Semantic Context Menus in KDE

Laura Drăgan
Digital Enterprise Research Institute (DERI)
National University of Ireland, Galway (NUIG)
laura.dragan@deri.org

Siegfried Handschuh
Digital Enterprise Research Institute (DERI)
National University of Ireland, Galway (NUIG)
siegfried.handschuh@deri.org

ABSTRACT
The Semantic Desktop brings the desktop data to a standardized form, enabling it to be better interlinked and as a result easier to browse and search. It is lifted from application specific formats and locations and made available to all the desktop applications. However, some applications might be better suited to display some resource types than others, or might provide specialized functions. Rather that duplicating those function, tools could just call them from the tool that provides them, if they were made available just like the data, in a standardized form. We propose a way for applications to advertise their functionality in a semantic way.

GENERAL TERMS AND KEYWORDS
Semantic Desktop, ontology

INTRODUCTION
Nepomuk-KDE [8] provides the framework for semantically enabled applications on the user's desktop. The central RDF\cite{rdf} repository is a central access point for semantic data to all the applications on the desktop - to store or to use.

A semantic application can, according to its functionality, create resources and store them in the RDF repository and/or create or edit properties of existing resources as well as relations between existing resources. However, the data from one application might be more appropriately used (visualized, edited, etc.) by another application from the desktop, like for instance showing contact information from a text file in the address book, where the data is better grouped and displayed. By not using the data in the best way possible, information is lost and meanings are hidden, therefore it is important that the user is made aware of all the possibilities available for the resources she is handling.

We propose a method for applications to advertise their capabilities in a semantic way. This implies that they register in the RDF repository the types of resources they can handle, and the actions they can perform. For this purpose we define an ontology for applications and actions. The ontology however is not sufficient, we also need a way for applications to use it, and a way for the system to be automatically kept up to date with the possible changes that might occur when applications are installed or deleted or upgraded.

Having the applications and actions data in RDF is still not sufficient for other semantic applications to use it. We provide a library for easy use of the new data available and we use our semantic note-taking tool SemNotes [7] as testing ground of our solution.

Related Work
In KDE, the KRunner application suggests possible actions available for text queries given by the user. It is however a tool for searching and launching files and applications by the user, and it is not suitable to be used in other tools. There exists a Nepomuk plugin for KRunner which provides search of Nepomuk resources and suggestions for possible actions for the results, but it does this using the mimetype of the resource, and hence it will not show a complete list of possible applications.

In Nepomuk-KDE, there is a plugin-based tool for annotation of resources. It has a Konqueror plugin that allows the annotation of web pages as well as resources opened with the nepomuk:/ protocol and also a Dolphin plugin for annotation of files. At the moment there are plugins for annotating web pages, files, personal information entities, for tagging, for relating resources to projects, or geographic resources with geonames. The AnnotationPlugins are described by desktop files having specific properties:

```
...
[PropertyDef::X-KDE-NepomukProperties]
Type=QStringList
[PropertyDef::X-KDE-NepomukResourceType]
Type=QStringList
[PropertyDef::X-Nepomuk-ResourceTypes]
Type=QStringList
```

The tagging plugin is described by a desktop file containing:

```
...
X-KDE-NepomukProperties=nao:hasTag
X-KDE-NepomukResourceType=nao:Tag
# we can tag everything
X-Nepomuk-ResourceTypes=*
```

Like our approach, the annotation service and plugins use the existing Nepomuk desktop ontologies in the desktop files that describe the plugins.

