

LUIA

*Learning Content Management System Using Innovative Semantic Web
Services Architecture*

IST- FP6 – 027149



Deliverable 3.2 Annotation Profile Specification

Matthias Palmér
Fredrik Enoksson
Mikael Nilsson
Ambjörn Naeve

Due date of deliverable: 28/02/2007

Actual submission date: 30/09/2007

Start date of the project: 01/03/2006

Duration: 30 Months

Matthias Palmér
Uppsala Learning Lab

Version 3.0, dated 31/08/2007

Change History

Version	Date	Status	Author (Partner)	Description
0.1	20060804	Initial draft	Matthias Palmér (ULL), Ambjörn Naeve (ULL), Fredrik Enoksson (ULL)	Internal review version
0.2	20060822	Initial draft	Matthias Palmér (ULL)	Adjusted according to reviewers comments
1.0	20060823	Final v1	Matthias Palmér (ULL)	Finalized according to template requirements
1.1	20070127	Initial draft of v2	Matthias Palmér (ULL),	More theory behind APs added
1.2	20070131	Initial draft of v2	Matthias Palmér (ULL), Mikael Nilsson (ULL), Ambjörn Naeve (ULL) Fredrik Enoksson (ULL)	Examples of how to use APs and suggested Syntax
2.0	20070302	Final v2	Matthias Palmér (ULL), Mikael Nilsson (ULL), Fredrik Enoksson (ULL)	Adjusted according to reviewers comments and document finalized

3.0	20070831	Final	Matthias Palmér (ULL) Ambjörn Naeve(ULL)	Adjusted according to EC reviewers comments
-----	----------	-------	---	--



Short explanation of changes in D3.2:

Section 1:

More discussion about the LUISA use-case and a comparison with other non-semantic annotation (editing) tools

Section 2:

Overall improvements on the language and clarity

Section 9

Glossary of terms updated with more terms

EXECUTIVE SUMMARY

This deliverable provides a specification of the Annotation Profile Model. The Annotation Profile Model is a platform and language independent information model that describes how to configure or automatically generate graphical user interfaces for form based editing of RDF metadata. The Annotation Profile Model is divided into the Graph Pattern Model and the Form Template Model, where the former is responsible for matching and expressing RDF graphs while the latter is responsible for presentation issues such as order, layout and style.

Furthermore, this deliverable specifies how existing expressions in schemas and ontologies must be considered when constructing annotation profiles. Possible syntaxes for annotation profiles are discussed but it is established that there will not be a single dedicated syntax. A mapping from the RDF query language SPARQL is briefly outlined, as well as how a possible editing extension to the Fresnel Display Vocabulary could be used.

Document Information

IST Project Number	FP6 – 027149	Acronym	LUISA
Full title	Learning Content Management System Using Innovative Semantic Web Services Architecture		
Project URL	http://www.luisa-project.eu		
Document URL			
EU Project officer	Kypros Kyprianou		

Deliverable	Number	3.2	Title	Annotation Profile Specification
Work package	Number	3	Title	Learning Object Annotation

Date of delivery	Contractual	28/02/2007	Actual	30/09/2007
Status	Version 3.0, dated 31/08/2007		final <input type="checkbox"/>	
Nature	Report <input type="checkbox"/>	Demonstrator <input type="checkbox"/>	Other <input checked="" type="checkbox"/>	
Dissemination Level	Public <input type="checkbox"/>	Consortium <input type="checkbox"/>		
Abstract (for dissemination)	<p>This deliverable provides a specification of the Annotation Profile Model. The Annotation Profile Model is a platform and language independent information model that describes how to configure or automatically generate graphical user interfaces for form based editing of RDF metadata. The Annotation Profile Model does not replace ontologies or schema information, instead it provides complimentary information that is necessary for annotation tools to behave consistently.</p> <p>The motivation for the specification is, first, to make the end user experience more flexible with regards to the set of metadata edited without compromising the usability or requiring further technical knowledge. Second, to allow a more a coherent end-user experience when editing metadata across tools. Third, to simplify for developers to include metadata editing in their tools by providing ready made reusable components/libraries based on this specification.</p>			
Keywords	Annotation profile, Metadata editing, Ontology driven editing, Form generation, Semantic Web, Query Language			

Authors (Partner)	Matthias Palmér (ULL), Fredrik Enoksson (ULL), Mikael Nilsson (ULL), Ambjörn Naeve (ULL)			
Responsible Author	Matthias Palmér		Email	matthias@nada.kth.se
	Partner	ULL	Phone	+46 18 471 62 90

Project Consortium Information

Partner	Acronym	Contact
Atos Origin S.A.E. (Coordinator)	ATOS 	Nuria de Lama Atos Origin S.A.E. c/ Albasanz 12 E-28037 Madrid, Spain Email: nuria.delama@atosorigin.com Tel.: +34 91 214 9321 Fax: +34 91 754 3252
University of Alcalá de Henares	UAH 	Dr. Miguel-Angel Sicilia Information Research Unit University of Alcalá Ctra. De Barcelona, Km 33.6 E-28871Alcalá de Henares (Madrid), Spain Email: msicilia@uah.es Tel.: +34 91 886 6603 Fax: +34 91 885 6646
University of Uppsala	ULL 	Dr. Ambjorn Naeve University of Uppsala Kyrkogårdsgatan 2 C Uppsala Email: amb@nada.kth.se Fax: +46 184-716-294
Open University	OU 	Dr. John Domingue Knowledge Media Institute, The Open University, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom Email: j.b.domingue@open.ac.uk Tel.: +44 1908 655014 Fax: +44 1908-653-169
University Henri Poincaré	UHP 	Dr. Monique Grandbastien University Henri Poincaré Vandoeuvre les Nancy 54506, PO Box 239, France. Email: monique.grandbastien@loria.fr Fax: : +33 383-278-319
Giunti Interactive Labs S.r.l.	GIUNTI 	Dr. Fabrizio Giorgini Giunti Interactive Labs S.r.l. Abbazia dell'Annunziata Via Portobello Baia del Silenzio 16039 Sestri Levante (GE), Italy Tel.: +39.0185.42123 Fax: +39.0185.43347
EADS FRANCE – Innovation works	EADS 	Anne Monceaux EADS FRANCE – Innovation works Avenue Didier Daurat - Centreda 1, Toulouse, 31700, France. Email: anne.monceaux@airbus.com Tel.: +33 561-184-725 Fax: +33 561-187-611

TABLE OF CONTENTS

EXECUTIVE SUMMARY	3
1 THE ROLE OF THIS SPECIFICATION WITHIN LUISA.....	8
2 INTRODUCTION.....	10
2.1 An Example Editor – The Book Example	10
2.2 Motivation	11
2.3 An Example Annotation Profile – the Book Example Continued	13
3 REQUIREMENTS	15
4 ANNOTATION PROFILE DESIGN	18
5 GRAPH PATTERN MODEL – MATCHING AND CREATING DATA	20
5.1 Constraints.....	20
5.2 Triple Patterns	22
5.3 Variable Bindings.....	23
5.4 Default Semantics.....	24
5.5 A Graph Pattern for the Book Example	27
6 FORM TEMPLATE – OUTLINING THE FORM	28
6.1 Form Items.....	28
6.2 Form Model.....	30
6.3 Default Semantics.....	31
6.3.1 Cardinality Restrictions	31
6.3.2 Finding Choices	32
6.3.3 Finding Labels and Descriptions for Form Items.....	33
6.3.4 Finding Labels and Descriptions on Choices	33
6.4 A Form Template for the Book Example	33
7 SYNTAX	35
7.1 Graph Pattern	35
7.1.1 RDF Query Language SPARQL	35
7.1.2 Edutella Query Language – QEL	36
7.2 Form Template	37
7.2.1 RDF Syntax.....	38
7.2.2 XML Syntax.....	38
7.3 Form Model.....	39



8 INTEROPERABILITY WITH OTHER STANDARDS/INITIATIVES	41
8.1 HTML Forms.....	41
8.2 XForms	41
8.3 Fresnel.....	41
8.3.1 The Role of Lenses	42
8.3.2 The Role of Formats	42
9 GLOSSARY OF TERMS	44
REFERENCES.....	46

1 THE ROLE OF THIS SPECIFICATION WITHIN LUISA

The purpose of LUISA is to show how Semantic Web Services can be successfully applied to the area of Technology Enhanced Learning. In practice this means enhancing Learning Management Systems, Learning Object Metadata Repositories, and supportive Learning Object Annotation tools with semantics.

