Improving Portlet Interoperability Through Deep Annotation

Oscar Díaz
The ONEKIN group
Univ. of the Basque Country
P.O. Box 649
San Sebastian, Spain, 20080
jipdigao@si.ehu.es

Jon Iturrioz
The ONEKIN group
Univ. of the Basque Country
P.O. Box 649
San Sebastian, Spain, 20080
jipitsaj@si.ehu.es

Arantza Irastorza
The ONEKIN group
Univ. of the Basque Country
P.O. Box 649
San Sebastian, Spain, 20080
jipirgoa@si.ehu.es

ABSTRACT

Portlets (i.e. multi-step, user-facing applications to be syndicated within a portal) are currently supported by most portal frameworks. However, there is not yet a definitive answer to portlet interoperation whereby data flows smoothly from one portlet to a neighbouring one. Both data-based and API-based approaches exhibit some drawbacks in either the limitation of the sharing scope or the standardization effort required. We argue that these limitations can be overcome by using deep annotation techniques. By providing additional markup about the background services, deep annotation strives to interact with these underlying services rather than with the HTML surface that conveys the markup. In this way, the *portlet producer* can extend a portlet markup, a fragment, with data about the processes whose rendering this fragment supports. Then, the *portlet consumer* (e.g. a portal) can use deep annotation to map an output process in fragment A to an input process in fragment B. This mapping results in fragment B having its input form (or other "input" widget) filled up. We consider deep annotation as particularly valid for portlet interoperation due to the controlled and cooperative environment that characterizes the portal setting.

Keywords: portlet interoperability, portal ontology, data-flow, deep-annotation, event.

General Terms: Design, Standardization.

Categories: D.2.11 - Software Architectures; D.2.12 - Interoperability; D.2.13 - Reusable Software; H.3.4 - Systems and Software; H.3.5 - Online Information Services.

1. INTRODUCTION

The significance of portal applications stems not only from being a handy way to access data but also from being the means of facilitating the *integration* with third party applications. This has led to the so-called *portal imperative*: the emergence of portal software as a universal integration mechanism [19].

Key to this view is the notion of *portlet*. Portlets are applications within a portal in much the same way as servlets are applications within a Web server. The difference stems from portlets being multi-step, user-facing applications. They are very much like Windows applications in a user desktop in the sense that a portlet renders markup *fragments* that are surrounded by a decoration containing controls. The portal page then, contains a number of portlets whose fragments can be arranged into columns and rows, and minimized, maximized, or arranged to suit the user needs.

However, aggregating portlets into a portal is more than merely invoking these services, or arranging their fragments together in the same portal page (i.e. the so-called *"side-by-side"* aggregation). Information contained in one portlet will surely be required in another, and forcing the individual user to manually copy and key in data from source to target portlets leads to frustration, lost productivity, and inevitable mistakes. And this situation certainly hinders the fulfillment of the portal imperative.

According to the IEEE Standard Computer Dictionary, interoperability means "the ability of two or more systems or components to exchange information and to use the information that has been exchanged". To achieve this end in a portlet context, distinct mechanisms have been proposed which can be classified as data-based and API-based. The former permits distinct portlets share a common piece of information but within the scope of the same producer. Portlets which pertain to distinct producers remain isolated. On the other hand, the API-based approach facilitates a programmatic interface for portlets to communicate their state to interested parties. Unfortunately, at the time of this writing, there is not yet an agreement on how to standardize this mechanism.

To overcome some of these drawbacks, this paper presents a *deep annotation approach* to portlet interoperation. Rather than resorting to back-end solutions, we support a front-end approach, i.e., the visual part of a portlet, the fragments, are supplemented with information about what these fragments render. This requires the creation, either manually or semi-automatically, of meta-data from existing information, a process known as *annotation* [5]. However, most of the approaches to annotation build on the assumption that the information sources are static (e.g. static HTML pages). This is not always the case for Web pages nor is it for portlets. As stated in [6], *"for dynamic web pages (e.g. ones that are generated from a database...) it does not seem to be useful to manually annotate every single page. Rather one wants to annotate the database in order to reuse it for one's own Semantic Web purpose"*. This leads to the notion of *deep annotation*.