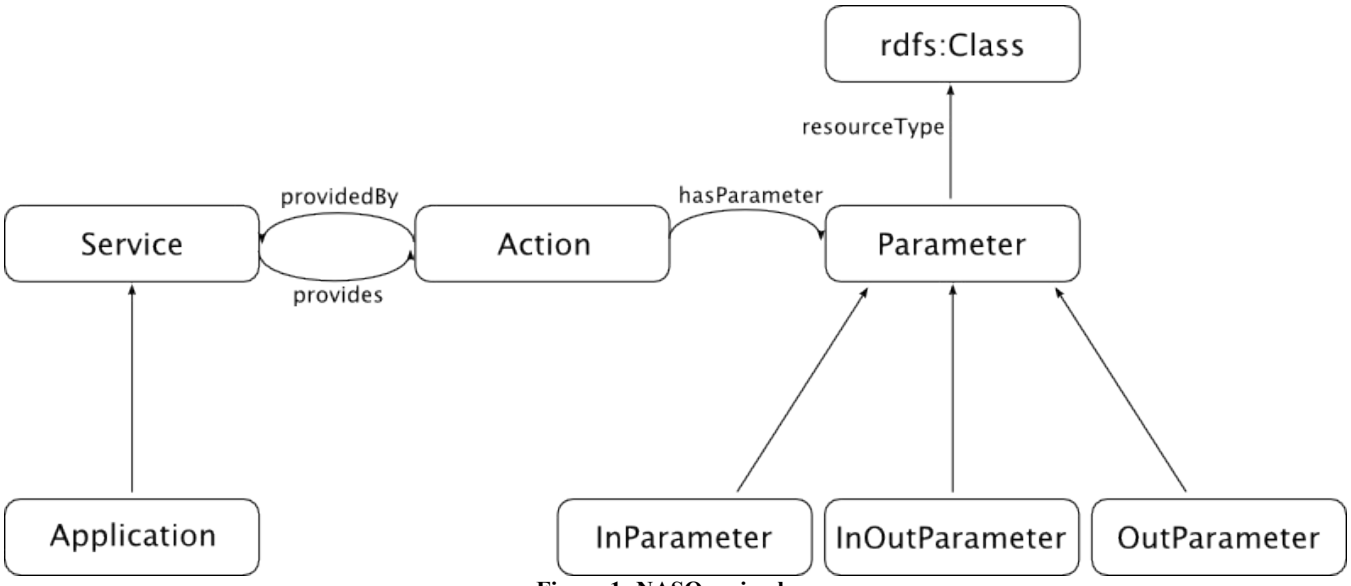

Figure 1: NASO main class

THE NEPOMUK APPLICATION AND SERVICE ONTOLOGY

We define and use an ontology to store information about the applications and services that are installed on the system. By adhering to this ontology, each application advertises the functions it provides. The ontology is still under community development at [6]. The Nepomuk Application and Service Ontology (NASO) is one of the Nepomuk ontologies, developed for the Nepomuk Semantic Desktop. It defines the classes Application and Service, where an application is considered to be a service with an user interface. Services and applications provide actions defined in the class Action. The main classes of the ontology and the way they are linked, as well as their key properties are shown in Figure 1.

Each Action instance is tied to the application or service that provides it. Each action is identified by a name (which does not have to be unique). Parameters are required to execute an action. There can be any number of parameters for an action. The parameters have a delegated/assigned type of resource they stand for. Actions can create resources as well as delete them; they can change resources or leave them unchanged after the action is executed. We define three subclasses of the class Parameter: InParameter represents a parameter that is not altered by the action; OutParameter represents a parameter that does not exist before the action is executed, and is created by it; InOutParameter represents a parameter that is modified by the action (this also includes deleting the resource).

Desktop Files

NASO is installed together with the rest of the Nepomuk ontologies [3] and is loaded when Nepomuk starts. Whenever a new application or service is installed on the system, the corresponding instances are added to Nepomuk. The application developers must define the actions that their application provides. This can be done in the desktop file of each application or service.

The desktop file is a configuration file that specifies how a particular program is to be launched, how it appears in menus, etc [2]. Therefore it is suitable that this file should contain also the actions provided by the application it describes.

Have a desktop file for each action defined by the application. The desktop file would have the type NepomukAction. We define the type with the service type:

```
[Desktop Entry]
Type=ServiceType
X-KDE-ServiceType=NepomukAction
Comment=Nepomuk Action
[PropertyDef::X-KDE-Nepomuk-Action-In-Types]
Type=QStringList
[PropertyDef::X-KDE-Nepomuk-Action-Out-Types]
Type=QStringList
[PropertyDef::X-KDE-Nepomuk-Action-In-Out-Types]
Type=QStringList
[PropertyDef::X-KDE-Nepomuk-Action-Name]
Type=QString
```

Our note-taking application SemNotes [7] provides the actions of creating and tagging a note. We define the action of creating a note in the desktop file:

```
[Desktop Entry]
Type=Service
Name=Create blank note
```

```
Comment=Create a blank note
X-KDE-ServiceTypes=NepomukAction
X-KDE-Nepomuk-Action-Name=semnotes-create-note
X-KDE-Nepomuk-Action-Out-Types=pimo:Note
```

There are no parameters required to execute the action, therefore no InParameters or InOutParameters need to be set.

