REWERSE©
reasoning on the web

A3-D12

An infrastructure for Web Service-based personalization in the Semantic Web: the Personal Reader Framework.

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Hannover/A3-D12/D/PU/b1
Responsible editors:	Nicola Henze
Reviewers:	Philipp Kärger
Contributing participants:	Hannover, Malta, Tekniker, Telefonica, Webxcerpt
Contributing workpackages:	I2, A3
Contractual date of deliverable:	February 29, 2008
Actual submission date:	February 29, 2008

Abstract
In this deliverable we describe a fully functional, Web Service-based framework – the Personal Reader Framework – for designing, developing and maintaining personalized information systems, with many realized application and demonstrators. Various Personalization Services are available in these demonstrators, which offer personal news recommendations, personal music recommendations, suggest study plans to students, annotate learning resources with relevant contextual information (exercises, quizzes, examples, summaries). Privacy protection is ensured by a centralized user modeling service. Access to user profile information is protected by user-specified policies.

Keyword List
semantic web, reasoning, personalization, user modeling, trust, access control, policies, personal reader

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

An infrastructure for Web Service-based personalization in the Semantic Web: the Personal Reader Framework.

Fabian Abel[1], **Juri Luca De Coi**[1], **Nicola Henze**[1], **Arne Wolf Koesling**[1], **Daniel Krause**[1], **Daniel Olmedilla**[1]

[1] L3S Research Center, University of Hannover, Germany
Email: {abel,decoi,henze,koesling,krause,olmedilla}@L3S.de

February 29, 2008

Abstract
In this deliverable we describe a fully functional, Web Service-based framework – the Personal Reader Framework – for designing, developing and maintaining personalized information systems, with many realized application and demonstrators. Various Personalization Services are available in these demonstrators, which offer personal news recommendations, personal music recommendations, suggest study plans to students, annotate learning resources with relevant contextual information (exercises, quizzes, examples, summaries). Privacy protection is ensured by a centralized user modeling service. Access to user profile information is protected by user-specified policies.

Keyword List
semantic web, reasoning, personalization, user modeling, trust, access control, policies, personal reader

Contents

1 Introduction

This deliverable reports about the achievements on the issue of personalized information portals, carried on within the working group A3, with a particular attention on the Personal Reader Framework, which has been developed during the REWERSE project.

Today, there exist various frameworks and toolkits like the Spring Framework[1], Ruby on Rails[2], etc. that facilitate development of modular Web applications. On the other hand, there are several Semantic Web frameworks like Jena[3], Sesame[4], etc. that enrich these Web applications with Semantic Web features. However, frameworks for developing Web information systems that exploit Semantic Web technologies and, in addition, support enhanced personalization and user adaptation functionalities are still missing. The Personal Reader Framework closes this gap.

The Personal Reader Framework is a fully functional, Web Service-based framework for designing, developing and maintaining personalized (Semantic) Web information systems. The core idea of the framework, which we introduce in Section 2, is to encapsulate the functionality of personalization of content within services. Thereby, personalization functionality can be re-used by various applications, which has – in addition to the features of the Personal Reader Framework – a positive effect on the development period of new applications. In Section 5 we will demonstrate this advantage on the basis of the many applications, which have been implemented by aid of the Personal Reader Framework during the REWERSE project.

An important feature of the Personal Reader Framework is the shared user modeling component, which is deployed as a service as well (see Section 3). User profiles are shared across different applications, which makes privacy issues crucial. In Section 4 we therefore present an advanced, novel mechanism to control access to RDF-based user profiles.

2 Personal Reader Framework

The *Personal Reader Framework* [Henze, 2005] (cf. Deliverable A3-D10) enables the creation of modular web service based applications (Figure 1 outlines its architecture). These applications are accessed by device-adaptable user interfaces (UI for short). *Syndication Services* implement the application logic and can be considered as the core of an application. By aid of a *Connector Service*, Syndication Services are able to discover and access *Personalization Services* dynamically, which aggregate domain-specific information in a personalized way. To gather information, Personalization Services access and process Semantic Web data sources. An important feature of the Personal Reader Framework is that new services can be integrated in a *plug-and-play* manner, hence no centralized component has to be modified and new services can be used immediately from all other services within the framework.

Both, Syndication and Personalization Services are able to access and store user data which is supplied by a centralized *User Modeling Service*. Therewith, the actual user profiles are also stored centrally in a shared repository which makes access control crucial and requires a sophisticated, fine-grained approach to control the access to these user information. In Section 4 we present an approach, which is based on user-adaptable policies and which fulfills these requirements. The Personal Reader Framework allows to model Syndications Services as state

[1]http://www.springframework.org
[2]http://www.rubyonrails.org
[3]http://jena.sourceforge.net
[4]http://www.openrdf.org

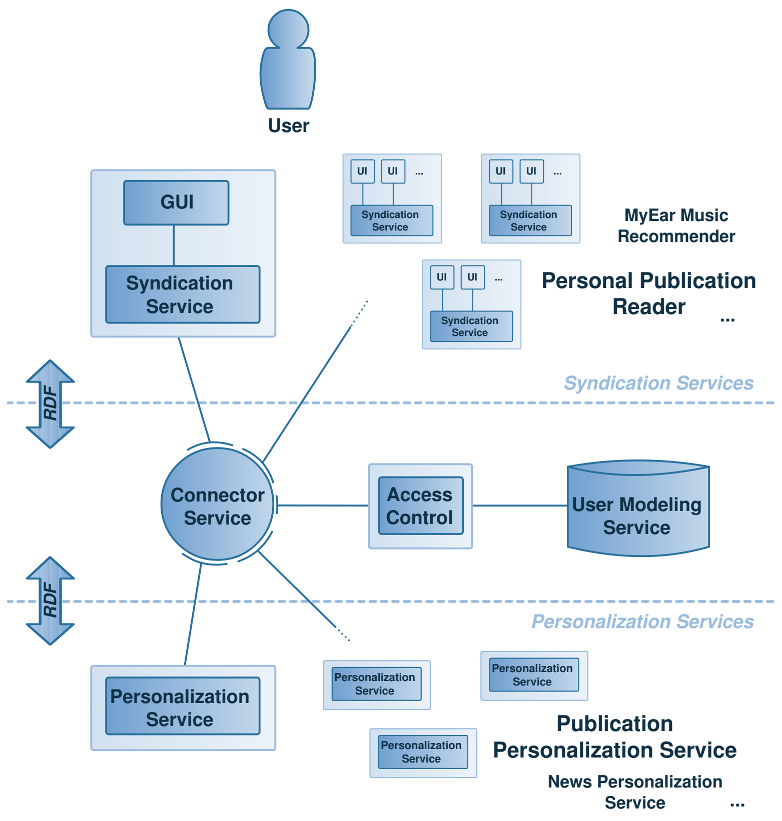


Figure 1: Personal Reader Architecture.

machines. This facilitates implementation of application logic in multiple ways. Developers just have to implement *action classes* and provide an RDF description of their state machine, which links those actions with states and events. Moreover, such formal description of application logic eases decoupling the functionality for capturing user observations.

Several applications have been implemented with the Personal Reader Framework – as presented in Deliverable A3-D10 – like the *Personal Publication Reader* [Baumgartner et al., 2005], the *MyEar* music recommender [Abel et al., 2006] or the *Personal Reader Agent*[5].

2.1 Personalization Services

The core idea of the Personal Reader Framework is to encapsulate personalization of content within services that can be re-used by various applications. Therefor, we introduced the novel concept of *Personalization Services*. Personalization Services are specialized on a certain domain of knowledge. They provide personalized content as RDF. In order to provide personalization features they are able to access and store user profile data that is shared accross services. Characteristic of Personalization Services ranges from services that simply wrap non-RDF data sources – e.g. a service that calls the Flickr API considering the user's preferences and trans-

[5]http://www.personal-reader.de/agent

forms the Flickr result into RDF using taxonomies like Dublin Core Metadata Element Set – to services that carry out more complex tasks – e.g. a music recommender service like MyEar. Personalization Services are described using well-known standards like OWL-S so that they can be discovered and utilized by other services at runtime. The Personal Reader Framework allows to describe Personalization Services semantically using a simplified vocabulary, which provides features to describe the inputs of services and is defined in the *Configuration Ontology.*

2.1.1 Configuration Ontology

The *Configuration Ontology* defines, on the one, the vocabulary that is needed to describe the inputs of Web Services and, on the other hand, concepts that are required for personalization functionalities. Figure 2 illustrates the concepts of the Configuration Ontology.

1. **Core Configurable Vocabulary (needed to describe a Configurable Web Service):**

   Configurable An instance of this class characterizes the *configurable* inputs of a Personalization Service. Therefor, a `name` and a `description` has to be defined. Example:

   ```
   (#MyEarConfigurable, name, "MyEar Configurable")
   (#MyEarConfigurable, description, "Configurable things of my MyEar Music Web Service")
   ```

   ConfigurableItem A `Configurable` consists of several `ConfigurableItems`. Example:

   ```
   (#MyEarConfigurable, hasConfigurableItem, #DurationItem)
   (#DurationItem, name, "Duration")
   (#DurationItem, description, "Duration of a Song that should be taken into account by my Web Service.")
   ```

   Input Every `ConfigurableItem` has at least one `Input`. We define two special Inputs: a `SelectionInput`, which allows only predefined values, and a `TextInput`, which allows arbitrary values. For an Input a `type`, a `minNumber-` and a `maxNumberOfInputValues` has to be specified. Example:

   ```
   (#DurationItem, input, #MinDurationInput)
   (#MinDurationInput, description, "The minimum duration of a song (in minutes)")
   (#MinDurationInput, type, http://www.w3.org/2001/XMLSchema#nonNegativeInteger)
   (#MinDurationInput, minNumberOfInputValues, 0)
   (#MinDurationInput, maxNumberOfInputValues, 1)

   (#DurationItem, input, #MaxDurationInput)
   ...
   ```

2. **User and their configured Personalization Services** – concepts needed to realize personalization functionalities:

   User This concept models the users of the *Personal Reader*. Therefor a `User` is a subclass of `foaf:Person` and is featured with an `username`, `password`, `name`, etc. and a list of `ConfiguredWebservices` (`hasConfiguredWebservice`). To link other descriptions, which characterize the user, we define the properties `researcherURI`, which points to the corresponding instance within our *Researcher Ontology*[6], and `foafURL`, which links the FOAF[7] description of the user. Example:

   ```
   (#abelFabian, username, "fabian")
   (#abelFabian, name, "Fabian Abel")
   (#abelFabian, foafURL, "http://www.fabianabel.de/foaf.rdf")
   (#abelFabian, researcherURI, http://www.personal-reader.de/rewerse#abelFabian)
   (#abelFabian, hasConfiguredWebservice, #abelFabianMyEarJazzConfigWS)
   (#abelFabian, hasConfiguredWebservice, #abelFabianPPRBioInformaticsConfigWS)
   ...
   ```

[6]The Researcher Ontology models the organizational structure of the REWERSE project: http://www.personal-reader.de/rdf/ResearcherOntology.owl

[7]Friend of a Friend: http://www.foaf-project.org/

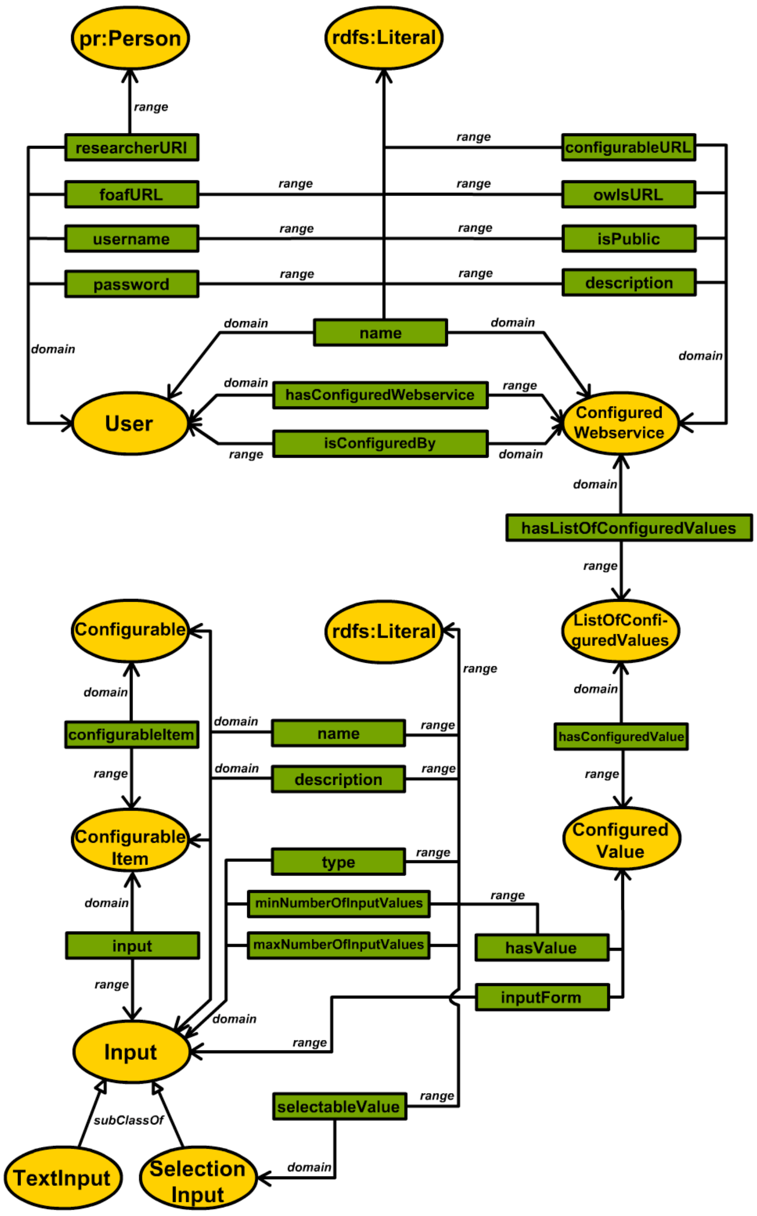


Figure 2: *Configuration Ontology* for describing adjustable inputs of Personalization Services.