Deep annotation has been proposed in [7] as an annotation process that *"utilizes information proper, information structures and information context in order to derive mappings between information structures"*. This process is called deep annotation *"as its purpose is not to provide semantic annotation about the surface of what is being annotated, this would be the web page* (in our case, the portlet fragment)*, but about the semantic structures in the background"* [1]. Deep annotation permits *querying parties* to interact with the background structure without the help of the HTML "surface". The HTML "surface" is used to obtain the underlying structure, e.g., the database schema. From then on, the underlying structure, the database, can be consulted without the need of the HTML page (e.g. through a Web service).

This paper presents how deep annotation is used for portlet interoperation. The key aspects of the approach can be summarized as follows:

1. portlets are characterized by their ontologies. Although none of the portlet standards (i.e. WSRP [12] and JSR168 [10]) contemplate this option, the extensibility mechanisms available in both standards can be used to extend the portlet description with an additional *ontology* property. Besides facilitating portlet interoperability, all the benefits of using explicit ontologies (e.g., better documentation, search, knowledge acquisition [9]) are brought to the portlet realm.

2. portlet fragments extend their markups with information about the processes these fragments support. So far, the fragment markup is geared towards rendering (e.g., XHTML). Now, this markup also conveys information about the underlying processes. This idea comes from deep annotation works.

3. portlet interoperability is achieved through mappings of the ontology instances. Mapping is necessary as portlet producers can have their own ontologies, and mapping is required to indicate how instantiations from one portlet “flows” to a neighbouring portlet.

Compared with back-end approaches, this mechanism makes explicit what is hidden in the data-based approach, and unlike the API-based proposal, requires no agreement with other portlet producers.

A final remark. As noted in [7], deep annotation relies on the cooperation of the markup producer who has to embed the “underlying information structure” into the HTML markup. Indeed, our approach rests on fragments being supplemented with information about the underlying processes. We argue that this assumption (i.e. producers cooperation) is valid here. The argument is two-fold. First, the additional effort required by this extra markup pays off in terms of achieving portlet interoperability. This in turn, leads to improve the user experience of the portals where these portlets are syndicated. Hence, portal masters will favor those portlet producers that facilitate this feature.

Second, the mistrust to share the ontology can be overcome by requiring prior registration. It is a common scenario to require a portal to register with the producer prior to use its portlets (e.g. for charging matters). Registration ensures a controlled environment where the producer can feel confident when disclosing its ontology.

The rest of this paper is organized as follows. First, section 2 outlines the notion of portlet. Next, the deep annotation process is particularized for our scenario where each contention is addressed in a separate section. Related work is presented in section 7. Finally, some conclusions are drawn.

2. A PORTLET BRIEF

Web services are an XML-centric means for integrating modular programs over the Web using open, standardized interfaces. However, the traditional use of Web services stops at the functional-integration layer. Web service standards facilitate the sharing of the business logic, but suggest that Web service consumers should write a new presentation layer on top of this business logic.

As an example, consider a Web service that offers two operations, namely, *searchFlight* and *bookFlight*. The former retrieves flights that match some input parameters (e.g. *departureAirport, flightDates* and so on). On the other hand, *bookFlight* takes the selected flight and payment data, and books a seat on this flight. This WSDL-based API can then be used by a consumer application as follows. First, the application would collect the *departureAirport, flightDates* and other parameters via an input form. Within the form, an *http* request might support a call to *searchFlight* which, in turn, returns a set of flights whose presentation is left to the calling application. Next, the user selects one of the flights and, through another form, the Web application collects the user’s information and payment data. This interaction will in turn invoke the *bookFlight* operation. This example illustrates the traditional approach where Web services provide the business logic, and both presentation and control layers are left to the calling application.

However, such an approach underscores the presentation layer. This layer not only addresses aesthetic aspects, but a whole range of concerns like usability issues, state management, error handling, client-side scripting, etc [16]. Indeed, most of the aspects that characterize a good Web site are related to interactive issues [11]. Recreating this interactive logic in each consumer application has potentially two main limitations, namely, it increases time-to-market, and it jeopardizes the company’s image.

To overcome these limitations, a portlet provides a coarser-grained component that includes both the business logic and the presentation logic. Portlets are currently a main building block for portal construction, being supported by the main IDE players[1]. A main step forward towards portlet interoperability between IDE vendors has been the delivery of the *Web Services for Remote Portlets* (WSRP) specification (April, 2003) [12] by OASIS, and the *Java Portlet Specification* (JSR168) [10] (October, 2003) by JCP. The goal is to define a component model that would enable portlets to be easily plugged into standards-compliant portals.

