Developing web information systems on the basis of domain ontology

P.A. Shapkin
Department of cybernetics
Moscow engineering physics institute (National research nuclear university)
Moscow, Russia
e-mail: psnet@yandex.ru

Abstract[1]

This article describes the of development web-oriented information systems on the basis of domain ontologies. Methods of representing and processing ontologies are considered.

1. Introduction

The development of information systems is a complicated task that requires a number of skills in different domains: the design of databases, programming of business logic, user interface implementation etc. The complexity rises even more if one considers the development of web-oriented information systems. It is caused by the use of different languages and development tools on different levels. Furthermore, many levels of web-application are functioning in separate, loosely coupled environments.

Common technique that helps to ease the development is to use standard software frameworks. Today the number of web frameworks constantly grows. It means that the industry needs more powerful tools than those that are presented by existing frameworks. In spite of using software frameworks the development of similar systems often requires repeated coding of similar functionalities. Repeated coding is often required even during the development of one system, e.g. when homogeneous objects are managed.

One of the main parts of information system is domain model. In case of object-oriented systems it contains the definitions of classes of objects. This paper proposes to use extended domain models represented in the form of domain ontologies as a foundation to define the functionalities of information systems.

The article is organized as follows: section 2 introduces the formalism of ontologies; section 3 discusses methods of developing ontology-based web-oriented information systems; section 4 concludes the paper.

Proceedings of the 11th international workshop on computer science and information technologies CSIT'2009, Crete, Greece, 2009

2. Description logics as a formalism for representing ontologies

Description logics [1] are a family of formal languages that are able to express notions of classes, subclasses and their properties. They have formal semantics which is based on the first order logic. Strictly speaking, they correspond to a restricted variant of first order logic which is decidable, unlike the first order logic by itself. The main building blocks in description logics are atomic concepts (unary predicates), which correspond to classes, atomic roles (binary predicates), which correspond to properties or relations, and individuals (constants), which correspond to instances. More complicated concepts and roles are built from simpler ones using concept and role constructors. Concept constructors include such operations as intersection of concepts, negation, role range restrictions and so on. They allow to define concepts in a manner close to the definitions of classes in object-oriented programming: using relations to other classes and enumerating properties (roles) that correspond to the instances of a given class. There are many variations of description logics, which include different sets of constructors. Some of them are more expressive, but also more complex and vice versa.

Description logics introduce several improvements in simple object-oriented models. We will show it on an example. Assume that there is a class Woman, and two roles: hasChild and hasFemaleRelative. The intersection role

$$\text{hasChild} \cap \text{hasFemaleRelative}$$

represents the role "has daughter", i.e. if we write a.hasDauther(b) we assume that b is simultaneously a child and a female relative of a. Now we can define the concept "woman having at most two daughters" in the following way:

$$\text{Woman} \cap \leq 2\ (\text{hasChild} \cap \text{hasFemaleRelative})$$

Thus, in description logics we can define some classes using other classes; classes and roles can have a name or they can be anonymous, like the role "has daughter", which is used to define a new concept but is not explicitly denoted as a separate role.

Another advantage of description logics is that it allows computing new classes or roles which are not explicitly defined in the initial ontology. The main form of inference in description logics are subsumption and proof of satisfiability of a given concept.

Determining subsumption is the problem of checking whether one concept is considered more general than another. In other words, subsumption checks whether the first concept always denotes a subset of the set denoted by the second one. For example, one might be interested in knowing whether Woman $\subseteq$ Mother (i.e. Woman subsumes Mother).

The problem of satisfiability is the problem of checking whether a concept expression does not necessarily denote the empty concept. In fact, concept satisfiability is a special case of subsumption, with the subsumer being the empty concept, meaning that a concept is not satisfiable.

It is remarkable that properties are separated from classes. An individual can have arbitrary set of properties. On the basis of these properties the reasoner can infer a class to which this individual belongs.

In the development of web applications the ontologies can be used on different levels:

- as a domain model;
- as a taxonomy of concepts used to describe the representation of data;
- as a description of services provided by the application;
- as a description of actions and controllers or the structure of pages in the application, and etc.

All of the data objects in the application can be represented as individuals. These individuals could be described in terms of ontologies defined for the application. Relying on these descriptions the application can decide which operations an object supports, how to represent this object to a user etc.

Ontologies can help to integrate different systems and to use existing standard components more. E.g. the developer of a new application can use standard ontologies to describe the domain of the application. This description will be based on concepts and roles denoted in standard ontologies, and there is a chance that the developer can later use standard libraries which make use of those ontologies.