ConfiguredWebservice This concept is used to store configurations of Web Services made by a user. The properties `name` and `description` allow to describe the concrete configuration. The boolean property `isPublic` indicates whether a `ConfiguredWebservice` can be accessed and re-used by other users than the user who configured it (`isConfiguredBy`). `owlsURL` points to the OWL-S[OWL-S, 2004] description of the Web Service that was configured by the user and `configurableURL` points to the `Configurbale` description. The values that belong to the concrete configuration are listed within the `ListOfConfiguredValues`. Example:

```
(#abelFabianMyEarJazzConfigWS, name, "Jazz Music")
(#abelFabianMyEarJazzConfigWS, description, "This configuration of the MyEar Music Web
                                             Service effects the Web Service to aggregate
                                             podcasting items that are related with Jazz.")
(#abelFabianMyEarJazzConfigWS, isPublic, "true")
(#abelFabianMyEarJazzConfigWS, isConfiguredBy, #abelFabian)
(#abelFabianMyEarJazzConfigWS, owlsURL, "...MyEar/rdf/MyEarOWLS.owl")
(#abelFabianMyEarJazzConfigWS, configurableURL, #MyEarConfigurable)
(#abelFabianMyEarJazzConfigWS, hasListOfConfiguredValues, #abelFabianMyEarJazzValueList)
```

ListOfConfiguredValues This is a list of the values that are configured by a user. Each `ConfiguredValue` has a `value` (range: typed Literals) and a reference to the `Input` (`inputForm`) which defines what is applicable in general. Example:

```
(#abelFabianMyEarJazzValueList, hasConfiguredValue, #abelFabianMyEarJazzValue1)
(#abelFabianMyEarJazzValue1, value, "3")
(#abelFabianMyEarJazzValue1, inputForm, #MinDurationInput)
(#abelFabianMyEarJazzValueList, hasConfiguredValue, #abelFabianMyEarJazzValue2)
...
```

2.1.2 Example: MyEar Personalization Service

The MyEar Personalization Service enables users to listen to their personalized *podcasting feed*. A podcasting feed is in fact a RSS 2.0 feed [RSS Advisory Board, 2002], whose items refer to audio files. The MyEar Personalization Service searches the web for podcasting feeds that suits to the users interest and then combines items from different feeds to present a personalized podcasting feed to the user.

Some extracts of the Configurable description are outlined in the examples of Section 2.1.1. In general the MyEar Configurable description consists of four *ConfigurableItems*:

myEarKeywordItem A keyword that should be within an item of a podcasting feed. `myEarKeywordItem` is defined as followed:

```
(#myEarKeywordItem, input, #myEarKeyword)
(#myEarKeyword, rdf:type, #TextInput)
(#myEarKeyword, minNumberOfInputValues, 1)
(#myEarKeyword, description, "Enter at least one keyword that should be within an item of
                              a podcasting feed, e.g. Jazz, Classic,..")
(#myEarKeyword, type, http://www.w3.org/2001/XMLSchema#string)
...
```

myEarItunesCategoryItem This item refers to a `SelectionInput` which only permits the selection of values that correspond to *itunes:category*[Apple Computer, Inc., 2006]:

```
(#myEarItunesCategoryItem, input, #myEarCategory)
(#myEarCategory, rdf:type, #SelectionInput)
(#myEarCategory, selectableValue, "Music")
(#myEarCategory, selectableValue, "Public Radio")
(#myEarCategory, selectableValue, "Arts")
...
(#myEarCategory, minNumberOfInputValues, 0)
(#myEarCategory, type, http://www.w3.org/2001/XMLSchema#string)
...
```

myEarDurationItem The duration item has two inputs which allow to specify minimum and maximum duration of audio files that should be included into the personalized podcasting feed:

```
(#myEarDurationItem, input, #myEarMinDuration)
(#myEarDurationItem, input, #myEarMaxDuration)
...
```

maxNumberOfGoogleCallsItem For the search of applicable podcasting feeds we use the *Google SOAP Search API*[8]. According to the Google API terms we only get 10 results per requests and thus the MyEar Music Web Service has to call the Google Search Web Service several times. By enabling the user to configure the maximum number of Google calls the user can affect the runtime of the MyEar podcasting item gathering process.

Each Personalization Service has to implement the interface `PService`, which defines the method `invoke()`. The invoke method has two parameters, the RDF data (*rdfRequest*), which embodies the `ListOfConfiguredValues` (see last example of Section 2.1.1), and a security token, which is e.g. needed when the Personalization Service accesses the User Modeling Service, or when another Personalization Service is called. The Personal Reader Framework provides abstract implementations of Personalization Services and various utility classes so that implementation of Personalization Services does not require any important overhead. The following code snippet shows how the MyEar implementation of the PService interface looks like.

```
public class MyEarPersonalizationService extends AbstratcPService implements PService {

  public String invoke(String rdfRequest, String securityToken) {
    Configuration userConfiguration =  JenaConfigurationFactory.constructConfiguration(rdfRequest);
    retrun MyEarLogic.getPersonalizedFeed(userConfiguration, securityToken);
  }

}
```

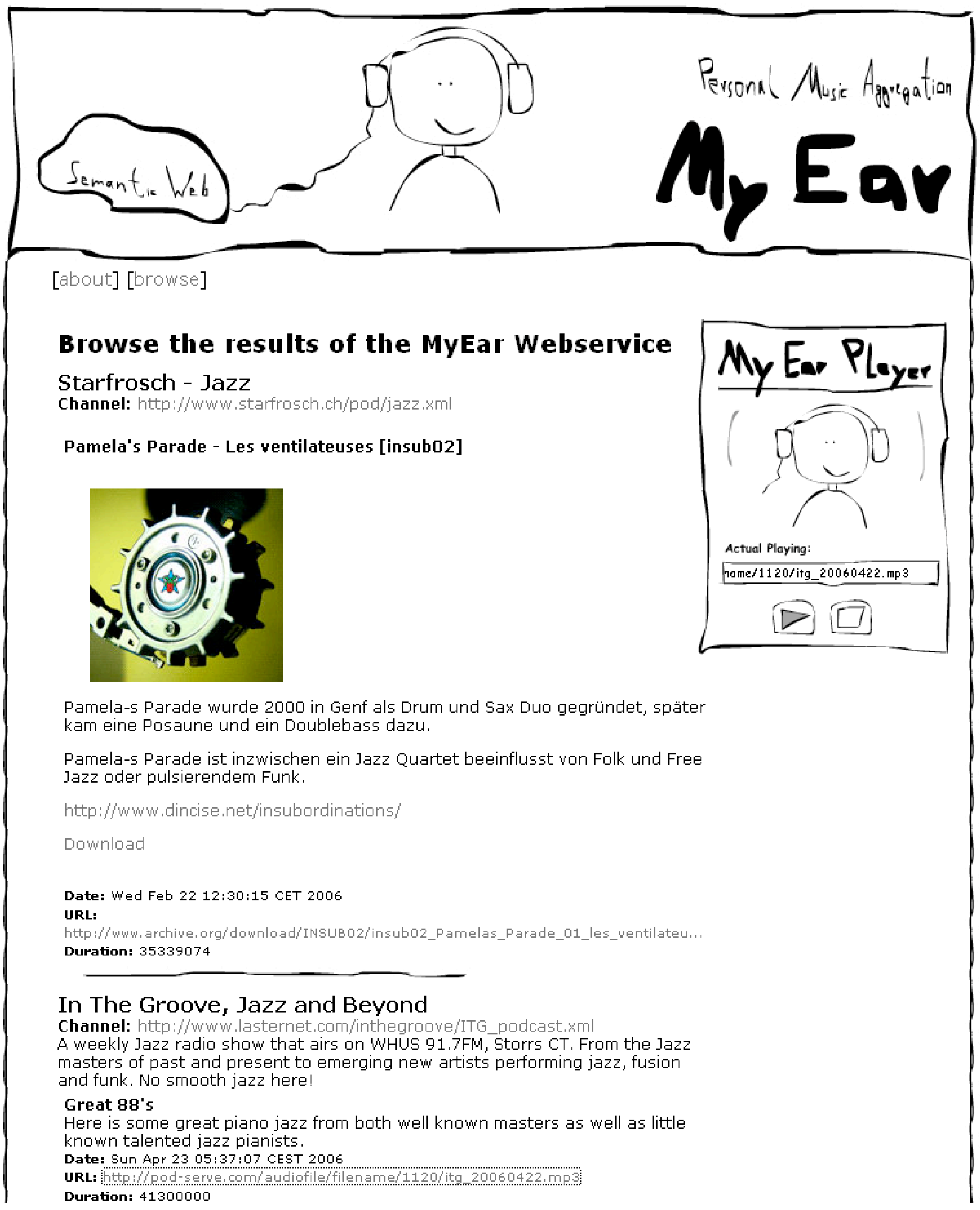
The result of the MyEar Personalization Service is a personalized RDF-based RSS feed, which is visualized by the user interface of the corresponding MyEar Syndication Service as shown in Figure 3.

2.2 Syndication Services

Syndication Services are responsible for syndicating the results of Personalization Services and for providing appropriate user interfaces that display the results of the syndicated results. Each Syndication Service provides at least one user interface, which can be interpreted as a user end point to a certain domain or task, and allows the user to benefit from many Personalization Services simultaneously, which are selected, combined and customized according to the requirements of the application, which is embodied by the Syndication Service, and according to the user needs. In general Syndication Services implement the application logic. They react to user actions forwarded from a user interface, syndicate content from Personalization Services considering user preferences, may update user profiles and return syndicated RDF content to the invoking user interface. Personalization Services deliver content as RDF. Hence, Syndication Services just have to be aware of the used vocabulary in order to combine content provided by different services.

The Personal Publication Reader [Baumgartner et al., 2005] is a good example of a Syndication Service. It makes use of a Personalization Service that provides information about publications and enriches this information with details about authors by utilizing on the one hand an *URI resolving service* and on the other hand a service that delivers details about persons.

[8]http://code.google.com/apis/soapsearch/



[about] [browse]

Browse the results of the MyEar Webservice

Starfrosch - Jazz
Channel: http://www.starfrosch.ch/pod/jazz.xml

Pamela's Parade - Les ventilateuses [insub02]

Pamela-s Parade wurde 2000 in Genf als Drum und Sax Duo gegründet, später kam eine Posaune und ein Doublebass dazu.

