       </AccessControl>
       <UsageControl>
        <Rights>
         <UseDownstream allowLazy="false">
          <ACUC reference="ACUCshipping@alice"/>
         </UseDownstream>
         <UseForPurpose>statistics</UseForPurpose>
         <UseForPurpose>shipping</UseForPurpose>
        </Rights>
        <Obligations>
         <DeleteWithin>P14D</DeleteWithin>
        </Obligations>
       </UsageControl>
      </ACUC>
     </UseDownstream>
     <UseForPurpose>statistics</UseForPurpose>
     <UseForPurpose>contact</UseForPurpose>
     <UseForPurpose>accountadmin</UseForPurpose>
    </Rights>
    <Obligations>
     <DeleteWithin>P2Y</DeleteWithin>
    </Obligations>
   </UsageControl>
  </ACUC>
 </Preference>
 ...
</Preferences>
```

Fig. 1. Excerpt from Alice's preferences.

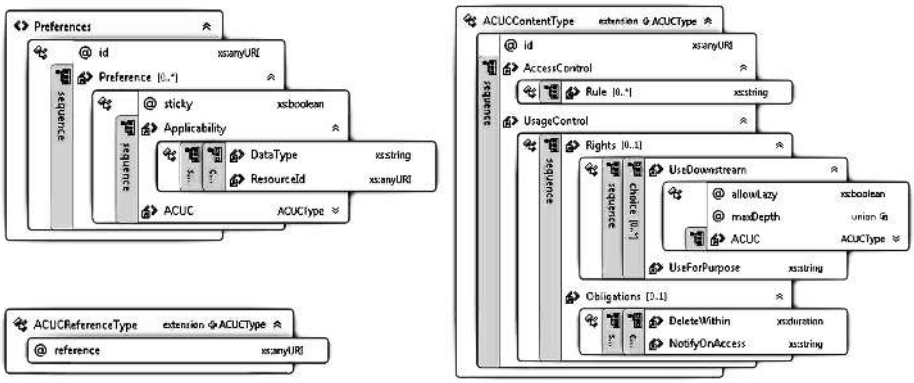


Fig. 2. Preferences and sticky policy language model

An obligation states an action that a data consumer is obliged to perform. We model two types of rights and two types of obligations here:

- `UseDownstream`: The right to forward the PII under given conditions to further data consumers. This is a crucial element in our policy language; we come back to its exact structure and meaning later.
- `UseForPurpose`: The right to use the PII for a specific purpose.
- `DeleteWithin`: The obligation to delete the PII within a given amount of time.
- `NotifyOnAccess`: The obligation to notify the user when the PII is accessed.

The complete language supports more rights and obligations, however, those four suffice to illustrate the ideas of this work. Empty usage control requirements mean that the data consumer is not allowed to store the PII at all. Multiple rights and obligations within a `UsageControl` element are combined by "and" semantics, meaning that the data consumer obtains all the specified rights and has to adhere to all of the specified obligations.

Policy model A data consumer's policy states which usage control requirements he is willing to adhere to when requesting a specific resource from a data provider. In addition the policy states the properties the data consumer is willing to disclose for fulfilling the data provider's access control rules for that resource.

Figure 5 shows the language schema of the server policy, which is obviously similar to that of the preferences. (The full schema is given in Appendix B.) The main difference is in the `AccessControl` element, which instead of access requirement rules now contains the properties that the data consumer can demonstrate to the data provider. To avoid having to repeat the same properties in multiple `Policy` elements, yet preserve the flexibility to use different properties for different types of PII, the server one can use the `id` attribute to assign a unique

identifier to an `AccessControl` element, so that it can be referred to from another `AccessControl` element using the `reference` attribute. In the following we provide example policies for the book shop and the shipping company.

The shop's policy stated in Figure 3 expresses that for collecting addresses, it is willing to authenticate as a shop or a book shop certified by CAx, and as bookshop.com certified by CAz. The address will be deleted after one year and used for statistics and account administration. Further, the shop wants to be able to forward it under the policy of the shipping company (that is specified below).

```
<Policies id="Policies@Shop">
 <Policy>
  <Applicability>
   <DataType> Address </DataType> </Applicability>
  <ACUC id="ACUCaddress@Shop">
   <AccessControl>
    <Property>CertifiedAsBy{bookshop, CAx}</Property>
    <Property>CertifiedAsBy{shop, CAx}</Property>
    <Property>CertifiedAsBy{bookshop.com, CAz}</Property>
   </AccessControl>
   <UsageControl>
    <Rights>
     <UseDownstream allowLazy="false">
      <ACUC reference="ACUCaddress@Shipping"/>
     </UseDownstream>
     <UseForPurpose>statistics</UseForPurpose>
     <UseForPurpose>accountadmin</UseForPurpose>
    </Rights>
    <Obligations>
     <DeleteWithin>P1Y</DeleteWithin>
    </Obligations>
   </UsageControl>
  </ACUC>
 </Policy>
 ...
</Policies>
```

Fig. 3. Excerpt from bookshop.com's policies.

Figure 4 depicts shipping.com's relevant policies. When collecting addresses, it is willing to authenticate as a shipping company certified by CAy and as shipping.com certified by CAz. It intends to use the address for shipping and statistical purposes and will delete it within one week.

Sticky policy model Sticky policies follow the same schema as preferences, but have the `sticky` attribute in the `Preference` element set to true. This is to

