Annotation profiles: Configuring forms to edit RDF

Matthias Palmér
Royal Institute of Technology, Sweden
matthias@csc.kth.se

Fredrik Enoksson
Royal Institute of Technology, Sweden
fen@csc.kth.se

Mikael Nilsson
Royal Institute of Technology, Sweden
mini@csc.kth.se

Ambjörn Naeve
Royal Institute of Technology, Sweden
amb@csc.kth.se

Abstract

Most of today's generic annotation tools for semantic web metadata (RDF) are designed for experts. People with no or little knowledge about RDF are therefore forced to use simplified and often domain-specific tools that work with fixed sets of metadata elements. This paper introduces the Annotation Profile Model as a configuration mechanism from which annotation tools can be automatically generated. The intention is to encourage metadata- or domain experts to define annotation profiles according to metadata vocabularies. This will allow end-users or administrators to select appropriate annotation profiles for the task at hand, and then an editor will be provided by the underlying system. This paper discusses the design of the Annotation Profile Model, which consists of a data-capturing part (the Graph Pattern Model) and a presentation part (the Form Template model). An implementation that can generate both web-based and standalone editors is also introduced.

Keywords: application profile; RDF editing; SPARQL; Fresnel; XForms.

1. Introduction

The intention of this paper is to outline how to build flexible, reusable and standards-compliant annotation tools that allow editing of metadata records, expressed in RDF, in an end-user friendly manner. The solution discussed here, the *Annotation Profile Model*, is a configuration mechanism for such annotation tools, used to adapt them to a specific metadata vocabulary, interaction style, and appearance. The term "annotation" in "annotation profile" and "annotation tool" was chosen to imply the human authoring of metadata. This should be contrasted with the term "application" in "application profiles" (Heery et al., 2000) where the main purpose is instead to guide application developers and improve interoperability.

1.1. Annotation Tools

Annotation tools with RDF support can be divided into three categories according to their adaptability to different kinds of metadata. First, *fixed annotation tools* have little or no adaptability as they are hard-coded to support a specific metadata vocabulary. Second, *configurable annotation tools* can be adapted by switching between configurations. Such configurations are typically developed by experts for compliance with specific metadata vocabularies and/or domain specific extensions. Third, *generic annotation tools* can edit any metadata by more or less exposing the underlying RDF data or ontologies.

For several reasons, *generic* annotation tools are not recommended for end-users with little or no technical background. First, expressing metadata directly in RDF requires expert knowledge and careful editing to comply with metadata the vocabularies used. Second, if the starting point is editing RDF rather than providing information according a specific standard or schema, guidance and editing scope will be lacking. Third, a user interface aimed at end-users should only require conceptual understanding of what metadata is.

The usefulness of *fixed* annotation tools can also be challenged but for different reasons than for generic annotation tools. One reason concerns the way in which metadata vocabularies are developed. Greater requirement for interoperability, not only by using specific standards but also

across standards and domains of use, requires adaptable annotation tools. Furthermore, if several persons are involved in an editing process, then - depending on their skill and competence - they may need to edit different kinds of metadata. In practice this translates to being able to handle greater diversity of metadata, for example to easily switch between or include new elements and vocabularies. Another reason is that in order to minimize the amount of reinvention (implicitly cost of developing metadata intensive applications), speed of development, as well as provide a more capable editing environment, annotation tools need to be independent and reusable components. Fixed annotation tools are per definition less independent and reusable than configurable annotation tools.

1.2. Configurable Annotation Tools

This leaves us with the category of *configurable* annotation tools. This paper focuses on a language-neutral interface that we term the Annotation Profile Model for capturing the configurations of configurable annotation tools, first introduced in Annotation Profile Specification (Palmér et al., n.d.). For simplicity, the Annotation Profile Model has been restricted to supporting only a *form-based* approach to editing. Other uses are not prohibited - only not prioritized in the design. An important requirement is that a form for editing can be generated automatically from an annotation profile. The structure, style, interactivity and which metadata record that is edited is specified in the annotation profile, and is consequently reflected in a generated form. A metadata record translates to a small graph around a central resource in RDF. The RDF statements of this graph may originate from a single metadata vocabulary or, if so desired, originate from several distinct vocabularies.

Annotation tools that are possible to configure do exist, for example *Reggie* that uses *schema files* (DSTC, n.d.) for configuration and the *Semantic MediaWiki* (Völkel, 2006) that uses *semantic templates* that can be configured in order to edit RDF. These configurations are neither complete nor general enough, for the way of configuring editors that we aim for. The annotation profile model described here is used in the SHAME library[1], a software library on top of which various configurable annotation tools can be built. The library has been developed by the authors and it will be briefly introduced at the end of this paper.