For the tagging of a note action, the required parameters are an existing note resource and an existing tag resource. We define the action in the desktop file:

```
[Desktop Entry]
Type=Service
Name=Tag Note
Comment=Tag a note
X-KDE-ServiceTypes=NepomukAction
X-KDE-Nepomuk-Action-Name=semnotes-tag-note
X-KDE-Nepomuk-Action-In-Types=nao:Tag
X-KDE-Nepomuk-Action-In-Out-Types=pimo:Note
```

There are no OutParameter types listed because there are no new resources created by the action. The action of tagging a note (existing note, and existing tag) will result in the creation of a new triple in the Nepomuk RDF repository, of the form:

```
<note URI> <nao:hasTag> <tag URI>
```

We choose to consider that only the note is altered by the action in this case, although the triple newly created has the tag as object. We will consider that only the resources that appear as the subject of a triple that is created, changed or removed are considered to be modified by the action and therefore only their types should appear in the list of InOutParameters. Resources that appear as objects of those triples are not considered to be modified, and therefore their types should be listed as InParameters. In this example, there are no types that should appear in the OutParameters list, since there are no new resources created by the action, therefore the line is omitted from the desktop file.

There are several other actions provided by the application. Each has to be described in its own desktop file, with the corresponding parameters and names. The actions are then listed in the application's desktop file by adding a new property:

```
X-KDE-Nepomuk-Actions=semnotes-create-
note;semnotes-tag-note;
```

An alternative is to create the link from the action to the application that provides it, instead of the other way around. It implies having a link to the application in the desktop file describing the action. This can be done in the file hierarchy, by having all the actions provided by a given application in the same folder having the application name. However, this method is restrictive and error prone. A more flexible solution is to link to the application by adding a new property in the desktop file of the action.

```
X-KDE-Nepomuk-Action-Provided-By=SemNotes
```

This option also allows us to reuse actions for several applications, in the case when the parameters match, by adding several applications in the list, therefore this is the better solution and the one we chose to implement.

The Nepomuk ActionLoader Service

Having all the actions provided by applications and services described in their respective desktop files, and the NASO ontology loaded in the system is not enough. We need a service that assures that all the newly defined actions are read and loaded, as well as checking for changes of existing actions. The system relies highly on Nepomuk, therefore it is natural that the needed service is a Nepomuk Service [9].

The ActionLoader service will be started automatically by the Nepomuk Server. It provides monitoring and on demand loading of actions. It depends on the storage service because the data about the actions is stored as RDF in the central repository. The two actions defined above are stored as:

```
<nepomuk:/app_semnotes>
a naso:Application ;
nao:prefLabel "Semantic Notes"^^xsd:string ;
naso:provides <nepomuk:/action_create_note .
<nepomuk:/action_create_note>
a naso:Action ;
nao:prefLabel "Create blank note"^^xsd:string ;
rdf:comment "Create a blank note"^^xsd:string ;
nao:identifier "semnotes-create-
note"^^xsd:string ;
naso:providedBy <nepomuk:/app_semnotes ;
naso:hasParameter <nepomuk:/create_note_param> .
<nepomuk:/create_note_param>
a naso:OutParameter, naso:Parameter ;
naso:resourceType pimo:Note .
<nepomuk:/action_tag_note>
a naso:Action ;
nao:prefLabel "Tag note"^^xsd:string ;
rdf:comment "Tag a note"^^xsd:string ;
nao:identifier "semnotes-tag-note"^^xsd:string ;
naso:providedBy <nepomuk:/app_semnotes> ;
naso:hasParameter <nepomuk:/tag_note_param1>,
<nepomuk:/tag_note_param2> .
<nepomuk:/tag_note_param1>
a naso:InParameter, naso:Parameter ;
```

```
naso:resourceType nao:Tag .

<nepomuk:/tag_note_param2>

a naso:InOutParameter, naso:Parameter ;

naso:resourceType pimo:Note .
```

THE ACTION QUERY SERVICE

Once the information about the possible actions is available in the local RDF store, it can be queried and used by applications which want to use the framework for providing new features of aggregation with other services. For this, each application can use the Nepomuk libraries, which provide access to the local RDF repository as well as querying functions [5]. However, because a large part of the usage will most likely be related to finding actions, there will be a lot of code duplication by various applications. To prevent this, we created a service called "ActionQuery" that retrieves the action data and presents it to its clients. The functions provided by the service return lists of actions (QList<Action>, where Action is a class representing the naso:Action type).

```
allActions();

actionsByName(QString); [or QRegExp]

actionsByApplicationName(QString);

actionsByApplicationUri(QUrl);

actionsByInParameterType(Nepomuk::Types::Class,
bool);

actionsByOutParameterType(Nepomuk::Types::Class,
bool);

actionsByInOutParameterType(Nepomuk::Types::Class,
bool);

actionsByParameterType(Nepomuk::Types::Class,
bool);

actionsByParameterTypes(QList<Nepomuk::Types::Clas
s>, bool);

actionsForResource(Nepomuk::Resource, bool);

actionsForResources(const
QList<Nepomuk::Resource>, bool);
```