Even if the developer constructs a new ontology for the application, the distributed nature of ontology representation languages enables to bind it to existing ontologies afterwards. If an object is defined in terms of an ad hoc ontology, its definition in terms of more general ontologies can be computed using inference procedures provided that relationships between those ontologies are defined.

3. Formal representation of an information system based on domain ontology

In this section we will give the formal model for a domain ontology based information system. Introduction of such a formal representation enables to build formal methods that are used to implement different functions of information system. The model is based on functional (applicative) style [2].

3.1. General model of an information system

We will consider an information system (IS) based on REST [3] and MVC [4] architectures.

According to the REST (Representational State Transfer) architecture each user request contains information about the resource it is addressed to, about the action that has to be executed, about the required response format and a number of additional parameters.

The Model-View-Controller (MVC) approach implies division of the whole application structure into three main parts: domain model, a set of views and a set of controllers. *Model* incorporates the definitions of structures needed to represent domain objects and implements required business logic. *Controller* is a component that is processing users' requests and executes required operations. Usually each controller contains definitions of some *actions*, which are commands that this controller supports. Different actions require different sets of parameters to be included in the request obtained from an external user. The interface of the system is defined by *views*. Views are responsible for rendering the results of executing controllers' actions. The results of that rendering are transmitted to the user as a response to his request. Usually views are bound to controllers' actions. The same action can have a number of views, which can have different formats. Thus, same actions can be invoked by different external agents, who require different response format, e.g. a trading system can supply the list of products both in a human-readable format such as HTML and in a format aimed at programmatic interpretation such as XML [5], RDF [6], JSON, etc.

The MVC approach enables to structure the application and break it to reusable parts. Nevertheless repeated coding is sometimes needed. E.g. in many cases the application contains many similar controllers because they implement homogeneous actions on similar objects. Different views can also have similar structures. Sometimes large part of views and controller actions can be generated automatically, but it requires extending the domain model by including additional metadata.

In the case of otology-based IS domain ontology can act as a model in MVC architecture. We will suppose that data is stored in a database, which we will define as a set of different data sources: RDBMS, external data services, etc. The main tasks of the information system are as follows:

- receive read/write requests from external users;
- get required data from the database;
- execute required data update operations;
- send results to the user in required format.

The operations that are executed by IS can be divided into two parts: database operations and generation of the response from the result of database operations.

Considered IS can be described by a tuple

$$< O, R, A, F, W, V, d >,$$

where O denotes the domain ontology; R is a set of REST resources in the IS; A is a set of actions supported by the IS; F is a set of supported response formats; W is a set of database operations; V is a set of views; $d : request \rightarrow string$ is a function that represents a universal controller.

Let analyze some the parts of this model in detail. Domain ontology is a formal definition of domain concepts and their properties. It contains definitions of concepts used to describe objects of the domain; it also contains metadata about domain objects, i.e. which of them are used as REST resources, etc. We will consider description logics as the representation of the ontology.

Database operations are used to execute a database query associated with a particular action on a particular resource. Such a construction can be described by a triple

$$(concept, action, operation),$$

where *concept* is a description logics expression of the associated concept, *action* is the associated action, *operation* is a function of type $R \rightarrow R$, that executes required database operations and returns the result.

Views are used to generate the response of the IS. Each view is associated with a concept, an action and format. Views can be represented by the following tuple

$$(concept, action, format, render),$$

where *concept* is a description logics expression of the associated concept, *action* and *format* are the associated action and format, *render* is a function of type $R \rightarrow TOut$, that renders its argument resource using the desired format. *Tout* is the type of output objects for the view.

Universal controller d is the only controller used in the IS. When the system receives a request controller has to execute the following operations:

- parse the request to extract the addressed resource, desired action, response format and additional parameters;
- choose and execute a database operation that matches the requested resource and action;
- render the results of the database operation using a view that matches the resource contained in the results.

This algorithm is illustrated by fig. 1.