1.3. Structure of this Paper

The rest of the paper will go through the Annotation Profile in some detail, however, for those who require a more complete description, the Annotation Profile Specification (Palmér et al., n.d.) provides much more detail. Section 2 goes through the requirements and section 3 compares the Annotation Profile Model with existing initiatives. In section 4 the Annotation Profile Model design is introduced and it is followed by a section on the implementation in the SHAME library. Finally there are some concluding remarks and acknowledgments.

2. Requirements

Annotation profiles are meant to be specific enough to allow user-friendly form-based editors to be generated automatically. What user-friendly means depends on who the user is. We foresee three different user roles involved in the editing process: *annotation profile author*, *annotation profile facilitator*, and *end-user*. Even though the same person can play more than one role, Annotation Profiles are mainly targeted at simplifying the editing process for end-users:

With these different roles in mind, we need to agree on what can and should be configured in advance in an annotation profile and what is suitable to leave to the end-user.

[1] Written in Java and available through LGPL license, see http://shame.sf.net.

TABLE 1. User roles, corresponding tasks and their required knowledge.

User role	Task	Required knowledge
Annotation Profile Author	Creates or modifies Annotation Profiles.	Experts on schemas/ontologies and/or domain experts
Annotation profile facilitator	Defines requirements for annotation profiles, selects annotation profiles and makes them available in an application for a specific group of people	Knowledgeable on the tasks of the group and where to find useful Annotation Profiles
End-user	Edits metadata in an editor generated from an Annotation Profile	Conceptual understanding of the metadata and the domain

First, the end user should not be bothered with the *value type*, that is, which RDF expression to use (for example a literal or a resource). Second, we believe that the *input style*, that is whether to input text by hand or choose from a predefined list, should also be defined by the annotation profile author. For example, when editing the title of a resource, it is natural to edit it directly as a text string (value type=literal, input style=edit). On the other hand, when providing type of a resource it is suitable to choose a URI from a list (value type=Resource, input style=select).

We will now continue by going through the requirements on Annotation Profiles divided into four categories: *completeness*, *structure*, *interaction*, and *presentation:*

TABLE 2. Four categories of requirements on Annotation Profiles.

Category	Description of category
Completeness	includes support for editing arbitrary well-formed RDF triples, i.e. according to all value types described above. Support for RDF containers and collections are also required. The Annotation Profile should specify which statements to edit and which to leave untouched.
Structure	includes cardinality constraints and order of selected statements. A direct correspondence between the graph structure and its presentation in the form should not be enforced. For example, it should be possible to hide a complicated graph-structure with intermediate resources from the end-user, and it should be possible to introduce pedagogical/cosmetic groupings of statements when the graph-structure is too flat.
Interaction	includes hints on how to choose values from vocabularies/ontologies, e.g. check-boxes, radio-buttons, drop-down menus, or search-dialogs. It also include mechanisms for string validation according to datatypes, control of auto-complete mechanisms etc.
Presentation	includes multilingual labels and descriptions to aid the user in deciding how to edit. Font, color, indentations, borders, and everything else that has to do with appearance is also included here.

Note that a few of these requirements overlap with what can be derived from RDF Schemas or OWL ontologies, e.g. cardinality constraints in OWL. When such information exists, profile-based editors are required to make use of it, allowing annotation profile authors to avoid duplication of information.

3. Background

This section presents existing initiatives and standards that are relevant for Annotation Profiles. We briefly investigate how they support the requirements in the previous section, see section 4,5, and 6 in (Enoksson et al., 2006) for a longer analysis.

3.1. Application Profiles

The concept of an Annotation Profile can in part be derived from the concept of an *Application Profile* (Heery et al., 2000). Application profiles specify which metadata to use in a specific application. Hence, a better name would be *the application's metadata profile*. The major purposes of an Application Profile are to be informative for developers, and to encourage interoperability. In contrast, an Annotation Profile has the additional purpose of allowing automatic generation of user interfaces for the metadata decided upon.

The main idea behind application profiles is to describe how a certain application can mix-and-match metadata terms/elements/properties from existing vocabularies/schemas/ontologies. In (Heery et al., 2000) the use of application profiles is described as:

> *Typically implementors are part of larger communities, they form part of a sector (education, cultural heritage, industry, government), possibly a subject-grouping, they are part of programmes with common funding, they work with others serving the same target audiences. In order to work effectively these communities need to share information about the way they are implementing standards. Communities can start to align practice and develop common approaches by sharing their application profiles.*

The paper (Nilsson et al., 2006) suggests a framework for Dublin Core metadata that aims for greater interoperability between metadata standards, within which application profiles play an integral part. There are also some attempts at formalizing application profiles, such as the initial guidelines available in (Baker et al., 2005). This document does not provide a machine-readable expression but briefly mentions earlier attempts at this, such as SCHEMAS and MEG. All these initiatives focus on reusing and refining terms accompanied with context-specific vocabulary restrictions. Even if application profiles would be ready for machine processing today, they still would not resolve issues regarding cardinality restrictions, order and grouping, labeling and layout in the user interface, as well as various other interaction-specific issues, even though an application profile can form a good basis for an annotation profile.