Pamela-s Parade ist inzwischen ein Jazz Quartet beeinflusst von Folk und Free Jazz oder pulsierendem Funk.

http://www.dincise.net/insubordinations/

Download

Date: Wed Feb 22 12:30:15 CET 2006
URL: http://www.archive.org/download/INSUB02/insub02_Pamelas_Parade_01_les_ventilateu...
Duration: 35339074

In The Groove, Jazz and Beyond
Channel: http://www.lasternet.com/inthegroove/ITG_podcast.xml
A weekly Jazz radio show that airs on WHUS 91.7FM, Storrs CT. From the Jazz masters of past and present to emerging new artists performing jazz, fusion and funk. No smooth jazz here!

Great 88's
Here is some great piano jazz from both well known masters as well as little known talented jazz pianists.
Date: Sun Apr 23 05:37:07 CEST 2006
URL: http://pod-serve.com/audiofile/filename/1120/itg_20060422.mp3
Duration: 41300000

Figure 3: MyEar User Iterface.

The *MyEar Music Syndicator* is another example of such a Syndication Service. It provides the user end point, which is shown in Figure 3, to specify requests for podcasts and to enable the user to listen to the generated personal podcasting feed.

The Personal Reader Framework provides abstract implementations of Syndication Services and a lot of utility classes in order to fasten and ease development of Syndication Services. For example, the framework offers a *generic state machine* implementation of a Syndication Service. Implementing a Syndication Service via the state machine mechanism requires the developer only to implement *actions* that should be executed whenever a certain *event* occurs. In addition, the developer has to provide an RDF description, which models possible *event-action-state sequences*, and JSPs for rendering the results.

2.2.1 Example: Personal Reader Agent Syndication Service

The *Personal Reader Agent* [Abel et al., 2006] is a generic Syndication Service. The Agent is a kind of wizard that allows to select, configure and call the Personalization Services and furthermore provides management functionality of users and their saved configurations. The feature of configuring Personalization Services is illustrated in Figure 4. The user interface of the Agent Syndication Service is basically just rendering the *Configurable description* (cf. Section 2.1.1) of a Personalization Service and allows the user to configure such service by hand.

Users of the Personal Reader Agent additionally are enabled to

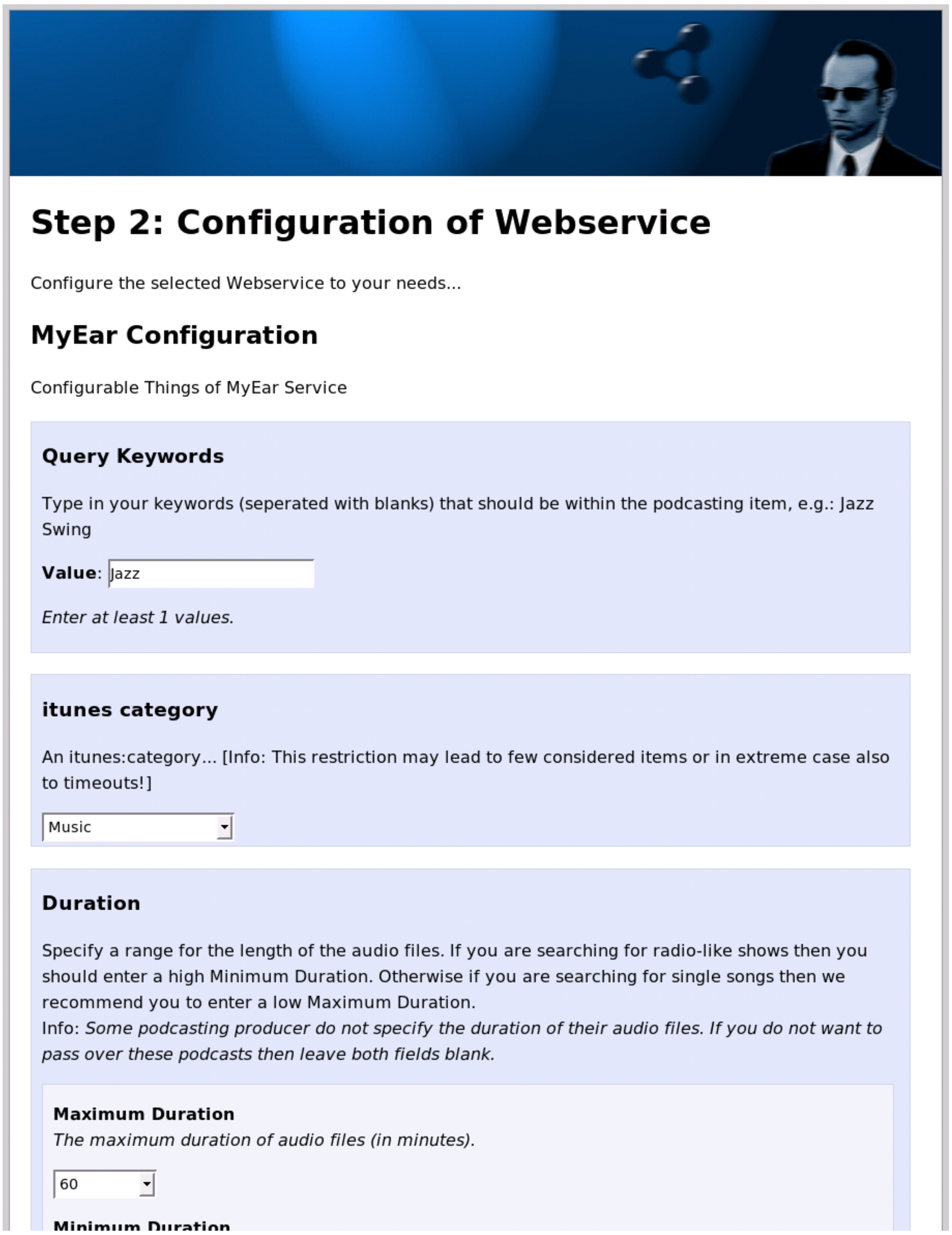
save configurations: as outlined in Section 2.1.1, the ontological model behind *saved configurations* is the concept of a `ConfiguredWebservice`. At this, users can specify `name`, `description` and `isPublic` on their own, `owlsURL` and `configurableURL` is set by the Agent and the `ListOfConfiguredValues` arises from the configuration step which is also performed by the users.

re-use their own configurations: in order to allow users a faster direct access to Personalization Services, they can call these services also with a *saved configuration*. Further the Agent provides some management functionality for configured services (*view*, *edit* and *delete*).

> **Relation within Researcher Ontology:** the Researcher Ontology defines persons and their involvements in working groups. If two persons (users) are involved in the same working group then the Agent suggests that configurations made by *User A* are also interesting for *User B*.
>
> **Relation within FOAF description:** FOAF defines among other things the relation *(Person A, knows, Person B)*. This relation can be used by the Agent to list configurations of other persons the user knows.

2.3 Connector Service

From a technical point of view, another component is required to maintain the communication between the Syndication Services providing the user interface, the Personalization Services, and the User Modeling Service, which is presented in Section 3. This is the so-called *Connector Service* (CService for short) which harvests Web service brokers, collects information about detected Personalization Services (for discovery, selection, customization, and invocation), and



Step 2: Configuration of Webservice

Configure the selected Webservice to your needs...

MyEar Configuration

Configurable Things of MyEar Service

Query Keywords

Type in your keywords (seperated with blanks) that should be within the podcasting item, e.g.: Jazz Swing

Value: Jazz

Enter at least 1 values.

itunes category

An itunes:category... [Info: This restriction may lead to few considered items or in extreme case also to timeouts!]

Music

Duration

Specify a range for the length of the audio files. If you are searching for radio-like shows then you should enter a high Minimum Duration. Otherwise if you are searching for single songs then we recommend you to enter a low Maximum Duration.
Info: *Some podcasting producer do not specify the duration of their audio files. If you do not want to pass over these podcasts then leave both fields blank.*

Maximum Duration
The maximum duration of audio files (in minutes).

60

Minimum Duration

Figure 4: Personal Reader Agent: Configuration of a Personalization Service.

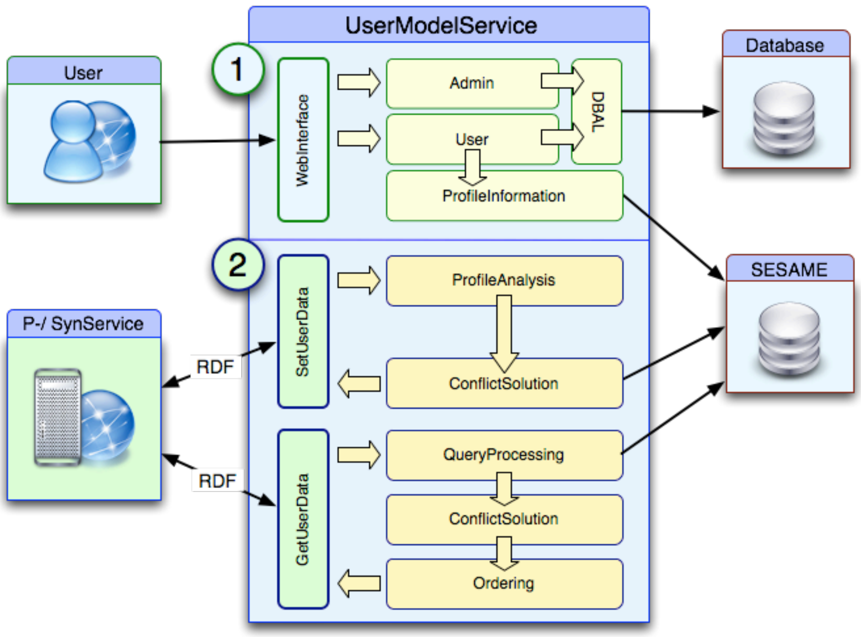


Figure 5: Architecture of the User Modeling Service.

for organizing the communication between all involved parties, including requests to the User Modeling Service.

The Connector Service decouples Personalization and Syndication Services. A service that needs certain functionality can query the Connector Service for a list of services, which may provide the requested functionality. Sophisticated service matchmaking is currently not implemented in the Connector Service and is part of our future work.

Another task of the Connector is to ensure security and privacy concerns at the service level, i.e. the Connector checks if a service caller is authorized to invoke a certain service. For example, the User Modeling Service, which provides access to user profiles can only be invoked by the Connector Service. Whenever another service wants to call the User Modeling Service then this has to be done via the Connector. The Connector Service checks signatures of the requests, which are sent by Personalization and Syndication Services, rejects malicious requests, signs trustworthy requests, and passes trustworthy requests to the service callee (e.g. to the User Modeling Service).

Another security mechanism that operates on the content level of user profiles – i.e. which service tries to access what information – is presented in Section 4.

3 User Modeling Service

For enabling the whole personalization process, a core information provider, which derives appropriate user profiles is essential. In our architecture, this is realized via a centralized *User Modeling Service*, which provides functionality for user modeling, maintaining and protecting information about a user on behalf of this user. The main reason for choosing a centralized

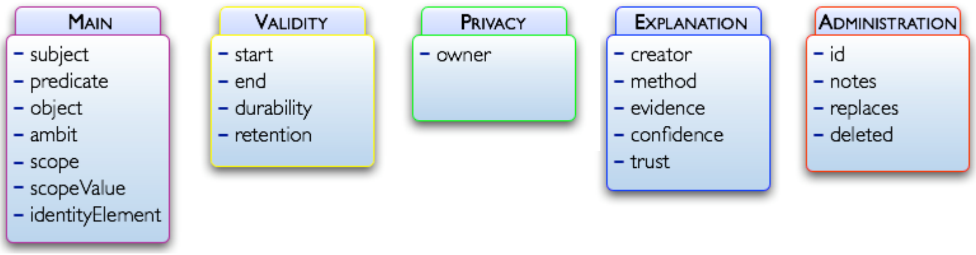


Figure 6: Main building blocks of *UserRDF* (properties of an UserRDF statement).

approach is to make user profile information re-usable for different applications. The central User Modeling Service receives updates from Syndication Services or Personalization Services in different domains, and allows for cross-domain re-use of user profile information (if the user wants that). The User Modeling Service stores and aggregates user profile data from various Personalization and Syndication Services. Because these services, which can be integrated at runtime and used immediately, aim on different domains, they also use different ontologies to express their knowledge about the user. For this reason we use RDF statements to store data domain-independently.

The architecture of the User Modeling Service is illustrated in Figure 5 and consists of two main independent components: (Figure 5.1) the Web interface and (Figure 5.2) the user modeling component.

