A USER INTERFACE TO DEFINE AND ADJUST POLICIES FOR DYNAMIC USER MODELS

Fabian Abel[1], Juri Luca De Coi[2], Nicola Henze[1], Arne Wolf Koesling[3], Daniel Krause[1] and Daniel Olmedilla[4]

[1]*Distributed Systems Institute/KBS, University of Hannover, D-30167 Hannover, Germany*

[2]*L3S Research Center, University of Hannover, D-30167 Hannover, Germany*

[3]*Peter L. Reichertz Institute for Medical Informatics, Hannover Medical School, D-30625 Hannover, Germany*

[4]*Telefonica Research & Development, 28043 Madrid, Spain*

Keywords: Policy, RDF, Access Control, User Interface.

Abstract: A fine-grained user-aware access control to user profile data is the key requirement for sharing user profiles among applications, and hence improving the effort of these systems massively. Policy languages like Protune can handle access restrictions very well but are too complicated to be used by non-experts. In this paper, we identify policy templates and embed them into a user interface that enables users to specify powerful access policies and makes them aware of the current and future consequences of their policies.

1 INTRODUCTION

Personalization encounters more and more attention as it promises to make programs more user friendly and content more appropriate. This information ideally should be learned by the system in interaction with the user and stored in a user profile. While user profiles that fit for one personalization component and that are implemented in one application have proved to work well, the benefit of a user profile will even increase if different applications can share its data. Especially in a service-based environment, where users invoke many different services and use them only for a short period of time, it is nearly impossible for a single service to generate such a user profile on its own as there is not enough interaction with the user.

Shared user profiles enable applications to utilize additional information that has been collected by other applications and therefore need a universal data storage format. The Resource Description Framework (RDF) provides such a generic format and hence, is used in the User Modeling Service (UMService) which is part of the Personal Reader Framework, a framework allowing the creation of web service-based applications. A serious problem of shared user profiles is that applications store sensitive information: While a trustful application known by the user is maybe allowed to access his bank account information, another application should not be allowed to access the same data. Therefore, an access control system is required that grants applications access to profile data only after the user has agreed. For this access control system, rule-based policy languages can be used very well as they allow precisely to specify which application can operate on which data at which time.

Rule-based policy languages, like Protune (Bonatti and Olmedilla, 2005a; Bonatti and Olmedilla, 2005b), can be used to define in a fine-grained fashion which web service is allowed to access which data, which credentials it has to provide and so on. In the domain of user profile data this problem is shifted to the task of selecting data from an RDF graph protected by an access policy. Policy languages can deal with this task but the resulting policies are usually too complicated to be specified by non-experts. A possible approach is to use predefined policy templates that can be completed easily by users. However, users cannot be fully aware about which data is covered by a policy if they do not have an appropriate user interface.

The contribution of this paper is to present a user interface that enables non-expert users to control the access to their RDF-based user profiles. Hence, we deduce access policy templates and embed them into the user interface. Then, we examine how to implement these policies in the Protune policy language. The user interface provides immediate feedback to the

Abel F., De Coi J., Henze N., Koesling A., Krause D. and Olmedilla D.
A USER INTERFACE TO DEFINE AND ADJUST POLICIES FOR DYNAMIC USER MODELS.
DOI: 10.5220/0001839801840191
In *Proceedings of the Fifth International Conference on Web Information Systems and Technologies (WEBIST 2009)*, page
ISBN: 978-989-8111-81-4

user which includes information about which part of the RDF data is covered by the policy and additionally which consequences the specified policy has.

The paper is structured as follows: In Section 2 we present underlying techniques like the Personal Reader Framework, the User Modeling Service and policies. Section 3 introduces the usage of policies for protecting user profiles. A user interface that enables users to specify these user profile policies is shown in Section 4 and 5. The related work can be found in Section 6. We conclude the paper and give an outlook to future work in Section 7.