3.2. RDF Schema and OWL

One approach is to use RDF Schema or OWL as the configuration mechanism. Unfortunately, neither of them is enough to cover our requirements. RDF Schema, or more correct RDF vocabulary language, provides many useful instructions for configuring an editor, e.g. labels and range restrictions. However, there is neither any cardinality restrictions, nor any support for ordering or grouping of properties, see section 5.2.2 in (Enoksson et al., 2006). OWL, on the other hand, provides cardinality restrictions but not much more from the configuration perspective, see section 5.2.3 in (Enoksson et al., 2006). Both RDF Schema and OWL lack the ability to select which properties to use, and they cannot express whether a blank node or URI should be used, neither whether a literal should have a language or not. Furthermore, there is little or no support for our requirements on the presentation or interaction categories outlined above.

3.3. Fresnel Display Vocabulary

Fresnel (Bizer et al., 2005) is a vocabulary for *displaying* RDF in a browser-independent fashion. Fresnel introduces the concept of a *lens* to specify a set of properties that should be displayed for a certain set of resources. Moreover, it provides *formats* for specifying formatting instructions on individual properties or sets of resources. Fresnel makes use of RDF Schemas when they are available for labeling and for prioritizing between lenses and formats. Since Fresnel is intended for presentation purposes only, it is hardly surprising that it does not support cardinality restrictions and language control. To extend Fresnel to be useful for editing seems to be feasible on a technical level, and the documents describing Fresnel clearly state that an editor extension is encouraged.

3.4. XForms

XForms (Boyer, n.d.) is a browser-independent technology intended, among other things, to replace forms in HTML. XForms splits into a *form model* and a *user interface*. The form model describes what XML data to edit. The user interface makes use of a fixed set of form controls, e.g. input, text-area, trigger, choice, label, etc., that can be freely mixed with e.g., XHTML. Each form control may be rendered differently depending on the type of browser, as well on as the supported media types.

XForms cannot be used to edit RDF. However, much inspiration can be found in the design of XForms. For example, the distinction between the form model and the user interface has inspired the separation of annotation profiles into *graph patterns* and *form templates*, see section 4.

4. Design of Annotation Profile Model

We will now introduce the Annotation Profile Model as a platform- and language-independent information model. This independence means that for every programming language and sometimes for each platform this interface needs to be realized as an API. Furthermore, acknowledging the usefulness of OWL, SPARQL, Fresnel etc., a dedicated syntax for representing and exchanging Annotation Profiles will not be provided. Instead, our focus is on allowing Annotation Profiles to be defined by expressing information that is complementary to existing expressions in OWL, SPARQL, Fresnel etc.

As made explicit in the structure category, a deviation between the original RDF graph and its presentation in the form is sometimes requested. Furthermore, the interaction and presentation categories provide requirements of another character, sometimes without any relation to the underlying representation in RDF. In order to address this we have divided the Annotation Profiles Model into two parts, the *graph pattern model* and the *form template model*:

- **The Graph Pattern Model** is responsible for capturing and creating subgraphs of triples, hence playing the roles of a query and template language at once. The Graph Pattern Model introduced below is heavily inspired by the SPARQL language, borrowing some of its terminology to increase understanding. However, SPARQL, as well as other RDF query languages, are ill-suited as a dedicated syntax for two reasons. First, they are too complex and include capabilities that makes creation of new triples undefined, e.g., disjunction. Second, choosing a single syntax, such as SPARQL, would make our approach less compatible with other approaches such as Fresnel. However, subsets and restrictions of existing query languages, such as SPARQL or QEL will be identified and mapped to the Graph Pattern Model.
- **The Form Template Model** provides order, grouping and pedagogical/cosmetic deviations from the RDF structure that is sometimes needed. The Form Template Model is a tree with references to variables of the Graph Pattern Model. The nodes of the tree also provide information on interactions, labels, descriptions and hooks for external style sheets etc.

The Annotation Profile Model has now been described as consisting of these two parts, the graph pattern and the form template. The generation of a GUI from a specific annotation profile requires several steps. First the RDF graph is bound in a intermediate and internal format, a set of *variable bindings*, via the graph pattern. Second the set of variable bindings is combined with the form template to generate an abstract representation of the form, the form model, to be displayed later in the GUI. Finally the GUI is generated from the form model and provides editing capabilities through manipulation of the form model. These manipulations are fed back into the graph pattern via the underlying variable bindings. See the FIG. 1 for an illustration of how of the data flows from the RDF graph to the user interface via the annotation profile.

