Ontology as an Information Base for Domain Oriented Portal Solutions

Michal Barla, Peter Bartalos, Peter Sivák, Kristián Szobi Michal Tvarožek, and Roman Filkorn

Slovak University of Technology, Faculty of Informatics and Information Technologies, Ilkovičova 3, 842 16 Bratislava, Slovakia,
{barla,bartalos,tvarozek,filkorn}@fiit.stuba.sk, szobi@chello.sk, psivak@mail.t-com.sk

Abstract. Ontologies are becoming increasingly accepted as a form for information representation of distributed web-based information and its processing. In our paper we build a web-based application around an ontology, which serves both as a data base and metadata source used in processing of the data. We propose techniques and methods that exploit this metadata to provide an easy and flexible implementation of the CRUD pattern. We identify patterns in the ontological representation of domain entities and transform them into web-based forms for data management. Based on this flexible framework we can model variations to adapt the application for specific sub-domains.

1 Introduction

A significant amount of resources has already been invested into the development of web-based applications that use ontologies for data storage [6]. Among other advantages, ontologies support the execution of semantic queries on an ontological database, explicit and inherent access to metadata and reasoning based on the formal representation of both the data and metadata.

By definition, an ontology is an explicit formal specification of a shared conceptualization (of a domain) [4]. Ontological languages provide means for the basic modeling of concepts such as classes, relations and various properties. Based on the corresponding language constructs, domain specific concepts can be defined and their semantics specified. Consequently, the model of a domain consists of both concepts defined by ontological languages, which are reused in other domain models and domain-specific concepts unique to a particular domain or set of domains.

If we create a model of a specific domain represented by an ontology, this model will necessarily have to be changed at some time in the future to compensate for changing requirements. Ontologies are especially suited for such change and provide flexible means of data storage.

However, the flexible data storage provided by ontologies would be of limited use unless applications are flexible enough to accommodate a similar degree of change. Otherwise, for any change in the ontology, the corresponding application code would have to be manually updated, which is unpractical for real-world applications.

Thus the apparent flexibility of ontologies results in the need for flexible applications. If we consider forms as the main mean of communication between portal and its users we come to the need for flexible form generation tools. These would be based not only on data stored in ontologies but also on additional (meta)data, which would be needed because domain ontologies shall not contain the information about the desired visual data representation, preserving their generality and delegating application-specific information to other sources.

2 Proposed approach

In our approach, we take advantage of the native availability of metadata in ontologies, which make the data self-descriptive and allows for effective searching in the stored data. Based on the assumption that an ontology may represent a formal model of an information domain, we propose the use of such metadata to build a domain oriented web portal solution for a particular domain.

In order to process ontologically structured data by the state of the art GUI frameworks, we implement mapping tools that transform data between its graph representation and an object-oriented representation. Consequently, we define the mapping of modeling concepts between these two modeling paradigms.

To design a flexible framework, we identify patterns as repeating structures (sets of concepts and their relations) in ontological representations and define consecutive data and processing around these patterns. We assume that similar patterns are shared between ontologies with the patterns themselves being defined using different types of ontological concepts – classes, relations between them, properties, instances and restrictions.

We map identified patterns onto sets of visual elements (graphical user interface widgets) that correspond to the data stored in a specific ontology. Based on this approach we implement the CRUD pattern (Create-Retrieve-Update-Delete) for a particular entity in a domain ontology.

Since the metadata available in ontologies are not always sufficient to fully create a satisfactory user interface, we define additional metadata, which describe the arrangement of visual components on web-based forms.

Finally, we automatically generate the forms corresponding to ontological concepts, where first the respective ontological patterns are identified. Next, the proper graphical representation is determined and lastly the form descriptions are saved and used during operation.

3 Object–ontology mapping

The creation of an object-oriented representation of ontological concepts introduces new challenges due to the fundamental differences of both representations. These come from the differences between description logic and object-oriented systems and lie primarily in the completeness and satisfiability. Ontologies have a significantly higher expressivity compared to object-oriented approaches [3].

We automatically generate a set of Java bean classes, each corresponding to an integral part of an entity described by the ontology. We represent literals by a simple data type field of the appropriate type and each object property by a separate Java bean.

In order to process these Java beans and store the values in an ontological repository, we need to generate additional metadata for the mapping between objects and classes and properties (RDF graphs) of the ontology. These metadata allow us to bind the ontological class to Java bean fields and include the name of the Java bean, the names of its fields, information about the corresponding OWL properties (e.g. multiplicity, data type) and the type of the object in the RDF triple (object type or a data type – literal).

