Jezabel: an RDF-driven web interface[1]

Tim Pierce
Information Infrastructure Laboratory
HP Laboratories Bristol
HPL-2002-317
November 20th , 2002*

RDF, semantic web, HTML, web interface design

RDF is designed to provide machines with a way of understanding information; it is not a human-readable format. As part of the building of an RDF-driven application for supporting W3C working group processes, we investigated how the information in the RDF database is used to create a web interface. Further, we used RDF to actually describe the appearance of the interface, providing a declarative approach to interface design and maintenance.

* Internal Accession Date Only Approved for External Publication
[1] Enquiries concerning this report should be directed to Andy Seaborne email: Andy_Seaborne@hpl.hp.com

Semantic Web Actions Database Project

13/9/02

The Jezabel Project

Tim Pierce

Jezabel Project: Tim Pierce

tmp99@ic.ac.uk

Background to the project

Teleconferences

The w3c is a truly international organisation, with people working from all over the world on projects together side-by-side. The great dispersion of people means that traditional meetings are out of the question, so to keep personal and group discussion going, teleconference meetings are used, where groups all around the world 'call-in' at a certain time and all can speak to one another.
`url: http://www.w3.org/Consortium/Process/`

To make the meeting work, keep things on track and keep order, a chair is required to chair the meeting. The chair will be responsible for a particular group, and will check on how work in progress is going, will bring up issues that need to be discussed and try to resolve problems etc., as well as keeping meetings to the point and trying to get all important issues discussed in the time period set aside for the meeting.

The chair therefore needs a way of knowing who is meant to be doing what, problems that have arisen that need to be discussed and a way of keeping track of resolution of problems and work finished. Other discussion usually still goes on in a shared mailing list of the group.

In particular the group I was working with studied the w3c rdf-core working group.

Actions, agendas and issues

In the w3c, things people are asked to do (or technically things people make a promise to do) are known as 'actions'. Actions are always assigned to a person or persons, they represent an immediate generally short-term task to be undertaken. Examples of actions could be things such as "Jon, write document on DAML types", "Aylene, review program submitted by Felix last week" or "Chris, give Leo at the pet-shop a call".

Problems, areas that are potentially the group's concern and simply areas that require group-discussion are usually referred to as 'issues'. Issues may be longstanding and very broad problems. They apply to the entire group, with no clear individual or individuals tasked with solving them (unlike actions). Examples of issues might be a question about whether the current work in progress is in keeping with the original goals of the group, problems with old parts of the group's work, disagreements in approaches between individuals in the group which need to be resolved and so on and so forth.

Actions are a useful tool for the chair or manager of a group to keep track of things that people are doing or should be doing and things that have been done. Actions generally go through some process of being created, being

worked on, being said to be complete (by the assignee of the action) and being accepted to be complete by the chair or manager. They often exist purely on a sheet of paper on the chair's desk or more professionally in simple databases. It will often be up to people assigned actions to make note of them for themselves, and actions are most often given out during a meeting – where they can be discussed and accepted, and further discussion and reports of completion etc. are usually then continued through email or at the next meeting.

Issues are stored in a similar way to actions and are problems that the manager or chair should address during a meeting. Issues like action have a definite life cycle where they are created, are active issues, and from there are reportedly fixed, decided not to be a problem, decided to not be in scope for the group or to be a problem that is not going to be fixed now but should probably be re-examined in future. A great number of issues exist at any one point, and for a partiuclar meeting, the issues going on are usually snap-shotted into an agenda.

An agenda is associated with a meeting and is an order of points that the chair should cover during the meeting. It often starts off with a roll-call, and after any other basics to starting the meeting will cover all of the currently open issues. Generally an issue will either be dismissed in one of the many ways described above, accpeted as solved, or will have an action created designed to solve it and be kept open. Open issues that have not been discussed before and are decided to be followed up therefore will often be resolved with an action when confronted in the meeting. Open issues that have already been discussed will often have actions assoctiated with them that upon completion should solve the issue. Thefore the chair will initiate discussion into 'un-actioned' issues and will check on progress of actions for currently existing issues.

Goals of the project

The goals of the project were firstly to examine the application and use of rdf and secondly to aid the process of group managing as regarding tele-conferencing and the management processes of actions, issues and agendas.

Oppertunities for automation

A number of areas that could quite obviously be improved with computerasiton/ automation as regarding the working of workgroups were found and briefly were as follows:

The action process:
Currently often existing only as a paper list and supported tenuously by emails in the working groups mailing list, a more rigorous definition of an action would be useful as well as tools to make records and management of them more easy and more accessible would be useful. For example being able to

quickly ‘find’ actions for any particular group or any particular person would be useful and is currently not remotely possible.

The issue process:
Issues suffer much the same problems as actions. they follow a similar life-cycle and are ultimately open to the same possibilites of aid by management and global viewing tools etc.

Roll-call
Roll-call is a procedure to discover exactly who is present at a meeting, and equally importantly who isn’t. It’s a simple procedure where names are simply read out one after another with people responding appropriately, however despite its simplicity it takes a long time – at least 5 minutes (and in an hour meeting that is almost 10% of the time). Generally in meetings there will be a few people who ‘phone-in’ to the teleconference with their own personal telephones. There is obviously scope here for mapping incoming phone-numbers to names, in helping automate the process to quicken the roll-call. Unfortunately a large proportion of the attendees of the meeting call in a small group from a group phone, with easily upwards of 8 people around any particular phone ‘hub’. Another area here is what is known as ‘regrets’, where a person will email the chair before a meeting to let them know that they will be unable to attend. This is time-consuming and in terms of computerisation could maybe even reach levels such as a person being able to email the manager their regret and having an email bot automatically spot the regret and cross them off the ‘automatically generated list’ for the meeting.

The project

My group decided to be concerned first and foremost with the management of actions, and to worry about the other aspects if time allowed. For this we first achieved a minimal action schema in rdf, based upon observations of their use in w3c teleconference meetings(in particular the rdfcore and daml ontology working groups), discussions with the working group members and discussions amongst ourselves. A mind was kept open whilst designing them of issues and agendas and the possibilities in future of combining them.
After initial preliminary observation and definition had been completed, the area that I was set to work on was an application to allow the easy viewing and management of actions. The application was to be web-based so as to allow as much scope for easy access from any part of the world as possible, which meant that a large part of what I had to do was approaching the problem of the manipulation and display of rdf action ‘objects’ through the medium of the internet. An rdf database was to be used as the basis of this front-end application in order to store and retrieve action objects in rdf, and this was designed by another member of the group, Alex Barnell. My application would build upon this technology.

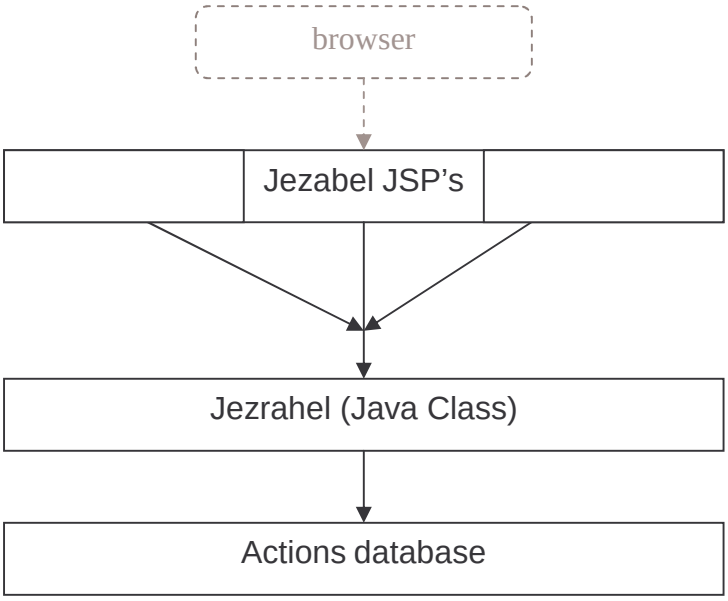
Jezabel

Background

Jezabel is a web-based graphical interface to a database of 'actions'. It must be a front-end allowing the user to create,observe and modify (i.e. manage) a collection of actions. It will be completely internet based, and interaction will be done through web-pages.

It will store data in *rdf*, will use JSP files to navigate and will utilise *SchemaExtension* (described in the next section of this report) interpretted by the *FormProducer* java class, which is a simple *SchemaExtension* interpretter in generating HTML representations of the rdf data

Overview of the Jezabel system



access heirachy in the Jezabel system

All database access by the jsp's is done indirectly through the Jezrahel java class. This is to force all Jezabel database access through a single point – which makes it much easier to maintain the system and also allows complex code to be hidden away from the JSP's which generally need to be maintained and modified frequently.

Browsers interact with the JSP's through the JSP Server (Jetty4.0.4). The JSP files were kept simple, and generally any 'complex' series of instructions that were required for a particular JSP were concentrated into a single method in the Jezrahel java class. No JSP was allowed direct access to the database, in order to force all access through a single database instance in the jezrahel java class, and thus keep tighter control over the program (and make maintainability issues such as changing databases etc. less of an issue (see page 11 – 'maintainability')).

