OntoPlay – a flexible user-interface for ontology-based systems*

Michał Drozdowicz[1], Maria Ganzha[1], Marcin Paprzycki[1], Paweł Szmeja[1], Katarzyna Wasielewska[1]

Systems Research Institute Polish Academy of Science, Warsaw, Poland
`(name.surname)@ibspan.waw.pl`

Abstract. One of the important issues when designing systems that are to use ontologies and semantic data processing is, how to build the interface between the user and the system. The aim of this paper is to discuss technical details of *OntoPlay*, a flexible interface for ontology-based systems. *OntoPlay* allows creation of dynamical interfaces that allow ontology-illiterate users to select a class or an individual on the basis of the ontology of the system. Furthermore, the *OntoPlay* is ontology agnostic and allows for ontology modification without the need for modification of the code of the user-interface.

1 Introduction

In computer science, concepts of ontology and semantic data processing have been around at least since the 1960's. Furthermore, since the introduction of the Semantic Web (2001; [10]) the research in this field has intensified, resulting, among others, in promises of ontologies being a "silver bullet" for "vampires" of heterogeneous data processing [3]. However, the uptake of ontology-based systems (and of the Semantic Web) is not matching promises found in seminal publications (e.g. [3, 9]). Thus far, successful applications of semantic technologies exist only in specialized areas (e.g. in biology and medicine [6,11,13]). In our opinion, one of the important causes of this situation is the fact that, much too often, in order to use them, semantic systems require (sometimes quite deep) knowledge about semantic technologies. Here, we introduce a web component enabling users to work with OWL 2.0[1] demarcated data, without understanding of ontologies and semantic data processing. In this way, semantic data processing may be hidden from the user in the same way as HTML and XML-demarcated content is invisible to users of standard browsers.

Let us make it clear that what we have in mind is not a semantic portal, understood as a system, where ontologically demarcated data is stored (or accessed from the Web) while a semantic browser is used in the same way as a search engine is used for HTML/XML demarcated data. We are interested in somewhat different use cases. Let us, therefore, start our considerations by describing two

* AT2012, 15-16 October 2012, Dubrovnik, Croatia. Copyright held by the author(s).
[1] `http://www.w3.org/TR/owl2-overview`

scenarios, a simple generic one, and a more complex, domain specific, one that we will focus on in the remaining parts of this paper.

1.1 Use case scenarios

The first, simplistic, scenario illustrates the problem (and its proposed solution) on the basis of a well-known sample ontology. Let us assume that a system based on semantic data processing is to be created for a pizza-delivery outlet. This system will use the standard Pizza Ontology[2] to internally represent the data ([2]). Obviously, this system has to have an interface for the client, who designs and orders her favorite pizza. To do this, she should be able, for instance, to specify the pizza base and select toppings from the available ones (defined in the ontology). From the user perspective, the result should be: great pizza. From the technical perspective, the result should be an OWL individual of class `Pizza`, which is a result of reaching an agreement / understanding between the user who does not "speak ontologies" and the system that "speaks ontologies only."

It is possible to develop the user interface by hiding the system behind a "standard browser" and build an "HTML-based" front-end that could transform HTML QueryStrings into SPARQL queries and execute them on the Jena-wrapped database[3]. On the "way back," the resulting individuals, would be translated to be displayed within the browser (see [7, 8], for an extended discussion of this approach). In this case, one of the problems is: how to deal with ontology changes (for example, how to update the interface when a new type of pizza topping, e.g. potatoes in yogurt, has been added to the available ingredients?). Furthermore, what happens if the pizza outlet starts selling pizzas with a cream-based sauce and a garlic-based sauce, in addition to the tomato-based one, and a new property (in this case `hasSauce`) has to be added to the ontology?

The second use case involves contract negotiations and a more complex ontology. Here, the *OntoPlay* front-end guides the users through the available (valid) choices and makes it possible for them to interact with the system, even if they do not know the structure and vocabulary of the underlying ontology. The scenario comes from the "Agents in Grid" (*AiG*) project (see, [14] and referenced to earlier work found there), where an agent-based system is developed to facilitate resource management in the Grid. In the *AiG* system, all data is ontologically demarcated and information is semantically processed. The *AiG* ontology (see, [4]) is much larger and more complex than the pizza ontology, as it consists of approximately 150 classes, and about 150 properties, and is likely to be further extended. One of the scenarios in the *AiG* project involves the user searching for Grid resources to execute a computational job. To do so, the user must first define criteria, based on which the Grid nodes of interest will be selected.