The mapping itself is performed by a pair of graph-to-bean and bean-to-graph transformers (analogical to O/R mapping in relational databases [2]). These transformers work the graph representation of RDF and use reflection to invoke *get* and *set* methods on the generated Java bean objects.

The transformation process is performed recursively, in each step performs mapping between one object and an integral part of the RDF graph, i.e. object property. Finally, a simple Java bean field is mapped to an RDF triple and vice versa, with the name of the field being used to determine the corresponding ontological property. Thus a simple data type field corresponds to a literal in the ontology and an object property corresponds to an instance of the respective Java bean class.

4 Pattern types

We identified two distinct types of patterns that can be applied at different levels of abstraction and are thus useful for a broad set of ontologies:

- *Widget patterns*, which are based on basic ontological language concepts and correspond to relatively simple configurations in ontologies.
- *Visual patterns*, which define the higher-level visual style of forms, the layout of individual widgets and other form controls (close to user interface design).

To achieve independence from the used ontology we base our metadata processing on basic ontological language concepts. Moreover, we specify a model for the representation of variability in information sub-domains, e.g.

specializing the structure and behavior of an instance of the CRUD pattern of a domain entity for some specified subset of users.

4.1 Widget patterns

Widget patterns focus on the structure of classes, subclasses, properties and possible restrictions. They determine the widgets used in a form to edit instances of classes.

4.2 Simple widget patterns

Primitive datatype properties

Patterns for *primitive datatype* properties include all properties, whose ranges are literals (string, integer, float, date, boolean etc.). The graphical representation of these patterns is straightforward – the *rdfs:label* of each property is displayed next to the input field for its value. This would generally be a dropdown list for Boolean values (true, false, undefined), a calendar for date values and a *textbox* for text strings. It is more convenient to use *textarea* input field for longer strings as it increases the readability of the form. To distinguish cases where a normal *textbox* and where a *textarea* should be used, we can either define additional metadata about desired form template or we can compare the average length of existing instances to a predefined threshold.

If the respective property has multiple cardinality, the above elements would be placed inside a special control called *repeater*, which enables users to add/remove more values by means of additional buttons for these actions.

Same-range object properties

Patterns for *same-range object properties* identify classes that have several object properties with multiple cardinality and the same range (class or a union of classes). In this case, the fields for the range are displayed only once and an additional component is used to distinguish the specific property that is being edited. Such a component can be either a dropdown list or a set of radio buttons. All these components are wrapped in a repeater to allow multiple values to be added.

For example, the *Prerequisite* class form the Job Offer domain ontology of the NAZOU [5] project. Each prerequisite has two multiple properties: the *Requires* and *Prefers* which have as their range a union of experience and qualification classifications. Thus, the visual representation provides a radio button to select between the *requires* and *prefers* properties while the rest of the widget is common for both properties and is used to assign experience and qualification prerequisites for job candidates.

Enumerations

Patterns for *enumerations* identify object properties, whose range is a class fully defined by its instances. It should not be possible to add new or edit existing instances. Enumerated classes are defined in the ontology itself, e.g. a class representing the days of the week or various time periods (hour, day, month etc.).

Several graphical representations of this pattern exist. If the cardinality of an object property whose range is an enumerated class is single then it can be represented by a dropdown list of its instances. If the cardinality is multiple, the mentioned dropdown list can be wrapped in a repeater or instances can be represented by a multi-choice *listbox*.

Besides enumerated classes, also classes which allow users to create new instances or choose an existing one can be identified. This information must be stored in the metadata for the appropriate class. In this case all fields necessary to create an instance would be displayed and a dropdown list or *listbox* with existing instances would be added as mentioned above.

4.3 Tree hierarchies

Since hierarchies offer a wide range of values, they must be structured in a way that allows users to easily understand and choose amongst them. For example, when users want to choose a country where a company is based, it might be convenient for them to first choose a continent, then a country on that continent, etc.

In ontology, there are two basic ways to represent tree hierarchies. The standard property *rdfs:subclassOf* between classes which represents an *is a* relation and/or custom defined properties between instances can be used, which define relationships between nodes in the tree hierarchy.

One can assume that if a class in an ontology has a property which is transitive and points to instances of the same class (its range and domain are the same), then it is used to represent some form of hierarchy. The job offer ontology of the NAZOU project [5] defined two properties in this way: the *isPartOf* property and the *consistsOf* property, which were mutually inverse and allowed for navigation in a hierarchy of regions.

Additional characteristic of a class is the number of levels of its subclasses and also whether or not a class is fully defined by its subclasses. A class is fully defined by its (direct) subclasses if every individual belonging to that class must belong to at least one of its (direct) subclasses.