```
<Policies id="Policies@Shipping">
 <Policy>
  <Applicability>
   <DataType> Address </DataType> </Applicability>
  <ACUC id="ACUCaddress@Shipping">
   <AccessControl>
    <Property>CertifiedAsBy{shipping,CAy}</Property>
    <Property>CertifiedAsBy{shipping.com,CAz}</Property>
   </AccessControl>
   <UsageControl>
    <Rights>
     <UseForPurpose>shipping</UseForPurpose>
     <UseForPurpose>statistics</UseForPurpose>
    </Rights>
    <Obligations>
     <DeleteWithin>P7D</DeleteWithin>
    </Obligations>
   </UsageControl>
  </ACUC>
 </Policy>
 ...
</Policies>
```

Fig. 4. Excerpt from shipping.com's policies.

indicate that this sticky policy originates from another party and must thus not be modified. Note that a data consumer may, in addition to the sticky policy, also have own preferences for forwarding previously received PII. Those preferences can, however, be changed and are therefore not sticky.

3.3 Downstream usage

The crucial aspect of our policy language is that it allows both the data provider and the data consumer to express to whom and under what conditions PII can or will be forwarded. These conditions are expressed in `UseDownstream` elements.

We first focus on `UseDownstream` elements occurring in the data provider's preferences. Each `UseDownstream` element contains exactly one `ACUC` child element. This `ACUC` element either contains a fully specified pair of access and usage control requirements, or another `ACUC` element is referenced with the `reference` attribute.

In the former case, the access control requirements specify to which downstream data consumers the PII can be forwarded, while the usage control requirements specify how these downstream consumers are supposed to treat it. Usage requirements can on their turn also contain `UseDownstream` elements that specify to whom and under what conditions the downstream consumer can further forward the PII, which on their turn can contain `UseDownstream` elements

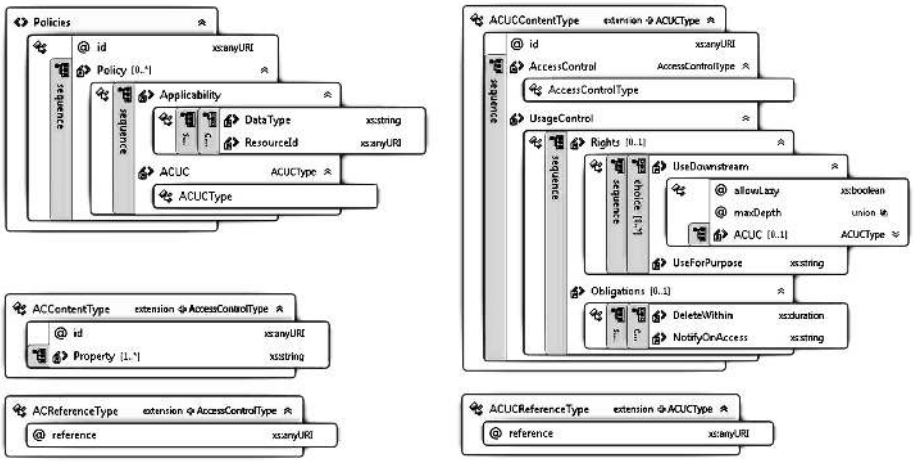


Fig. 5. Server-policy language model.

as well, etc. This mechanism enables the data provider to restrict the forwarding of his PII up to an arbitrary number of "hops". We refer to this approach as *nested* downstream usage control.

In the case where an `ACUC` element references the content of another `ACUC` element, for the sake of simplicity we insist that it can only refer to its closest ancestor `ACUC` element, i.e., the ancestor four levels higher in the XML tree. (We impose this restriction since it simplifies the matching procedure and there seem to be no convincing use cases for "cyclically recursive" policies with cycle length greater than one.) This means that the data consumer can then forward the PII under the same restrictions that were imposed on himself. We therefore call this approach *recursive* usage control. Note that our policy language allows to combine nested and recursive usage control by defining a chain of nested usage controls for the first number of hops and a final recursive usage control for any further hops.

In a data consumer's policy, each `UseDownstream` element contains *at most* one `ACUC` element. If present, it contains a set of properties describing to whom he plans to forward the PII, and a usage control policy describing how that downstream consumer will treat the data. This usage control policy could contain further `UseDownstream` elements, describing the next hops up to an arbitrary nesting degree. Alternatively, the `reference` attribute can be used to point to another `ACUC` element. This element may even be hosted directly by a downstream consumer (where we assume the `reference` to act as a URL).

In many situations, the downstream consumer or his policy are not be known at the time the PII is transmitted to the primary consumer. Rather than specifying all intended hops in full detail, the data consumer can indicate his willingness to enforce any restrictions imposed by the data provider by setting the `allowLazy` attribute of the `UseDownstream` element to `true` and omitting the

ACUC element. The matching between the data provider's preferences and the downstream consumer's policy is then done by the primary consumer at the time the PII is forwarded to the downstream consumer. We refer to the next section for more details on lazy matching. When the allowLazy attribute occurs in Preferences, it indicates whether the data provider allows the restrictions expressed in the child ACUC element to be matched lazily.

Finally, in the UseDownstream element a maxDepth attribute can be set to an integer or to unbounded to indicate how often a piece of PII can be forwarded at most. The intended behavior concerning this limit can be explained with a counter contained in sticky policies. The counter in a sticky policy that is attached to forwarded PII is decreased by one w.r.t. the previous sticky policy or w.r.t. maxDepth in case the PII is forwarded by the primary data provider.

3.4 To compose or not to compose.

To guarantee efficient matching of preferences and policies, our matching procedure sees UsageControl elements as monolithic blocks, and does *not* try to compose multiple UsageControl blocks when all of the corresponding AccessControl requirements are satisfied. For example, imagine that on top of the preferences described in Figure 1, Alice has a preference saying that electronics stores are allowed to use her address for marketing, as depicted in Figure 6. Suppose that

```
1   <Preference>
2    <Applicability>
3     <DataType> Address </DataType>
4    </Applicability>
5    <ACUC id="ACUCelshop@alice">
6     <AccessControl>
7      <Rule>CertifiedAsBy{electronicsshop,CAx}</Rule>
8     </AccessControl>
9     <UsageControl>
10     <Rights>
11      <UseForPurpose>marketing</UseForPurpose>
12     </Rights>
13    </UsageControl>
14   </ACUC>
15  </Preference>
```

Fig. 6. Additions to Alice's preferences from Figure 1.

Alice visits BEshop.com which is at the same time a book and electronics shop (and has the necessary credentials to prove this) and asks for Alice's address under the policy depicted in Figure 7 which says that she'll use her address for statistics, account administration, and marketing, and will delete it within one year.