As can be seen, both use case scenarios lead to the same problems. How to allow the user, who is not a specialist in semantic data processing, to: (i) interact with a "semantic system," (ii) formulate queries without knowledge of ontologies

[2] `http://www.co-ode.org/ontologies/pizza/pizza.owl`

[3] Jena – A Semantic Framework for Java. `http://jena.sourceforge.net`

and semantic query languages, and (iii) make the queries precise – matching the ontology, rather than performing natural language → ontology translation. Furthermore, it is very important to avoid building a specialized "static" front-end, which would be difficult to adapt to the changes in the ontology.

To address these problems we have developed an ontology-driven interface, that is generated incrementally and dynamically *from the ontology* by a special plugin. In other words, an interface that allows the user to "browse" the ontology used in the system, by providing information as to what is available, based on earlier selections. The result of this process is selection of an individual or a class used in semantic information processing inside the system (completely hidden from the user). In what follows we describe how the needed plugin was designed and implemented, and illustrate how it simplifies the use of semantic technologies.

2 Related work

Let us start from an interesting general observation. The Google query for a "semantic browser" leads to the page: `http://semanticweb.org/wiki/Category:Semantic_browser`, where 12 *Semantic Web Tools* are listed. However, all of them have a slightly different focus. For instance, they allow extraction of semantic data from web pages, visual browsing of linked data, building ontologies, visually representing ontologies and/or relations between concepts, etc. In other words, they support the needs of well-trained semantic data processing professionals. Of course, it may be the case that, upon in-depth studies, it would be possible to apply one of them to our use cases. However, an immediate support for the needed functionalities is not obvious. Finally, observe that only 3 of these 12 projects have been updated within the last 2 years.

Now, let us look more in depth into other applicable tools and technologies. They can be split into three groups: ontology workbenches, faceted browsers and SPARQL query editors. What follows is a discussion on representative tools of each of these categories, focusing on what we found missing for our application.

2.1 Ontology workbenches

Tools such as *Protégé*[4] or *TopBraid Composer*[5] allow creation and management of ontologies. Similarly, *WebProtege* [6] offers similar capabilities using a web interface. However, none of them can be used as an editor of OWL individuals or class expressions pluggable into a web site – they are targeted at the task of *developing ontologies* (rather than their use, in the context of the above scenarios).

[4] `http://protege.stanford.edu`
[5] `http://www.topquadrant.com/products/TB_Composer.html`
[6] `http://webprotege-beta.stanford.edu`

2.2 Faceted browsers

Faceted browsers let the user browse the data contained in an ontology by filtering the information on subsequent properties or dimensions. Here, the options the user has for defining the filters, are based on the previous user actions. In this aspect, these tools could be used, in the above mentioned scenarios, as search query editors, replacing the class expression editor feature of *OntoPlay*. Examples of such tools are: *Flamenco*[7], *Information Workbench*[8], *SlashFacet*[9], *jOWL*[10]. However, in contrast to our solution, these tools enable the user *only* to specify values of properties describing the root class. Therefore, it is not possible to nest conditions and restrict properties of classes other than the root class. Finally, the *mSpace* project[11] allows creation of queries also on values of properties, thus enabling one level of nesting, but only using text queries; it is not possible to use the same user-friendly interface as for the top level of properties.

2.3 Query editors

Finally, let us consider the SPARQL query editors, such as *SPARQLer*[12] or the tool provided by the *OpenLink Virtuoso* server[13]. They provide a text area into which the user can input a SPARQL query that is executed on the server and the result set is displayed. Some of these tools provide the user with a set of sample queries, others (e.g. the *Virtuoso*) present also a visual representation of the query, but they *require* that the user understands SPARQL and is able to hand-craft valid queries.

In summary, there exists a large body of work concerning web-based user interfaces to semantically demarcated information. However, we have not found any that could serve as a lightweight front-end allowing creation of nested descriptions of OWL individuals and class expressions in a manner usable by users with limited experience with semantic technologies. Therefore, let us now describe our approach to developing the interface to semantic systems.

3 OntoPlay – front-end requirements

Let us now consider the software requirements of the front-end. Recall that we assumed that users of the ontology-driven system can be "ontology-illiterate." Therefore, the interface should be straightforward and *hide* from them the very fact that they use ontologies and semantic data processing. In other words, the front-end should guide the user through defining the classes and/or individuals,

[7] `http://flamenco.berkeley.edu`
[8] `http://iwb.fluidops.com`
[9] `http://slashfacet.semanticweb.org`
[10] `http://jowl.ontologyonline.org`
[11] `http://www.mspace.fm`
[12] `http://sparql.org/query.html`
[13] `http://virtuoso.openlinksw.com`