If a class is fully defined by its subclasses and only has direct subclasses, we represent it by a dropdown list component that contains labels of these subclasses. Figure 1 and 2 shows the example of the *jo:Benefit* class. Its graphical representation is a simple dropdown list and since *jo:Benefit* is a multiple object property of another class (*jo:JobOffer*), it is wrapped in a repeater (bottom).

In this way, users choose the type of instance they want to create. If the class is not fully defined, the dropdown list will also contain the label of the parent class to enable users to create an instance of it instead of its subclasses.

If there is more than one level of subclasses, their presentation should enable users to browse them. If there are only a small number of classes, it is suitable to display them in a dropdown list as in the previous case and indicate the hierarchy by adding a symbol before the actual name of each class (for instance one dot for each level of hierarchy). This approach might not be suitable to display complex and deep hierarchies, where it is better to create a component which simulates the navigation in a tree. E.g. a *listbox*, which contains all classes of the same level and is redrawn with the classes of the next level when the user chooses one value.

The current location in the tree would be indicated next to the component and users would have the ability to return to a higher level in the hierarchy (e.g. a button, hyperlinks in the path).

The same approach can be applied to a hierarchy created by transitive properties between instances. Whether the user can choose a class or instance which is not a leaf of a tree hierarchy is determined from additional metadata about the class. Metadata are also used when both classes and instances are used to define a hierarchy.

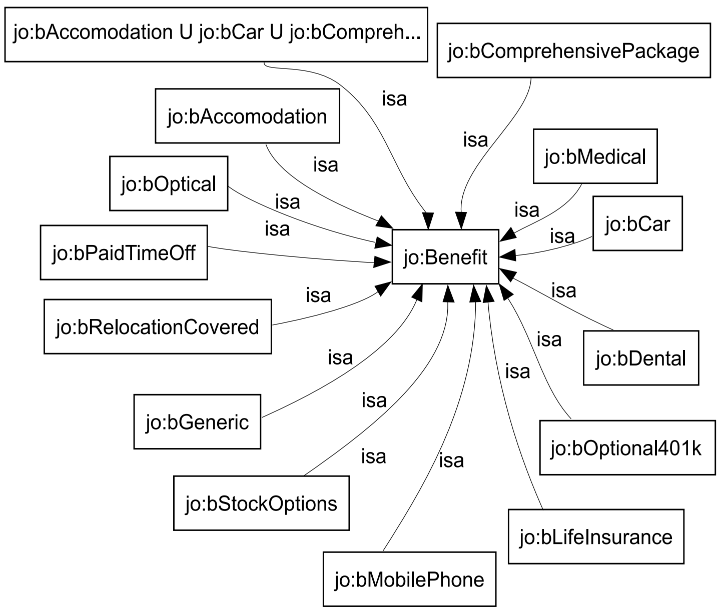
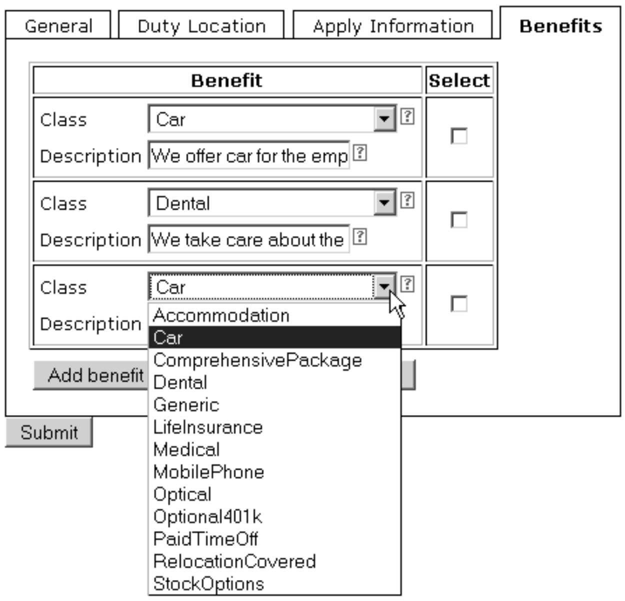


Fig. 1. Example of a class *jo:Benefit* that is fully defined by its direct subclasses.