```
<Policies id="Policies@BEshop">
 <Policy>
  <Applicability> <DataType> Address </DataType> </Applicability>
  <ACUC id="ACUCaddress@BEshop">
   <AccessControl>
    <Property>CertifiedAsBy{bookshop,CAx}</Property>
    <Property>CertifiedAsBy{electronicsshop,CAx}</Property>
    <Property>CertifiedAsBy{shop,CAx}</Property>
    <Property>CertifiedAsBy{BEshop.com,CAz}</Property>
   </AccessControl>
   <UsageControl>
    <Rights>
     <UseForPurpose>statistics</UseForPurpose>
     <UseForPurpose>accountadmin</UseForPurpose>
     <UseForPurpose>marketing</UseForPurpose>
    </Rights>
    <Obligations>
     <DeleteWithin>P1Y</DeleteWithin>
    </Obligations>
   </UsageControl>
  </ACUC>
 </Policy>
</Policies>
```

Fig. 7. Excerpt from BEshop.com's policies.

Intuitively, one may expect this policy to match the preferences, as the shop obtains the right to use the address for statistics and accountadmin based on its role as a bookshop, the right to use it for marketing based on its role as an electronics shop, and it adheres to the restriction of line 36 in Figure 1 to delete it within two years. Nonetheless, by our definition of matching the `ACUC` element `ACUCaddress@BEshop` in Figure 7 does matches neither the `ACUCbookshop@alice` element in Figure 1 nor the `ACUCelshop@alice` element in Figure 6 separately, so matching fails.

For a match to be found, the matching algorithm would have to be intelligent enough to "combine" rights and obligations from different `UsageControl` elements. Moreover, it would have to compute this combination for *every possible subset* of the `UsageControl` elements for which the corresponding `AccessControl` is satisfied. This causes an exponential blowup of the number of matchings to be performed, which of course we prefer to avoid.

Instead, we match `ACUC` elements atomically, and leave it either up to the data provider to explicitly add an `ACUC` element describing the case of combined book and electronics shops, or up to the shop to make two separate requests for Alice's address, once in its capacity of a book shop, and once in its capacity of an electronics shop.

4 Matching

Given a data provider's preferences and a consumer's policies, matching aims at automating the process of deciding whether the provider can safely transmit a piece of personal data. We introduce a '*more or equally permissive than*' operator to match preferences with policies. We say there is a *match* when the preferences are more or equally permissive than the policy.

4.1 Matching Privacy Preferences and Policies

To explain the matching procedure, we use a set-based representation of the XML structure described in the previous section. A `Preferences` element is represented by a set *Prefs* containing an element $Pref \in Prefs$ for each of its `Preference` child elements in the XML structure. $Pref.App$ represents a set containing all the PII owned by the user that is covered by the `Applicability` element, and $Pref.ACUC$ designates the contained `ACUC` child elements. The set $ACUC.AC$ is the set of access control rules (e.g., `CertAsBy`) contained in the `Rule` elements of the embedded `AccessControl`, while $ACUC.UC$ is the set of usage controls in terms of rights ($UC.Rights$) and obligations ($UC.Obls$), specified by the `Rights` and `Obligations` elements, respectively. We use an analogous notation for the consumers' policies.

Intuitively, preferences *Prefs* are more or equally permissive than policies *Pols*, denoted $Prefs \trianglerighteq Pols$, if the access control properties in *Pols* satisfy the rules in *Prefs* and if *Pols* asks for less rights and promises to adhere to stricter

obligations than specified in *Prefs*. We "overload" the notation of the $\trianglerighteq$ operator to compare not only preferences with policies, but also to compare rights, obligations, access control policies as well as usage control policies.

Matching of preferences and policies boils down to the matching of individual rights and obligations. To determine if there is a match between preferences *Prefs* and policies[2] *Pols*, it is verified if for each ACUC pair in a policy the user has a corresponding piece of PII with a more or equally permissive ACUC pair:

$$\begin{aligned} Prefs \trianglerighteq Pols \Leftrightarrow \forall Pol \in Pols \cdot \exists Pref \in Prefs \ \cdot \ \exists PII \in PIIs \cdot \\ PII \in (Pol.App \cap Pref.App) \cdot Pref.ACUC \trianglerighteq Pol.ACUC \end{aligned} \tag{1}$$

Above, *PIIs* is the set of all pieces of personal information that the user possesses, and *PII* can be any specific piece of PII in that set. In the following, we use the notations $*_{Pref}$ and $*_{Pol}$ to denote elements within preferences and policies respectively.

Pairs of access control and usage control policies are matched as follows:

$$\begin{aligned} ACUC_{Pref} \trianglerighteq ACUC_{Pol} \ \Leftrightarrow (ACUC_{Pref}.AC \trianglerighteq ACUC_{Pol}.AC) \wedge \\ (ACUC_{Pref}.UC \trianglerighteq ACUC_{Pol}.UC) \end{aligned} \tag{2}$$

Note that (2) is evaluated multiple times during the evaluation of (1). For example, $ACUC_{Pol}$ is instantiated subsequently with $Pol_i.ACUC$ for all Pol_i in *Pols*. The access control mechanism we employ is based on certified properties such as roles or IDs. To match access control requirements, it is verified if for each `Rule` in the preferences there is a corresponding `Property` in the policy:

$$AC_{Pref} \trianglerighteq AC_{Pol} \ \Leftrightarrow \ \forall r \in AC_{Pref} \cdot \exists r' \in AC_{Pol} \cdot r = r' \tag{3}$$

This could be extended to cover more sophisticated access control mechanisms such as claim-based access control or hierarchical roles. However, the matching becomes more complex when doing so (e.g., as environment attributes such as time of day cannot be pre-evaluated). Usage control requirements are matched as follows:

$$\begin{aligned} UC_{Pref} \trianglerighteq UC_{Pol} \ \Leftrightarrow \\ (\forall R \in UC_{Pol}.Rights \cdot \exists R' \in UC_{Pref}.Rights \cdot R' \trianglerighteq R) \wedge \\ (\forall O \in UC_{Pref}.Obls \cdot \exists O' \in UC_{Pol}.Obls \cdot O \trianglerighteq O') \end{aligned} \tag{4}$$

The matching of rights and obligations is specified for the different types of rights and obligations individually. In the following we give example specifications for the ones introduced in Section 3.2 (cf. [18] for more complex examples).

[2] Note that policies *Pols* are not complete privacy policies of a website but rather policies of specific inputs in a Web form or policies of credential attributes required to authenticate.

If obligations R and R' specify that the user must be notified when her PII is accessed, $R' \trianglerighteq R$ is evaluated with the appropriate $\trianglerighteq$ operator, i.e., (6) in this case. Letting obligation `DeleteWithin` with duration t be denoted as $DelWithin(t)$, matching is done as follows:

$$DelWithin(t) \trianglerighteq DelWithin(t') \;\Leftrightarrow\; t \geq t' \;. \tag{5}$$

Letting obligation `NotifyOnAccess` with contact information c be denoted as $NotifyOnAcc(c)$, matching is done as follows (where $*$ represents *any* string):

$$NotifyOnAcc(c) \trianglerighteq NotifyOnAcc(c') \;\Leftrightarrow\; c' = * \;\vee\; c = c' \;. \tag{6}$$

Letting right `UseForPurpose` with purpose p be denoted as $UseForPurp(p)$, matching is done as follows:

$$UseForPurp(p) \trianglerighteq UseForPurp(p') \;\Leftrightarrow\; p = p' \tag{7}$$

For the sake of readability, support for hierarchical purposes is not described here.

As downstream usage is the focus of this paper, the following two subsections explain the details of the matching procedure for downstream usage rights.

4.2 Proactive Matching of Downstream Rights

Supporting nested and recursive ACUC (access control and usage control) has an impact on matching. This section provides the intuition behind proactively matching a downstream structure, i.e., matching structures where both the preferences and the full chain of downstream usage policies are known at the time of matching.

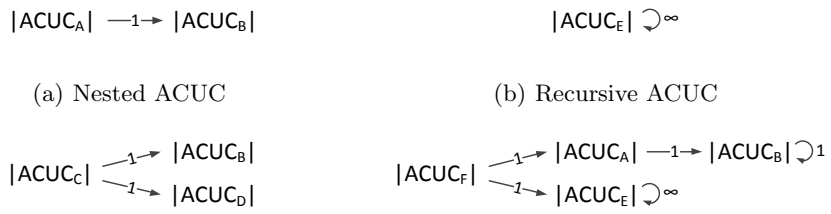