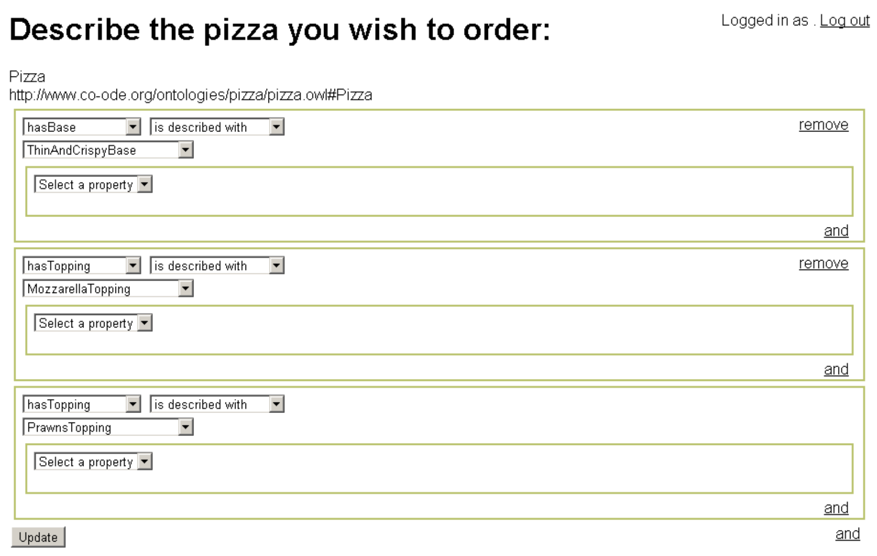
even at the expense of expressiveness. For instance, the system should allow the user to select the possible pizza individual on the basis of pizzas available in the pizza ontology. Similarly, in the *AiG* system, the front-end should support configuration of needed software and hardware, on the basis of the information found in the *Grid Ontology* (part of the *AiG* ontology, see [16]). Furthermore, observe that the "world changes rapidly" and, therefore, the system should be flexible and allow easy modification of the underlying ontologies. In other words, modifications of the vocabulary should cause as few as possible updates in other parts of the system. Ideally, the front end should be agnostic to the actual terms of the ontology, and depend only on some generic properties of the ontology structure. Let us now describe how such front-end can be built.

4 Condition builder

Considering the use cases and the requirements for the front-end component, the key challenge was a user friendly mechanism for creating OWL class expressions and individuals. This mechanism should allow selection of "properties of interest" from these found in the ontology. For example, it should display available pizzas (ontology classes, e.g. vegetarian pizzas), or allow their creation (selection of an instance, based on properties defined in the ontology, e.g. chicken and anchovies pizza with a yogurt sauce).

To describe the way that the *OntoPlay* plugin works, we start with an overview of its core – the *Condition Builder*. The process of establishing what the user is seeking, consists of defining conditions on different properties of the root class. The user can select the property she wishes to restrict, choose an appropriate operator and type in, or select, the value for that property. These steps can be applied to multiple properties of the same class. Let us look at some examples.

In the "pizza use case," let us assume that the user would like to order a thin crust pizza with mozzarella cheese and prawns. Specifically, we would like to build a condition that the pizza should have thin and crispy base and two toppings – mozzarella and prawns. Let us look at how this is specified in *OntoPlay*, using the actual classes and properties from the ontology. The root class is `Pizza` and the user would start by defining the value of the `hasBase` property to be an instance of the `ThinAndCrispyBase` class. She does so by first selecting the `hasBase` from the property list, then by selecting the "is described with" operator, and choosing the `ThinAndCrispyBase` from the class list. Both the property list and the class list are presented to her in a human-understandable way (see Figure 1). Note that the property list contains only these properties that have the `Pizza` class in their domain, and that the classes in the class list originate from the range of the `hasBase` property. This would make it much easier for the user to navigate even more complex ontologies. Afterwards, the user clicks the "and" link to add a new property restriction and completes analogous steps for the `hasTopping` property with a value from the `MozzarellaTopping` class (and once more for the same property, but from the `PrawnsTopping` class). These steps are available from a set of menus, while the "possibilities" that can be selected originate from the ontology itself. If required, the user can also link the conditions using the "or" operator.


Describe the pizza you wish to order:

Logged in as . Log out

Pizza
http://www.co-ode.org/ontologies/pizza/pizza.owl#Pizza

hasBase | is described with
ThinAndCrispyBase
Select a property

hasTopping | is described with
MozzarellaTopping
Select a property

hasTopping | is described with
PrawnsTopping
Select a property

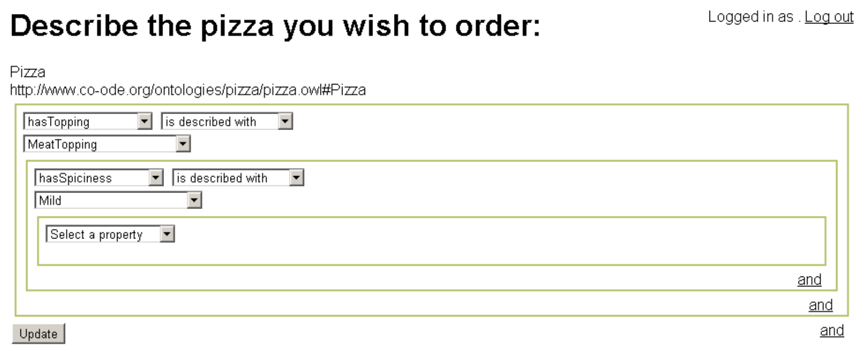
Fig. 1. Example of a definition of an individual