For example code to search for objects in the database of a partiuclar type and return them as an HTML table with each entry hyperlinked to another JSP, passing the URI of the object as a hidden field is encapsulated in a single Jezrahel method.

Style-sheets were used throughout in order to boost maintainability of the system.

Jezrahel

overview
The Jezrahel class provides support methods and database access for the Jezabel JSP files. The *RDFObjectDatabase* class by Alex Barnell was used to store and retrieve all database records.

The JSP's also depended on jezrahel for maintaining their state, which is rather awkward using JSP's. A specially written class called *ObjectIndex* which basically acts as an index, registering Objects with an integer identifier, was used in maintaining the state of different transactions going on at a particular time – since an integer is easy to pass over the get/post method of an HTML file (whereas the Jena Model Object that represents the state is not) whereupon the state Model Object could be retrieved.

Special methods that would return lists of RDFObjects fitting a certain description, with url links that would automatically go to the next correct JSP were used. For example: here using a list command in Jezrahel, the ListActions.JSP file asks that all RDFObjects with *rdf:type* action be drawn, with URL links to DisplayAction.JSP, with the uri of the RDFObject passed over as a hidden field.

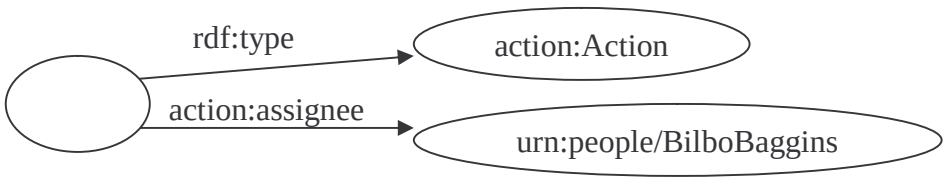
RDFObjects
RDFObjects as implemented by the RDFObjectDatabase class mentioned above correspond roughly to the concept a unit of information describing an individual of some description. It is a subsection of an rdf-graph, and will have one resource as a principle 'identifier'. An action, a person, a meeting, an agenda are all examples RDFObjects that appear in the rdf-graph of the action database. Qualitatively, an RDFObject can be fairly well imagined as an instance of an rdf-schema, e.g. an instance of a v-card. Most accurately an

RDFObject is a sub-graph of an original graph that starts with a principle resource in the original graph and has all the same properties and resources that the original resource has, but in this sub-graph the properties and objects connected to these ‘second generation’ resources will not be included. I.e. only one level of properties and objects from the principle resource in the original graph will be included in the RDFObject sub-graph. One exception is made, when one of the non-principal resources of the RDFObject is a blank resource, the properties and associated objects of the blank resource will be included in the RDFObject sub-graph. since 2 distinct statements with the same subject and property both going to a blank-node will be indistinguishable except for the further properties and objects attached to these blank-nodes.

In the RDFObjectDatabase, RDFObjects correspond roughly to records in a traditional database. The Jezabel system uses an RDFObjectDatabase for its data store.

Listing

The process of listing resources by *rdf:type* (or other property) became a very common problem. Because of this many listing methods were created within Jezrahel. Because of the ungracefulness of a uri and because indeed resources do not necessarily need to have a uri associated with themselves whatsoever, it is generally impractical to identify the entry of an rdf object in a list by the resource uri of the resource that represents it. Where no other possibility exists (as in the above picture), it is viable to use this method but ideally there will be some property such as ‘Title’ or ‘Name’ or ‘Identifier’ that can be used. Unfortunately no standard identifier property as such exists (although rdfv does suggest such a property (the describeBy property), I did not get a chance to introduce rdfv into the Jezabel system). Generally a ‘match’ resource would be passed, with properties from that resource and their associated objects corresponding to desired properties and objects of the target resources in the database. E.g searching for all actions for Bilbo Baggins a resource along the lines of the following blank resource would be passed:



Generally arrays of ‘frogger’s would be passed to a list instruction as one parameter, from which each array element would then be tried in order and the parts of the listing entry constructed (such as [action-name, assignee-name, date] might generate a listing entry “Rewrite document xyz, Roger, 6/4/2002”),

Frogger class

The Frogger class provided a simple self-contained object and associated set of methods that could record a ‘path’ of property-names of an rdf-graph and which could attempt to traverse a section of rdf-graph following these properties given a specific resource. So-called frogger as it would hop from one resource to the next and so-on.

Quite often you will want to get the family name of the person who is assigned to a particular action. A frogger instance representing the path *['AssignedTo', 'FamillyName']* can be quickly and easily constructed with the frogger class and then a method within the class called with a specific resource as a paramter can return the appropriate resources/literals as an iterator. In this example the name of someone assigned to a particular action can be quickly found. This rdf-path object can of course be easily passed between java methods and classes and thus very general methods requiring only 'Frogger's to be passed can be quickly and easily used. When designing menu's, or creating lists of actions with the action title, and then the name of the assignee to the action next to it, this class was extremely useful.

RDFDatabaseFrogger
When it turned out that each rdf object stored in the database would require a separate 'getObject' call, it was quick and painless to extend the Frogger class into RDFDatabaseFrogger, which simply had the getObject call added into one or 2 extended methods.

ObjectIndex Class
The ObjectIndex class was a class specially written to allow objects of any type to be freely passed from one JSP to another via a browser, which proves more difficult than one might think. especially when faced with issues of multiple users using the same pages etc.

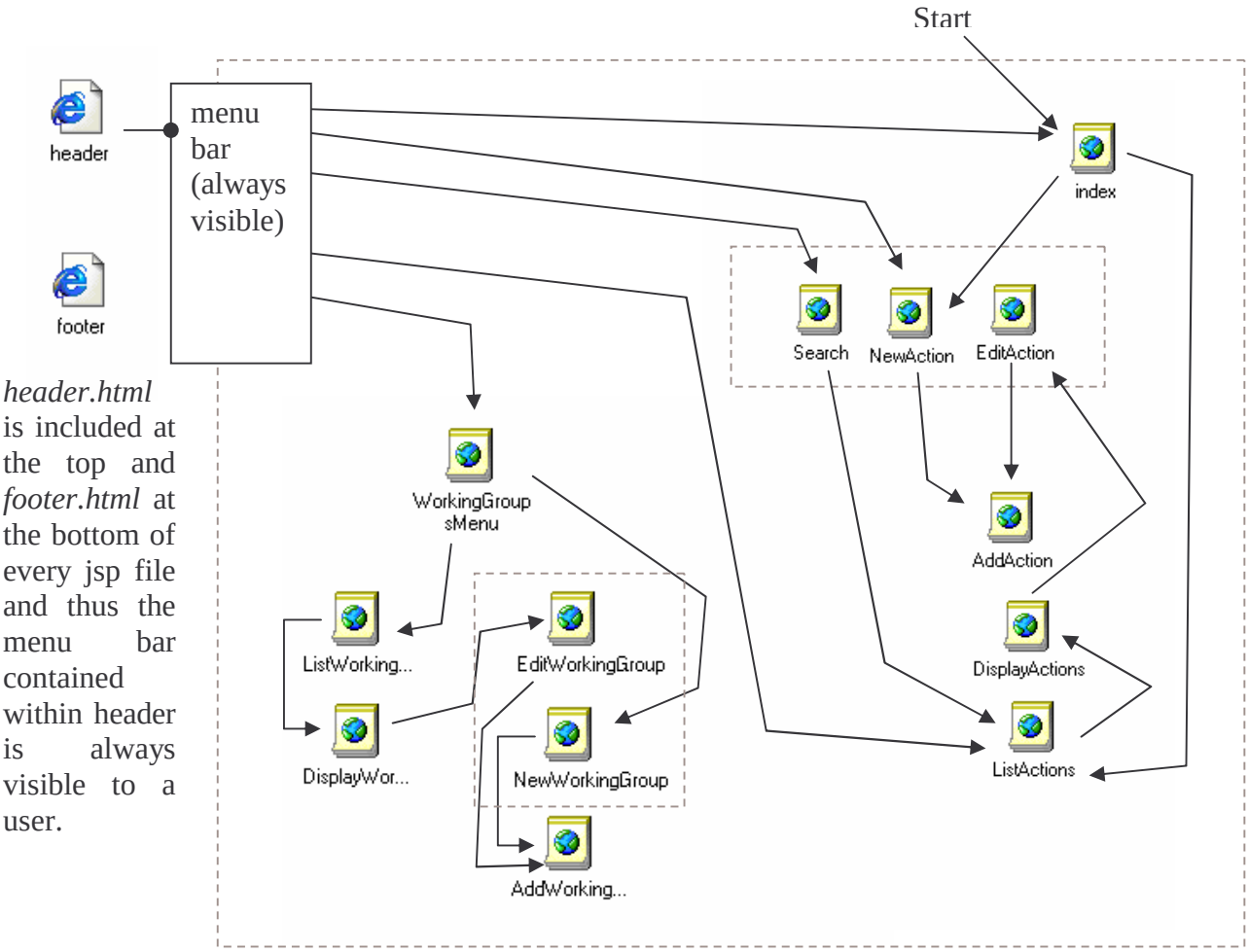
The principle is simply that the ObjectIndex resides in Jezrahel – of which only one instance on the server is ever running, and that the *singleton pattern* was then applied to the ObjectIndex class so that only one ObjectIndex within this Jezrahel class will exist, and thus one ObjectIndex will be shared by every program interfacing with Jezrahel. If an object is wanted to be added to the index it is simply added using the 'Add' method in the class, upon which an integer id value will be returned. If an integer id is passed to ObjectIndex, it will return the java Object corresponding to that id. It is then up to the JSP or other method that requested the Object to check its type etc. and handle it appropriately.