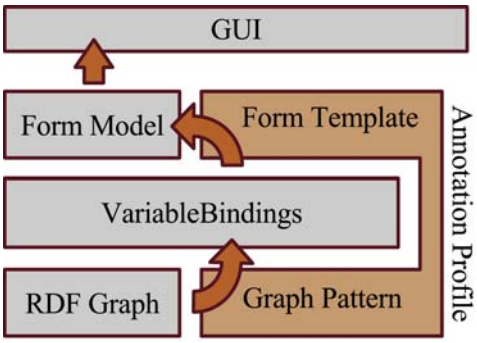

FIG. 1. Annotation Profile execution flow.

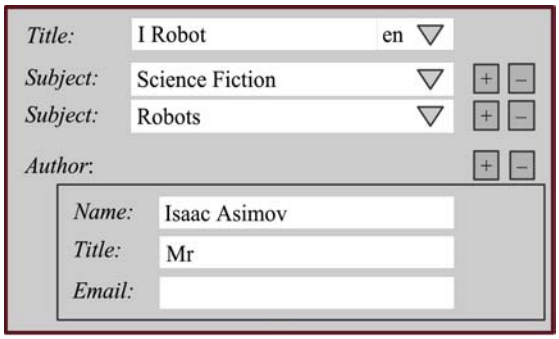

FIG. 2. An editor GUI for book metadata.

In the following subsections we start from the user perspective and introduce the form template model first as it has very close ties to the user interface. After that we go through the Graph Pattern Model with the aim to understand how data in RDF graphs are created and matched.

4.1. The Form Template Model

In order to understand the needs of form templates, let us consider editing the title-, subject-, and author items of a resource in a form, see the FIG. 2. In this form we see that the title item is a simple text field, the subject items are drop-down menus of available subjects taken from an ontology, and the author item is a sub-form containing text fields for the name-, title- and email item of the author. Breaking the form apart we say that the form consists of *items* - in this case four items - where the last item (the author item) has three *sub-items*. Let us now consider the constituents of an item.

First, each item has a *label*, in this case saying 'title', 'subject' (repeated twice), 'author', 'name', and 'title' respectively. Second, each item may provide input capabilities of various sorts, e.g., a text field for the title and drop-down menus for the subjects. Note that the author item does not provide input capabilities since it has sub-items. Third, items can be added or removed via specific '+' and '-' buttons. The availability of these buttons are due to cardinality restrictions. In the current example, an unlimited number of subjects and authors is allowed, and hence the corresponding items have buttons. The title item has a cardinality of one, and therefore it has no buttons. Furthermore, for every author, the author's name-, title-, and email item has a cardinality of one, and hence these items have no buttons either. Note that buttons on (parent) items should not be confused with buttons on sub-items (when they exist).

We now see how the user interface is composed of items. Moreover, which actual items that are visible in the user interface depends on the *Form Template* and the data extracted into variable bindings by the graph pattern. A Form Template provides order, structure, presentation specifics, variable references, and cardinality restrictions on a set of items. When data in the form of variable bindings is combined with a form template, the result is a *Form Model*, see the

illustration in FIG. 3. It is from the resulting Form Model that the user interface shown in FIG. 2 is generated.

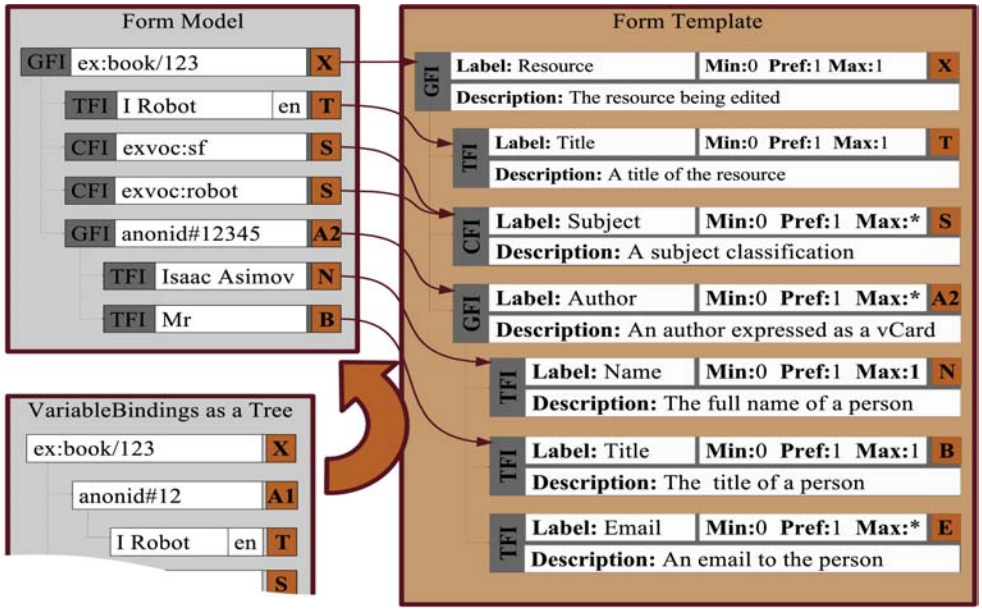

FIG. 3. VariableBindings combined with a Form Template produces a Form Model.