For a given pair $ACUC$, let $|ACUC|$ be the "local" ACUC, meaning containing only those restrictions and obligations that do not affect downstream usage, meaning

$$\begin{aligned} |ACUC|.AC &= ACUC.AC \\ |ACUC|.UC.Obls &= ACUC.UC.Obls \\ |ACUC|.UC.Rights &= \{R \in ACUC.UC.Rights : R \neq UseDS(\cdot,\cdot)\} \end{aligned}$$

We define the right of sharing downstream as $UseDS(\cdot,\cdot)$. Using this notation, we can represent the structure of an ACUC policy with downstream usage as a directed graph where each node represents a hop in the downstream usage. Each node is labeled with the local ACUC policy describing how the data are to be treated locally. Each edge represents the permission (in case of a provider's preferences) or intention (in case of a consumer's policy) to forward the data under the restrictions specified by the ACUC of the endpoint of the edge. For

instance, the case where $ACUC_A$ permits the right to share downstream under $ACUC_B$, but prohibits any further forwarding is depicted in Figure 8(a).

By the restrictions that we imposed on the connection among ACUC pairs, the structure of the graph is similar to that of a tree where the leaf nodes can optionally have a loop, representing recursion in the downstream usage policy. Figure 8(b) for example represents a simple recursive ACUC. Figure 8(c) is an example with two downstream ACUC policies. Figure 8(d) shows a deeper nested structure. Nodes in the graph can have multiple incident edges if multiple ACUCs



(a) Nested ACUC

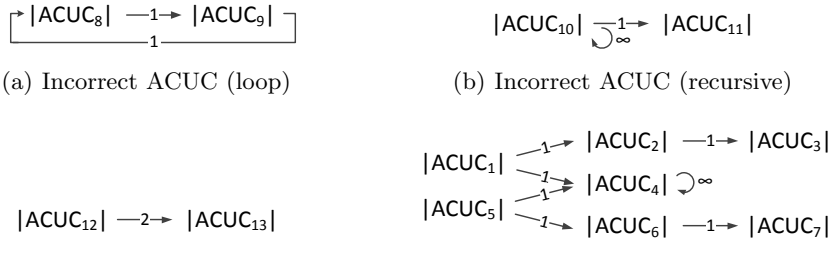
(b) Recursive ACUC

(c) ACUC with multiple downstream

(d) Deeper ACUC

Fig. 8. Examples of ACUC chaining

refer to it as part of their downstream rights. Figure 9(d) is correct for example, as by "doubling" node $ACUC_4$ it could be split into two separate trees rooted at $ACUC_1$ and $ACUC_5$. Figure 9(a) is not correct however because it contains a cycle: for the sake of readability and to avoid over-complex matching, cycles are forbidden. Figure 9(b) is not correct because only leaf nodes can be recursive. Figure 9(c) is not correct because nested downstream rights must have a depth equal to one. Intuitively, matching two ACUC pairs $ACUC_{Pref}$ taken from a



(a) Incorrect ACUC (loop)

(b) Incorrect ACUC (recursive)

(c) Incorrect ACUC (nested $\neq 1$)

(d) Correct ACUC

Fig. 9. Correct and incorrect chaining of ACUC

provider's preferences and $ACUC_{Pol}$ specified in a consumer's policy is done by

simultaneously going over the nodes in the two tree representations of $ACUC_{Pref}$ and $ACUC_{Pol}$ and verifying that it is possible to cover each branch of the policy-side tree with a more or equally permissive branch on the preference side. For instance, if $|ACUC_E| \trianglerighteq |ACUC_A|$ and $|ACUC_E| \trianglerighteq |ACUC_B|$ in Figures 8(b) and 8(a), then $ACUC_E \trianglerighteq ACUC_A$. However, it is impossible that $ACUC_A \trianglerighteq ACUC_E$ because $ACUC_E$ allows deeper downstream usage than $ACUC_A$.

Letting $UseDS(ACUC)$ denote a `UseDownstream` element with an `ACUC` child element represented by $ACUC$, the matching for downstream usage rights works according to the rule:

$$UseDS(ACUC) \trianglerighteq UseDS(ACUC') \;\Leftrightarrow\; ACUC \trianglerighteq ACUC' \tag{8}$$

We assume that an `ACUC` specifying recursive downstream sharing right with depth d is "folded out" into a graph of $d+1$ nested `ACUC`. Formula (8) is thus sufficient to handle nested as well as recursive access and usage control. Obviously, to improve performance when matching recursive `ACUC` and to avoid infinite loops in case of unlimited recursion depth, the concrete implementation of the matching algorithm must keep track of which combinations of `ACUC` elements have already been matched against one another and must take depth into account. We refer to Appendix A for a complete example of a proactive matching process.

4.3 Lazy Matching

In the previous section we focused on *proactive matching*, i.e., matching where all downstream policies are known beforehand. It is, however, not always possible to collect all policies during matching. For this reason, we also introduce *lazy matching*, which only takes into account the properties and policies of the data consumer, but not those of any downstream data consumers. Rather, the data consumer expresses that he is willing to impose whatever usage restrictions on downstream consumers that the data provider may specify.

Both types of matching imply that the sticky policy that the data consumer associates to the data must at least enforce the preferences of the data provider. On one hand, proactive matching allows to minimize the rights and maximize the obligations that are transferred as the matching procedure can already take the downstream consumers and their policies into account. On the other hand, lazy matching offers more flexibility and is the only option in dynamic settings, where either the downstream consumers or their policies are not known yet at the moment of matching, or where the access control policy depends on environment variables that will only be known when the data is actually forwarded.

Obviously, when all necessary information is available at the time of matching, proactive matching is preferable from a privacy perspective as it minimizes the rights that the provider has to give away. On the other hand, if the downstream consumers' policies change between the moment of matching and the actual moment of forwarding, then the transaction may fail even though the

new policy is still in accordance with the provider's preferences, but not with the (stricter) sticky policy that resulted earlier from the matching. Depending on the type of application, this type of failure may be acceptable or even recoverable by contacting the data subject to ask for additional rights.

To support lazy matching, we redefine formula (8) to take the `allowLazy` attribute into account, which is represented by a boolean value *lazy* here. Matching for downstream usage then follows the rule:

$$\begin{aligned} &UseDS(lazy, ACUC) \trianglerighteq UseDS(lazy', ACUC') \;\Leftrightarrow \\ &\quad (lazy \wedge lazy') \;\vee\; (ACUC \trianglerighteq ACUC') \end{aligned} \tag{9}$$

4.4 Creating Sticky Policies

A sticky policy specifies the commitment of the data consumer towards the data provider w.r.t. treatment of her shared data. We envision a sticky policy SP to be produced in case a matching procedure is successful. This sticky policy then never violates the preferences and is compliant with the policy, i.e., $Prefs \trianglerighteq SP \trianglerighteq Pols$ always holds. (Note that we assume that matching preferences with preferences is analog to the matching of preferences with policies.) For instance, the preferences may specify a maximum retention time of one year, the policy may specify that the data consumer needs to keep the data for six months, so the resulting sticky policy may mention any duration between six months and one year.

A privacy-conservative matching algorithm will choose a sticky policy that is as close as possible to the policy proposed by the consumer. In the example above, it would result in an obligation to delete the data within six months. Note that a sticky policy includes only those preferences that contain rights the data consumer gets or obligations he has to adhere to w.r.t. a given piece of PII.

Clearly, when a primary data consumer forwards the data to a downstream consumer, he must also attach a sticky policy. The data consumer may have its own preferences $Prefs'$ regarding data sharing. On top of his own preferences, the data consumer must enforce the sticky policy SP associated to this piece of data. In other words, the primary data consumer has to find a downstream sticky policy SP' so that $Prefs' \trianglerighteq SP' \trianglerighteq Pols'$ and, informally, $SP \trianglerighteq SP' \trianglerighteq Pols'$, where $Pols'$ is the policy of the downstream data consumer. More precisely, the latter evaluation is done by first extracting downstream preferences ($Prefs_{DS}$) from the sticky policy (SP) as depicted below, where $A \leftarrow B$ denotes the assignment to A of the value of B:

$$\begin{aligned} &\forall Pref \in SP \cdot (\forall R \cdot (R \in Pref.ACUC.UC.Rights \wedge R = UseDS(\cdot,\cdot)) \cdot \\ &\quad (Pref_{DS}.App \leftarrow Pref.App, \\ &\quad\; Pref_{DS}.ACUC \leftarrow R.ACUC, \\ &\quad\; Prefs_{DS} \leftarrow Prefs_{DS} \cup Pref_{DS})) \end{aligned} \tag{10}$$

Next $Prefs_{DS}$ is used for matching the downstream policy (i.e., $Prefs_{DS} \trianglerighteq Pols'$) and to create the downstream sticky policy SP' so that $Prefs_{DS} \trianglerighteq SP' \trianglerighteq Pols'$.

5 Conclusion and Future Work

This paper presents a simple yet expressive language to specify privacy policies and user preferences which may express downstream usage requirements. Our paper also describes a matching algorithm that, given a server's privacy policy, helps a user to decide whether her personal data can be shared with the server according to her preferences.

Since this paper mainly focuses on the downstream aspect of privacy matching, only basic examples of rights and obligations are provided. More details can be found in [18].

This work will be extended by looking at the future work described below:

Specification of logic-based representation of matching. We would like to specify the rules in a more formal way, e.g. using First Order Logic (FOL), Description Logic (DL) [19], Formula [20], or the Obligation Specification Language (OSL) [13] would be useful to verify matching, to reason on causes of mismatch, and to propose solutions (e.g. modified preferences). We started investigating different options and will continue this work in WP5.2.

Composition of policies. When combining pieces of data we can either keep separate preferences (or sticky policies) or compose preferences. In this case, each individual preference must be more (or equally) permissive than the composite preferences. When specifying the policy of a front end service, it is possible to either specify the policy of each downstream service or to combine those downstream policies into one composite policy. In this case, the composite policy must be more permissive than each individual policy. Specifying policy composition will be done in WP5.2 and WP6.3.

Obligation and authorization ontology. To be fully functional, one has to use our language in combination with a more complete ontology of authorization and obligation types. The definition of such an ontology is out of scope for this work, but we are planning to define more obligation types in a prototype implementation.

Integration into XACML. We consciously kept our policy language rather simple in expressing the applicability of a rule. By embedding our language into a practical language such as XACML, we could leverage the higher expressivity of the latter for our purposes. One issue that remains to be solved in this case, however, is what effect XACML's rule-combining algorithms will have on the embedded policies, in particular for combining algorithms where the effects of multiple rules have to be taken into account simultaneously. For some of these, one may have to compose the policies of multiple rules, similarly to what was suggested above when combining multiple pieces of data.

Matching of access control policies. By embedding our language into XACML, one can also profit from the expressivity of the latter in access control restrictions. One problem in that case, however, is that it becomes much harder to proactively match a downstream data consumer's properties against a specified access control policy, because not all the relevant attributes of the

downstream consumer may be known upfront. In particular, environment attributes (e.g., time of day, server load) may only be known at the time of actual access, further complicating the proactive matching procedure.
One idea to resolve this could be to design a "hybrid" between lazy and proactive matching, where both the data consumer's usage control policy does not contain the properties of the downstream consumer, but rather contains an access control policy that will be enforced on downstream consumers. Matching with a data provider's preferences involves checking whether the access control policy in the consumer's policy implies the one in the provider's preferences. Efficiently deciding implication of two access control policies may not be trivial for practical languages, however.

Comparing authorization and data sharing. In this work we focus on scenarios where the data is shared with the data consumer that must enforce usage control. In on-line scenarios, this could be implemented by keeping the data at user side and requiring the data consumer to request the data, for a specific purpose, each time it requires the data and to delete it immediately after usage. This approach would provide a better user control since the user would enforce usage control and be able at anytime to update personal data or to revoke access. Downstream sharing would be instantiated as delegation of rights, i.e. the data consumer authorizes a third party to access the user's personal data for a specific purpose. We want to evaluate whether the proposed language can be reused in such scenarios.

Acknowledgment

The authors would like to thank Ulrich Pinsdorf and Mario Verdicchio for their constructive feedback. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 216483 for the project PrimeLife.

References

1. Ashley, P., Hada, S., Karjoth, G., Powers, C., Schunter, M.: Enterprise privacy authorization language (EPAL 1.2) (2003)
2. W3C: The platform for privacy preferences 1.1 (P3P1.1) specification (November 2006)
3. W3C: A P3P preference exchange language 1.0 (APPEL1.0) (2002)
4. Becker, M.Y., Malkis, A., Bussard, L.: MSR-TR-2009-128: A framework for privacy preferences and data-handling policies. Technical report (September 2009)
5. Wang, X.: Mpeg-21 rights expression language: Enabling interoperable digital rights management. IEEE MultiMedia **11**(4) (2004) 84–87
6. ContentGuard: XrML 2.0 Technical Overview. `http://www.xrml.org/reference/XrMLTechnicalOverviewV1.pdf` (2002)
7. ODRL: Open Digital Rights Language (ODRL), version 1.1 (2002)
8. Ardagna, C.A., Cremonini, M., De Capitani di Vimercati, S., Samarati, P.: A privacy-aware access control system. J. Comput. Secur. **16**(4) (2008) 369–397

9. Park, J., Sandhu, R.: The UCONABC usage control model. ACM Trans. Inf. Syst. Secur. **7**(1) (2004) 128–174
10. Hilty, M., Basin, D., Pretschner, A.: On obligations. Lecture Notes in Computer Science **3679** (2005) 98–117
11. Hilty, M., Pretschner, A., Basin, D., Schaefer, C., Walter, T.: A policy language for distributed usage control. In Biskup, J., Lopez, J., eds.: 12th European Symposium on Research in Computer Security (ESORICS 2007). Volume 4734 of LNCS., Springer-Verlag (2007) 531–546
12. Leumann, M.: Policy evaluation and negotiation in distributed usage control (Master Thesis) (2007)
13. Pretschner, A., Schütz, F., Schaefer, C., Walter, T.: Policy evolution in distributed usage control. In: 4th Intl. Workshop on Security and Trust Management. Elsevier (June 2008)
14. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Computer **29**(2) (1996) 38–47
15. Bonatti, P., Samarati, P.: A unified framework for regulating access and information release on the web. Journal of Computer Security **10**(3) (2002) 241–272
16. Moses, T.: OASIS eXtensible Access Control Markup Language (XACML) Version 2.0. OASIS Standard oasis-access_control-xacml-2.0-core-spec-os, OASIS (February 2005)
17. Becker, M.Y., Fournet, C., Gordon, A.D.: SecPAL: Design and semantics of a decentralized authorization language. Journal of Computer Security (2009)
18. PrimeLife Project: Draft 2nd Design for Policy Languages and Protocols (Heartbeat: H 5.3.2). Technical report (July 2009)
19. Baader, F., Horrocks, I., Sattler, U.: Description Logics. In: Handbook of Knowledge Representation. Elsevier (2007)
20. Jackson, E.K., Schulte, W., Sztipanovits, J.: The power of rich syntax for model-based development. Technical report (2008)

A A Complete Example of Proactive Matching

This appendix shows how sample preferences and policies are matched using the rules specified in Section 4. Note that the policies and preferences are slightly different than the ones specified in Section 3.

A.1 Preferences