2 THE PERSONAL READER FRAMEWORK

The *Personal Reader Framework* (Henze, 2005) allows the creation of modular web service-based applications (Figure 1). These applications are accessed by user interfaces (UI for short). *Syndication Services* implement the application logic and can be considered as the core of an application. By means of a *Connector Service* all 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*. Several applications have been implemented with the Personal Reader Framework like the *Personal Publication Reader* (Abel et al., 2005), the *MyEar* music recommender (Henze and Krause, 2006) or the *Agent*[1].

2.1 The User Modeling Service

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. A statement contains the user as subject, a predicate from the service ontology and a value as object. Objects can also be more complicated and further RDF statements can be used to describe the objects in a more detailed way as outlined in the example scenario in section 2.2.1.

2.1.1 Access Control Layer

The access control layer of the User Modeling Service has to restrict the access to the data stored in the User Modeling Service. Therefore, a user should specify which web services are allowed to access which kind of data in the user profile and in which way. The environment of the access control layer is similar to a firewall: whenever an application tries to access a specific port, if an access rule for such application and port has been specified, the specified action (`allow` or `deny`) is performed. Otherwise the firewall asks the user how to behave. The firewall is at no time aware of which applications or ports exist in a system.

Similarly, as the framework allows to plugin new services immediately, the access control layer is not aware of which services will try to access which part of the user profile. Hence, specifying static access rules a priori like in other access control systems is not applicable.

Our access control layer solves this issue by a deny-by-default behavior. Every service that tries to access an RDF statement is rejected if no existing policy is applicable. The service is informed why it was rejected and will report this to the user. Afterwards, the user can enter the user interface of the access control layer to grant or deny access. The user interface can take the context into account, which contains the statements a service tried to access, and hence supports the user in specifying policies by reducing the choices to the affected statements. By allowing users to specify also general policies we try to avoid that the user is overwhelmed by too much interaction with the access control layer. Keeping user interaction low enhances usability and at the same time avoids that users ignore repeatedly displayed confirm messages. In the rest of the paper, we focus on granting read access. A similar approach can be used for write access requests.

2.2 Policies for Securing Data

Securing RDF data is different from securing usual datasets. Because RDF datasets can be considered as graphs we take into account this graph structure in order to provide a definition of “security”.

There are many possibilities to secure the data in the user profile, like black- or whitelisting of services

[1] http://www.personal-reader.de/Agent

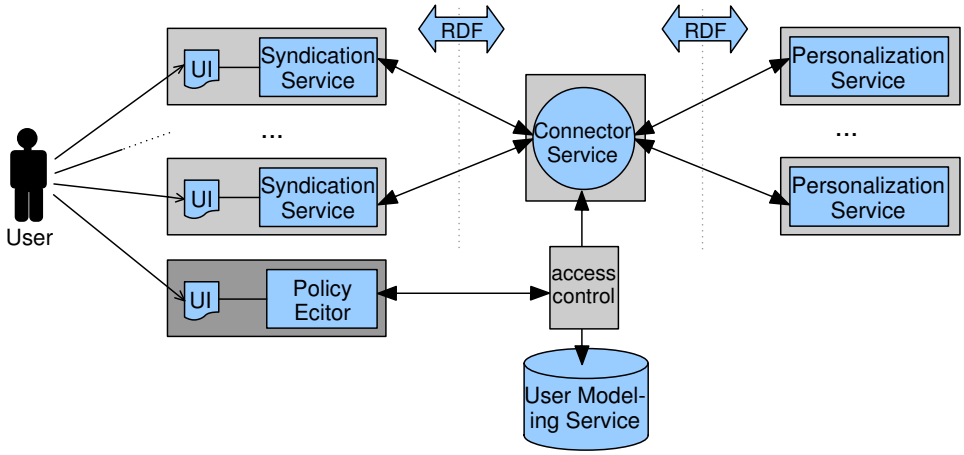

Figure 1: Personal Reader Architecture.