The Web interface provides management functionalities for users and administrators. Administration functionality is focussed on management of users of the Personal Reader infrastructure. From the user perspective, the Web interface of the User Modeling Service is the place where users have to login in order to fully utilize Personal Reader applications (*single sign on*), because Personalization and Syndication Services only have a chance to access the user profile if the corresponding user is logged in. After having logged in successfully, a session ID is generated, which is valid globally and – cryptographically secured – passed to the applications and services the user employs.

The user modeling component, which is deployed as a SOAP-based Web service, provides two main methods.

setUserProfile This method allows to store User profile information – i.e. RDF statements about a user – in the user profile database. At the moment we use Sesame as RDF repository. User profile statements may be formulated using arbitrary vocabularies. However, in order to profit from the full power of the User Modeling Service (e.g. conflict resolution) *UserRDF*, which is introduced below, has to be used as user profile vocabulary.

getUserProfile The method `getUserProfile` enables other services to receive user profile information. Queries can be formulated in SeRQL [Broekstra and Kampman, 2004] or *UserQL*, which is a query language that is optimized for UserRDF statements.

3.1 UserRDF and UserQL

Within the Personal Reader Framework user profiles are represented via RDF statements. Regarding a certain statement about a user, there are different kinds of metadata, which are

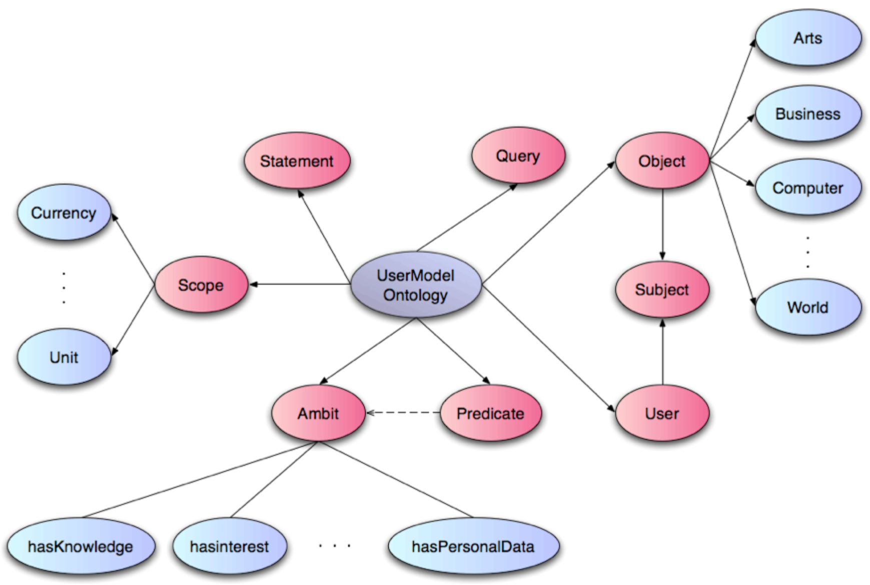


Figure 7: User Modeling Ontology.

also relevant to such a statement about a user, e.g., which service generated the statement, etc. Therefore, the User Modeling Service provides a format, which allows to enrich RDF statements with a well-defined set of metadata attributes and is called *UserRDF*. Furthermore, the user modeling component allows for advanced querying of such UserRDF statements via a corresponding lightweight query language, which is called *UserQL*.

Figure 6 lists the properties of an UserRDF statement, which is defined in the User Modeling Ontology (see *Statement* class within Figure 7), an ontology which is modeled on the General User Modeling Ontology (GUMO) [Heckmann et al., 2005]. The properties can be grouped into the following five different blocks.

Main. This building block contains the actual content of the statement, hence, the reificated RDF statement, which is embodied within *subject*, *predicate*, and *object*. The other attributes are used to characterize the predicate of the statement more precisely, e.g. *scope* is used to characterize the range of the property more precisely. The attribute *ambit* allows to categorize the UserRDF statement so that services that are not aware of the vocabulary, which is used to formulate the actual RDF statement, i.e., the ambit categorizes the property of the actual RDF statement. Ambits may be categories like *hasKnowledge*, *hasInterest*, etc. (cf. Figure 7).

Privacy. The security module provides just the property *owner*, which refers to the user to whom the statement belongs to[9].

Explanation. The attributes within the explanation building block can be exploited in order to describe who (*creator*) has generated the statement, how confident the creator is about

[9]Note that the subject of an UserRDF statement may be different from the user, e.g. if a statement about a user is composed of different statements.

the correctness of the statement (*confidence*), etc. The property *trust* can be adjusted by the user via the Web interface of the User Modeling Service and indicates the user's belief in the correctness of the statement.

Validity. The validity attributes allow to specify the expiration of the validity of a UserRDF statement.

Administration. An important aspect of the User Modeling Service is that UserRDF statements are never deleted. However, it is possible to mark a statement as *deleted* and to refer to the statement, which is replaced by a new statement (*replaces*).

In the following example, the statement about the user *Peter*, which was created by *WebServiceA*, is composed of three UserRDF statements. It says that *Peter* has a credit card, which itself has a number and certain type (*Creditcard*).

```
<rdf:RDF
  xmlns:ums="http://usermodelservice.org/#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

  <rdf:Description rdf:about="http://usermodelservice.org/Statement#1235">
    <rdf:type rdf:resource="http://usermodelservice.org/#Statement">
    <ums:subject rdf:resource="http://usermodelservice.org/User#Peter"/>
    <ums:predicate rdf:resource="http://example.org/predicate/#has"/>
    <ums:object rdf:resource="http://example.org/object/Creditcard#2345"/>
    <ums:ambit rdf:resource="http://usermodelservice.org/ambit/#hasPersonalData"/>
    <ums:owner rdf:resource="http://usermodelservice.org/User#Peter"/>
    <ums:creator rdf:resource="http://example.org#WebServiceA"/>
    <ums:confidence rdf:datatype="http://www.w3.org/2001/XMLSchema#int">100</ums:confidence>
  </rdf:Description>

  <rdf:Description rdf:about="http://usermodelservice.org/Statement#1236">
    <rdf:type rdf:resource="http://usermodelservice.org/#Statement">
    <ums:subject rdf:resource="http://example.org/object/Creditcard#2345"/>
    <ums:predicate rdf:resource="http://example.org/predicate/#hasNumber"/>
    <ums:object rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
      1234567
    </ums:object>
    <ums:owner rdf:resource="http://usermodelservice.org/User#Peter"/>
    <ums:creator rdf:resource="http://example.org#WebServiceA"/>
    <ums:confidence rdf:datatype="http://www.w3.org/2001/XMLSchema#int">100</ums:confidence>
  </rdf:Description>
  ...
  <rdf:Description rdf:about="http://example.org/object/Creditcard#2345">
    <rdf:type rdf:resource="http://example.org/object/#Creditcard">
  </rdf:Description>
</rdf:RDF>
```

Figure 8 lists the properties of an UserQL query, which is also defined in the User Modeling Ontology (see *Query* class within Figure 7). The properties can be grouped into the following three different blocks.

Match. This set of attributes corresponds to the building block *Main* of UserRDF. It is used to describe the kind of UserRDF statements that should be returned. Not specifying an attribute is equal to a wildcard.

Filter. The filter attributes can be applied in order to characterize the validity and confidence of the requested UserRDF statements.

Control. This building block contains properties that allow to order the resulting statements (*orderBy*), limit the amount of returned statements (*limitResult*), etc. The attribute

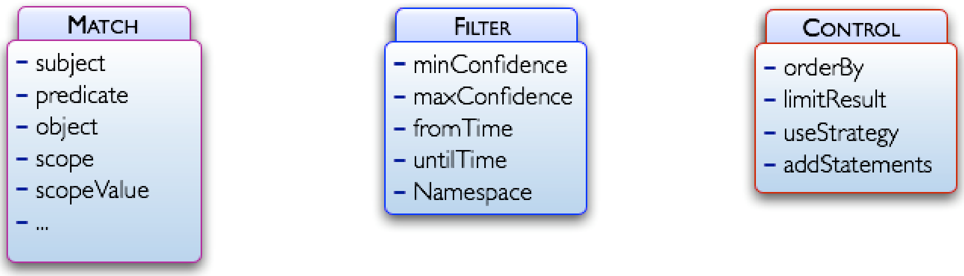


Figure 8: Main building blocks of *UserQL* (properties of an UserQL statement).

useStrategy enables the requester to specify what kind of conflict resolution strategy the User Modeling Service should use, e.g. *topKHighest* would return the top k (according to *limitResult*) statements, which have the highest confidence values.

The following UserQL example query would return all statements about the user *Peter*.

```
<rdf:RDF
  xmlns:ums="http://usermodelservice.org/#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Descriptionrdf:about="http://usermodelservice.org/Query#3456">
    <rdf:typerdf:resource="http://usermodelservice.org/#Query">
    <ums:subjectrdf:resource="http://usermodelservice.org/User#Peter"/>
  </rdf:Description>
</rdf:RDF>
```

Querying UserRDF statements can, of course, also be done via traditional RDF query languages like SPARQL or SeRQL as UserRDF statements conform to RDF. Protection of RDF statements about users is essential within the Personal Reader Framework, because user profiles are shared accross different applications. A fine-grained approach to control the access to user profile information is required. Section 4 describes the access control mechanism that is implemented as part of the Personal Reader Framework.

4 Access Control for User Profiles

Enabling behavior and content adaptation in distributed systems (like the Personal Reader infrastructure), in which users switch between different applications, requires the use of a shared user profile. Such a user profile is in charge of storing semantic data from different services, application domains, and users. As outlined in Section 3, RDF databases have been chosen to store these (meta)data, since they provide efficient access and high flexibility: arbitrary RDF data referring to various ontologies can be stored within the RDF database.

In service-oriented settings, which adhere to the Personal Reader architecture, different services may store or require sensitive data from the user profile the RDF repository. In our example (see figure 9) services store confidential contact information like email addresses or online e-commerce account information. It is crucial for the user to be able to specify which

```
<foaf:Person rdf:ID="Bob">
  <foaf:name>Bob</foaf:name>
  <foaf:mbox>bob@example.com</foaf:mbox>
  <foaf:phone>+49-511-123456</foaf:phone>
  <foaf:knows rdf:resource="#Alice"/>
</foaf:Person>

<foaf:Person rdf:ID="Alice">
  <foaf:name>Alice</foaf:name>
  <foaf:holdsAccount>
    <foaf:OnlineEcommerceAccount rdf:ID="aliceEC">
      <foaf:accountName>
        Bank Account of Alice
      </foaf:accountName>
    </foaf:OnlineEcommerceAccount>
  </foaf:holdsAccount>
  <foaf:currentProject
              rdf:resource="#rewerse"/>
  <foaf:knows rdf:resource="#Bob"/>
  <foaf:knows>
    <foaf:Person rdf:resource="#Bob">
      <foaf:phone>+49-511-654321</foaf:phone>
    </foaf:Person>
  </foaf:knows>
</foaf:Person>

<foaf:Project rdf:ID="rewerse">
  <foaf:theme rdf:resource="#SemanticWeb"/>
</foaf:Project>

<foaf:Document rdf:ID="SemanticWeb"/>
```

Figure 9: Example FOAF user profiles

(kind of) services are allowed to access and retrieve which part of the data stored in the user profile. For example, Alice must be able to allow a recommendation service to access information about her friends (linked via `foaf:knows`) but neither her private information (e.g., address or telephone number) nor the private information of her friends. Similarly, a means needs to be provided to allow Bob to restrict access to his online banking account data (`foaf:OnlineEcommerceAccount`) only to his banking service. Banking-related data may be defined as instances of a class `Banking` in some ontology, and the banking service may need to identify itself by providing some credential.

4.1 Related Approaches for RDF Access Control

An obvious approach, post-filtering results, which a query to a RDF database returns [Cozzi et al., 2006], is not an adequate solution, either: current RDF query languages allow to arbitrarily structure the results, as shown in the following example[10].

```
CONSTRUCT {CC} newNs:isOwnedBy {User}
  FROM {User} ex:hasCreditCard {CC};
              foaf:name {Name}
  WHERE Name = 'Alice'
```

Post-filtering the results of a query is hence not straightforward whenever their structure is not known in advance. It could be possible to break construct queries into a select query and the generation of the returned graph (construct), therefore avoiding this problem. However, the query response time may be too large since this approach cannot make use of repository optimizations and policies are enforced after all data (allowed and not allowed) has been retrieved. As an example, suppose an unauthorized requester submits a query asking for all available triples in the store. A post-filtering approach would retrieve all triples and then filter them all out. Especially for large result sets this would decrease performance dramatically.