It is not always the case that there is a one to one relationship between the Form Model and the Form Template. In the example above there are two subjects and no email of the author in the data, which is correspondingly reflected in the Form Model. The Form Template, being a template, does provide a single subject- and email item, but it allows the Form Model to deviate according to the specified cardinality restrictions. In general, branches of the Form Template that reference variables that have no corresponding VariableBindings will be left out of the Form Model. Branches that reference variables with multiple VariableBindings will be correspondingly multiplied in the Form Model.

Editing in the user interface is directly transferred into the nodes of the RDF graph via the VariableBindings. When data is missing, i.e., when there is no VariableBindings for certain variables, the Form Template is used to provide placeholders for editing in the user interface. This is the case for the email item in the example above. When such placeholders are triggered/filled-in, they generate new constructs in the RDF graph via the variable, the closest VariableBinding, and the corresponding parts of the graph pattern.

The following example shows an XML representation of the form template outlined above:

```
<FormTemplate  xmlns="http://kmr.nada.kth.se/APtags#">
 <Group max="1" vref="X">
   <Label lang="en">Resource</Label>
   <Description lang="en">The resource being edited</Description>
   <Text max="1" vref="T">
      <Label lang="en">Title</Label>
      <Description lang="en">A title of the resource</Description>
   </Text>
   <Choice vref="S">
      <Label lang="en">Subject</Label>
      <Description lang="en">A subject classification</Description>
   </Choice>
   <Group vref="A2">
     <Label lang="en">Author</Label>
```

```
    <Description lang="en">An author expressed in vCard</Description>
    <Text max="1" vref="N">
       <Label lang="en">Name</Label>
       <Description lang="en">The full name of a person.</Description>
    </Text>
    <Text max="1" vref="B">
       <Label lang="en">Title</Label>
       <Description lang="en">The title of a person.</Description>
    </Text>
    <Text max="1" vref="E">
       <Label lang="en">Email</label>
       <Description lang="en">An email to the person</Description>
    </Text>
   </Group>
 </Group>
</FormTemplate>
```

4.2. The Graph Pattern Model

The purpose of the graph pattern model is to provide access to RDF graphs through variables. Pragmatically this means both *capturing* existing RDF-data as well as *creating* new RDF-data. To capture existing data, a graph pattern behaves like a *query*, matching triples and consequently assigning literals and resources to variables. In the simplest case, a graph pattern prescribes an individual triple, in more complex cases, a graph pattern prescribes sub-graphs consisting of several interconnected triples. When creating new RDF-data, a graph pattern is used as a *template,* where variables indicate the points of possible user input.

Variables may be more or less constrained. From the perspective of capturing, this only affects the size of the set of matches. However, from the perspective of using the graph pattern as a template, i.e., from the perspective of creating new metadata, the number of constraints corresponds to the number of decisions forced on the end-users. From the discussion in section 2, we concluded that end-users should be required to have only conceptual understanding of metadata. This translates to constraining variables in the graph pattern so that every variable corresponds to a given input type, not exposing any details of RDF, such as the value type. This means more constraints on variables rather than less.

This leaves us with two choices. Either we require that the graph pattern always provides these constraints explicitly, or we provide a *graph pattern default semantics*. In compliant *graph pattern engines,* default semantics would be enforced. We have chosen an approach with a default semantics, since it places less restrictions on authors of graph patterns and it is less sensitive to lacking constraints. There is also the added value of being able to reuse query expressions for editing purposes without modifications.

The term *Graph Pattern* is borrowed from SPARQL, even though the intention here is a restricted form of use. Much like in SPARQL, the basics of the graph pattern is matching individual triples by *Triple Patterns.* In FIG. 4, we see how a graph pattern, which is built up of ten Triple Patterns, is used to capture an RDF graph into a tree of *VariableBindings*.