S_1: (John, phoneNumber, 123)
S_2: (John, hasFriend, Friend_1)
S_3: (Friend_1, phoneNumber, 234)
...
S_{m-4}: (John, hasFriend, $Friend_n$)
S_{m-3}: ($Friend_n$, phoneNumber, 345)
S_{m-2}: (John, hasFriend, Mary)
S_{m-1}: (Mary, phoneNumber, 456)
S_m : (John, loves, Mary)

Figure 2: John's user profile.

for specific RDF statements by means of access control lists. We do not want to mark resources as "accessible" or not in an automatic way, because the user should keep full control on which resources (not) to grant access for. But we also want to relieve the user from marking each resource individually, so we need a more flexible solution. We think that policies provide such a flexible solution. In the following we examine how Protune policies can be applied to RDF statements and graphs.

2.2.1 Scenario 1 - Sharing Telephone Number Information

Let assume that John's user profile contains the RDF statements illustrated in Figure 2.
John may want to make the phone numbers of his friends publicly available, but may want to hide statement S_m or maybe even statement S_{m-1}. The policy language Protune allows to define policies protecting such statements.

3 PROTUNE POLICY TEMPLATES FOR A USER MODELING SERVICE

The policies we need must be able to specify in a declarative way the prerequisites a service has to fulfill in order to access some resource. The policy language Protune (Bonatti and Olmedilla, 2005a; Bonatti and Olmedilla, 2005b) allows to formulate a broad range of policies like access control policies, privacy policies, reputation-based policies, provisional policies, and business rules. Some languages, like e.g. KAoS or Rei (Uszok et al., 2003; Kagal et al., 2003) adopt Description Logics (Baader et al., 2003) as underlying formalism, whereas Protune, which extends PAPL (Bonatti and Samarati, 2000) and PeerTrust (Gavriloaie et al., 2004), exploits Logic Programming and its SLDNF-based reasoning mechanism (Lloyd, 1987).

One of the main differences between Description Logics-based (DL-based) and Logic Programming-based (LP-based) policy languages can be found in the way they deal with negation: Description Logics allow to define negative information explicitly, whereas LP-based systems can deduce negative information by means of the so-called *negation as failure* inference rule. Whilst consistency problems may arise in DL-based systems (since both a statement and its negation can be asserted), LP-based systems do not have to deal with consistency issues, since they only allow the user to specify positive statements. LP-based policy languages like Protune may decide whether the user should only specify allow policies (thereby relying on the negation-as-failure inference rule) or the other way around. The first approach is usually preferred, since wrongly disclosing private in-

formation is a more serious issue than not disclosing information that should be publicly available.

In our framework we need both usual deny policies and deny-by-default policies: If a deny-by-default policy applies, the user is directed to the user interface to specify new policies, if a usual deny policy occurs the user is not informed since he already defined a policy. This feature allows us to implement in a very clean way the algorithm to be executed by the access control component (cf. figure 1), namely

```
if(a deny policy is defined) deny access
  else
   if(an allow policy is defined) allow access
    else
     deny access and ask the user
```

The access control component checks first whether a deny policy is applicable to the current access request and, if it is the case, denies access. If not, the system checks whether an allow policy is applicable. If this is not the case, access is denied and a message is sent to the user.

The following Protune policy applies to the RDF statements example given in the previous chapter. Its intended meaning is to allow services that belong to the user-defined group *trustedServices* to access the telephone numbers of John's friends, except Mary's number.

```
allow(access(rdfTriple(Y, phoneNumber, X))) :-
  requestingService(S),
  rdfTriple(S, memberOf, '#trustedServices'),
  rdfTriple('#john', hasFriend, Y),
  not Y = '#mary'.
```

Predicate *rdfTriple* retrieves RDF triples from some RDF repository, whereas predicate *requestingService* accesses runtime data in order to retrieve the value of the current requesting service. The rule the policy consists of can be read as a rule of a Logic Program, i.e., *allow(access(...))* is satisfied if predicate *requestingService*, all literals *rdfTriple* and the inequality are satisfied. Predicates which represent an action (i.e., *requestingService* and *rdfTriple*) are supposed to be satisfied if the action they represent has been successfully executed. The policy can therefore be read as follows: access to RDF triple *(Y, phoneNumber, X)* is allowed if the current requesting service (*S*) belongs to *trustedServices* and *X* is the phone number of someone who is a friend of John different than *Mary*.