Pragmatically, the two use-cases, industrial and academical, require a substantial amount of Learning Objects to work with. Since the use-cases have their own chosen methodology, subject domain, target group, and workflow, the descriptions of the Learning Objects will differ substantially. The use-cases will use both different parts of the basic LOM ontology and choose values from different domain-specific ontologies. Readily available LOM annotation tools, such as RELOAD¹, are in essence "non-semantic" editors, which are not appropriate for our use, since they allow free-text input in vital positions - instead of the carefully crafted semantics strived for in LUISA. When taking the step toward a "semantic LOM annotation tool", an adaption to the annotation context (the two LUISA use-cases) is required in order to allow end-users to input semantic data based on ontologies that are specific to these contexts. For example, the two LUISA use-case profiles will differ in defaults, mandatory fields, coverage, prerequisites (such as competencies) etc. Of course, providing two distinct implementations of the annotation tool is an option, but this would scale poorly as new use-cases would require new design and development.

As a solution, this report introduces and specifies the Annotation Profile Model from which Learning Object Annotation tools can be configured automatically. Note that the Annotation Profiles Model is not intended to replace ontologies but rather to complement them by providing the missing pieces that are necessary to automatically generate an Annotation Tool that is customized to a specific context. Consequently, in a later report [ref], two specific annotation profiles will be introduced - one for each of the LUISA use-cases.

Observation 1, the Annotation Profile Model are designed for working with RDF [1] metadata while the LUISA technology is centered around WSMO [2] and the corresponding WSML language [3]. However, even though WSML in itself is specified in a dedicated human readable formal syntax it has an RDF exchange syntax. This RDF exchange syntax will be used for internal representation in the LUISA Learning Object Metadata Repositories and therefore also by the annotation tools, consequently annotation profiles can be used without any problems. See also section 5.2.4 in Deliverable 3.1 – State of the art [4] for a discussion on the suitability of using the RDF exchange syntax.

Observation 2, the usefulness of the Annotation Profile Specification is neither limited to LUISA nor to describe only Learning Objects. Despite this generality,

1 <http://www.reload.ac.uk/>



fulfilling the user interfaces needed by LUISA use cases will not pose a problem since, by design, the Annotation Profile Specification provides fine grained restrictions with the sole purpose of being able to target specific user interface requirements.

2 INTRODUCTION

The purpose of the Annotation Profile Model is to provide an tool-independent description of how to *edit* statements in an RDF graph [1] in an end-user manner. Many specific approaches for editing RDF statements exist already, some are specific to singular vocabularies, others make use of specific configurations, RDF schemas [5] or OWL ontologies [6] to adapt to the vocabulary at hand. It is only the latter class of tools – which we will refer to as *configurable annotation tools* – that is the target of this specification.

An annotation profile is meant to be specific enough to allow the user interface of a configurable annotation tool – i.e. a form – to be generated automatically, without further human intervention. This allows very specific user interfaces that aid the end users to edit RDF according to specific vocabularies. A learning objects annotation profile according to the IEEE/LOM standard is a typical scenario where configurable annotation tools are useful. However, there are situations when the task is unspecific – for example, to edit generic, completely unknown RDF – in this situation a configurable annotation tool is not the right choice.

When designing the Annotation Profile Model much inspiration and lessons learned have been drawn from experiences of developing the SHAME library². Initiated in 2002, it has paved the way through numerous feature requests, redesigns, and changes in terminology. It should be noted that the Annotation Profile Model specified in this document deviates in several points from the state of art in SHAME. Some features have been left out since their usefulness does not out weight the complexity they introduce, one such example is allowing arbitrary deep metadata constructs to be edited via allowing loops in the Annotation Profile Model. Other features have been introduced or strengthened, for example, there is a stronger emphasis on using information expressed in schemas and ontologies when detecting cardinality and lists of selectable choices.

This report is organized as follows: The rest of section 2 will provide an example and some motivation for the Annotation Profile Model. Section 3 will list requirements for the Annotation Profile Model with respect to intended user roles. Section 4, 5, and 6 will introduce the Annotation Profile Model formally. Section 7 discusses alternative syntaxes. Finally, in section 8, there is an outlook towards related standards and initiatives and section 9 contains a glossary.

2.1 An Example Editor – The Book Example

Let us consider an example where we want to edit the title, subject, and author

2 The SHAME library is freely available from sourceforge under LGPL, see the projects homepage at <http://kmr.nada.kth.se/shame>

of an RDF description of a book. The following is a simple example of such an RDF description. The example uses Notation 3 syntax [7] without namespace declarations:

```
ex:book/123
  dc:title [ rdf:type rdf:Alt ;
            rdf:_1 "I Robot"^^en ] ;
  dc:subject exvoc:sf ;
  dc:subject exvoc:robot ;
  dc:author [ rdf:type foaf:Person ;
             vcard:FN "Isaac Asimov" ;
             vcard:TITLE "Mr" ] .
```

Figure 1 shows a typical form based user interface suitable for editing the example RDF above. Note that this kind of user interface is typical with regard to exposed functionality. Other user interfaces that expose the same kind of functionality but with a different design or interaction style is certainly possible.

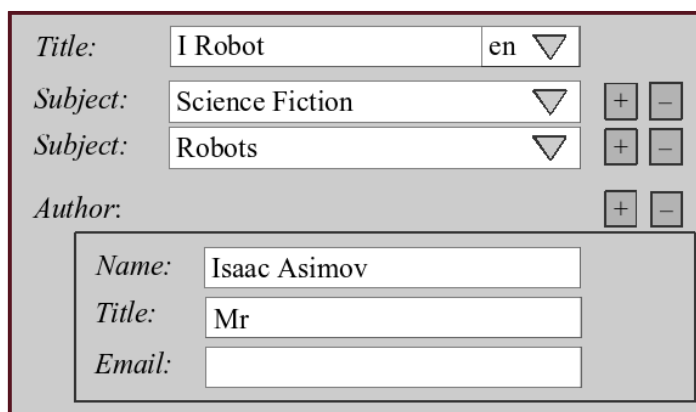


Figure 1: An example form-GUI for editing metadata for a book.

Note that the form-GUI does not correspond exactly to the RDF example as it provides the possibility to edit an email of the author which is not present in the RDF example. This is natural since the form is used for generating new RDF statements, not only editing existing RDF statements. In general, form-GUIs need to allow some fields to be left out, or duplicated according to the needs of the application.

2.2 Motivation

Annotation tools that work with RDF can be divided into three categories according to how adaptable they are in supporting different sets of metadata. First, *fixed annotation tools* have little or no adaptability as they are hard-coded

to support a specific metadata standard/schema. Second, *configurable annotation tools* can be adapted by switching between configurations. Such configurations are typically developed by experts for compliance with specific metadata standards/schemas and/or domain specific extensions. Third, *generic annotation tools* can edit any metadata by more or less exposing RDF to the end-user, in some cases supported by schemas and/or ontologies. In summary:

Table 1: Categories of annotation tools and when they are useful.

Category	Useful for
Fixed annotation tool	Small sets of well defined and non changing metadata with little use of external ontologies and schemas. A few very specific tools can be developed.
Configurable annotation tool	Flexible metadata with much reuse of ontologies and schemas. Many different tools can be generated for specific user roles or domain specific metadata. Good support for reusing and combining ontologies and schemas. Tools can easily be made multilingual. Tools are primarily intended for metadata instances rather than ontology development.
Generic annotation tool	Editing on the level of RDF or underlying ontologies/schemas. Powerful tools for experts.

The motivation for focusing on configurable rather than fixed annotation tools is supported by the following observation made in section 4 of deliverable 3.1, State of the art [4]:

“Annotation tools or editors for metadata are often a part of a larger application and the tool is not always recognized as an independent and reusable component. With the increasing need for interoperability, and the standardized solutions for metadata representation and exchange, the complexity and generic functionality of the annotation task has increased.”

In order to minimize the amount of reinvention (implicitly cost of developing metadata intense applications), speed of development, as well as provide a more capable editing environment, annotation tools need to be independent and reusable components. Furthermore, in order to provide useful and simple metadata editing to end users rather than experts, generic annotation tools is not an option (several such tools exists already). Finally, since ontologies and schemas are increasingly supported by flexible communities rather than big standardization bodies, the need to respond to changes in the ontologies and schemas increases. In this case, there is much to gain by having user interfaces that base their appearance, at least in part, on ontological or schema information rather than having to rely on developers to redesign the tool

accordingly. Hence, the aim is to enable better configurable annotation tools by introducing a common and well defined configuration mechanism, the *Annotation Profile Model*, which can be implemented across platforms and programming languages.