The following is two different listings of the graph pattern of the book example expressed in SPARQL:

```
SELECT *
WHERE {
  ?X dc:title ?A1.
    ?A1 rdf:type rdf:Alt.
    ?A1 rdfs:member ?T.
  ?X dc:subject ?S.
    ?S rdf:type exvoc:Genre.
  ?X dc:author ?A2.
    ?A2 vc:FN ?F.
    ?A2 vc:TITLE ?B.
```

```
SELECT *
WHERE {
  OPTIONAL{ ?X dc:title ?A1.
            ?A1 rdf:type rdf:Alt.
            OPTIONAL{?A1 rdfs:member ?T}}
  OPTIONAL{ ?X dc:subject ?S.
            ?S rdf:type exvoc:Genre }
  OPTIONAL{ ?X dc:author ?A2.
            OPTIONAL{ ?A2 vc:FN ?F }.
            OPTIONAL{ ?A2 vc:TITLE ?B }.
```

```
?A2 vc:EMAIL ?E}                        OPTIONAL{ ?A2 vc:EMAIL ?E }}
                                FILTER isBlank(?A1).
                                FILTER isLiteral(?T).
                                FILTER isURI(?S).
                                FILTER isBlank(?A2).
                                FILTER isLiteral(?F).
                                FILTER isLiteral(?B).
                                FILTER isLiteral(?E) }
```

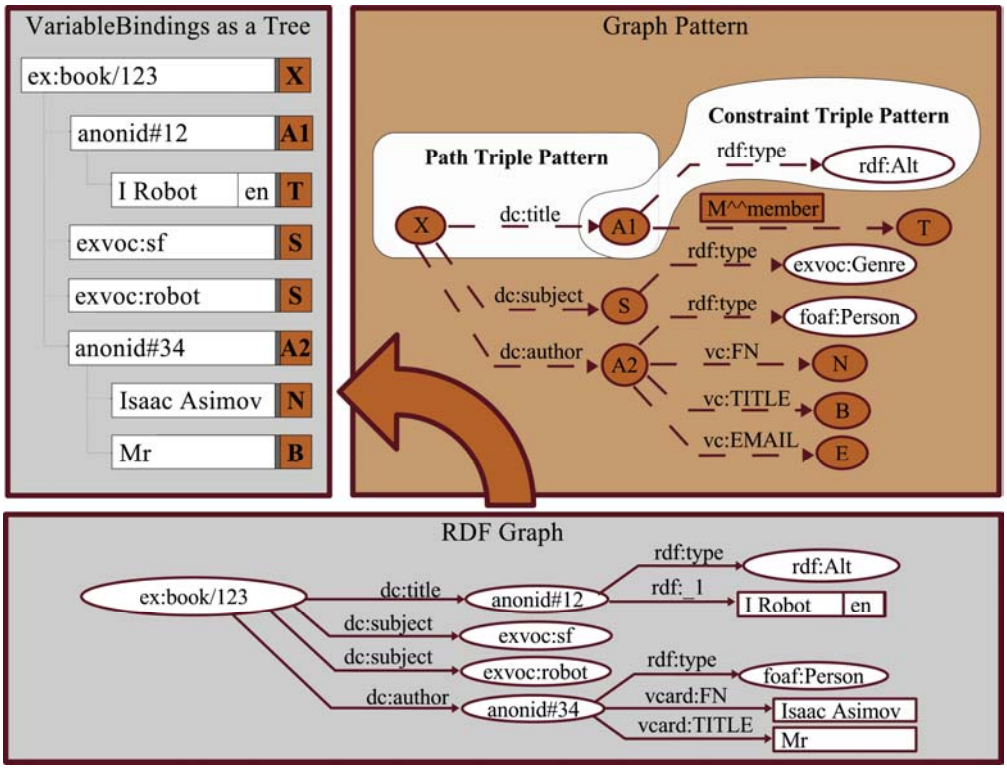

FIG 4. An RDF graph is matched into VariableBindings via a graph pattern.

The OPTIONAL and FILTER constructions shown to the right are not strictly necessary since they are automatically inferred through the default semantics, hence they are not shown in FIG. 4 either.

Note that the optional construction means that we are matching triples independently of each other, for example, if there is a title or not should not affect whether we match a subject or not. The required triples corresponds to what we termed *constraint triple patterns*, that is, triple patterns that have a fixed predicate and object. In most cases, constraint triple patterns are used to require the existence of specific typings as in the case of the title and subject in the book example.

The priority regarding enforcing default semantic constraints on the graph pattern are the following: first what is given in the graph pattern expression explicitly, second what can be found in RDF Schemas/OWL for the predicates and types indicated in the graph pattern, and finally, what is given by the default semantics of the Graph Pattern Model. See the annotation profile specification for a more full explanation of this.

5. Implementation

When designing the annotation profile model, much inspiration and lessons learned have been drawn from experiences of developing the SHAME library. It was first developed as part of the SCAM framework (Palmér et al., 2004) for the purpose of providing a web based user interface for editing RDF in an electronic portfolio application. Since then, it has been redesigned to be useful also in standalone applications. The SHAME library now almost fully implements the Annotation Profile Model and is useful for developing configurable annotation tools and integrating them into specific environments.