```
<Preferences id="Preferences@Alice">
  <Preference>
    <Applicability> <DataType> EMailAddress </DataType> </Applicability>
    <ACUC id="ACUCshipping@alice">
      <AccessControl>
        <Rule>CertifiedAsBy{role=shipping, issuer="CAy"}</Rule>
      </AccessControl>
      <UsageControl>
        <Rights>
          <UseForPurpose>statistics</UseForPurpose>
          <UseForPurpose>shipping</UseForPurpose>
        </Rights>
        <Obligations>
          <DeleteWithin>P7D</DeleteWithin>
        </Obligations>
      </UsageControl>
    </ACUC>
  </Preference>
  <Preference>
    <Applicability> <DataType> EMailAddress </DataType> </Applicability>
    <ACUC id="ACUCshop@alice">
      <AccessControl>
        <Rule>CertifiedAsBy{role=shop, issuer="CAx"}</Rule>
      </AccessControl>
      <UsageControl>
        <Rights>
          <UseDownstream allowLazy="false">
            <ACUC reference="ACUCshipping@alice"/>
          </UseDownstream>
          <UseForPurpose>statistics</UseForPurpose>
          <UseForPurpose>contact</UseForPurpose>
        </Rights>
        <Obligations>
          <DeleteWithin>P1Y</DeleteWithin>
        </Obligations>
      </UsageControl>
    </ACUC>
  </Preference>
</Preferences>
```

A.2 Policy