2.3 An Example Annotation Profile – the Book Example Continued

In this section we introduce annotation profiles by means of the book example. The example annotation profile will be able to edit metadata for books according to what was described in section 2.1, and the GUI will be similar to the sketch in figure 1. For the formal introduction to annotation profiles see sections 4, 5, and 6.

First of all, we acknowledge that the annotation profile consists of two parts, the *graph pattern* that captures RDF into variables and second the *form template* that describes the form-GUI and connects it to the variables. First, lets consider the form template expressed in XML:

```
<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 as an 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>
```

Observe that the first level of the form template, the top level group, is not made visible in the form-GUI in figure 1. Each form item provides cardinality

restrictions when the default cardinality is not sufficient, this translates to the availability of the '+' and '-' buttons in figure 1. Every form item references a variable through which data is read and written to the RDF graph. The variables are declared in the graph pattern. The graph pattern can be expressed in restricted versions of existing RDF query languages, below SPARQL is used (the namespace declarations are left out):

```
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 .
       ?A2 vc:EMAIL ?E .}
```

Note that if this query were to be interpreted by a regular SPARQL engine the RDF example above would not match since there is no email given in the RDF data. However, when interpreted as an annotation profile graph pattern, the dc:title, dc:subject, dc:author, vc:FN, vc:TITLE, and vc:EMAIL properties will be considered independent of each other, i.e. as if they were marked as OPTIONAL in Sparql. See section 5.2. for a longer explanation of this.

To be able to generate the GUI shown in figure 1 from this annotation profile, a three step procedure is applied: First, the graph pattern is used to match nodes of the RDF graph and make them easily accessible via corresponding variables. Second, these nodes matched to variables are combined with the form template to form an abstract representation, the form model, of how the GUI will look like. Third, the GUI is generated from the form model in a tool specific manner, e.g. as an standalone java application or part of a larger web application.

3 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. Three different user roles are involved in the editing process: *annotation profile author*, *annotation profile facilitator*, and *end-user*. Even though the same person can act in more than one role, annotation profiles are mainly targeted at simplifying the editing process for end-users:

Table 2: 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

With these different roles in mind, what can and should be expressed in an annotation profile? Let us consider an example:

When editing the title of a resource, it is necessary and natural to provide the title as a character string. But if it is also required to make it explicit that the character string is a Literal and not a URI, we are requiring knowledge possessed by an annotation profile author rather than an end-user. The end-user's knowledge should suffice to provide input according to one predefined *input style* - according to the following table:

Table 3: Listing of value types and preferred way of editing, i.e. various input styles. The superscripts PE, PS, OE, and OS stands for Predicate Editing/Select and Object Editing/Select correspondingly. Those marked with a star (*) are uncommon or strange and should be avoided.

Node Type (in RDF)	Input style (perspective of the end-user)	
	Edit	Select
node is Predicate		
URI node	edit URI ^{PE(*)}	select predicate ^{PS}

node is Object

Resource	blank node	–	select resource ^{OS1}
	URI node	edit URI ^{OE1}	select resource ^{OS1}
Literal	no language tag	edit string ^{OE2}	select string ^{OS2}
	language tag	edit string with language tag ^{OE3}	select string ^{OS2}
	with datatype	edit data ^{OE4}	select data ^{OS3}

The table shows four input styles for editing the object, OE1-OE4, and corresponding input styles for selecting the object, OS1-OS4. Furthermore, in certain situations it is needed to expose the predicate. This makes use of the input styles PE (similar to OE1) and PS (similar to OS1). The end-user's knowledge is in general not sufficient to decide *which* of these input styles to use, or - for that matter - which underlying *node type* it should correspond to. These decisions should already have been made by the annotation profile author and then encoded in the annotation profile.

The requirements on annotation profiles are divided into four categories: *completeness*, *structure*, *interaction*, and *presentation*:

Table 4 □: 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 node 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 the end user is supposed 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, annotation profile-based editors are required to make use of it, allowing annotation profile authors to avoid duplication of information.

4 ANNOTATION PROFILE DESIGN

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. Being language independent also means that we do not provide a single dedicated syntax for representing and exchanging annotation profiles. This allows us to acknowledge the usefulness of SPARQL [8], Fresnel [9] etc. and other already existing syntaxes and focus on how they can express part of the information needed for the Annotation Profile Model. Consequently, by taking the perspective that the Annotation Profile Model can be constructed by collecting partial information from several sources, it becomes natural to reuse existing expressions in OWL [6], RDF Schema [5] etc. See section 7 for a longer discussion on various syntaxes.

As made explicit in the structure requirements 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 the Annotation Profile Model is divided 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 role 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 provides 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 feed back into the graph pattern via the underlying variable bindings. See the figure 2 for an illustration of how of the data flows from the RDF graph to the user interface via the annotation profile.

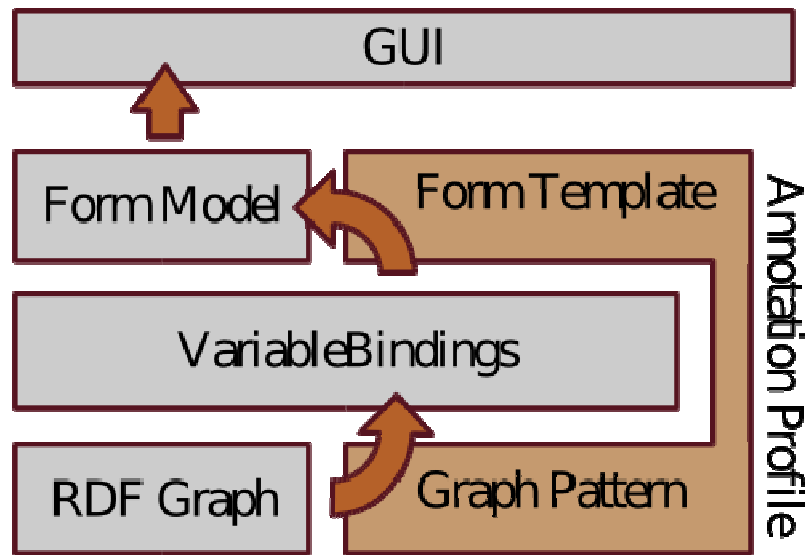


Figure 2: GUI generated from data and an annotation profile

5 GRAPH PATTERN MODEL – MATCHING AND CREATING DATA

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.

A graph pattern is built from *triple patterns* that are connected through common variables. To match an RDF triple each triple pattern consist of a single subject, predicate, and object *constraint*.

5.1 Constraints

There are three kind of constraints used in triple patterns: *constants* for requiring specific RDF nodes, *member constraints* that match any member property in RDF, and *variables* for capturing unknown RDF nodes and giving access to them for presentation and editing purposes. In triple patterns, constants may occur in predicate or object position, member constraints in predicate position only, and variables are allowed in all positions. In table 5, we see a characterization of the three different constraint types according to their allowed attributes.

Table 5: Constraint types and their attributes

Constraint Types	Attributes	Description
Constant	nodetype datatype language tag value	Matches a fixed URI or Literal. A specific datatype or language tag may be required for Literals.
Member	membertype	Matches all the predefined properties used to indicate members of RDF Containers or RDF Collections. E.g. <code>rdf:_1</code> , <code>rdf:_2</code> , etc. for RDF Containers and all <code>rdf:first</code> indirectly via <code>rdf:rest</code> in the RDF Collection construction.
Variable	nodetype datatype language tag identifier	Similar to Constant but makes the URI or Literal available for editing via the variable identifier.



In table 6, we see the permitted values listed per attribute.