3.1 Policy Templates for an RDF based User Profile

Since expressive policies become quickly hard to read for non-technical users we defined some general purpose policies in so-called templates.

3.1.1 Templates

Policy types can be defined in several ways:

1. One may group targets (in our case RDF statements or parts of them), so that the user is enabled to state, *what* triples should be accessible. Examples for such a group of targeted RDF statements are:
   - allow access to some specific phone numbers
   - allow access only to my own phone number
   - allow access only to my friends' phone numbers
2. Policies may also be grouped according to the requester, so that the user is enabled to state *who* gets access to the triples (i.e. allow access for one service or a specific group/category of services).

Protune policies allow the usage of both kind of policy types to protect specific RDF statements, a specific group of statements or, in general, an arbitrary part of an RDF graph. So, it is possible to

- specify RDF-predicates anywhere used in the user profile to be secured by a policy
- specify RDF-object/RDF-subject types anywhere used in the user profile
- specify RDF statements that contain information directly related to the user, like *(John, loves, Mary)*, and not just information indirectly related to the user, like *(Friend$_x$, phoneNumber, xyz)*
- specify meta-data predicates like requester or current time

Our user interface allows to define policies protecting *RDF graph patterns*. When defining a policy the user must instantiate such patterns and adapt them to the given context (see Figure 4).

3.1.2 Effects on the User Interface

If there is no policy defined on an RDF statement, an incoming request is denied by default and the accessing service will point the user to the user interface to define a new policy regulating the access to the RDF statement in the future. On the other hand, no user feedback is requested if a deny policy applies to the RDF statement and the current requester. Therefore, the service needs to distinguish between default denial and policy-based denial. Protune by itself uses only positive authorizations in order to avoid conflicts. For this reason we defined a deny predicate on top of Protune to enable also the definition of deny policies. However, if we allow for both positive and negative authorizations, conflicts can arise: This is the

case whenever a resource is affected by both an allow and a deny policy. To avoid such situations we designed our user interface in order to ensure that no conflict situations will arise or that they are solved in precedence.

When the user defines an allow policy affecting a resource that is already covered by a deny policy, the user interface will show a dialog, notifying the user that there is a conflict.

If the user does not want to allow access to the resource, the allow policy will still be defined (since in our framework deny policies have by default higher priority than allow policies),

otherwise the deny policy will be modified in order to exclude from its scope the affected resource. On the other hand when the user defines a deny policy affecting a resource that is already covered by an allow policy, the user interface will show a dialog, notifying the user that there is a conflict. If the user does not want to allow access to the resource, the deny policy will simply be added (for the same reason described above), otherwise a modified version of it will be added, which excludes from its scope the affected resource.

Finally, if the user model changes, new RDF statements can be automatically covered by existing policies. But the user has also the option to apply his policy only to RDF statements existing at policy creation time.

As soon as a service adds RDF statements, the user will be asked by the user interface whether his policy should also apply to the new statements.

4 USER INTERFACE FOR DEFINING ACCESS POLICIES

The interface that enables users to specify Protune access policies is called *Policy Editor* and operates on top of the access control layer of the User Modeling Service as outlined in Figure 1. If a service attempts to access user data for which no access policies have been defined yet, then the operation of the service fails and the user is forwarded to the Policy Editor. The interface which is shown to the user (see Figure 3) is adapted to the context of the failed operation. Such a context is given by the RDF statements which the service needed to access. 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 policies affecting them (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 such statements the interface additionally shows conflicting policies by marking affected policies and RDF statements.

Warnings make the user aware of critical policies. In Figure 3 the user wants to allow the access to *names* to all instances of a class *Contact*. But as the user may not be aware that such a policy would also disclose all future user profile entries containing a name, he is explicitly prompted for validation. If the user disagrees, he will be prompted whether the policy should be refined to cover only those name instances that are currently stored in the user profile.