Another way to address this problem is defining *a priori* which subsets of an RDF database can be accessed by some requester. For example, Named Graphs [Carroll et al., 2005] can be used to evaluate SPARQL queries [Prud'hommeaux and Seaborne, 2008]. [Dietzold and Auer, 2006] proposes a framework which first applies all rules to the whole RDF database and afterwards executes the query only on the subsets of allowed RDF triples. TriQL.P [Bizer and Oldakowski, 2004] allows the formulation of trust-policies in order to answer graph-based queries. Those queries describe conditions under which suiting data should be considered trustworthy. Access control based on identity could be performed if all requesters and their allowed graphs were known in advance.

However, this is not the case in our scenario presented above and since access to data may be granted or not depends on contextual conditions, these approaches do not apply. On the one hand, Named Graphs cannot be statically pre-computed for each possible combination of environmental factors, since their number would be too big; on the other, Named Graphs cannot be created at runtime, since the creation process would excessively slow down the response time. Furthermore, the plug-and-play nature of service-oriented architectures like the Personal Reader

[10]Our examples use SeRQL [Broekstra and Kampman, 2004] syntax (and for simplicity we do not include the namespace definitions), since SeRQL is the language we exploit in our implementation. However the ideas behind our solution are language-independent and can be applied to other RDF query languages.

Framework as well as the possibility that services dynamically change the RDF database itself by adding or removing data from the user profile would significantly complicate managing such named graphs.

[Reddivari et al., 2005] defines simple rule-based policies over the RDF database: such policies describe subgraphs on which actions like *read* and *update* can be executed: subgraphs are identified by specifying graph patterns. Some approaches also respect RDF Schema entailments [Jain and Farkas, 2006]. However, all these approaches require to instantiate the graph patterns, i.e., to generate one graph for each policy and execute the given query on each graph, hence leading to longer response times. Furthermore, these approaches cannot be applied to contextual queries either.

4.2 Access Control for RDF stores based on Query expansion

As shown in the previous section, existing work on RDF data protection does not suit the requirements of dynamic Semantic Web environments such as the Personal Reader Framework. Available solutions do not handle contextual information in a proper way, as they either require a large amount of memory or unacceptably increase the response time.

To address these problems we decided to enforce access control as a layer on top of RDF stores (also making our solution store independent). Our strategy is to pre-evaluate the contextual conditions of the policies, which do not depend on the content of the RDF store. Then, we expand the queries before they are sent to the database, therefore integrating the enforcement of the rest of (metadata) conditions with the query processing, thereby restricting the queries in such a way that they only utilize allowed RDF statements. This way, policies can hold a greater expressiveness and support both metadata and contextual conditions, while pushing part of its enforcement to the highly optimized query evaluation of the RDF store. In our scenario given a query, contextual conditions stated by the applicable policies are evaluated; if the evaluation succeeds, further constraints involving metadata (*graph pattern restrictions*) are conveniently translated and added to the query so that the underlying RDF store can enforce them in order to retrieve only allowed RDF statements. This approach allows to include more complex conditions without dramatically increasing the overhead produced by policy evaluation, and while relying on the underlying RDF store to evaluate RDF Schema capabilities (as discussed in [Jain and Farkas, 2006]).

4.2.1 RDF Queries

We assume disjoint, infinite sets I, B, and L, which denote IRIs, blank nodes and literals. In addition, let $Pred$, $Const$ and Var be mutually disjoint sets of predicates, constants and variables such that $Const = I \cup B \cup L$. Then (using similar notation as in [Polleres, 2007]) an RDF graph is a finite set of triples $I \cup B \times I \times Const$.

In the following we assume a query language with queries having the following structure (§6.19 in [Aduna, 2004])[11]:

```
SELECT/CONSTRUCT RF
  FROM PE
```

[11]Although our examples will use the syntax of the SeRQL query language, the results of this paper apply also to other languages with similar structure (e.g., SPARQL [Prud'hommeaux and Seaborne, 2008]).

```
WHERE BE
```

where

- RF is the result form, either a set of variables (projection in select queries) or a set of triples (construct clause in construct queries).
- PE is a path expression as defined below.
- BE is a boolean expression, that is, a string[12] representing a set of constraints in the form of (in)equality binary predicates and numerical operators such us greater than or lower than, connected by boolean connectives (AND and OR).

and a query will be denoted as $q = (RF, PE, BE)$. As today's established RDF query languages like SeRQL [Broekstra and Kampman, 2004] or SPARQL [Prud'hommeaux and Seaborne, 2008] do not support *insert* or *delete* operations yet, we focus on common *read* operations. An example query is provided in Figure 10. Without access control enforcement, this query would return an RDF graph containing all RDF triples matching the graph pattern defined in the FROM block, i.e., the query answer would include identifier and name of a person, her phone number(s) and the document(s) she is interested in.

We define a path expression as a triple (s, p, o) such that $s \in I \cup B \cup Var$, $p \in I \cup Var$ and $o \in Const \cup Var$. Hereafter we will use (s, p, o) and $triple(s, p, o)$ $(triple \in Pred)$ as synonyms. In addition, given an expression E (result form, path or boolean expression), we will denote by $vars(E)$ the set of all unbound variables occurring in E.

Definition – Introduction of new Variables: Given a path expression $e = (s, p, o)$ and a set of variable substitutions θ the function $disunify(e, \theta)$ returns the tuple (e', BE), where e' is a new pattern (s', p, o') and BE is a set of boolean expressions such that

- $$\begin{cases} s' = v_s \text{ and } be_s = (v_s = s) & \text{if } s \in I \cup B \\ s' = v_s \text{ and } be_s = (v_s = Value) & \text{if } s \in Var, [s = Value] \in \theta \\ s' = s \text{ and } be_s = \varepsilon & \text{otherwise} \end{cases}$$
- $$\begin{cases} o' = v_o \text{ and } be_o = (v_o = o) & \text{if } o \in Const \\ o' = v_o \text{ and } be_o = (v_o = Value) & \text{if } o \in Var, [o = Value] \in \theta \\ o' = o \text{ and } be_o = \varepsilon & \text{otherwise} \end{cases}$$

where v_s and v_o are fresh variables and $BE = \{be_s, be_o\}$. Intuitively, the variable substitutions for the subject and object of the pattern are extracted and converted into boolean expressions.

The purpose of this function is to extract variable substitutions in order to be able to reuse path expressions in the final RDF query, even if they are specified in different policies.

4.2.2 Specifying policies over RDF data

Using policies to restrict access to RDF statements requires to be able to specify graph patterns (path expressions and boolean expressions), such as one can do in an RDF query. In addition, it

[12]In the rest of the paper we also use BE to represent a set of boolean expressions. The exact meaning will be clear from the context.

```
CONSTRUCT * FROM
      {Person} foaf:name {Name};
               foaf:phone {Phone};
               foaf:interest {Document};
               foaf:holdsAccount {Account}
```

Figure 10: Example RDF query

is desired to have the ability of checking contextual properties such as the ones of the requester (possibly to be certified by credentials) or time (in case access is allowed only in a certain period of time). Therefore, we consider a policy rule *pol* to be a rule of the form

$$pred(triple(s,p,o)) \leftarrow$$
$$cp_1, \ldots, cp_n, pe_1, \ldots, pe_m, be_1, \ldots, be_p.$$

where $pred \in \{allow, disallow\}$, $triple(s,p,o)$ is a path expression as defined above, cp_i are contextual predicates (e.g., related to time, location, possession of credentials, etc.), pe_i are path expressions and be_i are boolean expressions. In the following we will refer to $H(pol)$ to the head of *pol*, $H^T(pol)$ to the triple in the head of *pol* and $B(pol)$ to the (possibly empty) body of *pol*.

Suppose that Alice specified the policies presented in Table 1 [13]. Instead of choosing a specific language, our policies are expressed in a high level syntax, which can be mapped to existing policy languages [14]. Their intended meaning is as follows:

1. anyone can receive Alice's phone number
2. the *RecommenderService* is not allowed to access the phone number(s) of members of the REWERSE project
3. recognized trusted services (which have to provide a suitable credential) are allowed to access the phone number(s) of people Alice knows
4. RDF statements containing *name* of entities different from Alice's boss Tom can be accessed during work time
5. this policy controls access to Alice's interests. Only interests related to her current project(s) can be accessed
6. a service can only access Alice's online ecommerce account if the service was invoked by a person which is known by Alice
7. only those services are allowed to access information about a person if they can supply the credential of this person

[13]Note that policies might also refer to named graphs, therefore allowing for approaches in which whole named graphs can be given access if the policy is satisfied.

[14]Although the final selection of the language will have an impact in the expressiveness and power of the kind of policies specified and contextual predicates supported

No.	Policy
pol_1	`allow access to`
	`triples (#alice, foaf:phone, Z).`
pol_2	`deny access to`
	`triples (X, foaf:phone, Z) IF`
	`(X, foaf:currentProject, #rewerse)  AND`
	`Requester = 'RecommenderService'.`
pol_3	`allow access to`
	`triples (X, foaf:phone, Z) IF`
	`Requester is certified by BBB AND`
	`(#alice, foaf:knows, X).`
pol_4	`allow access to`
	`triples (X, Y, Z)  IF`
	`Time is the current time AND`
	`09:00 < Time AND  Time < 17:00 AND`
	`Y = foaf:name AND X != #tom.`
pol_5	`allow access to`
	`triples (#alice, foaf:interest, Z) IF`
	`(Z, rdf:type, foaf:Document) AND`
	`(X, foaf:currentProject, P) AND`
	`(Z, foaf:topic, T) AND (P, foaf:theme, T).`
pol_6	`allow access to`
	`triples (#alice OnlineEcommerceAccount X) IF`
	`Invoker of Requester = Y AND`
	`(#alice foaf:knows Y).`
pol_7	`allow access to`
	`triples (X Y Z) IF`
	`(X rdfs:type foaf:Person) AND`
	`credential (Requester, C) AND`
	`issuer (C, X).`

Table 1: Example of high-level policies controlling access to RDF statements

Many algorithms could be exploited in order to evaluate policies and to handle conflicts which arise whenever two different policies allow and deny access to the same resource. However such algorithms are out of the scope of this paper. Therefore, in the following we assume a simple policy evaluation algorithm like the following one:

```
if a deny policy is applicable
    then access to the triple is denied
else if an allow policy is applicable
    then access to the triple is allowed
else access to the triple is denied
    (deny by default)
```

More advanced algorithms exploiting priorities or default precedences [Kagal et al., 2003] among policies could be used as well.

4.3 Policy Evaluation and Query Expansion

Given an RDF query, each RDF statement matching a pattern specified in the FROM block is accessed and, if the policies in force allow it, returned. Our approach consists of analyzing the set of RDF statements to be accessed and restricting it according to the policies in force. Contextual conditions (e.g., time constraints and conditions on properties of the requester) are evaluated by some policy engine, whereas other constraints are added to the given query and enforced during query processing.

To illustrate the algorithm step by step we consider the Sesame query defined in Figure 10.

Definition – Policy applicability: Given a path expression e, a set of policies P and a time-dependent state Σ [Bonatti and Olmedilla, 2005] (which in our case determines at each instant the extension of contextual predicates), we say that a policy $pol \in P$ is applicable to e (denoted by $\widehat{pol(e)}$) iff

- $\sigma' = mgu(e, H^T(pol))$, where mgu is the most general unifier
- $\exists \sigma, \sigma'' : \sigma = \sigma'\sigma'' \wedge \forall cp_i \in B(pol), P \cup \Sigma \models \sigma cp_i$
- if $\exists be_i \in B(pol) : \forall pe_i \in B(pol), vars(\sigma be_i) \cap (vars(\sigma pe_i) \cup vars(\sigma e)) = \emptyset \Rightarrow P \cup \Sigma \models \sigma be_i$

and its application is a function $e, pol \overset{P,\Sigma}{\longrightarrow} (PE, BE)$ such that for all pe_i, $disunify(pe_i, \theta) = (pe'_i, BE')$

- $PE = \{pe'_i | pe_i \in B(pol), pe'_i \neq pe_i\}$
- $\widetilde{BE} = \{\sigma be_i | be_i \in B(pol) \wedge \exists pe_i : vars(\sigma be_i) \cap (vars(\sigma pe_i) \cup vars(\sigma e)) \neq \emptyset\}$
- $BE = BE' \cup \widetilde{BE} \cup \{\sigma_i | \sigma_i = [X = Y] \wedge (X \in Const \vee Y \in Const)\}$

Intuitively, a policy pol is applicable to e if the triple the policy is protecting unifies with the path expression and all the contextual predicates and bound boolean expressions (or those not dependent of metadata expressions in the body of the policy) are satisfied. The return value is a set with the path expressions found in the body of the policy and all extracted boolean expressions which have not been evaluated and relate to the path expressions found.