SHAME is implemented in the Java programing language and requires an underlying API for working with RDF. The design allows the RDF API to be easily replaced allowing a tighter integration with tools with other requirements, however, currently only Jena is supported.

It is important to notice that SHAME is in itself not an annotation tool, instead it is a code library on top of which various annotation tools can be built. However, two proof of concept configurable annotation tools are provided in SHAME for demonstration purposes, first *Meditor,* a standalone Java Swing based application and second *Speed,* a simple web based solution using the Apache velocity template engine for web page creation. Both these tools are configurable in the respect that they allow end users to select among a list of available Annotation Profiles when editing. When integrating annotation tools built upon the SHAME library into a surrounding environment, some adaptation is needed to provide access to the RDF that should be edited. The SHAME library provides an interface to be implemented exactly for this purpose. For example, the Meditor tool implements the interface so that RDF is stored directly into files.

Since no dedicated syntax exists, SHAME has been designed to be easily adaptable to different syntaxes. Currently, a form template is expressed through an RDF/XML syntax, support for an XML syntax is planned. A graph pattern can be expressed in either SPARQL or QEL – Query Exchange Language – see (Nejdl et al., 2002) and (Nilsson et al., 2004). Detecting and merging restrictions from RDF Schema or OWL into the graph pattern remains to be fully supported.

We are currently developing another configurable annotation tool based on SHAME along the lines of Web2.0, that is, an AJAX based client. We have not chosen to support annotation profiles directly in the client, as it would mean redeveloping large parts of the SHAME library directly in javascript, which would also indirectly require an RDF API to be available. Instead, we have chosen to work directly with form models that are retrieved over a REST based protocol from a server. The server would need to use SHAME or some other compatible annotation tool library to provide the form models to the client as well as interact with underlying RDF storages.

6. Conclusions and Future Work

We have introduced the Annotation Profile Model as a complementary configuration mechanism to RDFS and OWL ontologies for automated configuration of annotation tools. The purpose of an annotation profile is to make the generated user interface simple enough to be useful for end-users without knowledge of RDF. Two new user roles were introduced by necessity: the annotation profile author and facilitator, which, correspondingly, authors and selects/customizes annotation profiles for the convenience of the end-user in a specific annotation tool environment.

Of the four categories of requirements, the Annotation Profile Model covers only completeness and structure fully. For the interaction category, fundamental support for indicating value type and input style are in place, however, more specific interactions, such as specifying that radio buttons or drop-down menus should be used, remain to be specified. For the presentation category, only labels and descriptions are in place. However, hooks for stylesheets are available for testing in the current implementation. Introducing better support for these two requirement categories will most likely be inspired by XForms.