An example of how the "ontological actions" are displayed to the user has been shown in Figure 1. Here, we present how the desired effect can be achieved by clicking on appropriate elements of the display. These elements are generated from the ontology through the selection process. The resulting OWL definition of the new pizza individual is shown in the following listing.

```
Individual: NewPizza
    Types: Pizza
    Facts:
        hasTopping _:MozzarellaTopping ,
        hasTopping _:PrawnsTopping ,
        hasBase _:ThinAndCrispyBase
```

An important aspect of the condition builder is the possibility to further specify values of properties using *anonymous nested individuals*. Let us assume that the user would like to build a pizza with a *mild meat topping*, without specifying the exact topping class (he is not sure what he would like and what are the possibilities, so he is looking for suggestions). To do so, he would select (by clicking on the appropriate display elements) the `hasTopping` class, choose the `MeatTopping` class and from the nested property list, select the `hasSpiciness` property and restrict it to the `Mild` class. This condition is presented in Figure 2.

For the *Agents in Grid* use case, let us define requirements for a Grid resource, assuming that the job requires an *Intel 64* processor with the first level cache greater than 3 MB. In this case, the user has to build a class representing


Describe the pizza you wish to order:

Logged in as . Log out

Pizza
http://www.co-ode.org/ontologies/pizza/pizza.owl#Pizza

hasTopping | is described with | MeatTopping

hasSpiciness | is described with | Mild

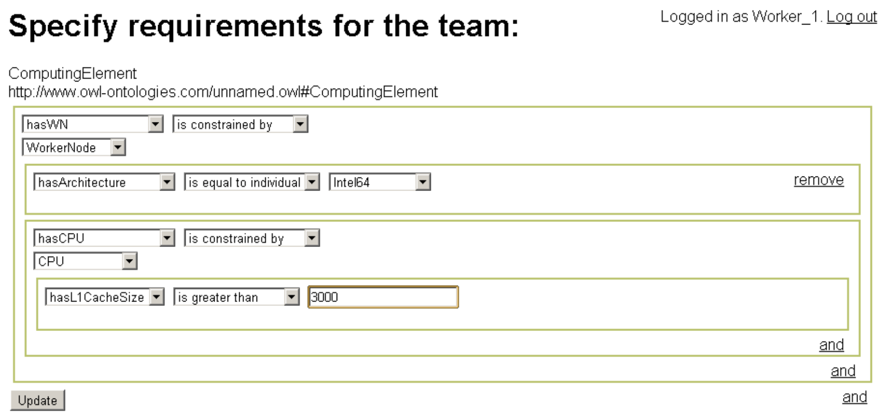
Select a property

and
and
and
Update

Fig. 2. Example of a nested individual

all individuals that satisfy these conditions. They can be defined in the application as presented in Figure 3. The user first specifies the need for a worker node, by setting the value of the `hasWN` to an instance of the `WorkerNode` class. Subsequently, she selects the `hasArchitecture` property equal to the individual `Intel64`. Next, the worker node should have the value of the `hasCPU` class constrained to an individual of the `CPU` class. Finally, she would restrict the property `hasL1Cache` of the `CPU` class to "greater than" 3000 KB. The result of such specification, translated into OWL, is shown in the following listing.

```
Class: TeamCondition
    EquivalentTo: ComputingElement that hasWN some
        (WorkerNode that hasArchitecture Intel64 and hasCPU some
            (CPU that hasL1Cache some integer[> 3000]))
```


Specify requirements for the team:

Logged in as Worker_1. Log out

ComputingElement
http://www.owl-ontologies.com/unnamed.owl#ComputingElement

hasWN | is constrained by | WorkerNode

hasArchitecture | is equal to individual | Intel64 | remove

hasCPU | is constrained by | CPU

hasL1CacheSize | is greater than | 3000

and
and
and
Update

Fig. 3. Example of a class constraint

Observe a subtle difference between defining individual and class expression. When defining the individual, for the datatype properties (e.g. the `hasClock-Speed`), the only operator would be "is equal to." However, when defining a class expression operators such as "is greater than" or "is less than" can be used. Obviously, the available operators depend on the range of the property (e.g. for a string property the "greater than" operator would not be applicable). Note also that, shown in the example, the condition builder enables for a number of nesting levels of conditions, while being able to chain the conditions on any level, allowing for arbitrarily complex specifications.