Following our example query, assuming contextual predicates are satisfied, then pol_4 is applicable to $(Person, foaf : name, Name)$ and returns $(\emptyset, \{[Person! = \#tom]\})$. In addition, pol_1, pol_2 and pol_3 are applicable to $(Person, foaf : phone, Phone)$ and returns $(\{Var8, foaf : currentProject, Var9\}, \{[Var8 = Person], [Var9 = \#rewerse]\})$, $(\{(Var1, foaf : knows, Var2)\}, \{[Var1 = \#alice], [Var2 = Person]\})$ and $(\emptyset, [Person = \#alice])$ respectively.

Before we describe the query expansion algorithm, and for sake of clarity, we describe the conditions under a query does not need to be evaluated since the result is empty. Intuitively, a query fails if there does not exist any allowed triple to be returned according to both the query and the applicable policies, that is if there exists a path expression for which no allowed triples exist (disallow by default) or if there exist a path expression for which a policy (which does not depend on path expressions) specifies that no triple is allowed (explicit disallow).

Definition – Success of a Query: Given a query $q = (RF, PE, BE)$, a set of policy rules P and a state Σ, we say that q fails if either of the following two conditions hold:

- $\exists e \in PE : pol \in P, H(pol) = allow(T) \wedge \widehat{pol(e)}$
- $\exists e \in PE : \exists pol \in P, H(pol) = disallow(T) \wedge \widehat{pol(e)} \wedge e, pol \xrightarrow{P,\Sigma} (\emptyset, \emptyset)$

Let's denote by $append(BE, Conn)$ (resp. $prefix(BE, Conn)$) a function that given a set of boolean expressions BE and a connective (e.g., AND or OR) returns a new boolean expression in which all the elements of BE are enclosed by brackets and connected (resp. prefixed) by $Conn$. The pre-filtering algorithm is defined as follows:

```
Input:
    a query $q = (RF, PE, BE)$,
    a set of policy rules $P$ and a state $\Sigma$
Output:
    $PE^{+}_{new}$ ≡ new optional path expressions
              (from allow policies)
    $PE^{-}_{new}$ ≡ new optional path expressions
              (from disallow policies)
    $BE^{+}_{new}$ ≡ conjunction of boolean expressions
              (from allow policies)
    $BE^{-}_{new}$ ≡ conjunction of boolean expressions
              (from disallow policies)

$policy\_prefiltering(q, P)$:
    $BE^{+}_{or}$ ≡ disjunction of boolean expressions
              (from allow policies)
    $BE^{-}_{or}$ ≡ disjunction of boolean expressions
              (from disallow policies)
    $P_{app}$ ≡ a set of applicable policies

01) $PE^{+}_{new} = PE^{-}_{new} = \emptyset$
02) $\forall e \in PE$
03)     $BE^{+}_{or} = BE^{-}_{or} = \emptyset$
        // check allow policies
04)     $P_{app} = \{pol | pol \in P \wedge H(pol) = allow(T) \wedge \widehat{pol(e)}\}$
05)     if $P_{app} = \emptyset$
            return query failure // no triples matching $e$ are allowed
```

```
06)     if $\exists pol \in P_{app} : e, pol \xrightarrow{P,\Sigma} (\emptyset, \emptyset)$
            // all triples matching $e$ are allowed without restrictions
        else
07)         $\forall pol \in P_{app}$
                $e, pol \xrightarrow{P,\Sigma} (PE', BE')$
08)             if $PE' = \emptyset$
09)                 $BE^{+}_{or} \cup = append(BE', `AND')$
10)             else if $\exists \theta, \widetilde{PE} \in PE^{+}_{new} : \theta = mgu(\widetilde{PE}, PE')$
11)                 $BE^{+}_{or} \cup = append(\theta BE', `AND')$
                else
12)                 $PE^{+}_{new} \cup = PE'$
13)                 $BE^{+}_{or} \cup = append(BE', `AND')$
14)     $BE{+}_{new} \cup = append(BE^{+}_{or}, `OR')$
        // check disallow policies
15)     $P_{app} = \{pol | pol \in P \wedge H(pol) = disallow(T) \wedge \widehat{pol(e)}\}$
16)     if $\exists pol \in P_{app} : e, pol \xrightarrow{P,\Sigma} (\emptyset, \emptyset)$
            return query failure // all triples matching $e$ are denied
17)         $\forall pol \in P_{app}$
18)             $e, pol \xrightarrow{P,\Sigma} (PE', BE')$
19)             if $PE' = \emptyset$
20)                 $BE^{-}_{or} \cup = append(BE', `AND')$
21)             else if $\exists \theta, \widetilde{PE} \in PE^{-}_{new} : \theta = mgu(\widetilde{PE}, PE')$
22)                 $BE^{-}_{or} \cup = append(\theta BE', `AND')$
                else
23)                 $PE^{-}_{new} \cup = PE'$
24)                 $BE^{-}_{or} \cup = append(BE', `AND')$

25)     $BE^{-}_{new} \cup = append(BE^{-}_{or}, `OR')$
```

Let $copy(RF, PE)$ be a function that copies (replacing previous content) into RF either the variables (for SELECT queries) or the path expressions (for CONSTRUCT queries) from PE.

```
Input:
    a query $q = (RF, PE, BE)$
    $PE^{+}_{new} \equiv$ new optional path expressions
                (from allow policies)
    $PE^{-}_{new} \equiv$ new optional path expressions
                (from disallow policies)
    $BE^{+}_{new} \equiv$ conjunction of boolean expressions
                (from allow policies)
    $BE^{-}_{new} \equiv$ conjunction of boolean expressions
                (from disallow policies)
Output:
    an expanded query
    $q = (RF^{+}, PE^{+}, BE^{+})\ MINUS\ (RF^{-}, PE^{-}, BE^{-})$

$expandQuery(q, PE^{+}_{new}, PE^{-}_{new}, BE^{+}_{new}, BE^{-}_{new})$
    $RF^{+} = RF^{-} = copy(RF, PE)$
    $PE^{+} = PE \cup prefix(PE^{+}_{new}, `OPT')$
```

$PE^- = PE \cup prefix(PE^-_{new}, `OPT')$
$BE^+ = BE \cup append(BE^+_{new}, `AND')$
$BE^- = BE \cup append(BE^-_{new}, `AND')$

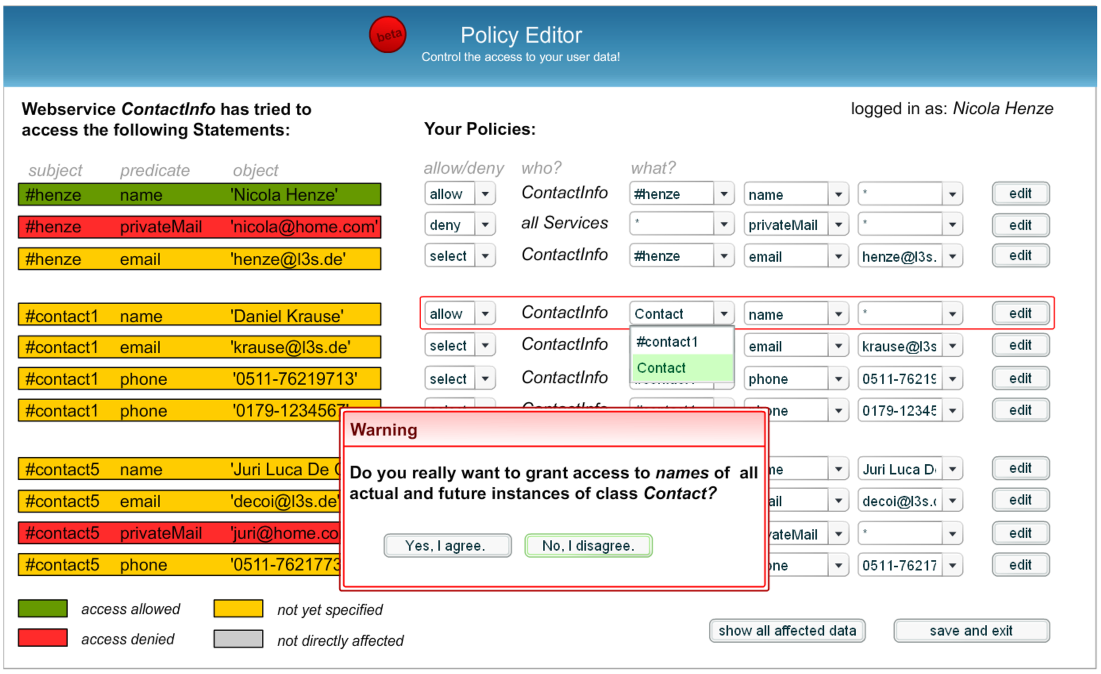
where the connective 'OPT' represents the "optional path expression" modifier in the chosen query language (e.g., '[' and ']' in SeRQL[Aduna, 2004]).

Briefly, the algorithm extracts the new path expressions found in the body of the policy rules. It extracts their variable bindings in order to reuse them in case they appear in more than one policy rule. However, if the same path expression is found in policies being applied to different from clauses, then they cannot be reused (since conditions on different expressions are connected conjunctively). After prefiltering each policy, a set of AND boolean expressions are extracted. The set of all boolean expressions from applicable allow policies to one from clause are connected by OR. The set of all boolean expressions applicable to different from clauses are connected by AND. From that query we have to remove the triples affected by disallow policies, which are specified in a similar fashion and added to the query using the MINUS operator.

The result of applying the above algorithm to the query in Figure 10 and the policies in Table 1 (assuming that time is 15:00, the requester is 'RecommenderService' and it is trusted) is shown in Figure 11.

```
  CONSTRUCT {Person} foaf:name {Name};
         foaf:phone {Phone};
         foaf:interest {Document}
         foaf:holdsAccount {Account}
  FROM {Person} foaf:name {Name};
         foaf:phone {Phone};
         foaf:interest {Document}
         foaf:holdsAccount {Account}
   [ {Var1} foaf:knows {Var2} ]
   [ {Var3} rdf:type {Var4},
     {Var3} foaf:topic {Var5},
     {Var6} foaf:currentProject {Var7},
     {Var7} foaf:theme {Var5} ]
  WHERE ( ( (Person != #tom) ) ) AND
        ( ( (Var2 = Person) AND
          (Var1 = #alice) ) OR
        ( (Person = #alice) ) ) AND
        ( ( (Var3 = Document) AND
          (Var2 = Person) AND
          (Person = #alice) AND
          (Var4 = foaf:Document) ) )
MINUS
  CONSTRUCT {Person} foaf:name {Name};
         foaf:phone {Phone};
         foaf:interest {Document}
         foaf:holdsAccount {Account}
  FROM {Person} foaf:name {Name};
     foaf:phone {Phone};
     foaf:interest {Document}
     foaf:holdsAccount {Account}
   [ {Var8} foaf:currentProject {Var9} ]
  WHERE ( ( (Var8 = Person) AND
          (Var9 = #rewerse) ) )
```

Figure 11: Expanded RDF query


Policy Editor

Control the access to your user data!

Webservice *ContactInfo* has tried to access the following Statements:

logged in as: *Nicola Henze*

subject	*predicate*	*object*
#henze	name	'Nicola Henze'
#henze	privateMail	'nicola@home.com'
#henze	email	'henze@l3s.de'
#contact1	name	'Daniel Krause'
#contact1	email	'krause@l3s.de'
#contact1	phone	'0511-76219713'
#contact1	phone	'0179-1234567'
#contact5	name	'Juri Luca De
#contact5	email	'decoi@l3s.de'
#contact5	privateMail	'juri@home.co
#contact5	phone	'0511-7621773

Your Policies:

allow/deny	*who?*	*what?*			
allow	*ContactInfo*	#henze	name	*	edit
deny	*all Services*	*	privateMail	*	edit
select	*ContactInfo*	#henze	email	henze@l3s.	edit
allow	*ContactInfo*	Contact	name	*	edit
select	*ContactInfo*	#contact1	email	krause@l3s	edit
select	*ContactInfo*	Contact	phone	0511-7621§	edit
			ne	0179-1234§	edit
			ne	Juri Luca D	edit
			ail	decoi@l3s.(	edit
			ateMail	*	edit
			ne	0511-76217	edit

Warning

Do you really want to grant access to *names* of all actual and future instances of class *Contact?*

🟩 *access allowed* 🟨 *not yet specified*

🟥 *access denied* ⬜ *not directly affected*

Figure 12: Policy Editor of the Personal Reader Framework (*basic view*).