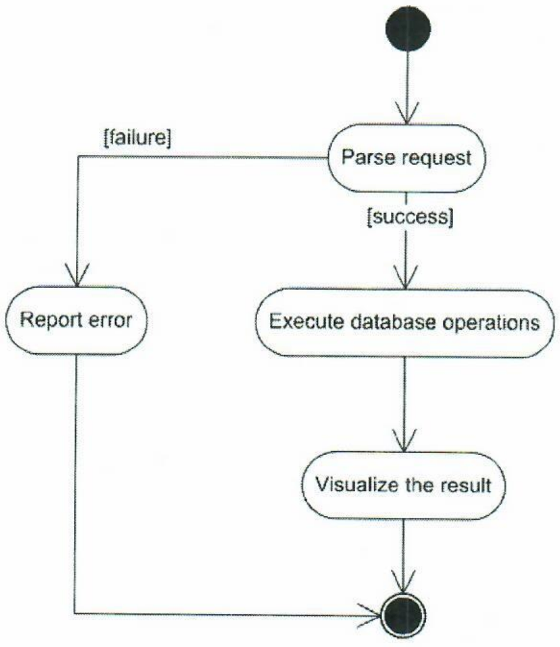

Fig. 1. Request processing algorithm

Request parsing is done by the function

$$analyzeRequest : Request \rightarrow Option\ (R \times A \times F).$$

Its argument is the request received from the user. The result of this function is either a triple of type $(R \times A \times F)$, which represents parsed request or a failure. To formalize this behaviour we use a parameterized monadic type *Option(T)*. If the parsing was completed successfully it returns an object *Some(R, A, F)*, in the case of failure an object None will be returned.

If *None* is returned as a result of request parsing, the controller stops request processing and returns an error report as a response to the user. If request parsing was successful its results its results are passed to the function

$$execute : (R \times A) \rightarrow (Option\ (R) \times A \times F),$$

it is responsible for executing matching database operations.

The result of the execution of database operations are passed to the function

$$render : (Option\ (R) \times A \times F) \rightarrow String,$$

which renders the results using the appropriate view and converts it into string that is returned in the response body to the user.

3.2. Method for processing semantic information

Execution of database operations and rendering system response is based on the same principle. Views and database operations are examples of *templates*. They are used by a *template system* that processes semantic information (i.e. RDF graphs) similar to the case of using XSLT to process XML data. When XSLT is used only to transform the data onto a different representation, we propose to use the same approach for general processing of RDF graphs. If the functions that are used in templates, besides the generation of an output have side effects, this approach can be used, e.g. for executing database operations.

A template system consists of a set of templates and a function that applies the system to an abject. This function chooses an appropriate template and executes it. Thus, a template system that processes values of type *TIn* and produces results of type *TOut* can be defined as a tuple

$$< Ts, e >,$$

where *Ts* is a set of templates and *e* is a function of type *TIn* → *TOut*.

The generic template has two major components: an expression that defines class of objects this template can be applied to (*match-expression*) and a function that is evaluated when the template is applied to an object.

In the case of processing an RDF-graph the input parameter for the application function has to be an identifier of a resource in this graph. The match-expression of the templates contains a definition of concept. If the input resource belongs to a concept defined in the match-expression of a template, this template is applied.

In the considered system templates have also an additional parameter "action". It is similar to "mode" parameter in XSL templates. Using different actions enables us to execute different operations for the same concepts.

Considered approach enables to use OOP methods to process semantic information such as RDF graphs equipped with ontology. When ontology concepts represent classes of objects, each template corresponds to a method of a class. The class (i.e. concept) to which a particular method (i.e. template) belongs is determined by the match-expression of that template. Thus, we assume that the proposed method is a universal tool for processing RDF data and can be used to involve ontologies into the object-oriented approach to programming.

4. Conclusion

This article introduced an approach to the development of web-oriented information systems based on domain ontologies. The approach enables to reduce repeated coding and gives control over the system by the means of domain metadata.

These ideas are implemented in a prototype information system with a simple ontology and user interface. The prototype is implemented on Java platform in the object-oriented functional language Scala. The implementation is in its initial state and many possibilities are not realized yet. Currently the system is designed to use RDF files as the data source.

Future work includes the realization of an internal ontology reasoner and testing the productivity of the system on different ontologies and different data stores.

References

1. Baader F., Calvanese D., McGuinness D.L., Nardi D., Patel-Schneider P.F. "The Description Logic Handbook: Theory, Implementation, and Applications". Cambridge University Press, 2003.
2. Wolfengagen V.E. "Combinatory logic in programming. Computations with objects through examples and exercises". 2-nd ed. Center JurInfoR, Moscow, 2003.
3. Fielding R.T. "Representational state transfer (REST)". Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine, 2000.
4. Gamma E., Helm R., Johnson R., Vlissides J. "Design patterns: elements of reusable object-oriented software". Addison-Wesley Reading, MA, 1995.
5. Bean J. "XML for Data Architects: Designing for Reuse and Integration". Morgan Kaufmann, San Francisco, 2003.
6. RDF Vocabulary Description Language 1.0: RDF Schema, from http://www.w3.org/TR/2004/REC-rdf-schema-20040210/