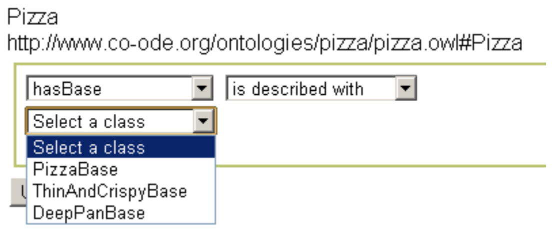


Fig. 4. List of classes before changing ontology

Another important feature of the *OntoPlay*, is that the only fixed dependence of a term from the ontology is the root class used in the condition builder. The remaining ontological data is inferred automatically and is *not* affected by changes in the ontology. To demonstrate this, let us show how the interface of the "pizza example" is changed after adding a new topping class. Before the change, the only available classes were `DeepPanBase` and `ThinAndCrispyBase`. Figure 4 shows these as options for the constraint on the `hasBase` property. After adding the `GreenPestoBase` to the ontology, and restarting the application, the user will be immediately able to select also the new class (see Figure 5). The change *did not* require any changes in the application code or configuration of the front-end.

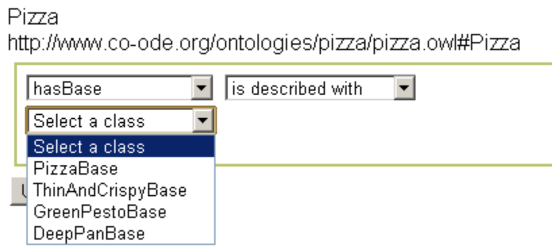


Fig. 5. List of classes after changing ontology

5 Front-end architecture

Let us now look into details of the architecture of the plugin, starting from the front-end. The user-accessible part of the system has been split into three components. Two of them are generic and always present, while the third is required only if the *OntoPlay* should be integrated with the *JADE*[14] agent framework (see, [4, 15]).

1. **Client** – provides HTML views and JavaScript logic for dynamic interaction.
2. **Server** – a web application acting as a controller and responsible for parsing and generating ontological data.
3. **Gateway** – an optional proxy subsystem, responsible for communication between the web application and the *JADE* agents.

5.1 Client

Thus far we have approached defining an OWL individual and/or a class from the user's perspective. Now, let us describe how this is achieved. First, let us recall that semantic processing is performed by the server while the client only receives what it needs. For example, the list of properties of a `Pizza` is sent from the server only when a user selects the `Pizza` class. Similarly, the list of individuals that can be set as a value of an object property is not retrieved until the user selects the property and an operator. For instance, in the *AiG* use case, if the user wants to specify the vendor of the CPU, the list of individuals of the `CPUVendor` class is sent to the Client component when the user selected the `hasVendor` property and the "is equal to individual" operator. More generally, actions undertaken by the user in the Client component generate a request to the Server component, the result of which is extending the user interface with elements that can be used to continue the specification process.

Every definition of a class expression or of an individual starts with a *condition box* bound to a class from the ontology. The condition box allows the user to choose a property that she wishes to restrict. The user can also add a new condition box tied to the same class by pressing the "and" link. When the user chooses the property, an appropriate property box is retrieved from the server. By default, the property box contains only a list of operators applicable to the current context (class expression, or individual) and the property type. Selecting an operator loads a property value box, which can contain different elements depending on the type of the property and of the operator. For simple datatype properties the user can type in the value. For object properties, the user can either select an individual (as a value of the property), or choose to describe the value as an anonymous, nested subclass or individual. Picking the latter option results in adding a new condition box, same as the root one, with the exception that the user needs to choose the class the condition is bound to. This is useful, for example, because it lets the user restrict the value of some property (e.g.

[14] `http://jade.tilab.com/`

hasMemory) to a subclass of the class defined in the range of the property (e.g. PhysicalMemory instead of Memory as in the range of the hasMemory property).

To implement the Client component, and to provide the user with a guided point and click interface, more than template-generated HTML is required. We decided to employ JavaScript and the jQuery library [15]. Specifically, the initial rendering of the client-side HTML and the JavaScript code contains only the condition box and a drop-down list with class properties. When the user makes a selection, an XMLHttpRequest is executed, to download a fragment of code that, in most cases, represents the operator list. As a result, the user interface is expanded. In the same way, when the user selects an operator, the code fragment defining the property value input is retrieved. After the user confirms her description by clicking the *Submit* button, the values from the user interface are parsed into an internal condition representation. This is accomplished by using the queryBuilder.js script that traverses the elements containing the selected classes, properties and values, and generates a class condition structure containing the class URI and, possibly, a list of property conditions. Each property condition might, in turn, contain a class expression, thus forming a tree. The condition object is then serialized into the JSON format, sent to the server, deserialized, and converted to an OWL ontology fragment. The following listing shows the result for the Pizza example from Section 1.1 (namespaces removed for brevity).

```
{ "classUri":"Pizza",
  "propertyConditions":[
    { "propertyUri":"hasBase",
      "operator":"describedWith",
      "classConstraintValue":{
        "classUri":"ThinAndCrispyBase",
        "propertyConditions":[ ] } },
    { "propertyUri":"hasTopping",
      "operator":"describedWith",
      "classConstraintValue":{
        "classUri":"MozzarellaTopping",
        "propertyConditions":[ ] } },
    { "propertyUri":"hasTopping",
      "operator":"describedWith",
      "classConstraintValue":{
        "classUri":"PrawnsTopping",
        "propertyConditions":[ ] } } ] }
```