Table 6: Attribute and their possible values

Attribute	Attribute Values
nodetype	One of the exact nodetypes UR, BR, OL, LL, DL, or one of the more generic nodetypes AR, PL, or AL if less specific matches are needed:
	UR <i>URI Resource</i> , i.e. an URI identifiable resource.
	BR <i>Blank Resource</i> , i.e. non identifiable outside of the RDF graph
	OL <i>Only Literal</i> , no language tag or datatype.
	LL <i>Language Literal</i> , i.e. literal with a language tag.
	DL <i>Datatype Literal</i> , i.e. literal with datatype.
	AR <i>Any Resource</i> , matches any of the exact nodetypes UR or BR.
	PL <i>Plain Literal</i> , matches any of the exact nodetypes OL or LL.
	AL <i>Any Literal</i> , matches any of the exact nodetypes OL, LL, or DL.
datatype	A URI indicating an RDF datatype.
language tag	String indicating the language tag of an RDF literal.
membertype	CONTAINER or COLLECTION, Indicating RDF Containers or RDF Collections.
value	A string that depending on the nodetype forms the constraint as a URI or Literal.
identifier	An identifier for the variable.

The attribute values are in some cases dependent on each other, e.g. the nodetype may prohibit or enforce language tag or datatype. See table 7 for details.

Table 7: Attribute constraints and comments.

Attribute	Restrictions/comments
datatype	A datatype is required for nodetype DL and forbidden otherwise.
language tag	A language tag is only allowed (never required) for nodetype LL.
value	When the nodetype is UR or AR it forms the constraint URI otherwise it is combined with either the language tag or the datatype to form the constraint Literal. It is not allowed for the nodetype BR.
identifier	The identifier need only to be unique within the graph pattern.

An RDF node matches a constraint if its nodetype, datatype, language tag, member type, and value attributes can be fulfilled (if they are specified). If the nodetype is LL and no language tag is specified the RDF node is required to have a language tag for a successful match, but it is not clear which. This situation occurs when you want to enforce end users to provide texts in one or several languages. The GUI will interpret the nodetype and lack of a language tag as an indicator to provide a list of languages to select among, in addition to the regular text input.

Note that neither RDF Schema nor OWL provides the possibility to prescribe the exact nodetype to use. Hence, to be able to present and edit RDF data created elsewhere in accordance with RDF Schemas or OWL ontologies, the generic nodeltypes AR, PL, or AL are necessary. However, when the graph pattern is used as a template, an exact nodetype is required. Deciding on a exact nodetype to use in these cases depends on where in the graph pattern the constraint occurs. This is part of the default semantics of graph patterns, described in section 5.4.

5.2 Triple Patterns

Triple Patterns consist of subject, predicate, and object constraint, where the constraints are either variables, constants or members. Variables are allowed to be shared between triple patterns by using the same identifier value. It is through common variables that triple patterns form a graph pattern. Triple Patterns may share variables *only* in the following ways, or combinations thereof:

Table 8: Variables can be shared between triples in the following ways.

Variable sharing	Description
Siblings	A variable occurs as subject in several triple patterns.

Predicate chained	A variable occurs as predicate in one triple pattern and subject in the rest.
Object chained	A variable occurs as object in one triple pattern and subject in the rest.

Furthermore, we also require that there must be a *single root* and *no loops*, which ensures that the triple patterns form a tree.

Triple Patterns that have constants in both predicate and object position are referred to as *Constraint Triple Patterns*. We will refer to Triple Patterns that are not Constraints as *Path Triple Patterns*. Finally, given this terminology for triple patterns, we can describe the matching requirements:

Table 9: RDF nodes and RDF triples are allowed to be matched according to the following table.

RDF	Match to	Successful if
node	Constant or Member	If the constant or members attributes can be fulfilled.
node	Variable	If the variables attributes can be fulfilled <i>and all constraint triple patterns where this variable is subject can be successfully matched</i> .
triple	Triple Pattern	The triple patterns subject, predicate, and object constraints can be successfully matched.

Formulated somewhat differently, a path triple pattern is only successfully matched if both its constraints can be fulfilled an all sibling, object, and predicate chained constraint triple patterns to the triple pattern can be successfully matched as well.

It is important to observe that path triple patterns that are siblings are matched independently of each other.

5.3 Variable Bindings

For constraints that are variables the purpose is – beyond participating in constraining the triple pattern – to give access to unknown RDF nodes for presentation and editing purposes. Hence, when an RDF node is matched to a variable the result is captured in a *variable binding*. When triple patterns share variables either as siblings, predicate, or object chained they automatically also share variable bindings.

Matching a triple means matching all the constraints at once. Hence, when

there are several variables, the resulting variable bindings are *dependent* on each other. Variable binding dependencies are based on their position in matched triple patterns. There are two simple rules: First, object position depends on subject or predicate position if it contains a variable. Second, predicate position depends on subject position. As a consequence of the shared variable bindings between triple patterns the dependencies will form a tree, similar to the graph pattern itself is a tree. However, as there might be no or multiple matchings to the same triple pattern the variable binding dependency tree will have some branches missing and some duplicated. Lets consider a simple example:

```
//Two triple patterns:
SELECT * WHERE
  ?X dc:knows ?F
  ?F foaf:name ?N.

//Four triples:
ex:pete foaf:knows ex:bob ;
ex:pete foaf:knows ex:steve ;
ex:bob foaf:name "Bob" ;
ex:steve foaf:name "Steve" ;

//The matched variable bindings, with dependencies indicated via '<':
(?X, ex:pete) < (?F, ex:bob) < (?N, "Bob")
              < (?F, ex:steve) < (?N, "Steve")
```

Without dependencies between variable bindings we would not know if “Steve” belongs to ex:bob or ex:steve. Observe that the dependency relation is transitive.

A *variable binding set* is a set of variable bindings that is the result of matching a graph pattern against a RDF graph. To be useful, a variable binding set should provide a way to discover the dependencies between individual variable bindings. The role of variable bindings as well as variable binding sets are to carry information internally between graph patterns, form templates and form models. Hence, their exact form is to be decided by each implementation.

5.4 Default Semantics

Variables may be more or less constrained. From the perspective of matching RDF data, this affects the amount of successful 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 3, we concluded that end-users should be required to have only conceptual understanding of metadata. E.g. they should not need to choose between optional input types. For the graph pattern this translates to constraining variables in the graph pattern so that every variable corresponds to an exact

node type and specified datatype when the nodetype is DL.

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* to be applied when needed, i.e. when editing. In compliant *graph pattern engines*, the default semantics would be enforced. We have chosen the approach with a default semantics, since it places less restrictions on authors of graph patterns and is less sensitive to lacking constraints.

In the following, a variable in object position will be referred to as an *object variable* and if there is a constant in predicate position in the same triple it will be considered the objects variables *leading property*. If the predicate position is a another variable and if there is a constraint triple pattern that indicates a super property, via the `rdfs:subpropertyOf` property, it will be considered the leading property instead. Furthermore, the variable in subject position in the triple will be considered the object variables *leading subject*. Finally, a variable is said to *match instances of a class* if there is a constraint triple pattern from the variable to the class or any subclass of it with the `rdf:type` as predicate.

Table 10: A graph pattern is constrained according to the following sources with lowest priority last.

Source	What to take into account
Graph pattern	Explicitly given Constraints.
RDFS	<p>On object variables, if a <code>rdfs:range</code> can be found for the leading property it will be used for constraining purposes. If the range is:</p> <ul style="list-style-type: none"> • an <code>rdfs:Datatype</code> – the nodetype is set to DL and the datatype attribute is set to this datatype • <code>rdfs:Literal</code> (or any subclass thereof except those that are datatypes) – the nodetype is set to OL. • any other class – the nodetype is set to AR and a constraint triple pattern is added with <code>rdf:type</code> as predicate and the specified class as object.
OWL	<p>OWL extends RDFS, hence the rules above is valid here as well.</p> <p>Additionally, an object variable is constrained according to a property restriction if it applies. If so, the value of <code>owl:allValuesFrom</code> is used in the same manner as the value of <code>rdfs:range</code> for RDFS outlined above. A property restriction applies on a object variable if:</p> <ul style="list-style-type: none"> • The leading property is indicated by the <code>owl:onProperty</code> and • The subject variable matches instances of the property

restriction class							
If a variable matches instances of a property restriction class with a owl:hasValue , a constraint triple pattern is added on the variable with the predicate being the value of the owl:onProperty and the object being the value of owl:hasProperty.							
Default semantics	Node type of a variable is set to:						
	<table border="1"> <tr> <td>OL</td> <td>If the variable occurs only in one triple pattern in object position, i.e. a leaf</td> </tr> <tr> <td>UR</td> <td>If in any predicate position <i>or</i> only in subject position <i>or</i> only in subject position of a constraint triple pattern. in</td> </tr> <tr> <td>BR</td> <td>if the variable is shared in an object chaining of two path triple patterns.</td> </tr> </table>	OL	If the variable occurs only in one triple pattern in object position, i.e. a leaf	UR	If in any predicate position <i>or</i> only in subject position <i>or</i> only in subject position of a constraint triple pattern. in	BR	if the variable is shared in an object chaining of two path triple patterns.
OL	If the variable occurs only in one triple pattern in object position, i.e. a leaf						
UR	If in any predicate position <i>or</i> only in subject position <i>or</i> only in subject position of a constraint triple pattern. in						
BR	if the variable is shared in an object chaining of two path triple patterns.						