Using this method it is very easy therefore to create a new Action in the index, and return its indexed id to the JSP that called the jezrahel function to create a new action in the first place. This id is then passed on as a hidden field to any other JSP's, which can then simply request its number from the index and retrieve the action, thus maintaining state of complex java objects through a browser.

JSP's

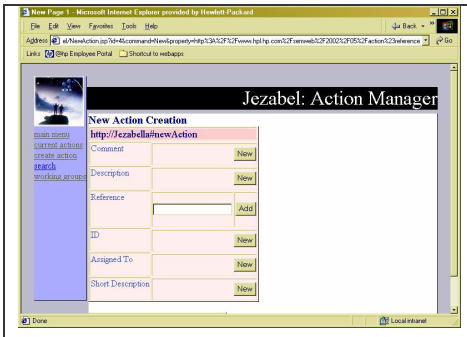
overview
All of the JSP's reside in the *jetty/webapps/jezabel* directory. Any particular page in the browser generally corresponds to a jsp – for example the 'main' front page corresponds to *index.jsp*, the 'list actions' button in this page then corresponds to the *ListActions.jsp* file. Where pages perform very similar operations in the underlying system – for example 'edit action' and 'create new action' for example, sometimes a new JSP was made called something like '*modifyAction.jsp*' – and JSP's such as *NewAction.jsp* and *EditAction.jsp* will barely consist of anything more than defining a String to say "new" or "edit" and then 'including' the modifyAction.jsp. (it is worth perhaps briefly mentioning before the detailed JSP section that every jsp file which is intended to be viewed as a standalone file will firstly have a captilised first-letter, and more importantly always include header and footer files, which are shared across all Jezabel JSP's).

In detail
The layout and interactions of the various JSP files were as follows:

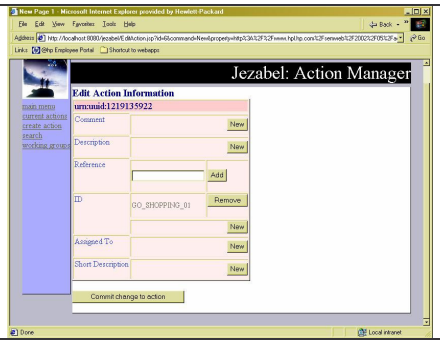


header.html is included at the top and *footer.html* at the bottom of every jsp file and thus the menu bar contained within header is always visible to a user.

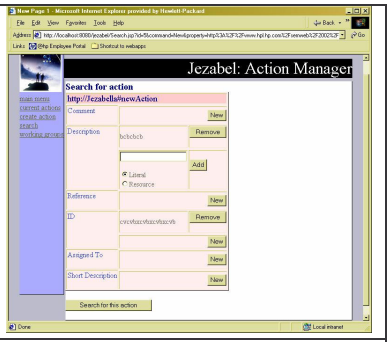
NewAction, EditAction & Search and the modifyAction.jsp



NewAction.jsp



EditAction.jsp



Search.jsp

Because they were so similar in function and implementation, the Search, NewAction and EditAction .JSP files are actually all inheritants of another file called modifyActions.jsp. Basically each file simply defines a String identifying which mode it is in and then includes the modifyActions.jsp file which will be able to act correctly according to the desired mode. The following is the entire contents of EditAction.jsp

EditAction.jsp

```
<%String mode="edit";%>
<%@include file="modifyAction.jsp"%>
```

NewAction and Search are the same format just with the String mode set to different identifying strings.

Each of them after any modification to the initial form has been made allow the user to submit the modified form compeltely.

When a fast addition of being able to apply the same functions as existed for actions on workgroups was required, modifyAction.jsp was copied to *modifyWorkgroup.jsp* and some equivalent files to the above called *EditWorkgroup.jsp* and *NewWorkgroup.jsp* were created in much the same way as above, with minimal modifications to replace references to action types to workgroup types.

ListActions

The ListActions.jsp file would use a call to jezrahel to list objects of type action, representing them by their shortID and the assignee of the action (If neither is present the uri of the action is used as a last resort), each with a hyperlink to DisplayActions.jsp with the uri of the action to display passed over the address line (e.g.“http://jezabel/DisplayActions.jsp?uri=uri:id1000234”).

ListWorkgroups was created as an almost exact copy of this file, changing a few things such as ‘action’ to ‘workgroup’ etc. so that workgroups would be listed and could be displayed with DisplayWorkgroups.

AddAction

It soon became apparent that task involved upon completing the NewAction.jsp or the EditAction.jsp was being able to create or replace an action in the database, and that these two tasks were not inherently that different. Therefore, the AddAction.jsp was created which would take as parameters, a parameter command, which would be set to either ‘create’ or ‘replace’ and a parameter called id, which would be the id of the action to add or replace in Jezrahel’s ObjectIndex.

Index

The index.jsp file was a simple front-page jsp for the system, simply to welcome the user to the site, and provide a few useful links (most of which are also present in the side-menu), and have a common place to return to after compelting other tasks. Things such as ‘news’ and an introduction to the site/system could be provided here.

A nice feature which appears here which doesn’t appear on the side-menu is the ability to look for actions for a particular person. This was done very simply by first using a jezrahel list function to generate a list of all people in the database (by doing a database search on all nodes with *rdf:type* person (as defined in the PersonSchema java file) and generating a drop-down menu from which a person can be picked. Upon initiating the requqest for a list of actions for a person and going to ListActions.jsp, the ListActions.jsp will check if the parameter person has been passed to it, and will create an appropriate listing

of actions if it has. If no people are in the database, or more importantly if the people in the database's resources have not been given *rdf:type* person(the only sure way of knowing a resource represents a person), then the program will miss them. Actions that have had no people assigned to them will not be accessible through this particular feature.

WorkingGroups
A simple menu, similar in appearance to the Index.jsp, and basically acting as the menu for the WorkingGroup functions (create new WorkingGroup, EditWorkingGroup etc.). These functions are not accessible from the side-menu, although this page is. Since WorkingGroup modifications, viewing etc. are unlikely to be used particularly regularly.

IteratorSponge
Very often it is useful to combine the results of two iterators. This class implements iterator, but also has an add method that takes an iterator as a parameter. When another iterator is passed to add, the add method will simply get all of its .next() entries until .hasNext() becomes false, absorbing its Objects like a sponge. Seems simple, but it turns out to be extremely useful when combining the results of multiple queries (resulting in multiple result iterators) whose results are to be processed in exactly the same way.

Post-analysis

Maintainability and re-useability of the system
At one stage, with very short notice I needed to add functionality to allow the same manipulation tools as existed for actions already for workgroups, and couldn't afford to spend too long on this area. Thankfully very little work was needed to add this extra functionality, and the majority of work involved was in changing url's to point to new areas. In the space of just over an hour and a half a fully operational workgroups management section was in place in the site. This was largely thanks to the fact that things had been written to be supported almost completely by rdf-schema's.

Problems with workgroup management
The schema for workgroups (written a few months before) included much use of *rdf:bag*'s, which unfortunately were not directly supported by the the SchemaExtension interpreter. The result was that to truly use bags they would have to be added into the database by an application other than jezabel, or by hand. The rdfv interpreter which was to supersede the SchemaExtension interpreter in the system was to take bags into account, but this stage was never reached. An application to handle bags should be simple and quick to implement.

Bug in search
The search function in the jezrahel contained a bug somewhere that I had to leave to attend to more pressing matters without having a chance to come back and correct it.

People management
A series of forms to allow the easy creation and modification of people would have added much to the value of the system. It was somewhat out of scope for this stage of the project however.