5.2 Server

In the server, the condition builder logic is handled by a controller class Constraints, and two sets of classes serving different parts of the user interface:

- Subclasses of the PropertyConditionRenderer class – responsible for rendering the property boxes for different property types
- Implementations of the PropertyValueRenderer interface – rendering the property value elements depending on selected operator

Having discussed bidirectional interactions between the Client and the Server components, let us look at interactions between the Server component and the

[15] http://www.jquery.com

ontology (provisioning of information from the ontology). The entry point to this subsystem is the `OntologyReader` class responsible of retrieving:

- information about an OWL class, including properties that can have it in their domain,
- OWL classes in the range of a given property,
- OWL individuals of classes in the range of a given property.

The `OntologyReader` uses a simplified, internal representation of the ontological information. This means that while the structure does not provide the flexibility of the OWL itself, it greatly simplifies the remaining parts of the plugin. The elements of the language that are modeled in this way are:

- `OwlElement` – base interface acting as a base for all other classes, defining *getters* for the *URI* and the local name.
- `OntoClass` – represents an OWL class and contains a list of properties that can describe it.
- `OwlIndividual` – view of the OWL individual with properties describing it.
- `OwlDatatypeProperty` – base class for classes representing different datatype properties, such as `IntegerProperty`, StringProperty, etc.
- `OwlObjectProperty` – class representing an object property.

It is now evident that, indeed, we use a very simplistic view of the OWL language, but it proved to be specific enough to satisfy the requirements of the employed user interface model. The current implementation supports the most common XSD data types, but the system was designed to be easily extendable. To achieve this, specific property factories (e.g. `StringPropertyFactory`) are configured on application start. Each factory contains logic for determining if it is appropriate for a given property from the ontology, and can create an instance of the matching property class (implementing the *Chain of Responsibility* design pattern [5]).

The last major part within the Server component is the `OntologyGenerator` class. Its responsibility is generating the RDF/XML fragments corresponding to the conditions defined in the Client component. As elsewhere, the logic of generating the ontology fragments has been split into classes depending on the condition type. The main classes of the component are:

- `OntologyGenerator` – the facade of the component, responsible also for initializing and managing instances of the *OWL API*[16] classes – `OWLDataFactory` and `OWLOntologyManager`,
- `IndividualGenerator` – responsible for generating OWL individuals,
- `ClassRestrictionGenerator` – responsible for generating OWL class expressions,
- `RestrictionFactory` and its subclasses, such as `ClassValueRestrictionFactory`, `IndividualValueRestrictionFactory` or `EqualToRestrictionFactory` – implements the logic of creating the actual property restrictions

[16] `http://owlapi.sourceforge.net/`

on individuals and class expressions. Here, each subclass corresponds to a particular type of a property and, as in other cases of the Server architecture, is centrally registered in the `PropertyTypeConfiguration` class.

The design of *OntoPlay* is meant to enable easy extension to include new property types (such as, for example, the XSD `xsd:negativeInteger` simple type). Such extensions should require as little modification of the existing codebase as possible, thus adhering to the *Open Closed Principle* [12]. For example, implementing such new datatype property, would involve only implementation of a new subclass of the `OwlDatatypeProperty` and the `OwlPropertyFactory` classes (optionally, a new implementation of the `PropertyValueRenderer`), and their registration (by modifying the `PropertyTypeConfiguration` class). No additional changes in the existing code of the application would be needed.

6 AiG Gateway

Thus far, we have deal with the Client and the Server components and their interactions. For many semantic applications this would be sufficient (e.g. for the "pizza parlor" from the first use case), however in the *AiG* we needed the web controllers to communicate with the *JADE* agents. In the *AiG* system, *JADE* agents are involved in semantic SLA negotiations [14]. Simple as it may seem, passing semantic information from the user interface to an agent system posed some interesting challenges; caused by the differences between the properties of agent communication and the request-response nature of the *Web*.

Sending messages to agents is rather simple. It is sufficient that the web application creates an *ACL* message and sends it to an appropriate agent using classes provided by the *JADE* framework. Specifically, these include the *Gateway* class, which can be used statically by controllers or servlets, and which is a proxy to a *GatewayAgent*, which is created for the web application and "forwards" the messages send to it by the *Gateway*.