Let’s go back to our sample application, but now delivering it as a portlet. A *bookFlight* portlet encapsulates the previous screenshot sequence, and regulates its rendering. Each output markup is referred to as a fragment. A portlet consumer (e.g. a portal) might register with the producer of *bookFlight* to syndicate this portlet as part of the portal’s offerings: the fragments of *bookFlight* are rendered in this portal. All the portal does is basically routing the interactions of the user with the fragment to the corresponding portlet producer.

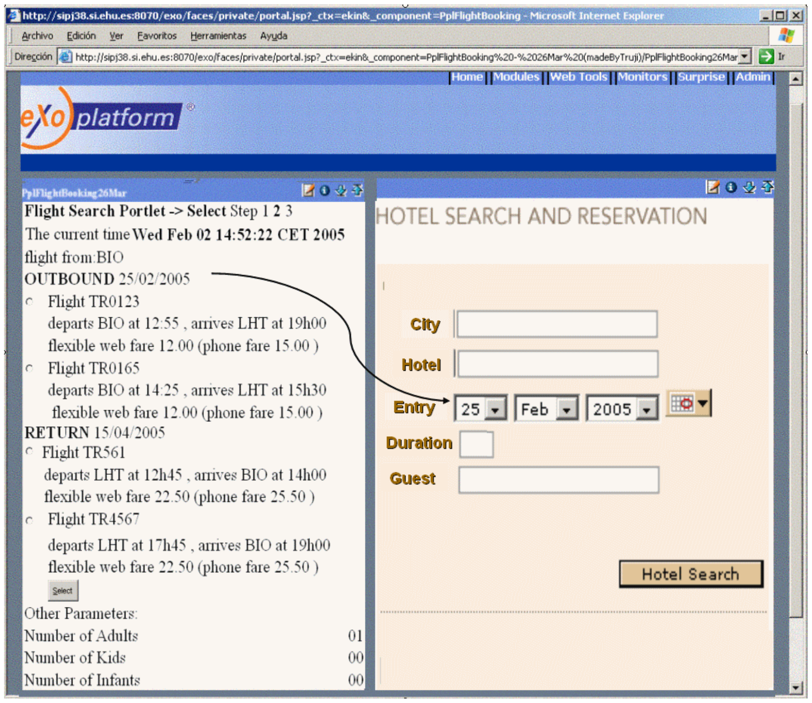
According with the WSRP protocol, the lifecycle of a portlet session begins when the first *getMarkup()* request is issued. This causes the first fragment to be rendered. Next, the user interacts with the fragment. If this interaction does not affect other portlets of the same producer being syndicated in the same portal (e.g. as a result of sharing some database), *getMarkup()* is invoked. Otherwise, *performBlockingInteraction()* is first issued, and second, *getMarkup()* is sent to all the portlets of this producer[2]. In this way, a single user interaction can change the output of distinct portlets. But this interoperation takes places at the back-end. Next section gives some additional details.

2.1 Portlet interoperation

For the purpose of this paper, we define portlet aggregation as the combination of a set of portlets to achieve a common goal. This aggregation can be totally unconstrained but even so, provide some

[1]Portlets are endorsed and supported by IBM, BEA, Oracle, Sybase, Viador, Verify, PeopleSoft, Plumtree and Vignette. It is important to notice that portlets are also referred by some vendors as *Web parts* or *gadgets*.

[2]The portal knows whether a user interaction affects distinct portlets through the links being clicked on. Links convey information about potential side-effects on other portlets of the provider.


http://sipj38.si.ehu.es:8070/exo/faces/private/portal.jsp?_ctx=ekin&_component=PplFlightBooking - Microsoft Internet Explorer
Archivo Edición Ver Favoritos Herramientas Ayuda
Dirección http://sipj38.si.ehu.es:8070/exo/faces/private/portal.jsp?_ctx=ekin&_component=PplFlightBooking%20-%2026Mar%20(madeByTruji)/PplFlightBooking26Mar
Home Modules Web Tools Monitors Surprise Admin
eXo platform
PplFlightBooking26Mar
HOTEL SEARCH AND RESERVATION
City
Hotel
Entry
Duration
Guest
Hotel Search
Select