Browser 'back' button vs. ObjectIndex
The ObjectIndex class, which gave the impression of application state being kept in the browser was actually keeping the state in the server and getting the browser to pass an integer as an http-parameter across jsp's which would then be used by the server to generate html using the state informtation it had for the particular integer id.
A slight problem occurred however regarding the browser's back-button. Internet users become very used to being able to just click 'back' and go back to the page they were on just before the current one. The implication to the user, and one that usually does not cause a problem is that the state of the page they are looking at is controlled by the browser. Unfortunately, in reality the browser is in fact simply plucking the previous web-page out of its cache, not truly getting it from the server.
The problem arises then that the user could for example add resources a,b,c,d and e in that order to a newly created action in the browser. The user might then press back twice and the browser will happily display html depicting an action with the newly added resources minus the last two, i.e. after pushing back twice it will seem as though only a,b,c are currently attached to the action. The user can then add another resource, z to the action expecting the next screen to display an action with a,b,c,z. instead however an action with a,b,c,d,e,z will appear, since clicking back does nothing to the actual action object being stored in the server.

Solution?
One solution would simply be to record an enumerated index of transactions performed on a particular action as the state, and then 'render' the transactions each time the object is needed to be displayed. Then as well as an integer id to identify the state, a second id could be passed with each page to indicate the last transaction performed.
Then the stages gone through when adding a,b,c,d,e might be something like:

primary id	secondary id	operation
3	1	add a
3	2	add b
3	3	add c
3	4	add d
3	5	add e

now if the user clicks back twice they will go back to the page with secondary id number 3. Then when they try to add resource z, the browser will pass the secondary id '3' on and the server would be able to render the action as the user expects, looking only at the first 3 operations and then adding 'z' and letting it become the new entry with id '4'. This would make the system behave as the user would expect.

Timeout in ObjectIndex

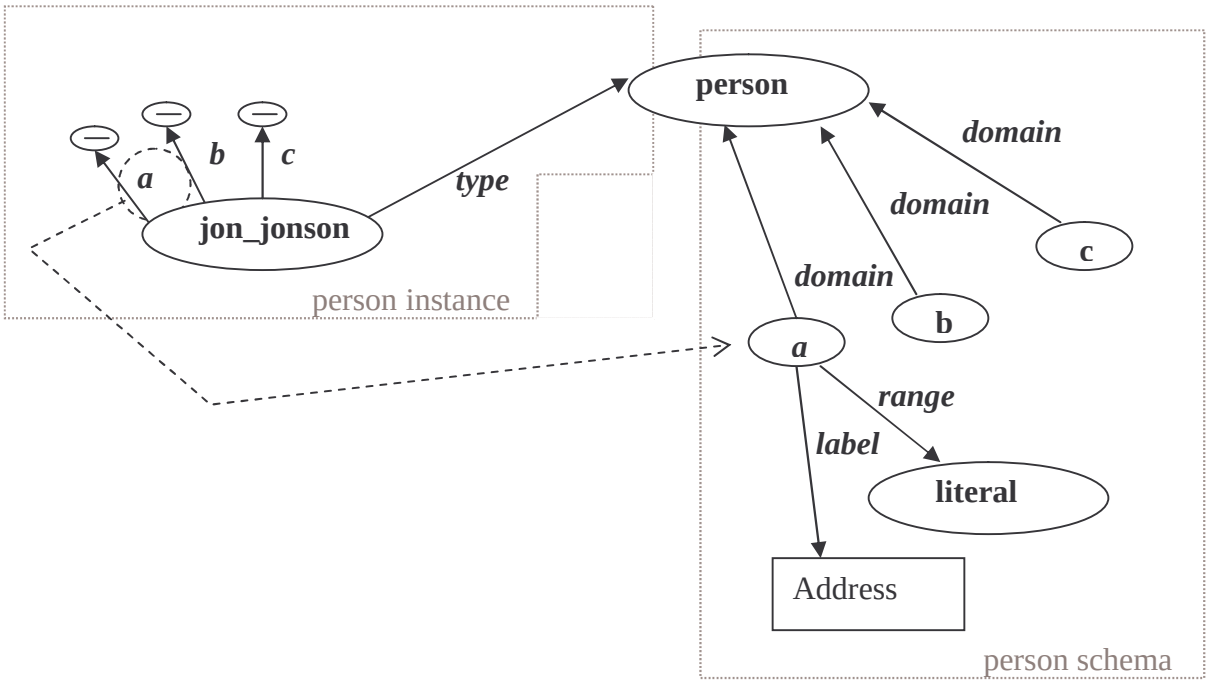
Since browser windows can simply be shut etc. it is very easy to be storing action states etc in the server that simply have no-one using them. The system has no way of knowing for example whether someone is simply taking their time filling in a form or whether the form along with its id has been closed and forgotten about. Small-scale this is not a problem, however potentially over time or with much use eventually these redundant states would put a strain on the server's resources. Therefore a simple timeout procedure such as every hour the ObjectIndex wiping any indexed items that have not been modified in the last 5 hours for example would be a simple fix.

SchemaExtension

namespace: http://www.hpl.hp.com/rdfs/schemaExtension#

Background to the approach

It soon became obvious writing applications to display rdf that *rdfs* rdf schema associated with the type of a particular instance is of great importance when trying to display the instance in a meaningful form. Properties of the schema such as the *rdfs:label* (abbreviated simply to 'label' in the diagram) become indispensible in reconstructing an understandble rdf-encoded rendition of a data instance. For example "http://www.hpl.hp.com/2002/08/example#completeAddress **:** HP, Walls Court Farm" can become "Address : HP, Walls Court Farm" – instantly more useful to the user. Using the types property, it is also possible to for example when creating an application which allows the user to create instance data to automatically be able to see if the data input should be added to the instance as a *resource* or *literal*, or if a choice should be given – such as when no type is defined (taken to mean both *resource* and *literal*) without the need for recourse to 'hard-coded' knowledge in the application.



Useful as these are, *RDFS* gives nowhere near information by itself to create useful or consistent forms. When displaying one implementation of a *vcard* for example, in excess of 30 fields appear that need to be displayed such as: given name, family name, photo-link, addressline1, addressline2, country, company, etc. A problem quickly arises however in that, there is deliberately no notion of ordering of a resources properties in *rdf.* This means that the order fields appear in is essentially random, and examples literally occurred where upon redrawing a form, the order of the fields would change. Since there is no notion of *ordering* this is completely accpetable and correct by *rdf* standards. A much greater problem than

the order changing between one form and the next is simply the ordering of the fields. For example:

addressline2: Filton Road
familyname : Smith
company : Hewlett-Packard
adressline1 : Walls Court Farm
given name : John

is typical example of the kinds of renditions of data that would be created by the '*vcards database'* web-application ###. Simplistic ordering strategies such as listing in alphabetical order obviously do not solve this problem either. Some scope existed – and had been half-heartedly implemented in the vcard schema I am using, by (perhaps incorrectly) subclassing - for example making streetname, state, country, zip etc. all subproperties of the *resource 'ADDRESS'* – which allowed the application to intelligently group certain properties. However the issue of ordering quickly re-emerges in this group, where there is no way to express using *rdfs* for example that conventially with western addresses we display the street-name before the town before the country, or simply (again by western convention) the 'given name' is written before the 'family name'.

There are many more issues connected with the representation of the rdf, and many features which would be extremely useful to have which are not represented in traditional *rdf* and *rdfs* terminology.

One solution is once again to hard-code the application with extra knowledge about the *rdf* data, such as the application knows to look for the *'http://www.randomSystem#title'* property and to draw this at the top of its rdf display window. This is a quick fix solution, and keeps extra information locally which is quite at odds with the 'globalised' and 'interconnected' perspective of the semantic web for which *rdf* was created, as well as turning up a plethora of maintainability and re-usability issues.

Another solution is to add extra information into the rdf-graph – in such a way as will naturally be ignored by any system that does not understand how to interpret it (in the same way that an internet-browser ignores html-tags it doesn't recognise) and have your application understand how to interpret this extra information. This is a much better solution, as it avoids many of the maintainability and re-usability (at least for your local system) issues of the previous solution, with the main problem simply being that anyone unfamiliar with the conventions and namespaces that you have added will be unable to glean the benefits of the additions.

Ideally the graphical representation or layout of the rdf data could be generated completely automatically from the rdf graph.

Therefore I will create a new schema underwhich useful layout information can be represented particularly as relating to the action database (Jezabel) project,

although I will certainly not be trying to create a complete system to fully solve the problem, but examining an approach.

Automaticness

A desire in this project was to have automatically generating forms, that is forms that can be generated from an application with minimum human intervention. Ideally if set-up correctly, things could be as simply as a command such as

DrawForm (Resource r, "uri:ViewX");

with all the information necessary to construct a pleasant, meaningful and clear representation of the data on-screen. I'll commonly refer to this as the 'layout' throughout.

This should not replace existing conventions such as rdfs, but instead augment it.

Also, the view or layout information should be encoded in rdf in such a way that it will not affect applications that do not recognise its schema, and also that if a particular graph has no information for the layout schema at all, then an acceptable representation of the graph will still be constructed.

The concept of views

It seemed from studying the problem that basically what would be extremeley useful would not be a way of saying 'this is how this data is represented' but rather 'this is one way the data can be represented' – and that really many different ways of representing the same data would be useful, or in fact necessary in order for the system(?) to be of any use. To allow this approach to work the idea of a 'view' was coined, which can be thought of as a 'schema' in say an SQL database – but obviously the word 'schema' couldn't really be used.