On the other hand, tracking responses to messages sent from the web application, requires an extension to the existing mechanisms. Controller actions are stateless by nature, and thus it is not possible to use an event based mechanism for notifying the user about new messages received by the *GatewayAgent* (for an in-depth discussion, see [7]). Instead, we have implemented a solution utilizing a message queue for storing replies to messages sent from the server. Specifically, all messages sent through the *AiG Gateway* are grouped into conversations, started separately when the user initiates a particular scenario that spans multiple messages to be exchanged between the user and the agents. The *GatewayAgent* keeps track of messages in conversations and through the *Gateway* proxy enables retrieving replies in a particular conversation. In response to the client initiated (*AJAX*) request for new messages from a given conversation, the web controller retrieves the data from the *Gateway* and sends it back to the client.

For instance, if we assume that our pizza system involves an agent based negotiation mechanism (e.g. for finding the cheapest restaurant), we could imagine that specifying the pizza the user wishes to order would initiate a new conversation with multiple agents. The *Gateway* would then gather all responses received from the restaurant agents, and the front-end would present them to the user.

7 Technologies used

The front-end application was developed on top of the *Play! Framework*[17] – a lightweight web framework with straightforward deployment and a rich plugin ecosystem. This framework also serves as the technological stack for the Server component of the ontology builder user interface, which comprises the web controllers as well as modules for reading ontological metadata and generating the OWL data from the descriptions provided by the user. The browser-side subsystem is implemented as a dynamic JavaScript application using jQuery – one of the most popular general purpose JavaScript libraries.

To interact with the OWL ontologies, the *OWL API* library is used as the base of the `OntologyGenerator`, enabling creation of ontology snippets. Initial implementation of the `OntologyReader` was built on the *Jena* framework, and used the Pellet[18] reasoner (the only reasoner directly compatible with *Jena*). Unfortunately, recently, when testing the solution with a wider set of ontologies we have found that there are situations where analogous operations of different reasoners bring inconsistent results. One of the examples involved sub-properties. Let us consider the `Pizza` class that has the property `hasIngredient` and its sub-property `hasBase`. The range of `hasIngredient` is the `Food` class, while the range of `hasBase` is `PizzaBase` (a subclass of `Food`). When asked for the range of `hasBase` property, Pellet, returns not only the `PizzaBase` class, but also `Food`. We have verified that using other reasoners, such as Fact++ and HermiT, an individual that is described by having the value of the `hasBase` equal to an individual of class `Food` but not `PizzaBase` (such as `IceCream`) is considered inconsistent with the ontology. Following this case, we have provided an alternative `OntologyReader` implementation, using the `OWL API` library and the HermiT reasoner. This implementation didn't exhibit such behavior.

Another difference between the Pellet and HermiT reasoners is their interpretation of the domain of a property, when the domain is defined as a union of classes. When Pellet is asked to list classes from a range of such property, it returns a list of all the conjuncts of the range expression. In contrast, HermiT provides the closest class that was a super-class of all conjuncts in the expression. For example, looking at a property – `hasMemory` – with the range of *PhysicalMemory OR* `VirtualMemory`, both classes being subclasses of the `Memory` class, retrieving the range of the property using Pellet would give us a set of `PhysicalMemory` and `VirtualMemory`, while the HermiT reasoner would instead only return the `Memory` class. This happens because HermiT assumes that the domain must be complete and does not return parts of the domain. In this case, a result of asking the reasoner for domains could not be `PhysicalMemory`, because

[17] `http://www.playframework.org`
[18] `http://clarkparsia.com/pellet`

the value of the property might in fact be of type `VirtualMemory`. The result of `Memory` is correct, because, regardless if the value is of type `VirtualMemory` or `PhysicalMemory` it is always of type `Memory`.

These cases became show that current implementation of the semantic reasoners differ in their semantics. These differences originate from differing interpretations / approaches to / vision of ontological reasoning. Since this is not the right venue to discuss these differences, let us only observe that issues like this make it impossible to develop reasoner-agnostic applications (i.e. it is *impossible* to simply replace one reasoner with another, without code modification). Instead, one has to consider the particular behavior of the specific reasoner while implementing functionality similar to that in the `OntologyReader` interface. We hope that, in the near future, this problem will be addressed in a way that will lead to a common semantics of semantic reasoning.

The remaining, auxiliary, part of the application – the *JADE Gateway* component is created as a standalone Java library, exposing an API for initiating agent conversations, sending messages and retrieving contents of a specific conversation. After some additional testing, the *JADE Gateway* and the ontology builder user interface are going to be released as *Play!* modules, to be easily integrated into other *Play!* applications. Furthermore, the *JADE Gateway* will be released as a *JADE* add-on.