```
<Policies id="Policies@Shop">
  <Policy>
    <Applicability> <DataType> EMailAddress </DataType> </Applicability>
    <ACUC id="ACUCemail@Shop">
      <AccessControl>
        <Property>CertifiedAsBy{role=shop, issuer="CAx"}</Property>
        <Property>CertifiedAsBy{id="www.bookstore.com", issuer="CAz"}</Property>
      </AccessControl>
      <UsageControl>
        <Rights>
          <UseDownstream allowLazy="false">
            <ACUC reference="ACUCemail@Shipping"/>
          </UseDownstream>
          <UseForPurpose>contact</UseForPurpose>
        </Rights>
        <Obligations>
          <DeleteWithin>P14D</DeleteWithin>
        </Obligations>
      </UsageControl>
    </ACUC>
  </Policy>
</Policies>
```

```
<Policies id="Policies@Shipping">
  <Policy>
    <Applicability> <DataType> EMailAddress </DataType> </Applicability>
    <ACUC id="ACUCemail@Shipping">
      <AccessControl>
        <Property>CertifiedAsBy{role=shipping, issuer="CAy"}</Property>
        <Property>CertifiedAsBy{id="www.shipping.com", issuer="CAz"}</Property>
      </AccessControl>
      <UsageControl>
        <Rights>
          <UseForPurpose>shipping</UseForPurpose>
          <UseForPurpose>statistics</UseForPurpose>
        </Rights>
        <Obligations>
          <DeleteWithin>P5D</DeleteWithin>
        </Obligations>
      </UsageControl>
    </ACUC>
  </Policy>
</Policies>
```

A.3 Matching

This section will show step by step the matching process to verify $Preferences@Alice \trianglerighteq Policies@Shop$.

Step 1: According to (1), $Preferences@Alice \trianglerighteq Policies@Shop$ is true because:

- $Prefs[0].App \supseteq Pols[0].App$, i.e. $EMailAddress \supseteq EMailAddress$ is true.
- $Prefs[0].ACUC \trianglerighteq Pols[0].ACUC$, i.e. $ACUCshop@alice \trianglerighteq ACUCemail@Shop$ is true (see Step 2).

Step 2: According to (2), $ACUCshop@alice \trianglerighteq ACUCemail@Shop$ is true because:

- $ACUCshop@alice.AC \trianglerighteq ACUCemail@Shop.AC$ is true, i.e. access is indeed granted according to access control rule: certified as "shop" by "CAx".
- $ACUCshop@alice.UC \trianglerighteq ACUCemail@Shop.UC$ is true (see Step 3).

Step 3: According to (4), $ACUCshop@alice.UC \trianglerighteq ACUCemail@Shop.UC$ is true because:

- AuthZ:
  - $UseDS(\ lazy : false, ACUCshipping@alice) \trianglerighteq UseDS(\ lazy : false, ACUCemail@Shipping)$ is true (see Step 4).
  - $UseForPurp(contact) \trianglerighteq UseForPurp(contact)$ is true according to (7).
- Obligations:
  - $DelWithin(1year) \trianglerighteq DelWithin(14days)$ is true according to (5).

Step 4: According to (9), $UseDS(\ lazy : false, ACUCshipping@alice) \trianglerighteq UseDS(\ lazy : false, ACUCemail@Shipping)$ is true because:

- $UseDS(lazy : false, ACUCemail@Shipping\}.lazy$ is $false$ and
- $ACUCshipping@alice \trianglerighteq ACUCemail@Shipping$ is true (see Step 5).

Step 5: According to (2), $ACUCshipping@alice \trianglerighteq ACUCemail@Shipping$ is true because:

- $ACUCshipping@alice.AC \trianglerighteq ACUCemail@Shipping.AC$ is true, i.e. access is indeed granted according to access control rule: certified as "shipping" by "CAy".
- $ACUCshipping@alice.UC \trianglerighteq ACUCemail@Shipping.UC$ is true (see Step 6).

Step 6: According to (4), $ACUCshipping@alice.UC \trianglerighteq ACUCemail@Shipping.UC$ is true because:

- AuthZ:
  - $UseForPurp(shipping) \trianglerighteq UseForPurp(shipping)$ is true according to (7).
  - $UseForPurp(statistics) \trianglerighteq UseForPurp(statistics)$ is true according to (7).
- Obligations:
  - $DelWithin(7days) \trianglerighteq DelWithin(5days)$ is true according to (5).

Policy matching in a more complex setting would imply more downstream preferences and policies and would require matching other types of obligations and authorizations. Applicability would also be more complex when hierarchy of data types would be used.

B Schema of Preferences, Policies, and Sticky Policies

For completeness, we provide here the full XML Schema definition of our language. We have separate schemas for preferences and policies due to some subtle differences between the two (e.g., `Rule` vs. `Property` elements and different default values of the `allowLazy` attribute). The sticky policy schema is the same as that of the preferences.

B.1 Preferences and Sticky Policies Schema