4.4 Policy Editor of the Personal Reader Framework

The interface that enables users to specify access policies operates on top of the access control layer of the User Modeling Service as outlined in Figure 1 and is called *Policy Editor*. If a service attempts to access user data for which there are no access policies defined yet, then the operation of the service fails and the user is forwarded to the Policy Editor. The overview which is then presented to the user (see Figure 12) is adapted to the context of the failed operation. Such a context is given by the RDF statements which should be accessed by a given service. Thus, the overview is split into a part which outlines these RDF statements, and a part which allows the specification of corresponding access policies. RDF statements are colored according to the effects of policies (e.g. if a statement is not affected by any policy it may be colored yellow, green statements indicate that at least one service is allowed to access, etc.). Next to the visualization of these effects the overview additionally alerts conflicting policies by marking concerned policies and RDF statements.

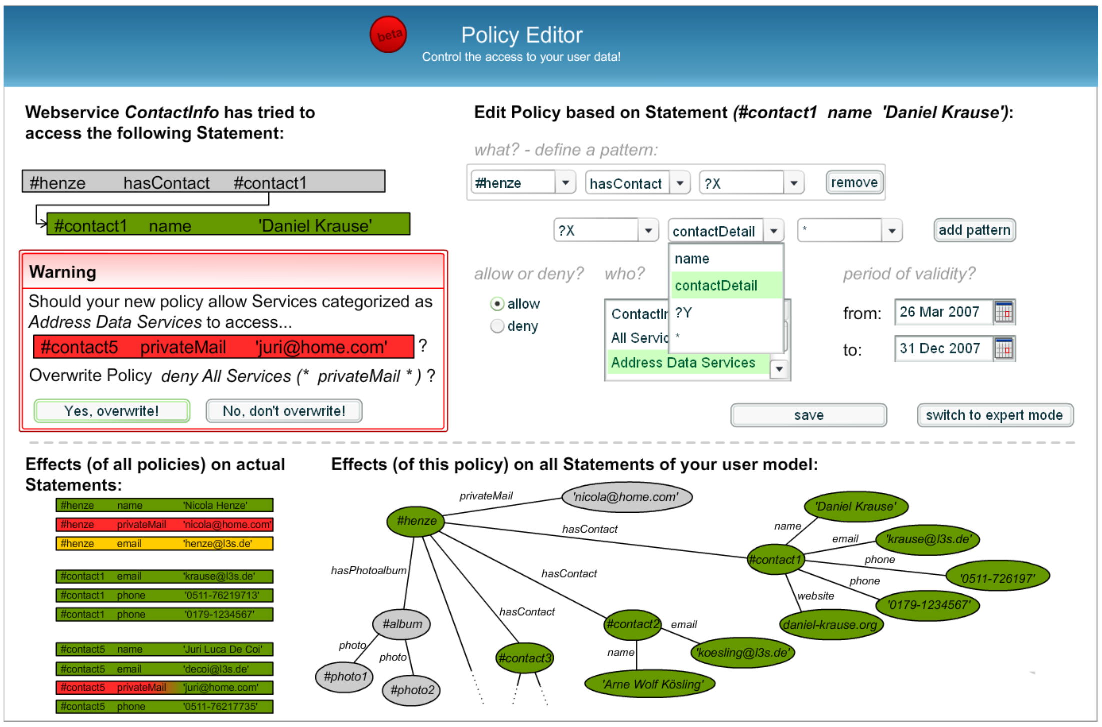
Warnings make the user aware of critical policies: In Figure 12 the user wants to allow the access to *names* of all instances of a class *Contact*. But as the user may not overlook which instances are added in the future, he is explicitly prompted for validation.

In general, policies are edited using a detailed perspective as illustrated in Figure 13. This perspective consists of two main fragments:

1. an arrangement of forms which allows the policy definition (top frame)

2. a view which clarifies the effects of the policy dynamically (bottom frame)

Users that do not use the *expert mode*, which would allow to specify Protune policies straight-


Policy Editor

Control the access to your user data!

Webservice *ContactInfo* has tried to access the following Statement:

#henze	hasContact	#contact1
#contact1	name	'Daniel Krause'

Warning

Should your new policy allow Services categorized as *Address Data Services* to access...

#contact5 privateMail 'juri@home.com' ?

Overwrite Policy *deny All Services (* privateMail *)* ?

Edit Policy based on Statement *(#contact1 name 'Daniel Krause')*:

what? - define a pattern:

#henze hasContact ?X

?X contactDetail *

allow or deny? [✓ allow] deny

who? Address Data Services

period of validity?

from: 26 Mar 2007

to: 31 Dec 2007

Effects (of all policies) on actual Statements:

#henze	name	'Nicola Henze'
#henze	privateMail	'nicola@home.com'
#henze	email	'henze@l3s.de'
#contact1	email	'krause@l3s.de'
#contact1	phone	'0511-76219713'
#contact1	phone	'0179-1234567'
#contact5	name	'Juri Luca De Coi'
#contact5	email	'decoi@l3s.de'
#contact5	privateMail	'juri@home.com'
#contact5	phone	'0511-76217735'

Effects (of this policy) on all Statements of your user model:

Figure 13: Policy Editor of the Personal Reader Framework (*detailed view*).

forward, just have to adjust a policy template on defining a concrete policy. This template consists of four components (see top right in Figure 13):

what The main task during creation of access policies is the specification of RDF graph patterns which identify statements that should be accessible or not. The predefined forms for defining these patterns are generated on basis of a partial RDF graph consisting of a certain RDF statement (here: *(#contact1, name, 'Daniel Krause')*) and its relation to the user *(#henze, hasContact, #contact1)*. To clarify this fact the RDF graph is presented to the user on the left hand.
To determine the options within the forms schema information of domain ontologies is utilized. In the given example the property *name* is part of the statement from which the forms are adapted. As *name* is a subproperty of *contactDetail* both appear within the opened combo box.
By clicking on *add pattern* or *remove* the user is enabled to add/remove RDF statement patterns to/from the overall graph pattern.

allow/deny The user can either allow or deny the access to RDF statements expressly.

who The policy has to be assigned to some services or category of services. For example to *ContactInfo*, the service trying to access user data, or to a category like *Address Data Services* with which *ContactInfo* is associated.

period of validity This parameter permits the temporal restriction of the policy.

According to Figure 13 the resulting Protune policy would be (without *period of validity*):

```
allow(access(rdfTriple(X, contactDetail, _))) :-
         requestingService(S),
         rdfTriple(S, memberOf, '#addressDataServices'),
         rdfTriple('#henze', hasContact, X).
```

Thus, *Address Data Services* are allowed to access all statements *(X, contactDetail, Y)* that match the RDF graph pattern *(#henze, hasContact, X), (X, contactDetail, Y)*. This policy overlaps with another policy that denies the access to statements of the form *(X, privateMail, Y)* wherefore a warning is presented to the user. This warning also lists the statements affected by this conflict: As *(#henze, privateMail, 'nicola@home.com')* does not suit, the pattern specified in Figure 13 *(#contact5, privateMail, 'juri@home.com')* is the only affected statement. By clicking on *"Yes, overwrite!"* the deny policy would be amended with the exception: *not rdfTriple(#contact5, privateMail, 'juri@home.com')*. Otherwise, by selecting *"No, don't overwrite!"* both policies would overlap. But as deny policies outrank allow policies (cf. previous sections) the affected statement would still be protected.

Next to such warnings the Policy Editor makes the user aware of how specified policies will influence the access to RDF statements. As *name*, *email*, etc. are subproperties of *contactDetail* the above policy permits access to a big part of the user's RDF graph which is consequently shown in green (see bottom of Figure 13).

5 Relevance of the Personal Reader Framework

The goal of the Personal Reader Framework is to ease and fasten implementation of Semantic Web applications, which provide personalization and user adaptation. The benefit of the framework is obvious if we look at the amount of applications that were developed in the period of the REWERSE project. Figure 14 illustrates when which of the various applications was mainly developed.

1. Framework Core (March 2004 - March 2005): Design and implementation of the core functionality of the Personal Reader Framework.

2. PReL (Java) (April 2004 - January 2005): Development of the Personal Reader for eLearning with a Java tutorial as example. While implementing the PReL some functionality was integrated into the Personal Reader Framework core.

3. PPR (August 2004 - May 2005): Implementation of the Personal Publication Reader [Baumgartner et al., 2005], which was awarded at the international Semantic Web Challenge 2005.

4. PReL (SemWeb) (February 2005 - April 2005): The development of the Personal Reader for learning Semantic Web topics was done much faster than the implementation of the *PReL (Java)* as functionality of the framework was re-used.

5. Semantic Portal (November 2004 - August 2005): For the implementation of the Semantic Portal for REWERSE (cf. [Abel and Henze, 2005]) some modules of the Personal Reader Framework could be utilized. As the Semantic Portal for REWERSE is itself not a *Personal Reader* it is marked with grey color.

6. PReL (September 2005 - November 2005): Quiz feature extension for the Personal Reader for eLearning.

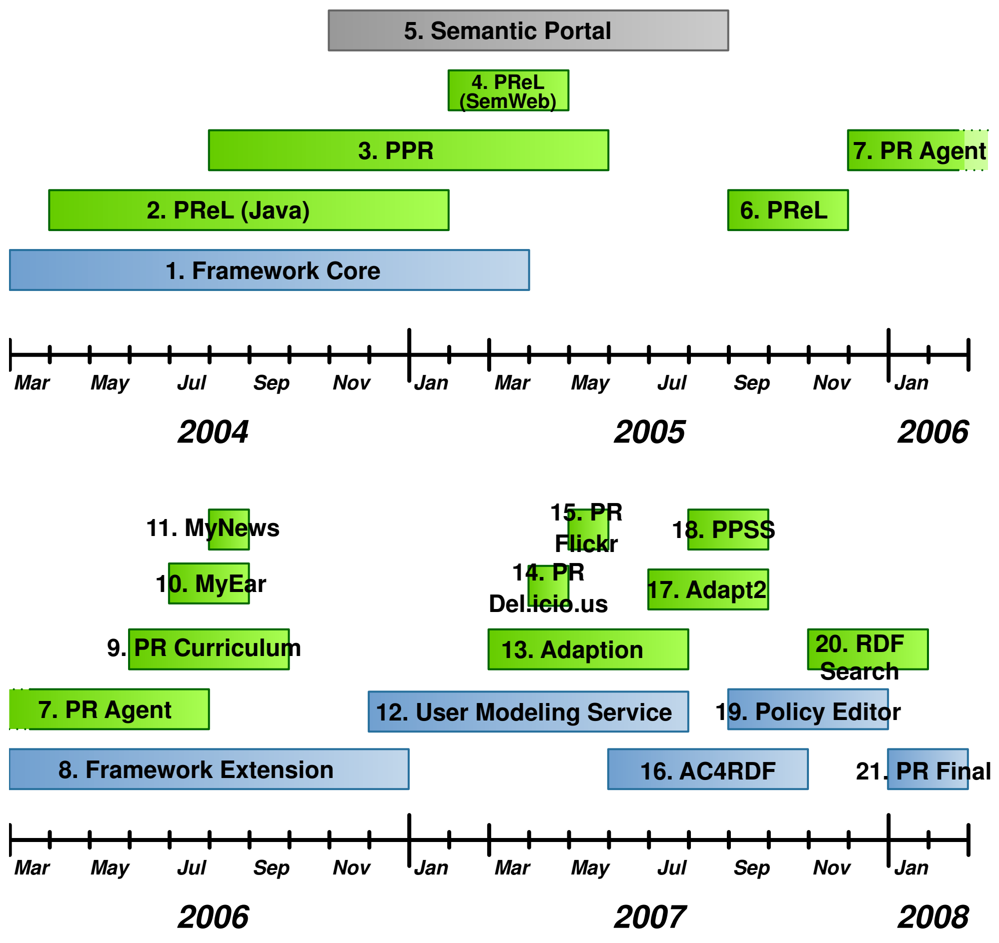


Figure 14: Development timeline – development of Personal Reader Framework (blue) and Personal Reader appllications (green).