Flight Search Portlet -> Select Step 1 2 3
The current time Wed Feb 02 14:52:22 CET 2005
flight from:BIO
OUTBOUND 25/02/2005
Flight TR0123
departs BIO at 12:55 , arrives LHT at 19h00
flexible web fare 12.00 (phone fare 15.00)
Flight TR0165
departs BIO at 14:25 , arrives LHT at 15h30
flexible web fare 12.00 (phone fare 15.00)
RETURN 15/04/2005
Flight TR561
departs LHT at 12h45 , arrives BIO at 14h00
flexible web fare 22.50 (phone fare 25.50)
Flight TR4567
departs LHT at 17h45 , arrives BIO at 19h00
flexible web fare 22.50 (phone fare 25.50)
Other Parameters:
Number of Adults 01
Number of Kids 00
Number of Infants 00

Entry 25 Feb 2005

Figure 1: Two portlets side-by-side: *bookFlight* on the left, and *bookHotel* on the right.

value to the user since portlets are simultaneously rendered in the same portal page. Here, the portal acts as a unified access point to the user. However, tighter forms of aggregation leverage portal functionality to that of a proper workplace where portlets share a common goal. This implies some kind of interoperation between the portlets. So far, the proposed mechanisms can be classified as data-based and API-based.

Data-based mechanisms permits distinct portlets share a common piece of information. This approach is followed by the notion of "portlet application" introduced by the JSR168 standard [10]. JSR168 defines a standard interface for portlets implemented for the Java platform and specifies the contract between a portlet and its container. In a J2EE architecture, a Web application refers to an aggregate of Web components such as JSP or Servlets which are packaged together into a WAR archive. Likewise, a "portlet application" is a Web application which includes a special kind of Web component, namely, the portlets. All the portlets contained within the scope of a "portlet application" can share some data. This is known as the "application" scope. Objects with an application scope can be shared among distinct requests issued by portlets which pertain to the same application. However, a portal normally frames portlets from distinct "portlet applications". Portlets which pertain to distinct "portlet applications" remain isolated.

API-based approaches provide a programmatic interface for portlets to communicate their state to interested parties. This approach has been proposed at both the portlet producer and the portlet consumer (e.g. a portal) side. At the producer side, the JSR168 envisages an event-based mechanism, similar to the one available for Java Beans that permits portlets to subscribe to events generated by other portlets.

As in the data-based case, the main drawback rests on the exchange being limited to a single producer. Therefore, if the exchange implies portlets from distinct producers then, this concern should be moved to the consumer side, e.g. the portal. An example of this latter approach is presented in [17]. To enable a portlet to be a source of data, fragments include a custom JSP tags that flag sharable data on the output markups. On the other hand, to enable a portlet to be a target, a new API is included that specifies the actions that can be invoked.

Unfortunately, there is not yet an agreement on how to standardize this mechanism. Indeed, standardizing this API would lead to commoditize one of the most valuable offerings of portal vendors. Hence, vendors might be inclined to retain this competitive advantage rather than commoditizing it, and enabling other companies to exploit their application logic and infrastructure functions on top of it. The WSRP committee is working actively on this issue. But, even if an API-based standard for portlet interoperation is finally agreed upon, ontology-based interoperation can facilitate the declarative specification of the mapping between the realms of two distinct portlet providers, rather than this mapping being hidden in the portlet code. Some experiences for Web services shed light on this topic [14][18].

Based on these observations, i.e. the limitation of the sharing scope and the standardization effort required, this work introduces an approach to portlet interoperability using deep annotation.

3. DEEP ANNOTATION

An annotation *"is a set of instantiations related to an ontology and referring to an HTML document"* [7]. Traditional annotation provides meta-data about the surface of what is being annotated, e.g., an HTML page. By contrast, deep annotation strives to capture the semantic structures in the background. For dynamic Web pages, now, the page also conveys the tables, attributes and the query used to recover the content being rendered in the page. This information structure/context (i.e. tables, attributes, query) can now be annotated (i.e. mapped) to the information structures/context of the