For example you may want for your 'CD collection rdf database' to have a view called 'simpleView' – which would perhaps just display the CD name and author. You could then have 'advancedView' which would add the date released and perhaps style and other information etc. to the simpleView 'fields'. Now from a slightly different perspective, we may want another view called 'editView' which would automatically mark that all of the fields in our simple CD collection example can be edited.

In a slightly more advanced system, we may wish for example that there is a view for employee data called 'managerEditView' – underwhich the name and address, 'project working on' etc. fields are marked as being modifiable, but the employee's salary is not.

In a more global system, we may wish that for some project information data there is a 'projectWorkerView' view in which it is marked that the 'comment' field should have means of adding information to it, but that no capability for removing existing 'comments' should be there so that people working on the project can easily add

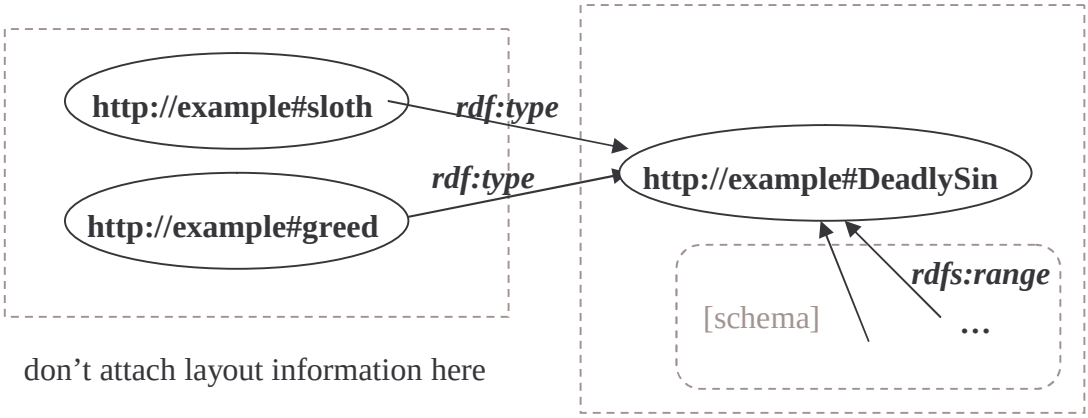
new comments to the rdf of the project, but can't remove existing comments - i.e. more advanced than simply marking something 'editable' or 'uneditable'.

Now that the concept of a 'view' has been put in place, it would seem sensible that since the basis is in rdf that each 'view' be identified by a uri, and that a rdf-type to identify a view be created.

Whereabouts should the view information go?

Now it is obvious that this extra layout information will be stored in rdf, it is necessary to decide where and how it will link to the information in the original graph. Logically and from the point of view of the other projects I'm working on it would seem that it is not practical or sensible to link it to actual instances of rdf data since inherently, rdf data of the same type is generally displayed in the same way. Thus the most sensible option would seem to be to link the extra information to the rdf-schema of the data and with the type resource itself, the connection to the actual data being the *rdf:type* property itself.

So in particular the *rdf:type* resource object connected to an instance of data, and the resources of type property connected to this resource. i.e:



don't attach layout information here

attach the layout information in here

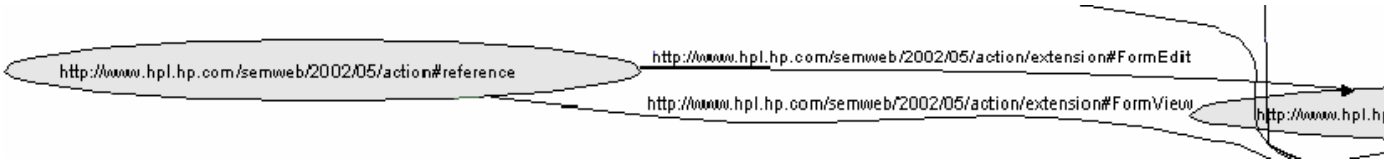
Another benefit of this approach is that the layout information is not attached to an instance of rdf-data directly, but implicitely through its type. Which means that the information becomes practically invisible to an onlooker. i.e. when viewing someone's vcard, there wont be any schema view information properties nestling alongside name and address properties. Whereas obvserving the schema associated with the data through the *rdf:type* property, these properties will be visible and seem quite appropriate there.

Evolution of the SchemaExtension schema

The first schema devised was called *SchemaExtension* since it seemed to be simply extending and augmenting the rdf-schema of objects. Under this system any particular 'view' would be given a uri and properties with these uri's would then act as 'gates' to the view's information.

The gates approach

The gates approach was simply that properties would act as gates in connecting information to nodes. It works on the principle that the application firstly has an idea which 'view' mode it is currently in (represented by a uri). It then starts at a resource describing say a property, and then looks for any properties connected to this resource with the same uri as the uri of the current view, and recursively goes down any and all of the branches it comes across, but the application will not go down properties that are not of the same uri-name as the current view mode. hence the notion of a gate. The nodes on the other end of the gate property will (may) then have extra information describing the layout.



In this example, some of these gate properties can be seen in action, taken from the original jezabel system:

http://www.hpl.hp.com/semweb/2002/05/action/extension#FormEdit

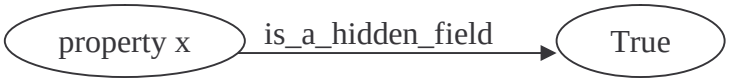
And *http://www.hpl.hp.com/semweb/2002/05/action/extension#FormView*

There is then one special 'gate' property:

http://www.hpl.hp.com/rdfs/schemaExtension#All

a sort of 'default' or 'open to all' gate. The idea being that regardless of view-mode this will be traversed by the application. Hence if you want a field to by default appear hidden, then you can simply send a schemaExtension#All property off to a piece of 'hidden field' rdf. It is also the principle connecting property throughout schemaExtension.

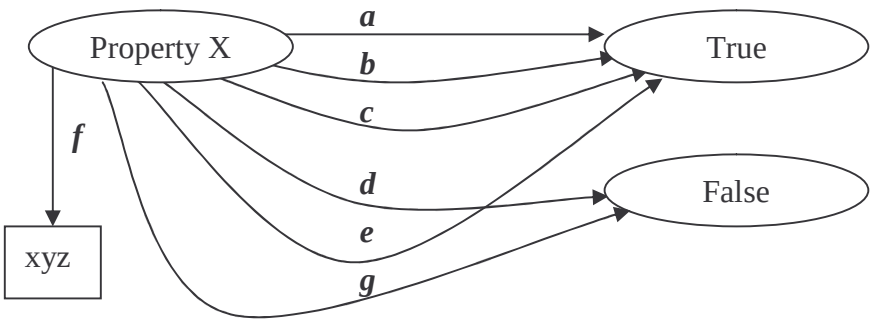
I deliberately tried to make the SchemaExtension schema extremely versatile and flexible, but also efficient when being processed by an application or modified by a person. For this reason I decided that firstly there would not be properties such as



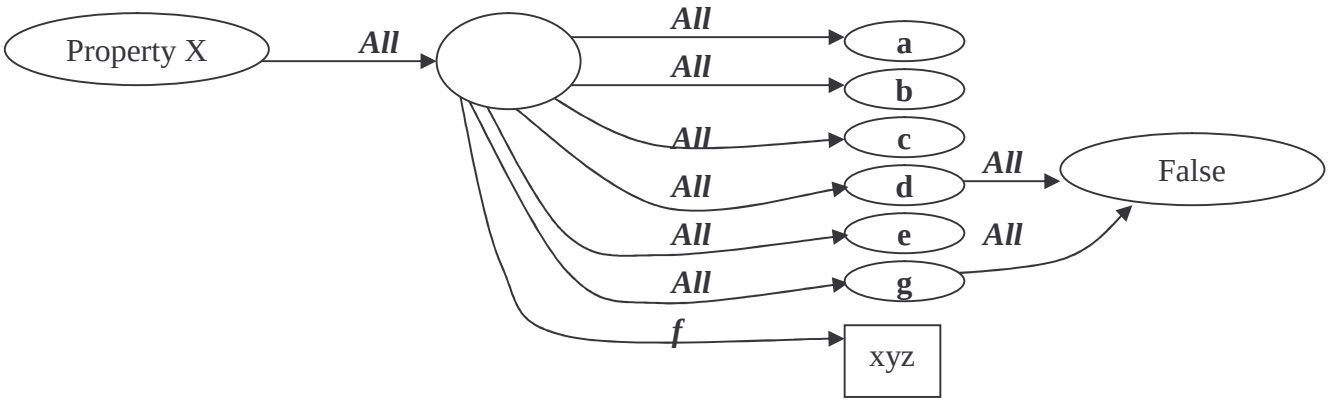
but instead:

Thus an application now only needs to search any properties of uri SE#All and the current view uri, and does not need to check in turn each possible attribute such as ‘is_a_hidden_field’, ‘is_an_editable_field’, ‘is_a_persistant_field’ etc. through every other property in the schema (although it turns out that this application oriented approach is in fact a bad approach – this is discussed more fully at the end of this chapter).
The other important factor in this apporach was maintainability. It seemed that by attempting to group all of the information off these stalk-like gate properties that in order to say completely remove the SchemaExtension information, potentially one single property could be removed or changed to do so, as opposed to having to scan for every instance of every possible property in the schema (See diagram below)

So:

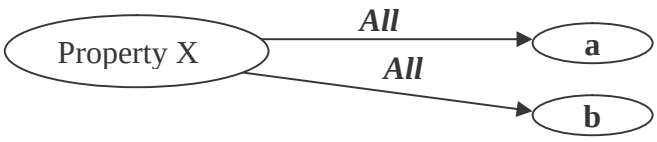


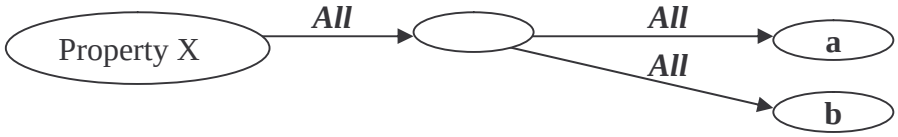
becomes:



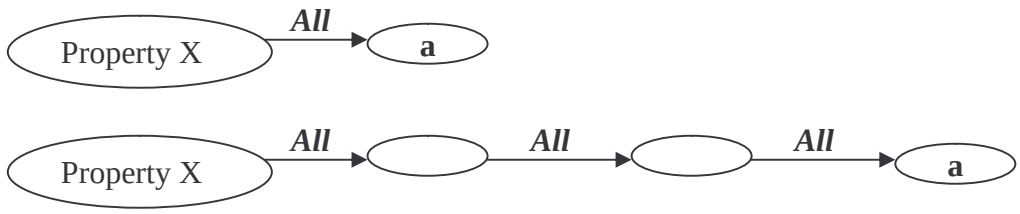
The understanding is that attaching ‘false’ or ‘true’ linked to something makes it false or true, but that without attaching anything, the implication is that its true.

Note that there is no difference in the result of the application’s interpretation of the following:



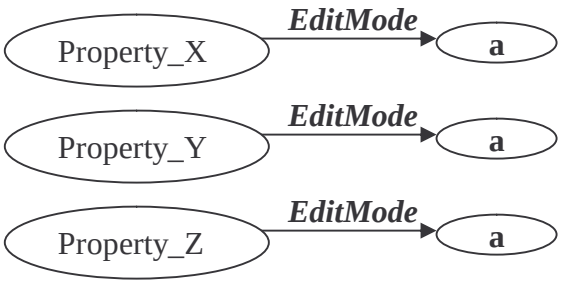


Or indeed:

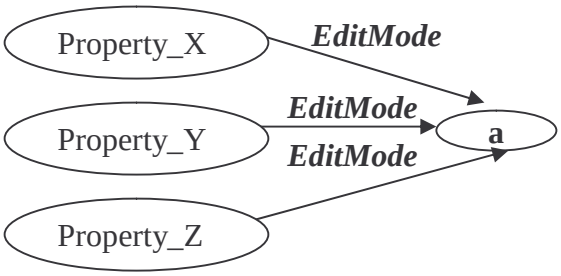


The versatility of this approach allows the programmer to create some very nice designs and structures in RDF. It is very common for example to want different fields in a structure to have the same SchemaExtension attributes set. For example in a standard ‘EditMode’ you will want all of the fields in a normal schema to be editable.

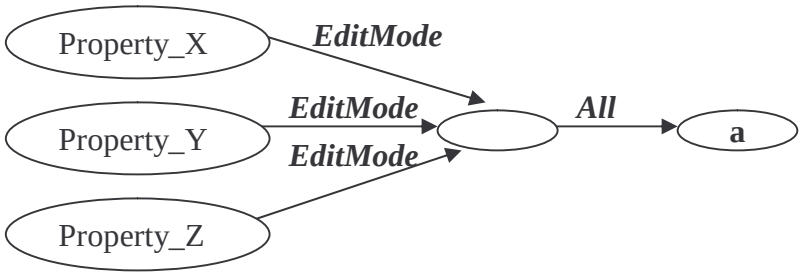
instead of representing them like this:



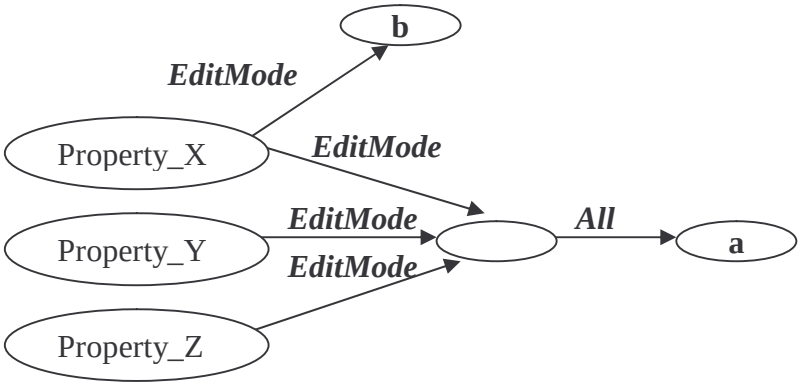
they can be modularised like this:



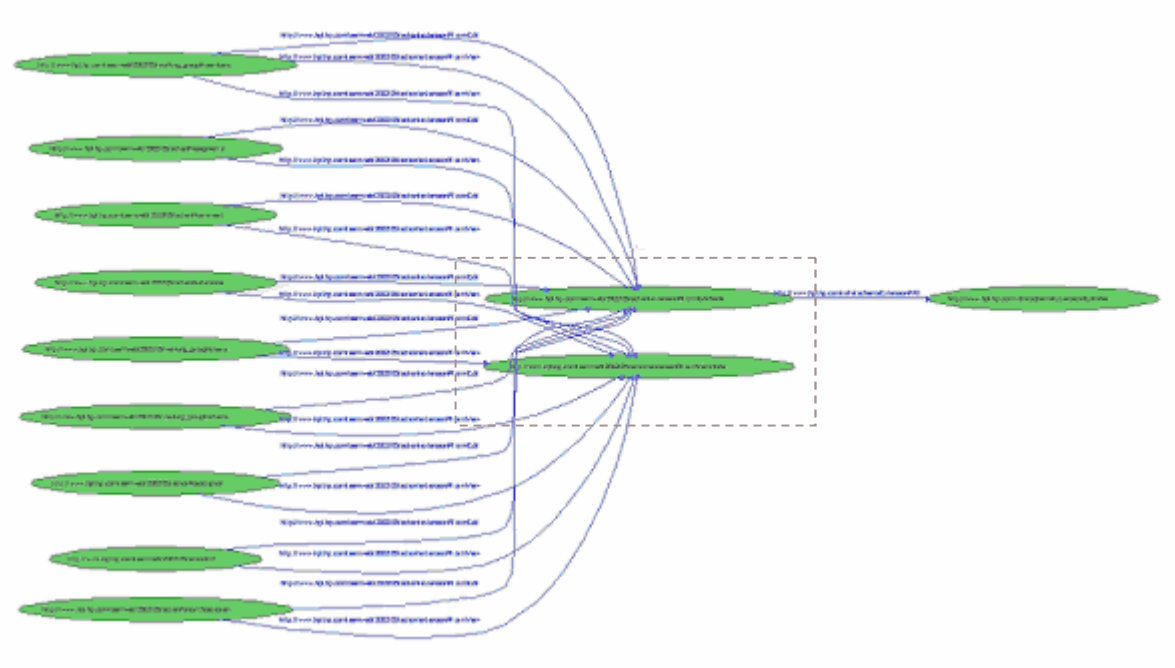
or better as:



Now say for example Property_X also needs the standard ‘EditMode’ properties that all the others take, but also requires say ‘b’ when in ‘EditMode’. Then it is easy and painless to add in:



Note in the above two examples how the EditMode gate properties ensure that a can only be reached when in EditMode. Once passed the first ‘gates’ it is unneccasary, and in fact un-useful to use gates when they arent needed and best to use the ‘All’ property as frequently as possible.



In this SchemaExtension rdf, this structure as described in the last 2 examples has been used extensively. Each of the resources on the left is a property of the Action schema###, and each has a gate property: *http://www.hpl.hp.com/semweb/2002/05/action/extension#FormEdit* and *http://www.hpl.hp.com/semweb/2002/05/action/extension#FormView*, each linking to a resource called …#FormEditNode and …#FormViewNode – although the uri’s of these resources aren’t necessary and should be ignored by any application, they aid human understanding when looking at the graph – these two resources have been highlighted with a bounding box.

Where does SchemaExtension apply?