5.5 A Graph Pattern for the Book Example

In figure 3 we see how the RDF graph have been captured into a set of variable bindings via a graph pattern, both the RDF graph and the graph pattern corresponds to the book example introduced in section 2.3. The graph pattern, the box at the top to the right, is illustrated as a graph with nodes and labeled arcs. The nodes and the labels on arcs are constraints, i.e. constants, variables and in one case a member. An arc and its label together with the connected nodes forms a triple pattern, we see two path and constraint triple pattern marked for clarity.

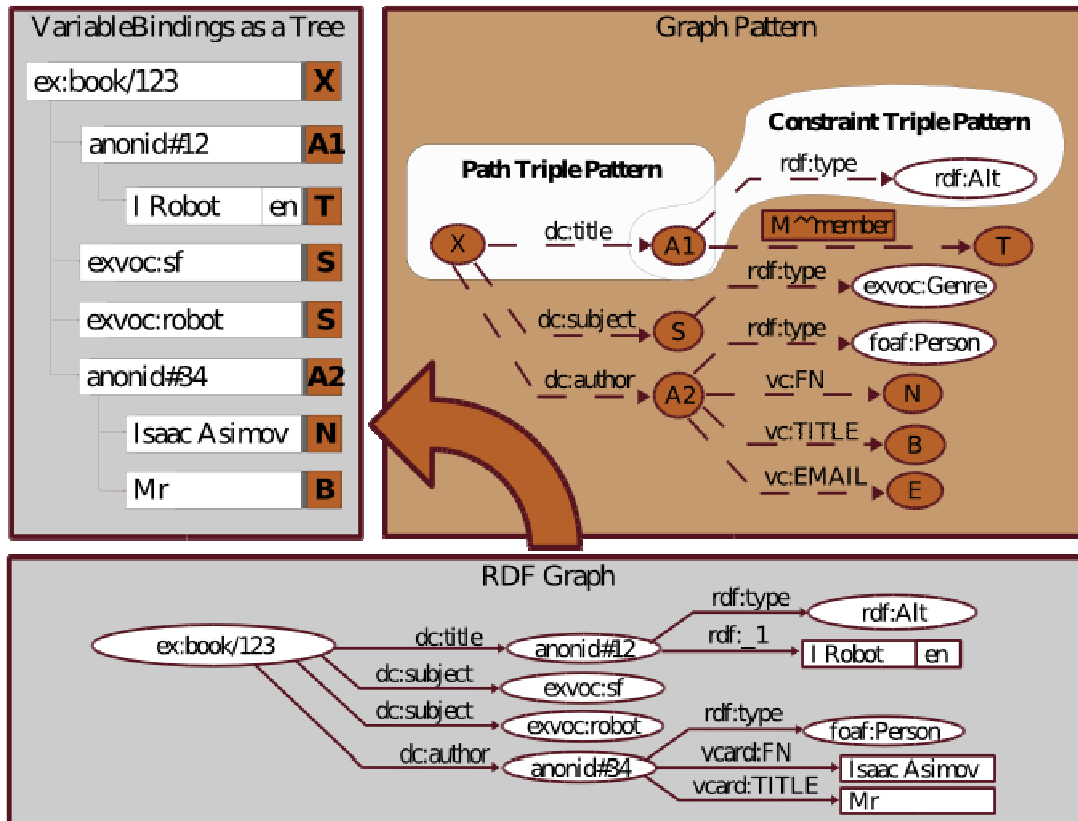


Figure 3: The RDF graph for the book example is captured into a tree of variable bindings via the book example graph pattern

The variable bindings are organized into a tree according to their dependencies. Observe that the RDF graph consists of two subjects but no email. This is reflected in the variable binding by two variable bindings for the variable S and no variable binding for the variable E.

6 FORM TEMPLATE – OUTLINING THE FORM

To build a form based user interface automatically from a configuration we need to identify generic building blocks. For the forms we aim for, the building blocks need to consist of a label, an input field, and some kind of cardinality control. We will refer to these building blocks as form items and together they form the form template. There are three kinds of form items, *group*, *text*, and *choice* form items configured by a range of attributes. Group form items create the tree structure of the form template and corresponds to grouping of information that might or might not have a correspondence in the RDF data, they can be seen as forms within forms. It is seldom motivated to allow group form items to have an input field themselves, instead it is delegated to subitems. Text form items provide the *edit* input style and choice form items provides the *select* input style of predefined values. See the section 3 for the different input styles required.

6.1 Form Items

There are three kinds of form items, described in the table below:

Table 11: Different kind of form items, their attributes and purpose.

Types of form items (FI)	Abbreviated	Description
Text Form Item	TFI	Used for displaying and editing: <ol style="list-style-type: none"> plain text possibly providing a language or a datatype. URI as plain text
Choice Form Item	CFI	Used for displaying and choosing predefined resources or literals.
Group Form Item	GFI	Used for displaying and editing: <ol style="list-style-type: none"> Intermediate resources and their further properties Groups of properties that fit together Group form items accomplish this by having other form items as children.

Every form item are characterized by its attributes. All attributes listed in the table below are optional, with the exception of *variableref* for TFI and CFI unless the value attribute is provided.

Table 12: The form items attributes, what they apply to and their allowed values.

Attribute	Appli	Attribute Values
-----------	-------	------------------

es to

label	All FI	A short text string that will appear first in the form item presentation, may be provided translated into several languages.
description	All FI	A descriptive text that typically appears as a tooltip or is accessible via a help button. It may be translated into several languages.
variable-reference	All FI	A reference to a variable in the graph pattern.
value	CFI, TFI	A predefined value to be presented in this form item. If a variable is referenced, this value is added to the variable binding set.
enabled	CFI, TFI	True if you are allowed to use this form item for editing, false otherwise.
min-cardinality	All FI	The minimal amount of occurrences of this form item in the form model relative to the parent form item.
max-cardinality	All FI	The maximal amount of occurrences of this form item in the form model relative to the parent form item.
preferred-cardinality	All FI	The preferred amount of ready-to-fill-in form item in the form model relative to the parent form item. Hence, when initializing the form model for some RDF data, this is the desired number of form items. If existing data yields a lower amount of form items, additional, empty, form items are generated.
cardinality	All FI	Setting this attribute is the same as setting the min, max, and preferred cardinality to the same value.
choices	CFI	Indicates a set of potential values to choose among. Either explicitly via a list of choices given in a format which is specific to each syntax, or indirectly via a query string which generates a list of choices when executed.
class	All FI	This attribute assigns a class name or set of class names to an element, similar to HTML. The intention is to provide hooks for CSS.
id	All FI	This attribute assigns a unique identifier to an form item, providing a hook for CSS and other purposes.

For choice form items a list of choices is expected to be provided in one way or another. The following attributes are available for each choice.

Table 13: Choice attributes and their possible values.

Attribute	Attribute value
-----------	-----------------

label	A short text string that will be used to present the choice, may be translated into several languages.
description	A descriptive text that typically appears as a tooltip or is accessible via a help button. It may be translated into several languages.
value	A RDF node, i.e. a resource or literal, that this choice represents.

Note that the value attribute always has to be present. If the label is missing, the value, possibly truncated, should be used instead.

6.2 Form Model

A form template is strictly a template, i.e. there is no reference to any specific RDF data. When combining a form template with RDF data (captured into a variable binding set by the graph pattern) we get something that we will refer to as the *form model*. The form model is an abstract representation of how the GUI will be drawn³.

A form model consists of nodes ordered into a tree, much like the form template. Each node references a form item from the form template and optionally a variable binding. A correctly built form model must fulfill the following requirements:

- Every variable binding of the variable binding set must be referenced exactly once.
- A single root node is allowed and it must reference the root of the form template.
- For every child node:
 - If a variable binding is referenced, it must be dependent on all variable bindings referenced higher up in the node tree.
 - The referenced form item must be a child of the form item referenced by the parent node.
 - If a variable binding is referenced its variable must be the same as in the referenced form item.