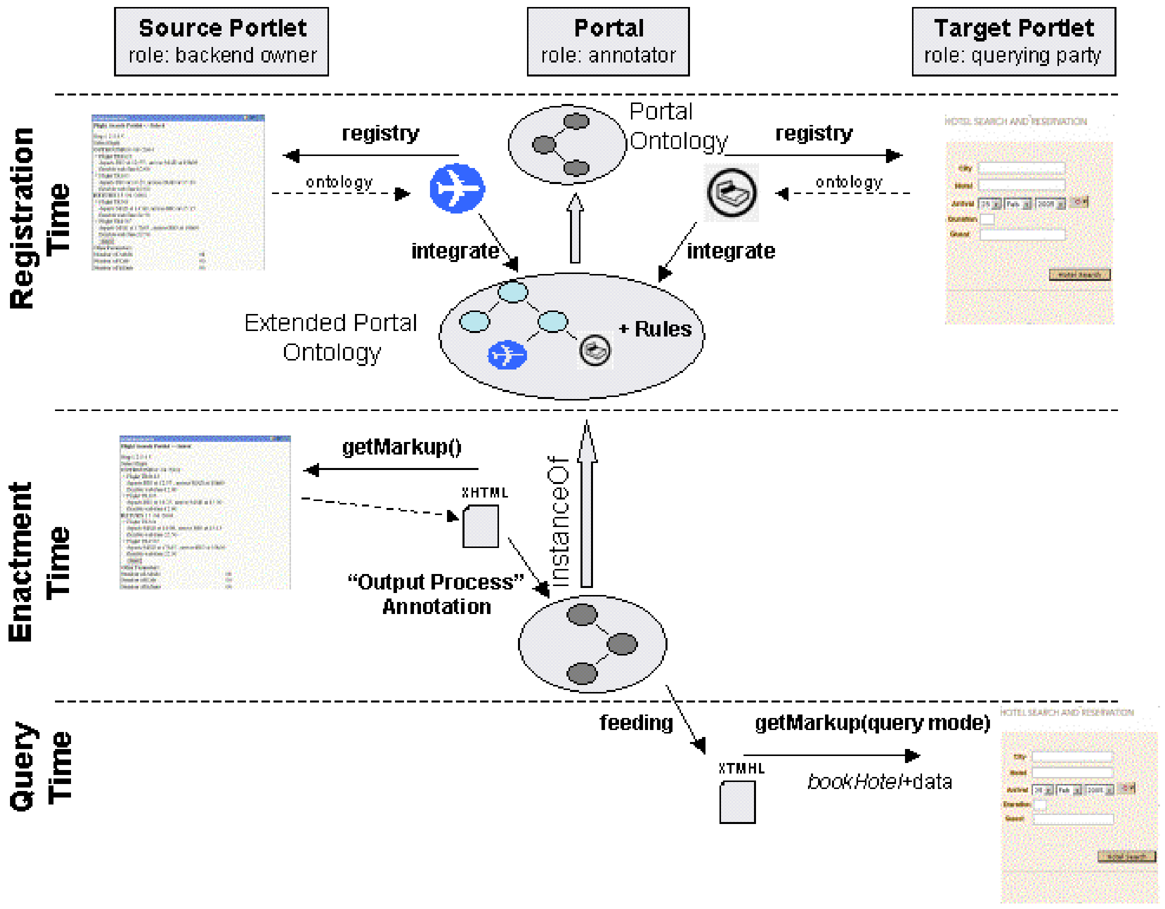


Figure 2: An architecture for deep annotation adapted for the portlet case.

client, and in so doing, permits the client to consult the database without resorting to the HTML surface.