It is worth noting that the *AJAX* functionality for listening on agent's responses has been developed as a jQuery plugin, enabling its easy embedding into any HTML page. Currently, the plugin is implemented using a simple polling mechanism controlled on the browser-side with a request sent to the Server component every specific number of seconds. In the future versions of the *JADE Gateway* this mechanism will be replaced with a less server-consuming implementation based on the *Comet / Long Polling* mechanism [1].

8 Concluding remarks

One of key issues of semantic systems (including systems involving various forms of semantic negotiations) is their usability among ontology-illiterate users. The aim of this paper was to present an in-depth discussion of the development of a front-end application for semantic (and agent-semantic) systems that solves this problem. The developed *OntoPlay* plugin allows creation of ontology-driven user interfaces for any semantic system. These interfaces are generated dynamically on the basis of user selections, from entities featured in the ontology. Furthermore, *OntoPlay* is agnostic to the changes in the underlying ontology, thus simplifying semantic system adaptation (i.e. changes in the ontology automatically materialize in the interface). Finally, *OntoPlay* can be used to connect a web-based front-end application to a *JADE*-based agent system (e.g. facilitating autonomous agent negotiations). Code of *OntoPlay* is available as part of the *AiG* project at *http://gridagents.svn.sourceforge.net/viewvc/gridagents/trunk/AiG/QueryBuilder/* and we welcome comments and suggestions for its further development.

References

1. Comet and Reverse Ajax: The next-generation Ajax 2.0. `http://dl.acm.org/citation.cfm?id=1453096`.
2. Protègè documentation: ontology development 101. `http://protege.stanford.edu/publications/ontology_development/ontology101.html`.
3. Fensel Dieter. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag, New York, 2003.
4. Michał Drozdowicz, Katarzyna Wasielewska, Maria Ganzha, Marcin Paprzycki, Naoual Attaui, Ivan Lirkov, Richard Olejnik, Dana Petcu, and Costin Badica. Trends in parallel, distributed, grid and cloud computing for engineering. chapter Ontology for Contract Negotiations in Agent-based Grid Resource Management System. Saxe-Coburg Publications, Stirlingshire, UK, 2011.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, January 1995.
6. Yong Gao, June Kinoshita, Elizabeth Wu, Eric Miller, Ryan Lee, Andy Seaborne, Steve Cayzer, and Tim Clark. Swan: A distributed knowledge infrastructure for alzheimer disease research. *Web Semant.*, 4(3):222–228, September 2006.
7. Maciej Gawinecki, Minor Gordon, Paweł Kaczmarek, and Marcin Paprzycki. The problem of agent-client communication on the internet. In *Scalable Computing: Practice and Experience*, volume 6(1), pages 111–123, 2005.
8. Minor Gordon, Marcin Paprzycki, and Violetta Galant. Agent-client interaction in a Web-based E-commerce system. In Dan Grigoras, editor, *Proceedings of the International Symposium on Parallel and Distributed Computing*, pages 1–10, Ias,i, Romania, 2002. University of Ias,i Press.
9. James Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, 16(2):30–37.
10. Tim Berners Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American Magazine*, 2001.
11. Phillip Lord, Sean Bechhofer, Mark D. Wilkinson, Gary Schiltz, Damian Gessler, Duncan Hull, Carole Goble, , Lincoln Stein, Damian Gessler, and Duncan Hull. Applying semantic web services to bioinformatics experiences gained, lessons learnt, 2004.
12. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.
13. Christian J. Stoeckert and Helen Parkinson. The MGED ontology: A framework for describing functional genomics experiments. *Comprative and Functional Genomics*, 4(1):127–132, 2003.
14. Katarzyna Wasielewska, Michał Drozdowicz, Maria Ganzha, Marcin Paprzycki, Naoual Attaui, Dana Petcu, Costin Badica, Richard Olejnik, and Ivan Lirkov. Trends in Parallel, Distributed, Grid and Cloud Computing for engineering. chapter Negotiations in an Agent-based Grid Resource Brokering Systems. Saxe-Coburg Publications, Stirlingshire, UK, 2011.
15. Katarzyna Wasielewska, Michał Drozdowicz, Paweł Szmeja, Maria Ganzha, Marcin Paprzycki, Ivan Lirkov, Dana Petcu, and Costin Badica. Agents in grid system—design and implementation. In *Large Scale Scientific Computing*, volume 7116 of *LNCS*, pages 654–661. Springer Germany, 2012.
16. Wei Xing, Marios D. Dikaiakos, Rizos Sakellariou, Salvatore Orlando, and Domenico Laforenza. Design and Development of a Core Grid Ontology. In *Proc. of the CoreGRID Workshop: Integrated research in Grid Computing*, pages 21–31, 2005.