In general, policies are edited using the interface depicted in Figure 4. This interface consists of two main parts which allow to:

1. define policies (top frame)
2. dynamically show the effects of the policy (bottom frame)

An *expert mode* is also available, which allows to directly input Protune policies. Users that do not use the *expert mode* just have to instantiate a template consisting of four steps (see top right in Figure 4):

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 4 the resulting Protune policy would be (without *period of validity*):

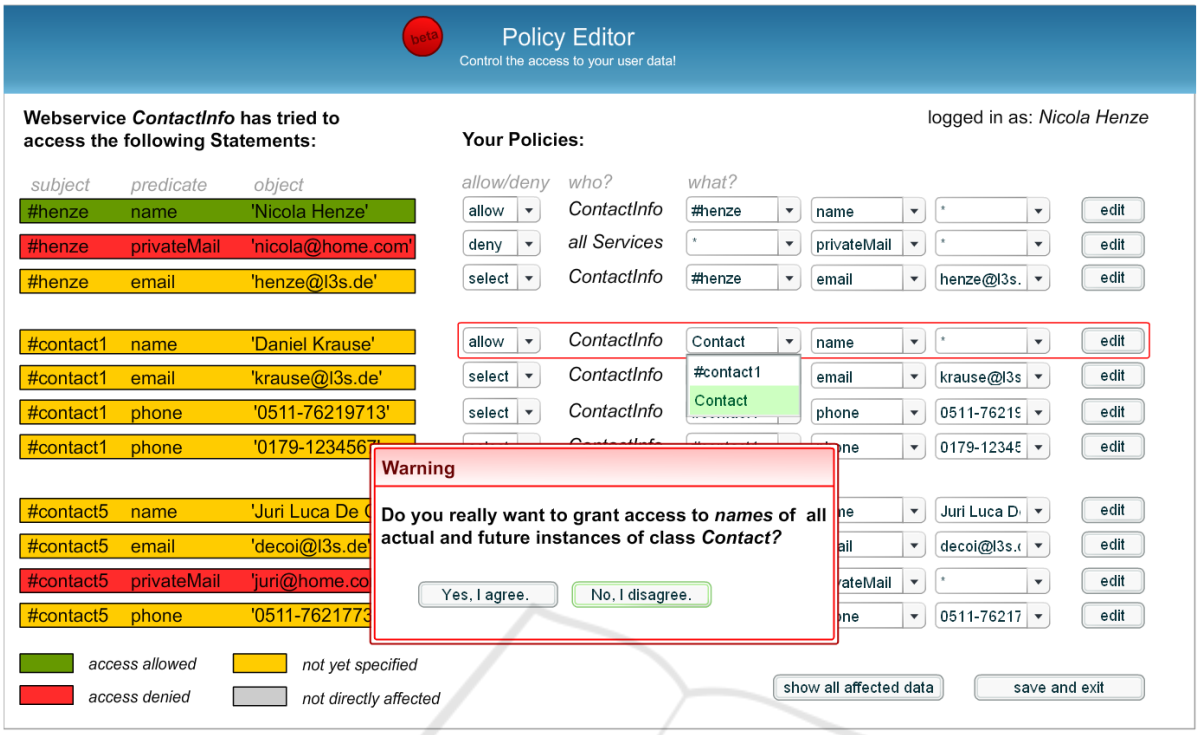

Figure 3: Defining Policies - Overview.