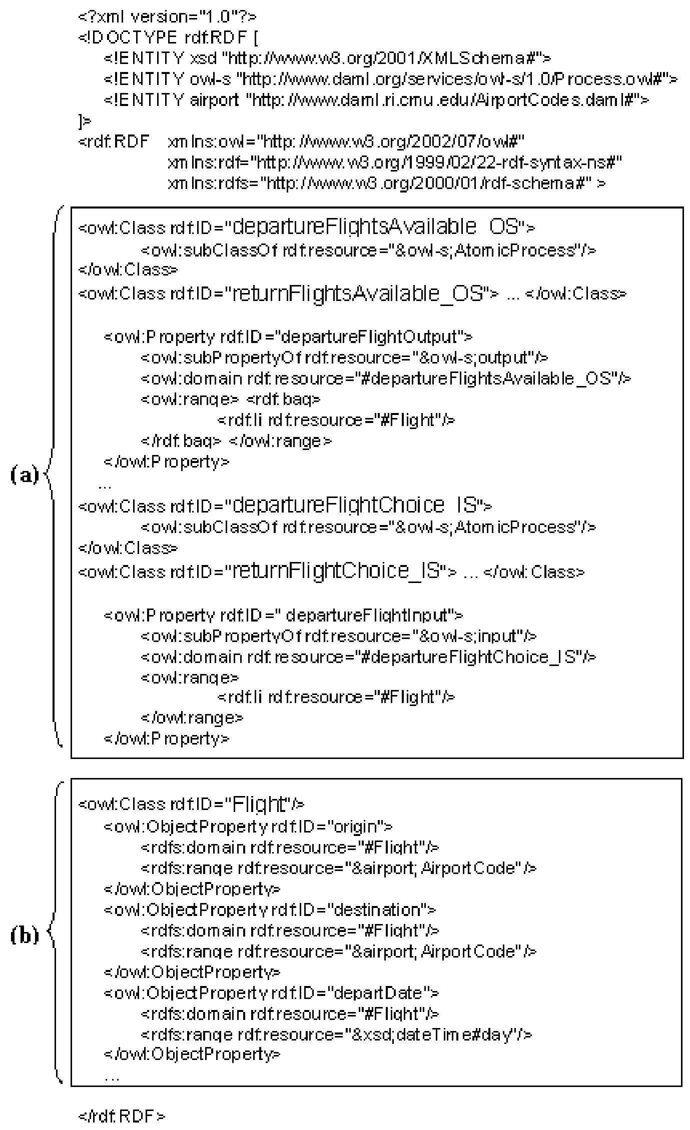
According with the proponents, deep annotation involves three actors: the backend owner (e.g., the database administrator), the annotator, and the querying party. If the backend resource is a database as illustrated in [7], then these actors interact as follows:

1. The *backend owner* produces server-side web page markup according to the database's information structures. The outcome is a set of HTML pages that convey not only the data but also which database columns provide the data (among other aspects).
2. The *annotator* produces client-side annotations which conform to the client ontology and the server-side markup. In this context, an annotation is a set of instantiations related to a (client) ontology and referring to a (server-based) HTML document.
3. The *annotator* publishes the client ontology and the mapping rules derived from annotations. The goal of the mapping process is to give interested parties access to the source data. All information, including the structure of all tables involved in a Web site query, must be published so that users can retrieve data.
4. The *querying party* loads client's ontology and mapping rules, and uses them to query the information source via a web service API, and without the intervention of the HTML page.

This paper argues that this approach can also be used for portlet interoperation. As an example, consider a portal that syndicates two portlets, one for flight booking, *bookFlight*, and the other for hotel booking, *bookHotel* (see figure 1). We want these two portlets to interoperate so that data can flow smoothly from the former to the latter. That is, *bookHotel* can render the fragment which prompts for the *entry-date* already filled up from the *arrival-date* obtained after enacting *bookFlight*.

In this scenario, the *backend owner* corresponds to the source portlet *bookFlight;* the *querying party* maps to the target portlet *bookHotel*; and the *annotator* role is played by the portal. Figure 2 gives an overview of this approach:

- At registration time, the portal loads the ontologies for the distinct portlets, and integrates them into the portal's ontology.
- At enactment time, fragments are annotated according with the portal's ontology. The portal keeps track of the distinct interactions with the portlets in terms of instantiations of the portal's ontology.
- At query time, target portlets can use these instantiations "to feed" their fragments.



```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
    <!ENTITY owl-s "http://www.daml.org/services/owl-s/1.0/Process.owl#">
    <!ENTITY airport "http://www.daml.ri.cmu.edu/AirportCodes.daml#">
]>
<rdf:RDF   xmlns:owl="http://www.w3.org/2002/07/owl#"
           xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >
```