7. PR Agent (December 2005 - July 2006): Design and implementation of the Personal Reader Agent [Abel et al., 2006].

8. Framework Extension (March 2006 - December 2006): The development of the Personal Reader Agent gained new features that were worth to be integrated also as features within the Personal Reader Framework. During the period of extending the Personal Reader Framework the core functionalities of the framework were renewed.

9. PR Curriculum (June 2006 - September 2006): Design and implementation of the Personal Curriculum Planner [Baldoni et al., 2006, Baldoni and Marengo, 2007], which is a service-oriented personalization system, set in an educational framework, based on a semantic annotation of courses, given at knowledge level.

10. MyEar (July 2006 - August 2006): The development of the MyEar Music Recommender

[Abel et al., 2006] also gained a generic Personalization Service for personalizing RSS feeds.

11. MyNews (August 2006): The generic Personalization Service for personalizing RSS feeds was re-used in order to realize MyNews, which enables users to browse or subscribe to personalized news feeds.
12. User Modeling Service (December 2006 - July 2007): Design and implementation of the User Modeling Service for the Personal Reader Framework.
13. Adaption (March 2007 - July 2007): Adaptation of Personal Reader applications was necessary in order to benefit from the new User Modeling Service – before, user profiles were stored in a shared RDF repository.
14. PR Deli.icio.us (April 2007): The Personal Reader for Del.icio.us[15] bookmarks re-used Personalization Services, which were originally developed in the context of the Personal Publication Reader and the MyEar Music Recommender. Therewith, the time for developing PR Del.icio.us was minimized.
15. PR Flickr (May 2007): The Personal Reader for Flickr[16] benefited from existing Personalization Services and the Framework infrastructure as well as the Personal Reader for Del.icio.us.
16. AC4RDF (June 2007 - October 2007): The Access Control for RDF stores (AC4RDF) mechanism, which was presented in Section 4, was developed in collaboration with working group I2 during the last period of the REWERSE project.
17. Adapt2 (July 2007 - September 2007): A Personal Reader, which connects the Personal Reader infrastructure with the *Advanced Distributed Architecture for Personalized Teaching & Training* (ADAPT2) [Brusilovsky et al., 2005], which aims at providing personalization and adaptation services for developers of otherwise not personalized content, was developed in Summer 2007. The Personal Reader for Adapt2 made use of different Personalization Services that were already existent within the Personal Reader environment, e.g. the User Mapping Service that was originally implemented for the Personal Publication Reader.
18. PPSS (August 2007 - September 2007): Development of the Personalized Preference Search Service[17] (PPSS) [Kärger et al., 2007] in collaboration with working group I2.
19. Policy Editor (September 2007 - December 2007): Design and implementation of the Policy Editor, which was presented in Section 4.4.
20. RDF Search (November 2007 - January 2008): Design and implementation of an RDF (Meta) Search Engine, which extends Sindice[18] [Tummarello et al., 2007] and other RDF search engines like Watson[19]. The RDF (Meta) Search Engine was – as all application listed in Figure 14 – realized by aid of the Personal Reader Framework and by utilizing the generic Personalization Service for personalizing RSS feeds.

[15]http://del.icio.us
[16]http://flickr.com
[17]http://www.personal-reader.de/PreferenceQueryGUI
[18]http://sindice.com
[19]http://watson.kmi.open.ac.uk

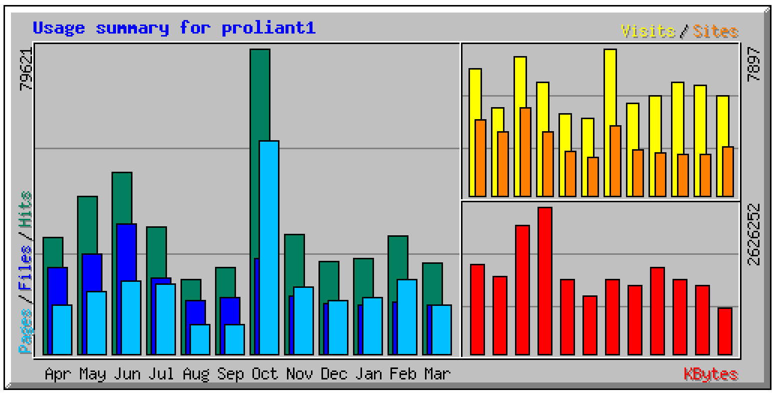


Figure 15: Web usage statistics of Personal Reader website from April 2007 till March 2008.

21. PR Final (January 2008 - February 2008): Finilizing of the Personal Reader Framework.

As indicated in Figure 14 the average time for developing Personal Reader applications, thus Semantic Web applications that provide personalization functionalities in order to enhance exploration and reading of Web content, decreased significantly during the REWERSE project. The reason for this decrease of development time is the Personal Reader Framework itself, which reduces programming efforts, and the set of existing Personalization Services, which could be re-used by different Personal Reader applications as mentioned above.

Figure 15 shows the access statistics for the website of the Personal Reader framework[20]. This website promotes the framework itself and the various Personal Reader applications. The daily average of more than 180 visits (in the last year of the REWERSE project) illustrates that people are interested in the Personal Reader applications and the Personal Reader Framework. The big access increase in October can be partially explained by the International Semantic Web Conference 2007[21], where the Personal Reader team presented a paper about access control in RDF stores [Abel et al., 2007]. Detailed access statistics of the Personal Reader website can be gathered from: http://www.personal-reader.de/webalizer/.

6 Conclusion

In this deliverable we describe the Personal Reader Framework, which enables design, implementation, and maintenance of personalized information systems. We focus especially on the user modeling part and presented an advanced mechanism to control access to RDF-based user profiles. Furthermore, we list the various applications, which were realized by aid of the Personal Reader Framework, and show how the development time of such applications decreased during the REWERSE project as the framework provided more and more features, and the amount of re-usable services increased.

[20]http://www.personal-reader.de
[21]http://www.iswc07.org

Acknowledgement

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

References

[Abel et al., 2006] Abel, F., Brunkhorst, I., Henze, N., Krause, D., Mushtaq, K., Nasirifar, P., and Tomaschweski, K. (2006). Personal reader agent: Personalized access to configurable web services. Technical report, Distributed Systems Institute, Semantic Web Group, University of Hannover.

[Abel et al., 2007] Abel, F., De Coi, J. L., Henze, N., Koesling, A. W., Krause, D., and Olmedilla, D. (2007). Enabling advanced and context-dependent access control in RDF stores. In Aberer, K., Choi, K.-S., and et al., N. N., editors, *International Semantic Web Conference 2007 (ISWC 2007) (to appear)*, Busan, Korea.

[Abel and Henze, 2005] Abel, F. and Henze, N. (2005). User Awareness and Personalization in Semantic Portals. In *Proceedings of 4th International Semantic Web Conference, Galway, Ireland (6th–10th November 2005).* Digital Enterprise Research Institute.

[Aduna, 2004] Aduna, B. (2004). The SeRQL query language (revision 1.2). `http://www.openrdf.org/doc/sesame/users/ch06.html`.

[Apple Computer, Inc., 2006] Apple Computer, Inc. (2006). Podcasting and Itunes: Technical Specification. `http://www.apple.com/itunes/podcasts/techspecs.html`.

[Baldoni et al., 2006] Baldoni, M., Baroglio, C., Brunkhorst, I., Henze, N., Marengo, E., and Patti, V. (2006). A Personalization Service for Curriculum Planning. In Herder, E. and Heckmann, D., editors, *Proc. of the 14th Workshop on Adaptivity and User Modeling in Interactive Systems, ABIS 2006*, pages 17–20, Hildesheim, Germany.

[Baldoni and Marengo, 2007] Baldoni, M. and Marengo, E. (2007). Curriculum Model Checking: Declarative Representation and Verification of Properties. In Duval, E. and Klamma, R., editors, *Proc. of EC-TEL 2007 - Second European Conference on Technology Enhanced Learning*, LNCS. Springer.

[Baumgartner et al., 2005] Baumgartner, R., Henze, N., and Herzog, M. (2005). The personal publication reader: Illustrating web data extraction, personalization and reasoning for the semantic web. In *ESWC*, pages 515–530.

[Bizer and Oldakowski, 2004] Bizer, C. and Oldakowski, R. (2004). Using context- and content-based trust policies on the Semantic Web. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 228–229, New York, NY, USA. ACM Press.

[Bonatti and Olmedilla, 2005] Bonatti, P. A. and Olmedilla, D. (2005). Driving and monitoring provisional trust negotiation with metapolicies. In *6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*, pages 14–23, Stockholm, Sweden. IEEE Computer Society.

[Broekstra and Kampman, 2004] Broekstra, J. and Kampman, A. (2004). SeRQL: An RDF query and transformation language.

[Brusilovsky et al., 2005] Brusilovsky, P., Sosnovsky, S., and Yudelson, M. (2005). Ontology-based framework for user model interoperability in distributed learning environments. In *E-Learn 2005*.

[Carroll et al., 2005] Carroll, J. J., Bizer, C., Hayes, P., and Stickler, P. (2005). Named graphs, provenance and trust. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 613–622, New York, NY, USA. ACM Press.

[Cozzi et al., 2006] Cozzi, A., Farrell, S., Lau, T., Smith, B. A., Drews, C., Lin, J., Stachel, B., and Moran, T. P. (2006). Activity management as a web service. *IBM Systems Journal*, 45(4):695–712.

[Dietzold and Auer, 2006] Dietzold, S. and Auer, S. (2006). Access control on rdf triple stores from a semantic wiki perspective. In *Scripting for the Semantic Web Workshop at 3rd European Semantic Web Conference (ESWC)*.

[Heckmann et al., 2005] Heckmann, D., Schwartz, T., Brandherm, B., Schmitz, M., and von Wilamowitz-Moellendorff, M. (2005). Gumo - the general user model ontology. In Ardissono, L., Brna, P., and Mitrovic, A., editors, *User Modeling*, volume 3538 of *Lecture Notes in Computer Science*, pages 428–432. Springer.

[Henze, 2005] Henze, N. (2005). Personalization services for the semantic web: The personal reader framework. In *Framework 6 Project Collaboration for the Future Semantic Web Workshop at European Semantic Web Conference ESWC 2005*, Heraklion, Greece.

[Jain and Farkas, 2006] Jain, A. and Farkas, C. (2006). Secure resource description framework: an access control model. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 121–129, New York, NY, USA. ACM Press.

[Kagal et al., 2003] Kagal, L., Finin, T. W., and Joshi, A. (2003). A policy language for a pervasive computing environment. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003), 4-6 June 2003, Lake Como, Italy*, pages 63–. IEEE Computer Society.

[Kärger et al., 2007] Kärger, P., Abel, F., Herder, E., Olmedilla, D., and Siberski, W. (2007). Exploiting preference queries for searching learning resources. In Duval, E., Klamma, R., and Wolpers, M., editors, *EC-TEL*, volume 4753 of *Lecture Notes in Computer Science*, pages 143–157. Springer.

[OWL-S, 2004] OWL-S (2004). OWL-S: Web Ontology Language for Services, W3C Submission. http://www.org/Submission/2004/07/.

[Polleres, 2007] Polleres, A. (2007). From SPARQL to rules (and back). In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 787–796, New York, NY, USA. ACM Press.

[Prud'hommeaux and Seaborne, 2008] Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`.

[Reddivari et al., 2005] Reddivari, P., Finin, T., and Joshi, A. (2005). Policy based access control for a RDF store. In *Proceedings of the Policy Management for the Web Workshop*, A WWW 2005 Workshop, pages 78–83. W3C.

[RSS Advisory Board, 2002] RSS Advisory Board (2002). Really Simple Syndication 2.0 specification. `http://www.rssboard.org/rss-2-0/`.

[Tummarello et al., 2007] Tummarello, G., Oren, E., and Delbru, R. (2007). Sindice.com: Weaving the open linked data. In Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L. J. B., Golbeck, J., Mika, P., Maynard, D., Schreiber, G., and Cudr-Mauroux, P., editors, *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea*, volume 4825 of *LNCS*, pages 547–560, Berlin, Heidelberg. Springer Verlag.