The names of the functions are suggestive regarding the actual functionality they provide. The service allows searching by the action names or by the application that provides them. Actions can have more than one parameter, so the functions searching by a certain type of resource as In, Out or InOut parameters also have a boolean parameter selecting whether the query should be restricted only to actions that match exactly. If the parameter is set to false, the function returns all actions that match the condition, including the ones that have other parameters that do not appear in the list given. Accordingly, the search by a list of parameter types will return all the actions that match exactly all the types given if the boolean parameter is set to true, or include the actions that have more parameter types than the ones requested for, when the boolean parameter is set to false. Searching by a specific resource will call the search by parameter type, with the type of the given resource.

When searching for actions by parameter type, we also take into account the inheritance relation between RDF classes. For instance if an action takes as parameter a resource of type Contact, it will be returned when searching for actions that take as parameter any of the subclasses of class Contact: PersonContact and OrganizationContact.

Having the actions that match a query, applications can use them as factories to create specific instances for specific resources. For an action instance to be run, all parameters should have a resource value assigned to them of the required type. Taking for example the "Tag note" action described earlier, we must assign both parameters to specific resources of type Note and Tag respectively. If a specific note and a tag are give, there is only one possible action that has to be generated. However, if only one of the two required resources is given, the other can be filled in by querying the RDF repository for all the resources of the required type. If the note to tag is given, but not also the tag, we can create as many possible actions as there are tags in the RDF store. If the tag is given but not the note, we will have as many possible actions as there are notes in the system. The algorithm works the same for any number of required parameters, ignoring of course the OutParameter types, for which the corresponding resources will be created only after the actions are executed.

Semantic Context Menus in SemNotes

We use our semantic note-taking application SemNotes to showcase the ActionQuery Service and the use of actions provided by external tools. We enhance the context menus in SemNotes with actions provided by other applications.

SemNotes is a semantic note-taking application that automatically identifies the resources mentioned in the text and links them to the note. The linking is done on two levels. The text identified as representing an existing resource is transformed in a link to the corresponding resource URI. Also, a RDF triple is created in the repository connecting the note to the resource. The user can select the types of resources that she wants to be identified and linked in the notes. All types of resources available in the repository can hypothetically be linked, but only linking some of them adds value to the user. For instance linking a tag in the text is not necessary, because the application provides a tagging function.

Having the resources ready identified in the text of the note, we propose to make the user aware of the possible actions that can be performed on them. We do that by using context menus to list the actions that can be taken. This way, the possibilities are presented on request, therefore not disrupting the user activity.

According to the KDE Usability Project [1] context menus

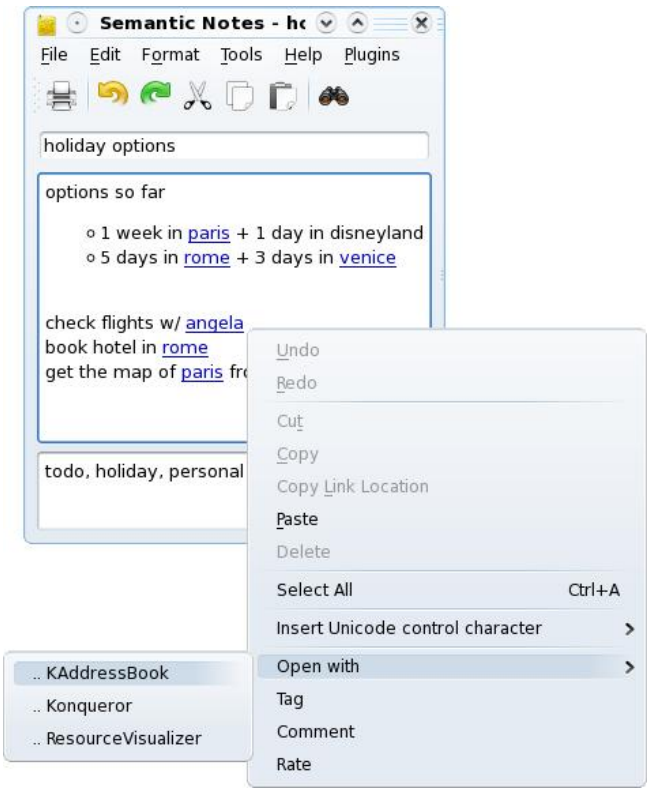

Figure 2: Context menu additions from naso: Actions plus tagging, rating, and comments.