Generally SchemaExtension properties are attached to resources of type property in the schema and to other SchemaExtension nodes. As mentioned in the post-analysis of SchemaExtension,

Overview of the SchemaExtension schema

Properties:

Where no namespace is indicated, the 'SE:' namespace as defined by the URI http://www.hpl.hp.com/rdfs/SchemaExtension# can be assumed.

Property(predicate) name	Domain	Range
rdfs:Label	rdfs:Class	Literal
Help	rdfs:Property	Literal
SchemaExtension	rdfs:Resource	rdfs:Resource
All		
Any		

Type Resources:
I.e. resources that ultimately subclass rdfs:Class, and may be 'rdf:type'-d to. Once again, absence of a namespace indicates the 'rdfv:' namespace.

Resource URI	rdf:type
Persistent	rdfs:Resource
Editable	rdfs:Resource
Hidden	rdfs:Resource
False	rdfs:Resource

Resources
Type resources are of course resources themselves, but it seems to make more sense to describe them seperately above.

Resource URI	rdf:type
true	Boolean
false	Boolean

Post-analysis of SchemaExtension

Unrigorous typing system
SchemaExtension was originally written almost as a test, simply exploring the idea of storing extra display information hidden in the schema of an rdf graph. It got absorbed into the jezabel system early on since it was to hand. It is far from perfect however. A particular problem is the *rdf:type*-ing of the schema, which is virtually non-existant. This makes it hard to construct rigorous rdf structures and get a clear view of how the SchemaExtension schema should be used.

For example, even though qualitatively it can be seen that SchemaExtension properties get added to either resources of type *rdf:property* or to a node of the SchemaExstension type, without an actual SchemaExtension type being defined the best that can be said is that SchemaExtension properties get added to 'a resource'. Which is unnecassarily vague and dangerously general(open to misinterpretation).
Then there is the concept of a view. A 'view' is very much an entity, or an individual, and should really have its own type. Views should be able to be described in rdf. Unfortunately, there is no view type, and they are not even represented as actual resources (although it may be argued that properties are resources). This makes saying anything about them or even detecting them in an rdf-graph extremely difficult. What way is there of knowing if *XYZ:former* is a view or merely a piece of unknown information?
Some withstraints in the form of rdf-types would have been very useful and provided a much more descriptive schema, which in semantic terms is for all practicalities, essential.

Extra information for bags etc.
A bag gives no restriction, and rdfs gives no way of stating what kind of types should be allowed into a bag. This however is important information if wanting to create forms to manipulate bags adequately. For example knowing that only people should be added to a bag and not actions or any other resource or object in the database. This problem was not addressed in SchemaExtension, but ought to be in future systems/schemas.
More than this, really being able to describe a certain kind of object that is wanted, perhaps by example, for example being able to say "only actions assigned to chris" "*rdf:type action*", "*action:assigned urn:people:chris*" would be useful.

Form element descriptions
Knowledge can be programmed into the interpretter of form-elements (drop-down menu's, checkboxes, pictures, graphs, etc.) to use when generating forms to allow the manipulation of object properties. It would be more descriptive and allow for greater control of form-appearance to be passed to the rdf-graph of the form's object to have a schema property and resources that allowed you to associate certain 'standard' form controls, such as drop-down menu's, multiple choice fields, simple graphs etc. For example a property 'score' going to a literal 44 could automatically draw a pie chart with 44% of the chart shaded in instead of writing the integer 44, using extra knowledge attached to the score property in the rdf-schema.

Colours
I tend to feel that colours of forms, form elements, fonts , styles and so on and so forth are somewhat out of scope for the goals this area of the project is trying to achieve, since these are things that ultimately do not make a vast amount of difference to the usability of a form.

The editable attribute
Simply having a field editable which generates a form that allows instances of properties to be added and removed is not nearly descriptive enough. Very quickly examples where only adding functionality should be present on a form (for

example when adding comments to a message board – functionality to remove messages is not especially helpful), and presumably only removing properties similarly. As well as of course wanting both add and remove functionality (which is most common). Therefore the ability to independently describe a property for a particular view as having add capabilities and remove capabilities would be a definite advantage.

Strings vs. writers

In the SchemaExtension interpretting class 'SchemaFormer', html was for convinience passed around as 'String's. When an html form was generated for example, it would be returned to the calling class as a String. This is fine for the size of the generated html forms used in this project, but in greater scope, a String can only store 50k of data. Therefore ideally a java Reader/Writer should have strictly been used, and future projects should take note of this.

One instance

Examples soon occur where it becomes obvious that giving the user means to add more than instance of a property in a particular object would be meaningless, even though there is no restriciton in this (which SchemaExtension mirrors). One example where more than one instance of a property would be unuseful is in the example of an 'id' associated with an objbect. A stronger example is that of the 'creation date' of an object – where describing more than one creation date of something renders both dates next to useless.

A means of letting the form creater know that the ability to create more than one instance for a particular property should not be allowed would be very useful.

Problems in the gates approach

The gates approach of using a property as something of a bridge which only allows applications of a current internal view-mode that corresponds to the uri of the property is inherently flawed.

Firstly it is flawed because the property in keeping with the semantic web is supposed to be describing a relationship between the subject and the object of the property's statement. The way that I used the property in SchemaExtension makes sense only in a procedural context.

Secondly as mentioned before, it is not acceptable to describe a view simply as a property. With no type associated the resulting graph becomes completely confusing, with it being unclear whether a property is a gate or simply a property from an unknown schema and with little formal way of making a distinction.

A re-think is needed in this area.

Rdfv: - a layout schema

http://www.hpl.hp.com/2002/08/rdf-view

Background

After creating and using SchemaExtension, it became obvious that there were lots of things that would be useful to add and that different fundamental approaches would have been much better, such change was needed that I decided to rewrite SchemaExtension under a different name altogether (to prevent confusion, since other components of the project were already depending on SchemaExtension).

Of particular consideration was taken the points and problems observed with SchemaExtension – the precursor to rdfv, in summary these were an unrigorous type system, the problems associated with the first 'gates' approach, the inability to describe form elements to use with certain fields on a form, no way to suggest whether one instance of a property is appropriate and lack of ability to describe clearly enough how a property may be edited.

Evolution of the Rdfv schema

Of classes:

Types were of minimal concern when SchemaExtension was being created. It was only much later when things had gone too far to be easily altered that I wanted a much stronger and rigourous type system. For this reason the resources that provide the actual rdfv layout data can now be called of rdf:type rdfv:Rdfv.

The idea of a beholder class, was introduced, in which a beholder beholds a view, which basically means that views can be represented by a resource, rather than by the property of a uri.

A proper Boolean class was introduced, rather than just having a 'false' resource. It includes both True and False.

A FormElement class was introduced which would allow the graphical designer to indicate that a certain property in a class was particularly apt for being represented by a certain FormElement, and with this, the DropDownMenu type was introduced – a sub-class of FormElement.

Operation

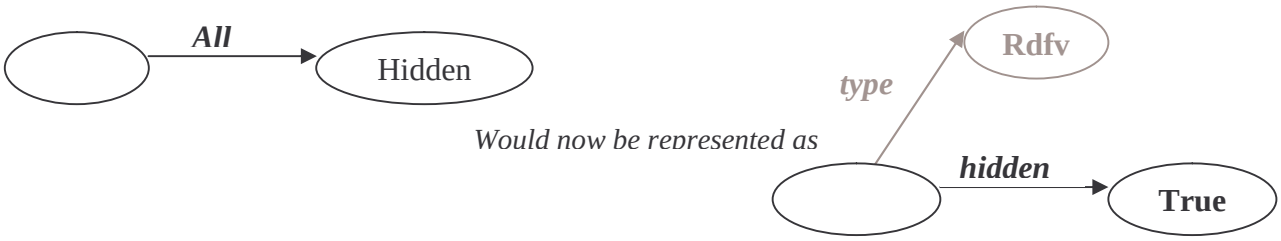
Firstly I wanted to change the way the gates idea had been implemented completely. the SchemaExtension notation was not in keeping with the style and function of the semantic web - to have properties that act as corridors with selective doors at the end controlling the flow of the applications interpretation is not the way the semantic web was really envisaged and is very application-oriented, and more importantly not meaningful. It also makes giving a 'view' a type very difficult, and lastly the notion of a 'view' represented in a graph purely by the

actual uri of a property just does not make for a sound approach (much better to be a resource). Better would be to have a property that indicates that this code is applicable to a partiuclar view, which could then go to a resource which would represent the particular view. This was exacted by adding the intendedBeholder property to the schema, in conjunction with the Beholder class (which gives a 'view' a type).