Observe that we do not require every node to have a variable binding. There is two reasons for this. First, if a group form item is used for purely formatting purposes, there is no variable and consequently no variable binding. Second, if a text or choice form item have a fixed value set via the value attribute there is

³ Following the well known MVC pattern the form model corresponds to the model part, hence the name.

no variable and consequently no variable binding.

The above requirements do not take into account minimal, preferred or maximal cardinality as introduced in section 6.1. The reason is that there are no suitable way to enforce such restrictions without arbitrary removing or creating values. Instead, cardinality restrictions should be used by the GUI, to remind, forbid or recommend the user on the amount of values provided before ending the editing session.

6.3 Default Semantics

For some of the information provided as attributes for the form items there are alternative sources, i.e. RDFS, OWL as well as a default semantics when possible. To be able to draw information from RDFS or OWL we need to identify the relation between a form item and an RDFS or OWL expression. This relation goes via referenced variables, their *leading predicate*, *leading subject*, as well as the classes the variables *match instances for*. See section 5.4 for a definition of these terms.

In the subsections below we go through alternatives for finding out form item cardinality, lists of suitable choices, and finally labels and descriptions on both form items and choices.

6.3.1 Cardinality Restrictions

Cardinality can be specified in the form template, drawn from OWL ontologies. If not specified, the default semantics is applied, see table 14.

Table 14: Alternative ways to find out Cardinality restrictions for form items with the lowest priority last.

Source	Cardinality restriction on a form item
Form template	According to attributes cardinality, minCardinality, maxCardinality, and preferredCardinality.
RDFS	RDF Schema does not provide any cardinality restrictions.
OWL	For a form item that references an object variable, cardinality restrictions are detected in two ways: First, if the leading property is a functional property the maxCardinality is fixed to 1. Second, if the leading subject is an instance of a property restriction class restricting the leading property, the corresponding owl:cardinality, owl:minCardinality, owl:maxCardinality are used.
Default Semantics	minCardinality 0

maxCardinality	infinity
preferredCardinality	1 for Text and Choice, 0 for GroupFormItems.

6.3.2 Finding Choices

When detecting choices for a choice form item there are three cases, fixed list, query string or graph pattern constraints, see table below for details.

Table 15: Choice lists can be generated in several ways, lowest priority last.

Case	Choice detection mechanisms for a choice form item
Fixed List	A list of choices are given via the 'choices' attribute on the form item.
Query string	A query string expressed in some query language, e.g. SPARQL or FSL (Fresnel Selector Language [10]), is given via the 'choices' attribute on the form item. When executed against an RDF graph the resulting matches constitutes valid choices.
Graph pattern constraint	If the form item has a variable and the variable occurs as subject in constraint triple patterns in the graph pattern these constraint triple patterns can be used to generate a query. When this query is executed against some RDF graph the resulting matches constitutes valid choices.

For query strings and graph pattern constraints the RDF graph executed against may be OWL ontologies, RDF Schemas or plain RDF graphs that are bundled with the annotation profile or other RDF data sources provided by the annotation tool.

6.3.3 Finding Labels and Descriptions for Form Items

First of all, the label and description attributes on the form items are considered. Second, for form items that references object variables, the label and description can be taken from `rdfs:label` and `rdfs:comment` on the object variables leading property if stated in RDFS or OWL. Moreover, properties that are declared as OWL annotation properties could be useful, but are left to implementor to decide if and how.

6.3.4 Finding Labels and Descriptions on Choices

If the choices are given in a fixed list, the labels and descriptions, if there are

any, are given explicitly. If choices are found through query strings or graph pattern constraints the labels and descriptions should be detected from OWL ontologies or RDF Schemas. In these cases, the label and description can be taken from `rdfs:label` and `rdfs:comment` on the choice resource. Other annotation properties may also be considered but if and how are left to the implementors to decide.

6.4 A Form Template for the Book Example

In the figure 4 we see how the form template of the book example transforms a set of variable bindings to a form model. The form template (shown the box on the right) contains a tree of form items where the most common attributes are displayed in a compact format. Every form item is shown as a box where the type, e.g. GFI, TFI, or CFI is shown on the left, the referenced variable is shown to the right and in the middle are the label, description and cardinality attributes shown.

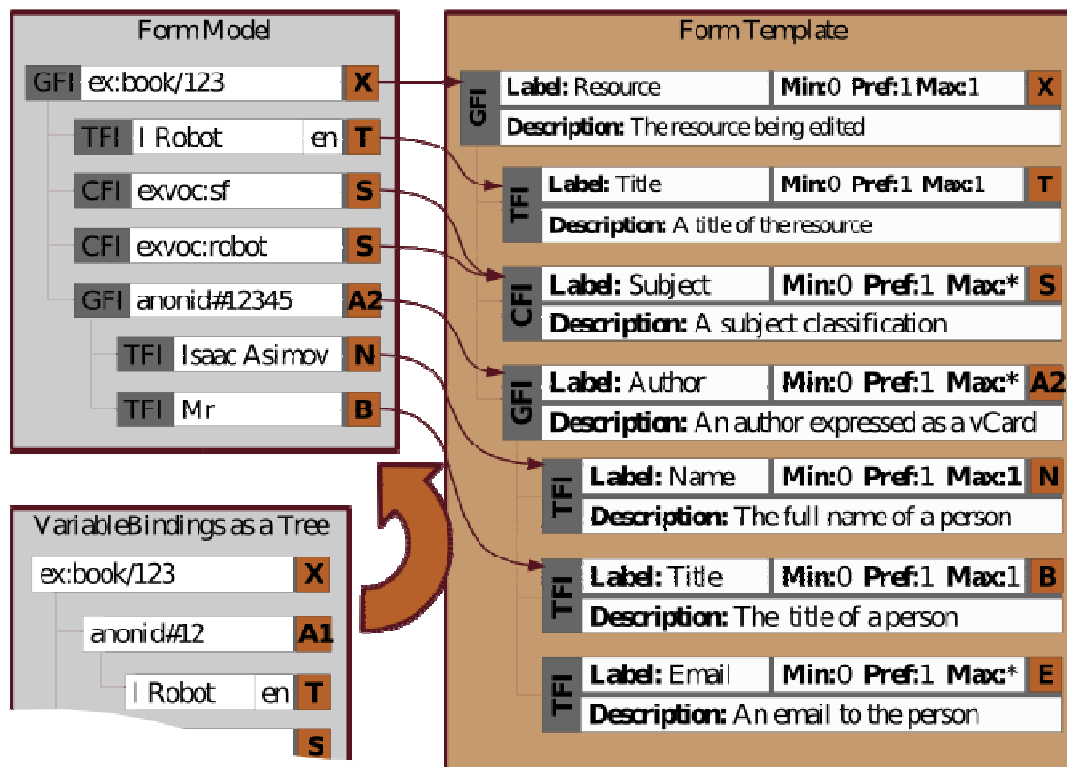


Figure 4: Variable bindings for the book example is transformed into a

At the bottom to the left we see part of the variable binding set (as a tree) that was the output of the graph pattern binding to the RDF data, see figure 3 in section 5.5. The form model is generated from the form template by duplicating or removing form items according to number of variable bindings for the variables referenced. The references to the original form items in the form



template is kept so that the presentation code, e.g. the GUI, can make use of labels, descriptions etc.

Observe that the more complicated title expression, behind an intermediate resource captured in the A1 variable, can be hidden by simply referencing the T variable directly. This is not always preferred, e.g. the author construction cannot be flattened as it would destroy the possibility of adding several authors (which requires the '+' and '-' buttons to be available on the author resource rather on the authors name, title, or email). It is also possible to include virtual groupings that does not correspond to intermediate resources (simply by not referencing a variable). But this is not shown in this example.

7 SYNTAX

This document does not endorse a single syntax, rather it gives examples of some syntaxes that can be used to implement the form template and graph pattern constructs.

7.1 Graph Pattern

As mentioned in section 4 we will not introduce a dedicated syntax, instead we will see how two query languages can be restricted to fit our needs.