```
<xs:schema xmlns="http://www.primelife.eu/wp5.2/downstream/preferences"
targetNamespace="http://www.primelife.eu/wp5.2/downstream/preferences"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="Preferences">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="Preference"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:anyURI"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Preference">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Applicability"/>
        <xs:element ref="ACUC"/>
      </xs:sequence>
      <xs:attribute name="sticky" type="xs:boolean" default="false"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Applicability">
    <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:choice>
          <xs:element ref="DataType"/>
          <xs:element ref="ResourceId"/>
          <!-- New ways of addressing resources to be inserted here -->
                </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="DataType" type="xs:string"/>
  <xs:element name="ResourceId" type="xs:anyURI"/>

  <xs:element name="ACUC" type="ACUCType"/>
  <xs:complexType name="ACUCType" abstract="true">
```

```
40      <xs:sequence />
41    </xs:complexType>
42
43    <xs:complexType name="ACUCReferenceType">
44      <xs:complexContent mixed="false">
45        <xs:extension base="ACUCType">
46          <xs:attribute name="reference" type="xs:anyURI" use="required" />
47        </xs:extension>
48      </xs:complexContent>
49    </xs:complexType>
50
51    <xs:complexType name="ACUCContentType">
52      <xs:complexContent mixed="false">
53        <xs:extension base="ACUCType">
54          <xs:sequence>
55            <xs:element ref="AccessControl"/>
56            <xs:element ref="UsageControl"/>
57          </xs:sequence>
58          <xs:attribute name="id" type="xs:anyURI" use="optional" />
59        </xs:extension>
60      </xs:complexContent>
61    </xs:complexType>
62
63    <xs:element name="AccessControl">
64      <xs:complexType>
65        <xs:sequence>
66          <xs:element minOccurs="0" maxOccurs="unbounded" ref="Rule" />
67        </xs:sequence>
68      </xs:complexType>
69      </xs:element>
70
71    <xs:element name="Rule" type="xs:string" />
72
73    <xs:element name="UsageControl">
74      <xs:complexType>
75        <xs:sequence>
76          <xs:element ref="Rights" minOccurs="0"/>
77          <xs:element ref="Obligations" minOccurs="0"/>
78        </xs:sequence>
79      </xs:complexType>
80    </xs:element>
81
82    <xs:element name="Rights">
83      <xs:complexType>
84        <xs:sequence>
85          <xs:choice minOccurs="0" maxOccurs="unbounded">
86            <xs:element ref="UseDownstream" />
87            <xs:element ref="UseForPurpose"/>
88            <!-- New right types to be inserted here -->
89          </xs:choice>
```

```
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="UseDownstream">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ACUC"/>
      </xs:sequence>
      <xs:attribute name="allowLazy" type="xs:boolean" default="true"/>
      <xs:attribute name="maxDepth">
        <xs:simpleType>
          <xs:union>
            <xs:simpleType>
              <xs:restriction base="xs:integer"/>
            </xs:simpleType>
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="unbounded"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:union>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>

  <xs:element name="UseForPurpose" type="xs:string"/>

  <xs:element name="Obligations">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:choice>
          <xs:element ref="DeleteWithin"/>
          <xs:element ref="NotifyOnAccess"/>
          <!-- New obligation types to be inserted here -->
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="DeleteWithin" type="xs:duration"/>
  <xs:element name="NotifyOnAccess" type="xs:string"/>

</xs:schema>
```

B.2 Policies Schema

```
<xs:schema
xmlns="http://www.primelife.eu/wp5.2/downstream/policies"
targetNamespace="http://www.primelife.eu/wp5.2/downstream/policies"
```

```
xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="Policies">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="Policy"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:anyURI"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="Policy">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Applicability"/>
        <xs:element maxOccurs="1" ref="ACUC"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Applicability">
    <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:choice>
          <xs:element ref="DataType"/>
          <xs:element ref="ResourceId"/>
          <!-- New ways of addressing resources to be inserted here -->
                </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="DataType" type="xs:string"/>
  <xs:element name="ResourceId" type="xs:anyURI"/>


  <xs:element name="ACUC" type="ACUCType"/>
  <xs:complexType name="ACUCType" abstract="true">
    <xs:sequence />
  </xs:complexType>

  <xs:complexType name="ACUCReferenceType">
    <xs:complexContent mixed="false">
      <xs:extension base="ACUCType">
        <xs:attribute name="reference" type="xs:anyURI" use="required" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="ACUCContentType">
    <xs:complexContent mixed="false">
```

```
54          <xs:extension base="ACUCType">
55            <xs:sequence>
56              <xs:element ref="AccessControl"/>
57              <xs:element ref="UsageControl"/>
58            </xs:sequence>
59            <xs:attribute name="id" type="xs:anyURI" use="optional" />
60          </xs:extension>
61        </xs:complexContent>
62      </xs:complexType>
63
64      <xs:element name="AccessControl" type="AccessControlType"/>
65
66      <xs:complexType name="AccessControlType" abstract="true">
67        <xs:sequence />
68      </xs:complexType>
69
70      <xs:complexType name="ACReferenceType">
71        <xs:complexContent>
72          <xs:extension base="AccessControlType">
73            <xs:attribute name="reference" type="xs:anyURI" use="required" />
74          </xs:extension>
75        </xs:complexContent>
76      </xs:complexType>
77
78      <xs:complexType name="ACContentType">
79        <xs:complexContent>
80          <xs:extension base="AccessControlType">
81            <xs:sequence>
82              <xs:element ref="Property" maxOccurs="unbounded"/>
83                    </xs:sequence>
84            <xs:attribute name="id" type="xs:anyURI"/>
85          </xs:extension>
86        </xs:complexContent>
87      </xs:complexType>
88
89      <xs:element name="Property" type="xs:string"/>
90
91      <xs:element name="UsageControl">
92        <xs:complexType>
93          <xs:sequence>
94            <xs:element ref="Rights" minOccurs="0"/>
95            <xs:element ref="Obligations" minOccurs="0"/>
96          </xs:sequence>
97        </xs:complexType>
98      </xs:element>
99
100     <xs:element name="Rights">
101       <xs:complexType>
102         <xs:sequence>
103           <xs:choice minOccurs="0" maxOccurs="unbounded">
```

```
104             <xs:element ref="UseDownstream" />
105             <xs:element ref="UseForPurpose" />
106             <!-- New right types to be inserted here -->
107           </xs:choice>
108         </xs:sequence>
109       </xs:complexType>
110     </xs:element>
111
112     <xs:element name="UseDownstream">
113       <xs:complexType>
114         <xs:sequence>
115           <xs:element ref="ACUC" minOccurs="0" maxOccurs="1"/>
116         </xs:sequence>
117         <xs:attribute name="allowLazy" type="xs:boolean" default="false"/>
118         <xs:attribute name="maxDepth">
119           <xs:simpleType>
120             <xs:union>
121               <xs:simpleType>
122                 <xs:restriction base="xs:integer"/>
123               </xs:simpleType>
124               <xs:simpleType>
125                 <xs:restriction base="xs:string">
126                   <xs:enumeration value="unbounded"/>
127                 </xs:restriction>
128               </xs:simpleType>
129             </xs:union>
130           </xs:simpleType>
131         </xs:attribute>
132       </xs:complexType>
133     </xs:element>
134
135     <xs:element name="UseForPurpose" type="xs:string"/>
136
137     <xs:element name="Obligations">
138       <xs:complexType>
139         <xs:sequence maxOccurs="unbounded">
140           <xs:choice>
141             <xs:element ref="DeleteWithin"/>
142             <xs:element ref="NotifyOnAccess"/>
143             <!-- New obligation types to be inserted here -->
144           </xs:choice>
145         </xs:sequence>
146       </xs:complexType>
147     </xs:element>
148
149     <xs:element name="DeleteWithin" type="xs:duration"/>
150     <xs:element name="NotifyOnAccess" type="xs:string"/>
151
152   </xs:schema>
```