***Web Forms Generation***

Web forms modelled in RaUL are reusable RDF graphs, each of which is assigned a URI. They can easily be reused, either by directly retrieving them from a remote ActiveRaUL service, or by deploying a RaUL Web form model in a local ActiveRaUL service. To support the developer in defining a RaUL Web form model, we have extended the ActiveRaUL service with a deployment endpoint that upon invocation generates a Web form from an arbitrary user submitted ontology.

Here, we briefly describe the algorithm implemented in the ActiveRaUL service. The most challenging part in creating a web form from an ontology is the mismatch between the graph nature of the ontology and the tree structure of a Web form as mentioned in the introduction. For example, it is impossible to display all the properties of directly or indirectly related classes on a Web form, as this may introduce infinite loops, because of the graph structure or renders a unusable Web form due to the number of allowed properties. Thus, we have introduced some termination conditions to stop the algorithm, either by determining the ranges of the property path or by the type of OWL property restrictions (see below).

The ***generateWebForms*** function takes as input a domain ontology O(including imported ontologies) submitted by the user. The algorithm queries for all domain classs Od.C** and those classes of imported ontologies (Oi.C)*,* that are range for some domain properties *Od.P(a,b) &&b:Oi.C)*. This filtering is introduced to reduce the complexity of web forms, for the naive users and make it the responsibility of reasoner to reason about class and properties hierarchies from user provided domain instances.

***generateWebForms Algorithm***

|  |
| --- |
| ***Input:*** Domain Ontology O(N) = {O1, O2, …, ON}  ***Initialisation:***  Construct class set (Ci) ={Ci | Ci Od.C*OR (Oi.C Od.P(a,b) b:Oi.C )*}  ***FOR*** class C***i***** ***do***  *Traversal history map <k,v> ={}*  ***modelOntologyclass(****Ci* ***,*** *“”* ***);***  ***END FOR*** |

Next for each class we create a WidgetContainer in the RaUL model and add the appropriate form element fields inside that container via the ***modelOntologyclass*** *function*. For each generated web form we keep a map *****for already traversed class URIs to avoid duplicates in the user interface.*

*The* ***modelOntologyclass*** *procedure takes a class Ci  for which a Web form will be created and the property 'P' for which this class is a range. All the properties that have Ci as its domain are recorded along with their ranges in another map i <pj, rpj> where pis a property of C*i *and rpj is the range of property pj.  rpj is a class URI if pj is an object property or an xsd datatype if its a datatype property. We add this map to the initial map* ****where *Ci is key and i is its value, to record the traversed class URIs that would be used only if modeling a class properties on this web form would cause cycles other than detected through OWL properties restrictions. Next we process each entry μj of the map i to create a RaUL WidgetElement inside the WidgetContainer. To determine the actual type of the WidgetElement we first* distinguish between datatype properties and object properties. For each datatype property, if the xsd datatype of the range *rpj i*s boolean, radio buttons are created for boolean options, else a form control instance of type textbox is created for each xsd type contained in *rpj for this property*.

***modelOntologyclass procedure***

|  |
| --- |
| ***Input:*** Domain class Ci , Property P| P(x,y) * y:Ci*  ***Initialisation:***   * + Constructor class Ci properties and range map i <pj, rpj>   + *.put(Ci , i )*   ***FOR*** *entry μj i <pj, rpj>* ***do***  ***if***  *μj.**pj is datatype property*  ***if*** *μj.rpj is boolean*  *radiobuttons*  ***else***  *textbox of xsd:datatype=μj.rpj*  ***else*** *μj.pj is object property*  *link to add existing instances of μj.rpj from KB*  ***if*** *μj.rpj has instances*  *listbox L(Ik) = {Ik | Ik : μj.rpj)}*  ***else if***  *Pj is inverse Of P*  *drop this property*  ***else if***  *(Pj is transitive)||(Pj is symmetric)*  *textbox for instance of type μj.rpj*  ***else if (*** *listProperties(μj.rpj).size <1 ) ||*  *( .containsKey() == true)*  *textbox for instance of type μj.rpj*  ***else***  ***modelOntologyclass(****μj.rpj* ***,*** *pj* ***)***  ***end if***  ***end if***  ***END FOR*** |

*For each Object property we want to present the user with links to existing instances of rpj (class URI in case of an object property) in the Web Form, by creating a list of existing instances that can be explored via a browsing interface. To add new value for object property users are provided different types of widget elements on the basis of OWL property restrictions or range class. If property pj has range rj and rj has instances defined in the local ontology then a RaUL ListBox is created for user whole list elements are all the instances of type rj*

*Next, we handle different OWL properties restrictions. They are important for two reasons 1) to create a satisfiable Abox 2) to convert the graph structure of the ontology to a compatible tree structure for web forms. If property pj is owl:inverseOf P, we drop this property because its inverse property has already been handled by algorithm while modeling some previous classs. If property is owl:TransitiveProperty or owl:SymmetricProperty then we create a simple textbox to add a new instance of*  *rj . We stop processing the properties of range rj to break the loop. Further, if rj has no property with it or rj is already in map* ****(which means that range for the property is already traversed). We create a textbox. Finally, if there exists some *properties whose domain is rj, we create a widgetElement Group or Dynamic Group and add new widgetElements inside it according to the same modelOntologyclass procedure. This recurive procedure is called until all the object properties meet some loop termination conditions.*

*We have deliberately omitted the logic of owl:FunctionalProperty from algorithm for brevity. By default, users are allowed to add multiple values for a property by pressing a + button if it is a non functional property which actually results in a change of the Web form model after submission as well as the data provided. Similarly a user can modify a web form by eliminating a property from the model with '-' sign shown next to '+' button.*

***Visualisation of generated Web form using the RaUL JavaScript library***

*The ActiveRaUL Web service can be invoked by any REST client, including, of course, Web browsers. However, for HTTP requests beyond the GET and POST requests supported by HTML forms in Web browsers, for example, to submit an arbitrary ontology file (or a reference to one) or for submitting data through an already generated RaUL Web form, our RaUL JavaScript library is required.*

*When a developer invokes ActiveRaUL (via the JavaScript library or a RESTful client like curl) to generate a Web form-based user interface for an arbitrary ontology, the service returns a URI to the newly generated user interface. When retrieving this URI through a browser, an HTML Web form will be generated that uses the RaUL JavaScript client library for some dynamic behaviour, the parsing and the server communication via HTTP requests.*

*Submitting the SSN ontology from our use case to ActiveRaUL yields the user interface shown in figure. XYZ. User interfaces generated from ontologies involving many classes are divided on a per class basis, whereby a Listbox is created that lets the user choose for which class he wants to create an instance.*

the form elements are displayed based on direct and indirect links to this class specification from its ythe

he user can create and edit class and relation instances in a simple Web form and transparently submit these changes through the RaUL client library in the browser.