7.1.1 RDF Query Language SPARQL

The SPARQL triple patterns corresponds nicely to the annotation profile triple patterns. For nodetype restrictions of the variable the following SPARQL FILTER restrictions will be used:

Table 16: Suitable SPARQL FILTERS for the different nodetypes

Annotation profile nodetype	Sparql FILTER expression for a variable A.
UR	<code>isIRI(?A) && !isBlank(?A)</code>
BR	<code>isBlank(?A)</code>
OL	<code>isLiteral(?A) && lang(?A)=""</code>
LL	<code>isLiteral(?A) && lang(?A)!=""</code>
DL	<code>isLiteral(?A) && datatype(?A)!=""</code>
AR	<code>isIRI(?A)</code>
PL	<code>isLiteral(?A) && datatype(?A)=""</code>
AL	<code>isLiteral(?A)</code>

Actual datatype or language restrictions in the expressions above are modified with the provided values. All other SPARQL modifiers are ignored, e.g. other FILTER expressions, OPTIONAL, UNION, other Result Forms, dataset specifications, solution modifiers etc. In fact the OPTIONAL modifier requires some extra clarification. Even though it is ignored, when the SPARQL query is interpret as a annotation profile graph pattern some triple patterns are considered as optional. According to the discussion on default semantic in section 5.4, a specific nodetype is also inferred whenever needed. As an example, consider the book example from section 2.1 again (repeated below for your convenience):

```

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 .
      ?A2 vc:EMAIL ?E .}

```

When interpreted as an annotation profile graph pattern the corresponding expression in SPARQL would be:

```

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 } .
                  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) . }

```

7.1.2 Edutella Query Language – QEL

QEL [11] is a language for matching RDF inspired by datalog. It's built-in predicates correspond quite nicely to the needs of annotation profile graph patterns:

Table 17: QEL predicates and how they should be interpreted

QEL predicate	Graph pattern				
qel:s(S,P,O)	Same as a triple pattern with the three constraints S, P, and O				
qel:nodetype(C, type)	Corresponds to a nodetype attribute on the C constraint. The nodetype constraint 'type' may be one of: <table border="1" data-bbox="523 1899 1367 2033"> <tbody> <tr> <td>qel:Literal</td> <td>corresponds to AL</td> </tr> <tr> <td>qel:Resource</td> <td>corresponds to AR</td> </tr> </tbody> </table>	qel:Literal	corresponds to AL	qel:Resource	corresponds to AR
qel:Literal	corresponds to AL				
qel:Resource	corresponds to AR				

	qel:NonAnonymousResource	corresponds to UR
	qel:AnonymousResource	corresponds to BR
qel:member(S,O)	The predicate corresponds to a triple pattern with a member constraint in the predicate position. The membertype is RDF Container.	
qel:datatype(C, type)	Specifies that the datatype attribute of the C constraint is type, any URI is accepted.	
qel:language(C,"en")	Specifies that the language attribute of the C constraint is "en".	

The other predicates are ignored as well as, the outer join mark, disjunction through multiple rules with the same head, and recursively defined rules. The book example would be expressed as:

```
?(X,A1,T,S,A2,F,B,E)-
  qel:s(X,dc:title,A1),
    qel:s(A1,rdf:type,rdf:Alt),
    qel:member(A1,S),
  qel:s(X,dc:subject,S),
    qel:s(S,rdf:type,exvoc:Genre),
  qel:s(X,dc:author,A2),
    qel:s(A2,vc:FN,F),
    qel:s(A2,vc:TITLE,B),
    qel:s(A2,vc:EMAIL,E).
```

As for SPARQL examples above, namespace definitions have been left out.

7.2 Form Template

Compared with graph patterns there are no corresponding language that can be restricted to work as a syntax for the form template, the requirements are to specific. Instead two purpose specific syntaxes have been developed, first an quite verbose RDF syntax and later a more concise XML syntax.

7.2.1 RDF Syntax

The RDF syntax defines a couple properties and classes (the form items) in the namespace `http://kmr.nada.kth.se/shame/example#form` where the `GroupFormItem` is a subclass of `rdf:Seq`. Hence, nesting form items is done through the regular member operations in RDF containers. Label and descriptions are expressed using the `dc:title` and `dc:description`. The other attributes of form items defined in section 6.1 have their own corresponding properties. Below are the book example expressed using the RDF/XML encoding, truncated for space reasons.

```
<form:Form rdf:about='http://kmr.nada.kth.se/shame/example#form'>
  <rdf:li>
    <form:GroupFormItem>
      <dc:title>
        <rdf:Alt>
          <rdf:li xml:lang='en'>Resource</rdf:li>
        </rdf:Alt>
      </dc:title>
      <dc:description>
        <rdf:Alt>
          <rdf:li xml:lang='en'>The resource being edited</rdf:li>
        </rdf:Alt>
      </dc:description>
      <form:variable
rdf:resource='http://kmr.nada.kth.se/shame/example#X' />
      <rdf:li>
        <form:TextFormItem>
          <dc:title>
            <rdf:Alt>
              <rdf:li xml:lang='en'>Title</rdf:li>
            </rdf:Alt>
          </dc:title>
          <dc:description>
            <rdf:Alt>
              <rdf:li xml:lang='en'>A title of the resource</rdf:li>
            </rdf:Alt>
          </dc:description>
          <form:variable
rdf:resource='http://kmr.nada.kth.se/shame/example#T' />
            </form:TextFormItem>

... abbreviated for space reasons.

      </rdf:li>
    </form:GroupFormItem>
  </rdf:li>
</form:Form>
```

7.2.2 XML Syntax

The XML syntax is very simple with the form items as three different tags with all the attributes described in 6.1 as XML attributes except the label and description which are nested tags in their own right. Group Form items nest other form items within them, and the order specified in the XML is preserved when read in as a form template. See the book example in section 2.1 for an example

7.3 Form Model

The need to provide a syntax for the form model may be questioned since it in many cases it is an intermediate and automatically generated format used only to generate the GUI. Hence, in many cases when the GUI is built directly, it is never serialized into a syntax but only available as an object model in the API.

However, with the increasing popularity of AJAX based solutions the form model need to be transported from the server, where the annotation profile is applied to an RDF graph to the client where the GUI generation is done. A second situation when a syntax for the form model would come in handy is when doing integration with XForms. Having a XML syntax for the form model would allow a specific XForm to be generated for every annotation profile.

The form model was originally considered as a special case of the form template, where the form items of the form template was simply duplicated/removed and complemented with the value from the variable binding. Hence, the syntax for the form model looked the same. However, a cleaner syntax for the form model was required to minimize duplication of information as well as make the separation between the static form template and dynamically generated form model clearer.

A proposed XML syntax is exemplified by the book example below:

```
<FormModel xmlns="http://kmr.nada.kth.se/APtags#">
  <Group vref="X">
    <Text vref="T" lang="en">I Robot</Text>
    <Choice vref="S" choicelistref="1" choiceref="2"/>
    <Choice vref="S" choicelistref="1" choiceref="3"/>
    <Group vref="A2">
      <Text vref="N">Isaac Asimov</Text>
      <Text vref="B">Mr</Text>
      <Text vref="E"></Text>
    </Group>
  </Group>

  <ChoiceList id="1">
    <Choice id="1">
      <Label lang="en">Fantasy</Label>
    </Choice>
    <Choice id="2">
      <Label lang="en">Science Fiction</Label>
    </Choice>
    <Choice id="3">
      <Label lang="en">Robots</Label>
    </Choice>
  </ChoiceList>
</FormModel>
```

The reference between the form model nodes and the corresponding form items in the form template is done by referencing the same variable. The ChoiceList construct may be copied from an explicit fixed list in the form template, generated from a query specified in the form template or found through restrictions expressed in the graph pattern. An XML Schema is in development for this syntax.

8 INTEROPERABILITY WITH OTHER STANDARDS/INITIATIVES

Interoperability with query languages have already been discussed in the section on syntax and need not be repeated here. Instead the focus will be on alternative standards and initiatives for presenting and or editing of metadata in a form based manner.

8.1 HTML Forms

HTML Forms [12] provides the basics for building forms in webpages, although limited compared to XForms, much can be achieved by using HTML, CSS, and server side support. A configurable annotation tool driven by annotation profiles has been proved to work using the Java Servlet technology⁴ that, based on available metadata for the resource being edited, constructs a webpage with a prefilled HTML form. The user edits the data and posts the form data back to the server when finished or when the form need to change substantially, for example when adding or removing fields.