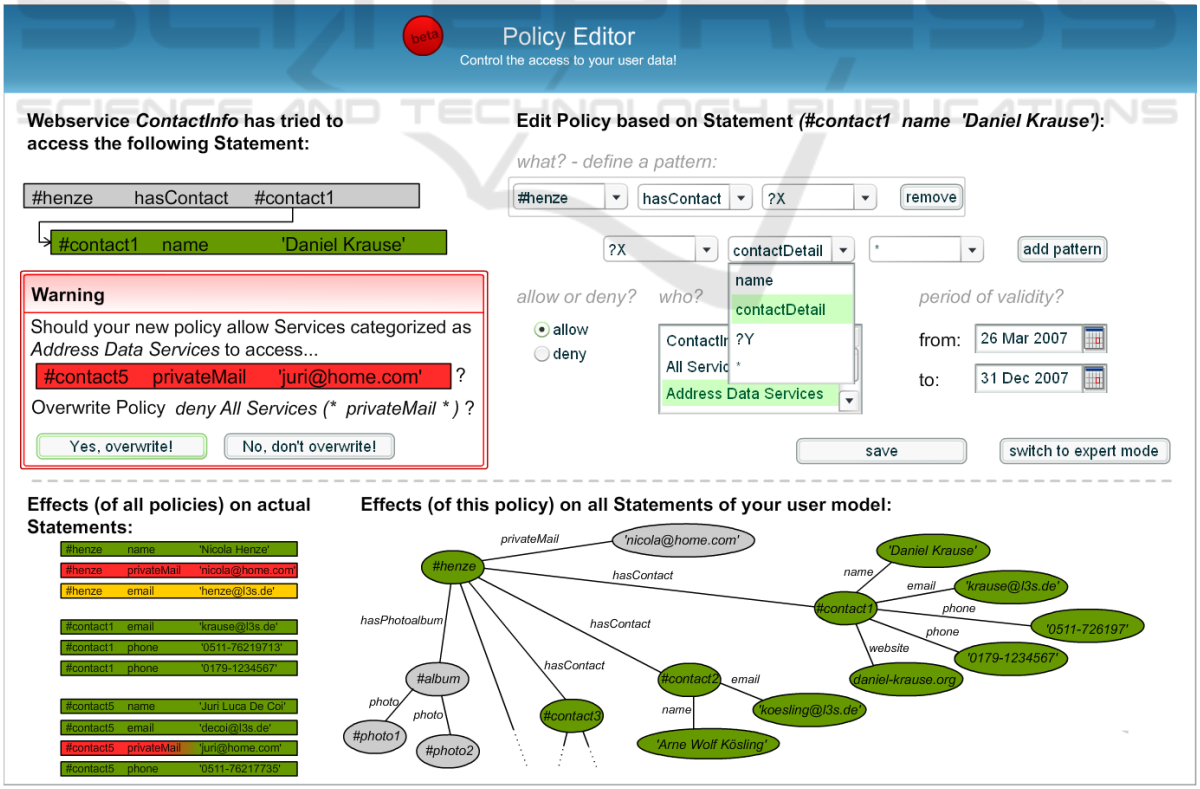

Figure 4: Editing a policy in a detailed view.

```
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 4 *(#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. section 3) 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 4).

5 CURRENT IMPLEMENTATION AND DISCUSSION

In the current implementation[2] the user interface for defining access control policies for RDF data supports already the core functionality described in Section 4. We furthermore integrated the prototype into the Personal Reader framework (cf. Section 2) and applied AC4RDF (Abel et al., 2007) together with Protune to enforce the policies defined by the users. AC4RDF is an access control mechanism for RDF stores. It rewrites queries so that they respect the access policies specified via the user interface, i.e. policies as presented in Section 3.

Regarding usability issues, the main advantage of our user interface are:

- *Easy-to-use* – the users do not need to learn any policy language, policies are created by specifying simple pattern.
- *Scrutability* – users can inspect the effect of the policy immediately as the RDF data is colored either red (access not allowed) or green (Access Allowed)
- *Awareness of Effects* – whenever a change in a policy will disclose data in the future, it is not visualized in the current graph. Hence, users get a confirmation message to make the aware of the effects of the changes.

6 RELATED WORK

Controlling the access to RDF data stores can be realized within the RDF query layer. As outlined above, we utilize AC4RDF (Abel et al., 2007) together with Protune in our current implementation to enforce the policies formulated by the users via the easy-to-use editor presented in this paper. Another approach to access control for RDF data is discussed in (Dietzold and Auer, 2006), where access to RDF data is restricted by defining views, which correspond to RDF subgraphs an inquirer is allowed to access. However this approach does not make use of policies and thus, cannot benefit from their advantages (e. g. negotiation, declarative and intuitive structure, and facility of fine-grained adjustments) and broad range of tool support (e. g. upcoming possibilities like formulating policy rules in natural language [3]) policy languages like Protune offer.

Beside the policy language Protune, there exist other policy languages like e.g. KAoS (Uszok et al., 2003) or Rei (Kagal et al., 2003) that may be utilized to secure RDF statements. But we didn't choose those languages for protecting RDF graph patterns but Protune instead, because in our approach we make use of properties like e.g. recursive definitions and variables and we also want to exploit advanced features like negotiations in the future. For some policy languages there are policy editors available. Because of their complexity, most of these editors are difficult to use. KPAT (Uszok et al., 2004), a policy editor for KAoS, offers to constrain the creation of policies with forms to ease the usage. However, none of the editors deals with the visualization of the policies' consequences or takes RDF graph structure into account to deduce policy templates.