Benefit		Select
Class	Car	☐
Description	We offer car for the emp	
Class	Dental	☐
Description	We take care about the	
Class	Car	☐
Description		

Fig. 2. Visual representation of benefits

4.4 Visual patterns

Visual patterns describe the concepts of sub forms and identify the typical structure, when a class has an object property pointing to another class (which can also have object properties). So, if class A has an object property pointing to class B, there are several possibilities how to represent it in a form for class A:

- Dedicating an area of the form for the properties of class B, usually bounded by a rectangle. This representation is suitable when class B has only few datatype properties.
- Dedicating a special part of the screen to display any object property of class A and creating a button for each object property of class B. Users choose the property, they want to edit by clicking the appropriate button. If a user clicks on a button of a property in class B, the content of the dedicated part would be redrawn and would contain the form for the editing of the instance of class B.
- Creating a button for each object property of class A. If a user clicks a button for class B, a pop-up window would be shown, where the object property could be edited. This solution is not desirable since pop-ups are usually annoying and are often filtered by web browsers.
- Using a tabbed interface, where each tab would represent one object property of class A. Selecting a tab for class B makes it editable. If class B also contained other object properties, a second row of tabs would be available.

5 Form Generation: Mapping patterns to form controls

The basic principle of dynamic form generation lies in the use of data stored in the ontology along with the appropriate metadata to identify ontological patterns. Patterns are used to define a binding between the data in an ontology and their graphical representation to create forms for specific classes and form controls for specific properties.

Since one pattern can have more than one graphical representation, additional metadata must be used to choose the most appropriate one.

Form generation is a recursive process which begins from the given identifier of a class from an ontology for which the form should be generated and continues via its object-type properties. During form generation, the visual description and the data model of the form must be generated. Furthermore, the corresponding implementation related objects such as Java classes (e.g., Java beans), which store form data must also be generated as well as mapping rules between these classes and the respective ontology. These mapping rules are used by the previously mentioned graph-to-bean and bean-to-graph transformers.

The proposed method matches the structure of each class to patterns described in the previous section. This determines whether the process of form generation is recursively applied to object properties of the class or is terminated by defining a set of simple widgets for display. The matching itself is done by examining the conformance of a selected class to a sequence of patterns in predefined order.

5.1 Form layout

The previously described *widget patterns* map the structure of an ontology to the form controls but do not describe the layout of the form. Although we already discussed the visual representation of object properties between classes (*Visual patterns*), none of the proposed solutions (e.g., tabbed interface) provide information about ordering of the visualization. Moreover, these representations do not allow the personalization of forms preventing us from creating various specialized form templates which fit the needs of individual users. These include hiding of unnecessary form elements or pre-filling forms with default data.

Another problem is that ontologies normally do not contain application-specific information, e.g. whether users are only allowed to choose from existing instances or have to create new ones. Finally, ontologies may contain concepts that should not be displayed on specific forms at all.

Since the lack of flexible form layout support would degrade the proposed solution we defined an additional ontology (fig. 3) which contains information about the order of form tabs and their titles as well as the order of the class properties displayed in these tabs. Additional metadata define whether users are allowed to create new instances of certain classes or are only allowed to choose from existing ones, or the combination of both. The ontology can also describe predefined values for specific fields thus allowing for the creation of forms which are more user-friendly compared to forms generated by generic ontology editors such as Protégé[1].