8.2 XForms

The separation of the form template and the form model is in part made to open up for future integration with XForms [13]. The XForms model is any kind of instance XML that is allowed to be updated by the XForms processor according to binding expressions, schema constraints, form controls and java scripts bound to form events. Hence, it would be feasible to use the annotation profile form model as the Xforms model and precompile the annotation profile form template into binding expressions, javascripts bound to events, and a default structure of form controls mixed with for example some flavor of HTML. The form controls could even be modified afterwards if specific appearance where required.

The details of how XForms integration is done remains to be decided upon and proven in practice.

8.3 Fresnel

Fresnel [9] is a browser-independent display vocabulary for *presenting* RDF models. Let us shortly introduce the basics of Fresnel. The following is quoted from the Fresnel Manual [9]:

"Fresnel introduces lenses and formats. Lenses define which properties of an RDF resource are displayed and how these properties are ordered. Fresnel formats determine how the selected properties are rendered by

4 In combination with a template language, JSP (Java Servlet Pages) and Velocity (Apache project) has been tested.

specifying RDF-specific formatting attributes and by providing hooks to CSS, which is used to specify fonts, colors, margins, borders, and other decorative elements.“

The Fresnel manual also lists five design goals:

- *useful for rendering different output formats like HTML, SVG, PDF, plain text, and others,*
- *applicable across different RDF display paradigms, (nested box-based textual representation à la XHTML+CSS, node-link diagram, etc.),*
- *built on existing Web technology,*
- *extensible for more specialized needs,*
- *easy to learn and use.*

All of these goals are good reasons for why Fresnel is interesting as a syntax for annotation profiles. However, it is only a display vocabulary and will need to be extended to provide editing specific instructions to be truly useful.

Hence, a *Fresnel Editing Extension*, from now on abbreviated FEE, would be highly interesting. Especially if it could be done in a manner which is compatible with annotation profiles. In the following we shortly consider the basics of Fresnel from the perspective of annotation profiles and a potential FEE.

8.3.1 The Role of Lenses

Lenses are suitable for capturing all of the features in the completeness and structure categories introduced in section 3.

In Fresnel there is a class called *PropertyDescription* that is used whenever you need to say something explicit about how a property is used in a certain lens. In many cases you can manage without this class by listing the properties via their URIs directly. From the perspective of FEE, *PropertyDescription* is the natural place to express many of the attributes of the graph pattern constraints, e.g. exact nodetype. If no *PropertyDescription* is used, the default semantics will provide the necessary information.

Features such as selection and order of properties is already supported in Fresnel Lenses via the *showProperties* property. Grouping of properties is not supported in Fresnel but is easily introduced by allowing lenses to occur in the list of properties pointed to by the *showProperties* property (since Lenses are in principle groupings of properties).

Cardinality constraints are also easily introduced into the *PropertyDescription* class mentioned above.

8.3.2 The Role of Formats

The distinction between Lenses and Formats explicitly states that:

“Fresnel formats determine how the selected properties are rendered by specifying RDF-specific formatting attributes and by providing hooks to CSS, which is used to specify fonts, colors, margins, borders, and other decorative elements.”

This is a clear match for the features in the *presentation* category.

Today, the features in the *interaction* category have no correspondence in Fresnel formats or lenses, because the latter was designed for display and not editing. However, the resemblance between the presentation and the interaction category features provides a strong argument for keeping the latter in the format as well.

9 GLOSSARY OF TERMS

Below is a list of terms that are introduced in this document. Terminology or acronyms that are not specific for this document have been left out. For explanations of such terms, the reader is advised to use some other source - such as the online dictionary Wikipedia⁵.

Annotation Profile – An instance of (= configuration expressed according to) the Annotation Profile Model.

Annotation Profile Model – A platform and language-independent information model that is used to express the information necessary for the automatic generation of a Configurable Annotation Tool.

Configurable Annotation Tool – A tool that is capable of changing the metadata it can edit by means of a configuration. An example of such a configuration is an Annotation Profile.

Constraint – There are three kinds of Constraints used in Triple Patterns, *constant*, *member*, and *variable*. Constants are used to restrict triples to fixed values in some position. Members are used to match any RDF member relations in triples. Variables are used to bind captured or fabricated RDF nodes into Variable Bindings, which are later exposed through the user interface of the annotation tool.

Form Item – Items that are arranged into a tree that constitutes the Form Template Model. There are three kinds of Form Items: *group*, *text*, and *choice* corresponding to (1) the non-leaves of the tree, (2) input in the form of plain text, and (3) input from controlled vocabularies.

Form Model – An abstract representation of the form that the GUI of the Configurable Annotation Tool will present when the Form Model is applied to an RDF graph. A Form Model is the result of applying a specific Annotation Profile to an RDF graph.

Form Template – The second part of the Annotation Profile. A Form Template is expressed according to (= is an instance of) the Form Template Model. A Form Template encodes how to generate and work with a Form Model.

Form Template Model – The second part of the Annotation Profile Model. A Form Template Model specifies the appearance of the form GUI of the annotation tool, including order, labels, indentation, and cardinality.

Graph Pattern – The first part of the Annotation Profile, expressed according to the Graph Pattern Model. A Graph Pattern encodes how to match and create RDF graphs. Graph Patterns are often expressed in a restricted form of an RDF query language such as SPARQL.

⁵ <http://wikipedia.org>

Graph Pattern Model – The first part of the Annotation Profile Model. TA Graph Pattern Model has the double purpose of describing how to match existing RDF graphs as well as how to create them. The Graph Pattern Model consists of triple patterns that are built from Constraints.

Triple Pattern – A set of Triple Patterns together form a Graph Pattern. A Triple Pattern consists of three Constraints used for matching RDF triples, one for the subject, predicate, and object position respectively.

Variable – A special kind of Constraint used in Triple Patterns. When the same Variable occurs in several Triple Patterns, it effectively connects the Triple Patterns into a Graph Pattern. Variables are also referenced from Form Items, providing a way to expose the data of the generated Variable Bindings - according to the guiding of a Form Template.

Variable Binding – An intermediate format that is the result of applying a Graph Pattern to an RDF graph. In the next step, when the Variable Bindings are combined with the Form Template, a Form Model is produced.

REFERENCES

- [1] Klyne G., Carroll J., Resource Description Framework (RDF): Concepts and Abstract Syntax, <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>
- [2] Roman, D. et. al., D2v1.3. Web Service Modeling Ontology (WSMO), <http://www.wsmo.org/TR/d2/v1.3/>
- [3] Bruijn, J., Lausen, H., Krummenacher, R., Polleres, A., Predoiu, L., Kifer, M., Fensel, D., D16.1v0.21 The Web Service Modeling Language WSML, <http://www.wsmo.org/TR/d16/d16.1/v0.21/>
- [4] Enoksson, F., Palmér, M., Naeve, A., Arroyo, S., Fuschi, D., Pariente, T., State of the Art: SWS Infrastructure, Annotation, LCMS, http://luisa.atosorigin.es/www/images/comission_deliverables/d3.1_stateoftheart_v1.0_final.pdf
- [5] Brickley D., Guha R., RDF Vocabulary Description Language 1.0: RDF Schema, <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>
- [6] Patel-Schneider P., Hayes P., Horrocks I., OWL Web Ontology Language Semantics and Abstract Syntax, <http://www.w3.org/TR/2004/REC-owl-semantics-20040210/>
- [7] Tim Berners-Lee, Notation 3 (N3) A readable RDF syntax, <http://www.w3.org/DesignIssues/Notation3>
- [8] Prud'hommeaux, E., Seaborne, A., SPARQL Query Language for RDF, <http://www.w3.org/TR/2006/WD-rdf-sparql-query-20061004/>
- [9] Bizer C., Lee R., Pietriga E., Fresnel - Display Vocabulary for RDF, <http://www.w3.org/2005/04/fresnel-info/manual-20050726/>
- [10] Pietriga, E., Fresnel Selector Language for RDF (FSL), <http://www.w3.org/2005/04/fresnel-info/fsl>
- [11] Nilsson, M., Siberski, W., RDF Query Exchange Language (QEL) - concepts, semantics and RDF syntax, <http://edutella.jxta.org/spec/qel.html>
- [12] Raggett, D., Le Hors, A., Jacobs, I., HTML 4.01 Specification, <http://www.w3.org/TR/html4/>
- [13] Boyer, J., Landwehr, D., Merrick, R., Raman, T., Dubinko, M., Klotz, L., XForms 1.0 (Second Edition), <http://www.w3.org/TR/2006/REC-xforms-20060314/>