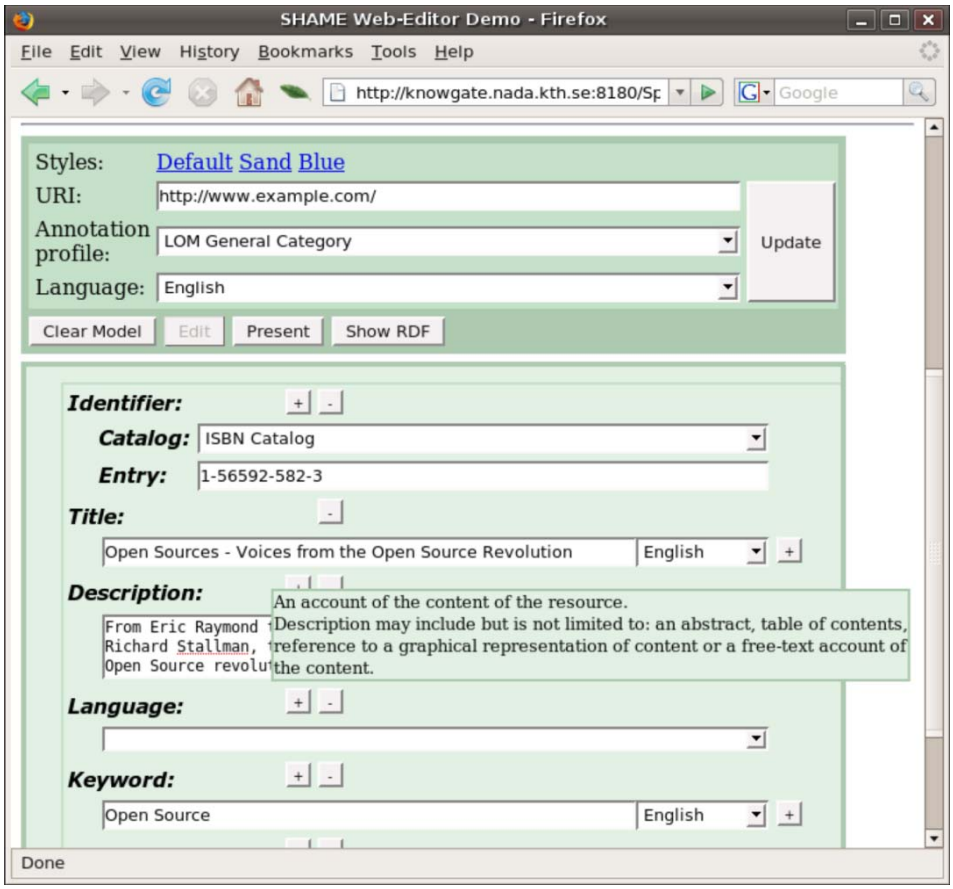

FIG 5. The annotation tool Speed, configured by an annotation profile for the general category of LOM used to edit a book. The demo allows the end user to change both the annotation profile and the language for the user interface.

The SHAME library supports nearly all of the features of the Annotation Profile Model, only some issues with RDF Schema/OWL remains. Beyond making the SHAME library compliant with Annotation Profile Model the plan is to build a RESTful annotation tool based on Web2.0 techniques. Another priority is to support more alternative syntaxes as well as align with a future editing extension of Fresnel. When the Dublin Core Application Profile is a reality, we will also look into the option of doing automatic conversions or providing complementary information that would make DCAP expressions directly compatible. For the time being we will closely watch the development and try to influence in a direction that makes the two initiatives as compatible as possible.

Finally, it is interesting to note that the approach of dividing the functionality in a graph pattern and a form template with communication through VariableBindings makes it possible to imagine editing other data models than RDF by replacing the query/template language. For example, it would be feasible to edit the content of a relational database through SQL queries, since SQL resultset would correspond nicely to the set of variable bindings.

7. Acknowledgements

This work has been carried out with financial support from the EU-FP6 projects LUISA and Prolearn, which the authors gratefully acknowledge. We also want to thank Henrik Eriksson, Jöran Stark, and Jan Danils who did the first implementations of the SHAME library in 2002-2003.

References

Baker, Tomas, Makx Dekkers, Thomas Fischer, and Rachel Heery. (2005). *Dublin Core application profile guidelines*. Retrieved March 27, 2007, from http://dublincore.org/usage/documents/profile-guidelines/.

Bizer, Chris, Ryan Lee, and Emmanuel Pietriga. (2005). *Fresnel—Display vocabulary for RDF*. Retrieved March 27, 2007, from http://www.w3.org/2005/04/fresnel-info/manual/.

Boyer, John M. (n.d.). *XForms 1.1*. Retrieved April 16, 2007, from http://www.w3.org/TR/xforms11/.

DSTC, Resource Discovery Unit. (n.d.). *Reggie—The metadata editor.* Retrieved April 16, 2007, from http://metadata.net/dstc.

Enoksson, Fredrik, Matthias Palmér, Ambjörn Naeve, Sinuhe Arroyo, David Fuschi, and Tomas Pariente. (2006). *D3.1: State of the art: SWS infrastructure, annotation, LCMS.* Retrieved April 16, 2007, from http://www.luisa-project.eu

Heery, Rachel, and Manjula Patel. (2000). *Application profiles: Mixing and matching metadata schemas*. Retrieved March 27, 2007, from http://www.ariadne.ac.uk/issue25/app-profiles/.

Nejdl, Wolfgang, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. (2002). Edutella: A P2P networking infrastructure based on RDF. *Proceedings of the 11th World Wide Web Conference*.

Nilsson, Mikael, and Wolf Siberski. (2004). *RDF Query Exchange Language (QEL)—Concepts, semantics and RDF syntax*. Retrieved July 2, 2007, from http://edutella.jxta.org/spec/qel.html.

Nilsson, Mikael, Pete Johnston, Ambjörn Naeve, and Andy Powell. (2006). Towards an interoperability framework for metadata standards. *Proceedings of the International Conference on Dublin Core and Metadata Applications: Metadata for Knowledge and Learning, Colima, Mexico.*

Palmér, Matthias, Ambjörn Naeve, and Fredrik Paulsson. (2004). The SCAM framework: Helping Semantic Web applications to store and access metadata. *Proceedings of the European Semantic Web Symposium*.

Palmér, Matthias, Fredrik Enoksson, and Ambjörn Naeve. (n.d.). *D3.2: Annotation profile specification*. Retrieved April 16, 2007, from http://www.luisa-project.eu.

Völkel, Max, Markus Krötzsch, Denny Vrandecic, Heiko Haller, and Rudi Studer. (2006). Semantic Wikipedia. *Proceedings of the 15th International World Wide Web Conference*.