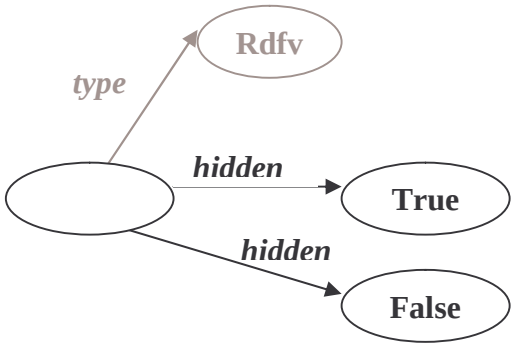
Not unrelatedly I also added a single specific property called associatedRdfv – which basically means that whatever is described by the Rdfv resource on the other end of the associatedRdfv property can be taken to apply to the current Object as well as whatever other rdfv information is described. 'Rules' for describing what to do when 2 pieces of associated rdfv describe different things for the same rdfv properties is discussed further below.

Being able to suggest when only one instance of a particular property should really exist would have been very useful. I decided not to put in a 'required' type, since firstly rdf does not naturally have required properties and secondly since I am only really concerned with the layout here. The one-instance rule above for example is only an indication to the interpretting application that the user should not be able to add more than one instance of a property, it is not in any way suggesting that the model might be 'incorrect' or inconsistent or wrong if more than one instance already exists.

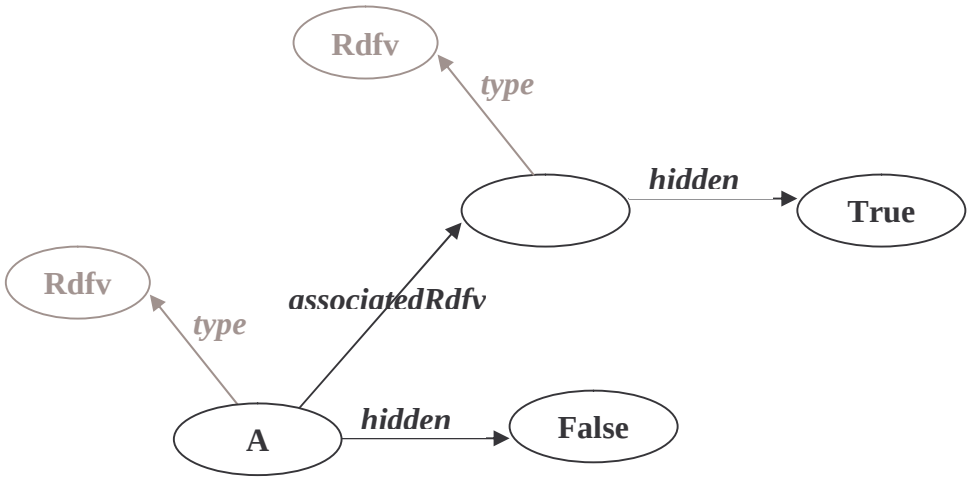
Lastly I wanted to get rid of the 'hidden', 'editable' and 'persistent' resources, and replace them as properties that would go to a boolean value, and also add more functionally useful boolean preoperties to be added to an Rdfv resource.



As with SchemaExtension, it quickly becomes possible with this system to say things that don't make sense, but do make valid rdf and fit valid rdfs schemas. For example:



or



I decided to do things completely differently to the SchemaExtension approach, and rather than having complex extra knowledge of things like whether something that is declared Hidden can be made un-hidden, and to re-organise the idea of ‘priorities’ or castes again completely differently so that the proximity in terms of levels of detachment from the original object or property that the Rdfv is describing is what decides which description of an rdfv property such as ‘hidden’ or ‘add’ is taken as the ‘true’ or perhaps ‘truest’ description. Thus in the above example from resource A, the rdfv property ‘hidden’ would be taken to be false.

Equally when a case exists where two statements with equal ‘importance’ state opposite things, such as an rdfv property is both true and false, to simply not define what happens, and leave it up to the designer to not design graphs like this, and leave it up to the interpretter to act in whatever way it deems appropriate. From a human or ‘semantic’ point of view equally if something is said to be true and false it is not clear what to do, and thus I feel able to justify this approach.

The one-instance property was added to solve the one instance problem of SchemaExtension, the describeBy property was added in light of the fact that a ‘human’ name for an rdf object is often required, but that there is no ‘standard’ property for describing this. For example it would add info that an issue is best described by its ‘title’. This is simply so that any application automatically has a good idea how to gracefully represent an object. so that urn:people:Patrick can become merely ‘Patrick’ (by adding a describeBy property from the people class resource to the firstName property described in the schema.

Overview of the Rdfv schema

Properties:
Where no namespace is indicated, the ‘rdfv:’ namespace as defined by the URI above can be assumed.

Property(predicate) name	Domain	Range
associatedRDFV	-	Rdfv
intendedBeholder	Rdfv	Beholder
hidden	Rdfv	Boolean
persistant	Rdfv	Boolean
add	Rdfv	Boolean
remove	Rdfv	Boolean
representWith	Rdfv	FormElement
rdfs:label	Rdfv	rdf:Literal
help	Rdfv	rdf:Literal
describeBy	rdf:Class	rdf:Property
oneInstance	rdf:Property	Boolean
match	DropDownList	rdfs:Resource
order	rdf:Class	rdfs:seq

Type Resources:
I.e. resources that ultimately subclass rdfs:Class, and may be ‘rdf:type’-d to. Once again, absence of a namespace indicates the ‘rdfv:’ namespace.

Resource URI	rdf:type
Rdfv	rdfs:Class
Boolean	rdfs:Class
Beholder	rdfs:Class
FormElement	rdfs:Class
DropDownList	FormElement

Resources
Type resources are of course resources themselves, but it seems to make more sense to describe them seperately above.

Resource URI	rdf:type
true	Boolean
false	Boolean

Schema in depth

Types:

Rdfv
A resource with associated rdfv ‘code’.

Boolean
A simple class with two resource instances: true, and false

Beholder
A resource description of a ‘view’.

FormElement
A resource corresponding to a form-element such as a pie-chart, radio-buttons or drop-down menu.

DropDownList (subClass of FormElement)
A resource corresponding to a formElement of type drop-down list – associated with a drop down list form element.

Properties:

associatedRDFV
domain: any
range: rdfv:Rdfv
Currently only of any use when the domain is of type rdfv:Rdfv, rdfs:Class or rdf:Property, although in light of expandability this has not been defined. Wherever this property appears connected to any subject, and going to an object of type rdrv:Rdfv, it is taken as an indication that the rdfv object is associated with the subject (extraordinarily).

intendedBeholder
domain: Rdfv
range: Beholder
A Beholder is a resource whose uri identifies a particular view of the graph. Pieces of rdfv information in a graph can only ever be accepted and ‘put into effect’ in association with a particular Beholder. Thus if no beholders are defined in a graph, then no rdfv should ever be activated. It is quite acceptable, and in fact recommended to have more than one beholder associated with a piece of rdfv.

hidden
domain: Rdfv
range: Boolean
When true indicates that in this view this information should be hidden from the viewer. A field can be assumed not to be hidden if no information is given.

persistant
domain: Rdfv

range: Boolean
When true indicates that in this view, if the property described in the schema has no corresponding property in the instance of the Object described by the schema then the property should still be displayed, simply with no field.

add
domain: Rdfv
range: Boolean
When true indicates that in this view, the form should include methods to add instances for the described property.

remove
domain: Rdfv
range: Boolean
When true indicates that in this view, the form should include methods to allow instances of this property in the relevant RDFObject to be removed.

representWith
domain: Rdfv
range: FormElement
Allows an object of type FormElement to be associated with a particular field. For example could suggest a drop-down menu would be a very appropriate form element to use.

rdfs:label
domain:
range: rdf:Literal
Imported from the rdf-schema schema. Suggests a better rdfs:label than any described in the actual rdfs of the object, if any exists.

help
domain: Rdfv
range: rdf:Literal
This is ‘help’ associated with the property. For example it could be represented in an html form as a little piece of text which appears when the mouse is held in the same spot over a the associated property.

describeBy
domain: rdf:Class
range: rdf:Property
This is to suggest allow a meaningful name to be associated with an instance of a class. It could automatically suggest for example that a person is best described by their name, or that an action by its ‘short description’.

oneInstance
domain: Rdfv
range: Boolean
This is a recommendation that the form should not give facilities to add extra instances of the property if any already exist.

match
domain: DropDownList
range: rdfs:Resource
For use by the DropDownList resource, the resource at the end of the ‘match’ resource will be an ‘object by example’ of elements to use in the drop-down list.

Type Resources:
I.e. resources that ultimately subclass rdfs:Class, and may be ‘rdf:type’-d to. Once again, absence of a namespace indicates the ‘rdfv:’ namespace.

Future

Expand formElement classes
Many more elements than simply ‘DropDownList’ should be added for good functionality, and the DropDownList class itself perhaps edited and refined somewhat since its description is currently a little vague.

Order
The all-important ‘ordering’ problem was not resolved with this version of rdfv (or SchemaExtension for that matter). The problem is that there is no suggestion in rdfs of which order properties of an object should be displayed, and is discussed early on in SchemaExtension. It is a very real issue and should be approached and resolved if meaningful forms depictions of rdf-data are truly required.