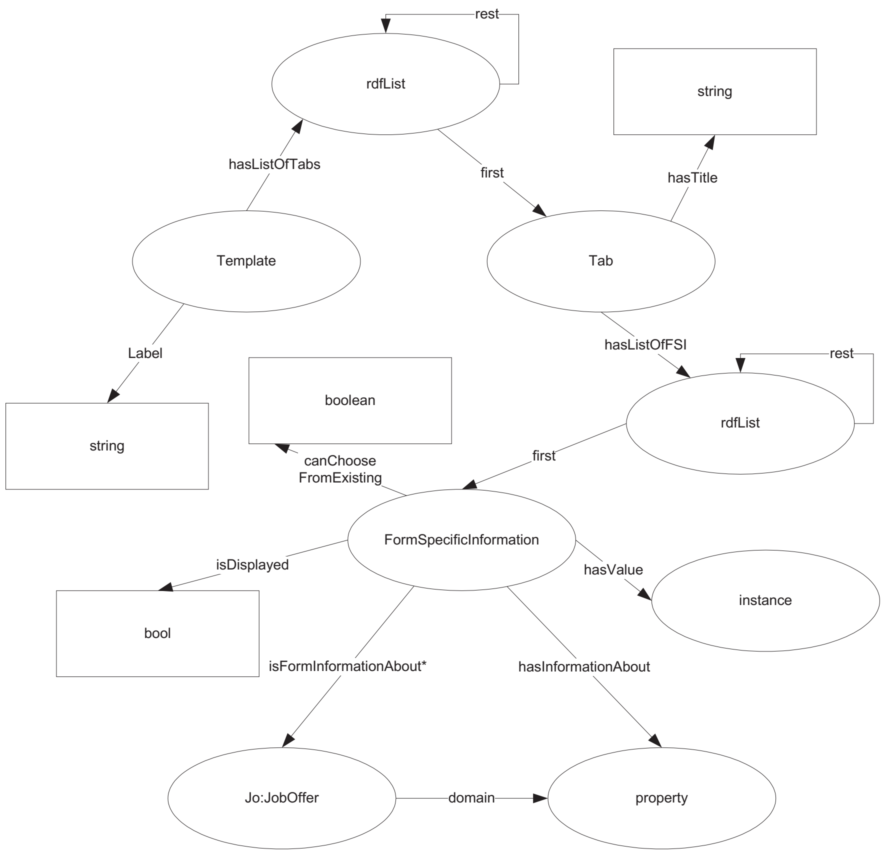


Fig. 3. Meta-model of a form layout information.

The model in fig. 3 shows that our solution supports the reuse of its parts since one Tab instance can be used in multiple lists and one *FormSpecificInformation* instance can be used in multiple tabs. This allows for quick customization of a form template, where some parts of a form can be reused and new layout can be defined for the rest of the form.

The fact that the model is directly connected to a domain ontology (class *jo:JobOffer* in our case) allows for easy personalization of a form template to meet the needs of organizations and individuals. If a system determines that a

[1] Protégé ontology editor, `http://protege.stanford.edu`

user always fills in the same value in a field (e.g., duty location or qualification prerequisites) it can create a template for this user which does have these fields pre-filled with the appropriate values (instances in the ontology). On the other hand, if the user is constantly ignoring some field of the template (e.g., level of management or salary bonus) the system can hide these fields from the template while still allowing their use if user explicitly requests the complete form.

The identification of the template that is used for a specific user is stored in a user model which is used for adaptation throughout the whole system if we consider the CRUD pattern as a part of a larger system.

6 Conclusions

We briefly described one of the current trends – the drive towards web applications built around the semantic web principles and technologies. We also explored the problems introduced by the dependency of portal systems on ontologies as means for information storage.

We proposed dynamic form generation as a possible approach which would accommodate changes in ontologies by increasing the flexibility of web portals. We identified and described several patterns in ontologies and their mapping to form controls. With this knowledge we designed an algorithm that dynamically generates form descriptions from ontologies, which can be used by portal solutions to display and process forms for instances from these ontologies. We find this approach suitable for easy creation of flexible forms for portal solutions with similar tasks. Typical usage is in the domain of offers (job offers, realty offers, vacation offers etc.) and in solutions where CRUD pattern is to be realized by system's users.

To verify the feasibility of proposed solution, we created a job offer portal that uses the job offer domain ontology developed as a part of the NAZOU project [5]. The portal enables users to input and publish job offers by filling in various aspects of the respective job offers in appropriately generated web-based forms.

Future work might include the identification of more complex patterns in ontologies and their mapping to form controls. Exploring the possibilities offered by adaptive hypermedia technologies and their relevance to dynamic form generation is another promising direction of research.

References

1. Aldred L, Dumas M, Heravizadeh M, Hofstede A (2002) Ontology Markup for Web Forms Generation. In: Workshop on Real World RDF and Semantic Web Applications

2. Ambler S W (2006) Mapping Objects to Relational Databases: O/R Mapping In Detail.
`http://www.agiledata.org/essays/mappingObjects.html`
3. Battle S, Jiménez D, Kalyanpur A, Padget J (2004) Automatic Mapping of OWL Ontologies into Java. In: Frank Maurer and Günther Ruhe (eds) Proc. of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004)
4. Benjamins V R, Fensel D, Studer R (1998) Knowledge engineering : Principles and methods. Data and Knowledge Engineering, 25(1-2):161-197.
5. Bieliková M, Návrat P, Rozinajová V (2005) Methods and Tools for Acquiring and Presenting Information and Knowledge in the Web. In: Proc. of International Conference on Computer Systems and Technologies – CompSysTech' 2005. Varna, Bulgaria
6. Dong J S (2004) Software modeling techniques and the semantic Web In: Proc. of 26th International Conference on Software Engineering (ICSE'04), pp. 724-725, IEEE
7. Obrst L (2003) Ontologies for semantically interoperable systems. In: Proceedings of the twelfth international conference on Information and knowledge management (CIKM '03). CM Press 366–369, New Orleans, LA, USA