are ``menus called by user interaction that provide a set of commands related to the context of where the interaction takes place within the interface object. They offer only items that are applicable or relevant to the object or region at the location of the focus or the pointer." We modify the common understanding of the term by not considering just the actions possible in the environment of the current/active application (from which the menu is called), but the actions possible in all (or most) of the applications that support actions on the selected resource. The semantic context menus should be activated only when the object is identified as an existing resource in the RDF repository. It can further be restricted so that the menu is activated only for some of the available types of resources. In the LinkedEditor of SemNotes, the resources identified in text are transformed automatically into links that point to the URI of the resource, therefore it is when these links are selected that we build the enhanced context menus.

To build the menu for a selected resource in the note, we use the ActionQuery service to retrieve the actions that work on it. Furthermore, we need only the actions that have exactly one required parameter. One type of these actions are the "Open with .." actions that should be provided by most applications for the types of resources they work on. We call the function:

```
actions = actionQuery.actionsForResource(resource,
true);
```

We created a SemanticContextMenu class that creates the list of QActions that would then be added to the default context menu of the editor window. Its function

```
QList<QAction*> processActions(Nepomuk::Resource,
QList<Action>);
```

takes as input the list of Actions returned by the ActionQuery service and returns the menu items to be appended to the context menu. Because on Nepomuk-KDE any kind of resource can be tagged, rated or commented, we also chose to add these actions to the context menu (see Figure 2). Tagging is an action that requires 2 parameters, the resource being tagged and the tag. The resource is set, when the context menu is generated, but the tag parameter has to be filled in by us. Generating the menu items for tagging the resource with all the existing tags can lead to a very long list of menu items which would make it very hard to use. Hence, we chose to make the standard metadata menu items open dialogs for data input by the user. This approach, although not optimal, is satisfactory.

CONCLUSION AND FUTURE WORK

In this paper we presented a way for applications to advertise their functionality in a semantic form. We devised an ontology for describing the possible actions of applications. Further, we present a desktop file system and a service for populating the ontology and keeping the information up to date. Having the data in RDF format and available to all the tools from the desktop is however not enough, we need a way to query and use the data easily. The ActionQuery library provides the means for developers to aggregate functionality from other applications in their tool. We demonstrate the use of the system for providing semantic context menus in our semantic note-taking application SemNotes.

In the current implementation we consider actions taking as parameters only resources. However, there is the possibility of actions having simple types as parameters, like strings, integers or boolean values. Adding simple types as parameters is not a trivial extension of the system, but it is important. Possible actions with simple types as parameters include changing property values of resources, creating new resources of certain RDF types starting from given basic values.

Another interesting direction is pipe-lining of actions. That means using the output of one action as input for another action, like in Unix pipes.

ACKNOWLEDGEMENTS

The work presented in this paper was supported (in part) by the Líon-2 project supported by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 and (in part) by the European project NEPOMUK No FP6-027705.

We would like to thank Sebastian Trüg and Charlie Abela for the insightful discussions on the subject of this paper.

REFERENCES

1. Context Menu. http://techbase.kde.org/Projects/Usability/HIG/SOU_Workspace/Context_Menu
2. Desktop Entry Specification. http://standards.freedesktop.org/desktop-entry-spec
3. NEPOMUK Ontologies. http://www.semanticdesktop.org/ontologies/
4. Resource Description Framework (RDF). http://www.w3.org/RDF/
5. The Nepomuk Meta Data Library. http://api.kde.org/4.x-api/kdelibs-apidocs/nepomuk/html/
6. Nepomuk Application and Service Ontology, 2009. http://dev.nepomuk.semanticdesktop.org/wiki/NasoOntology
7. L. Dragan. SemNotes. http://smile.deri.ie/projects/semn
8. S. Trueg. Nepomuk - The Social Semantic Desktop. http://nepomuk.kde.org/
9. S. Trueg. Nepomuk Services. http://techbase.kde.org/Development/Tutorials/Metadata/Nepomuk/NepomukServices