[2]The policy editor user interface is made available via: http://www.personal-reader.de/wp/projects/policy-editor

[3]Expressing Protune rules in ACE: http://attempto.ifi.unizh.ch/site/

7 CONCLUSIONS AND FURTHER WORK

In this paper we presented a user interface that enables non-expert users to control the access to their RDF-based user profiles. We used the policy framework Protune to enforce the underlying access control layer and outlined how to use policy templates to define access control policies. Furthermore, we discussed how to deal with conflicting policies and how a user interface helps to understand complex and expressive policies and their consequences. We presented the current implementation of the proposed user interface. In the future we will evaluate the user interface by testing its usability. This includes to check whether a user will be able to express his intention and whether he can be made aware of the consequences of his policies.

REFERENCES

Abel, F., Baumgartner, R., Brooks, A., Enzi, C., Gottlob, G., Henze, N., Herzog, M., Kriesell, M., Nejdl, W., and Tomaschewski, K. (2005). The personal publication reader, semantic web challenge 2005. In *4th International Semantic Web Conference*.

Abel, F., Coi, J. L. D., Henze, N., Koesling, A. W., Krause, D., and Olmedilla, D. (2007). Enabling advanced and context-dependent access control in rdf stores. In *6th International Semantic Web Conference*, pages 1–14.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Bonatti, P. A. and Olmedilla, D. (2005a). 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.

Bonatti, P. A. and Olmedilla, D. (2005b). Policy language specification. Technical report, Working Group I2, EU NoE REWERSE.

Bonatti, P. A. and Samarati, P. (2000). Regulating service access and information release on the web. In *ACM Conference on Computer and Communications Security*, pages 134–143.

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)*.

Gavriloaie, R., Nejdl, W., Olmedilla, D., Seamons, K. E., and Winslett, M. (2004). No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In *1st European Semantic Web Symposium (ESWS 2004)*, volume 3053 of *Lecture Notes in Computer Science*, Heraklion, Crete, Greece. Springer.

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.

Henze, N. and Krause, D. (2006). Personalized access to web services in the semantic web. In *SWUI 2006 - 3rd International Semantic Web User Interaction Workshop*, Athens, Georgia, USA.

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)*, Lake Como, Italy. IEEE Computer Society.

Lloyd, J. W. (1987). *Foundations of Logic Programming, 2nd Edition*. Springer.

Uszok, A., Bradshaw, J. M., Jeffers, R., Suri, N., Hayes, P. J., Breedy, M. R., Bunch, L., Johnson, M., Kulkarni, S., and Lott, J. (2003). Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, Lake Como, Italy. IEEE Computer Society.

Uszok, A., Bradshaw, J. M., Johnson, M., Jeffers, R., Tate, A., Dalton, J., and Aitken, S. (2004). Kaos policy management for semantic web services. *IEEE Intelligent Systems*, 19(4):32–41.