(a)

```
<owl:Class rdf:ID="departureFlightsAvailable_OS">
        <owl:subClassOf rdf:resource="&owl-s;AtomicProcess"/>
</owl:Class>
<owl:Class rdf:ID="returnFlightsAvailable_OS"> ... </owl:Class>

    <owl:Property rdf:ID="departureFlightOutput">
        <owl:subPropertyOf rdf:resource="&owl-s;output"/>
        <owl:domain rdf:resource="#departureFlightsAvailable_OS"/>
        <owl:range> <rdf:bag>
                <rdf:li rdf:resource="#Flight"/>
        </rdf:bag> </owl:range>
    </owl:Property>
    ...
<owl:Class rdf:ID="departureFlightChoice_IS">
        <owl:subClassOf rdf:resource="&owl-s;AtomicProcess"/>
</owl:Class>
<owl:Class rdf:ID="returnFlightChoice_IS"> ... </owl:Class>

    <owl:Property rdf:ID=" departureFlightInput">
        <owl:subPropertyOf rdf:resource="&owl-s;input"/>
        <owl:domain rdf:resource="#departureFlightChoice_IS"/>
        <owl:range>
                <rdf:li rdf:resource="#Flight"/>
        </owl:range>
    </owl:Property>
```

(b)

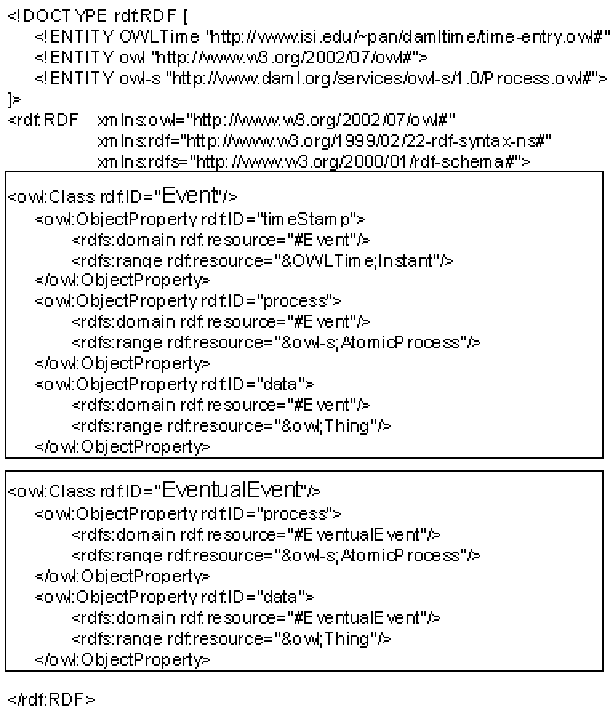
```
<owl:Class rdf:ID="Flight"/>
    <owl:ObjectProperty rdf:ID="origin">
        <rdfs:domain rdf:resource="#Flight"/>
        <rdfs:range rdf:resource="&airport; AirportCode"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="destination">
        <rdfs:domain rdf:resource="#Flight"/>
        <rdfs:range rdf:resource="&airport; AirportCode"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="departDate">
        <rdfs:domain rdf:resource="#Flight"/>
        <rdfs:range rdf:resource="&xsd;dateTime#day"/>
    </owl:ObjectProperty>
    ...

</rdf:RDF>
```

Figure 3: The portlet's ontology: task ontology (a) + domain ontology (b).

This scenario raises the following issues:

1. Defining the ontologies for the portlets and the portal.
2. Fragment annotation, i.e. producing a set of instantiations related to the portal's ontology and referring to the fragment markups of a source portlet.
3. Fragment querying, i.e. "feeding" the markup of a target portlet from annotations kept by the portal.

Next sections address these concerns with the help of a running example.



```
<!DOCTYPE rdf:RDF [
    <!ENTITY OWLTime "http://www.isi.edu/~pan/damltime/time-entry.owl#"
    <!ENTITY owl "http://www.w3.org/2002/07/owl#">
    <!ENTITY owl-s "http://www.daml.org/services/owl-s/1.0/Process.owl#">
]>
<rdf:RDF   xmlns:owl="http://www.w3.org/2002/07/owl#"
           xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<owl:Class rdf:ID="Event"/>
    <owl:ObjectProperty rdf:ID="timeStamp">
        <rdfs:domain rdf:resource="#Event"/>
        <rdfs:range rdf:resource="&OWLTime;Instant"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="process">
        <rdfs:domain rdf:resource="#Event"/>
        <rdfs:range rdf:resource="&owl-s;AtomicProcess"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="data">
        <rdfs:domain rdf:resource="#Event"/>
        <rdfs:range rdf:resource="&owl;Thing"/>
    </owl:ObjectProperty>

<owl:Class rdf:ID="EventualEvent"/>
    <owl:ObjectProperty rdf:ID="process">
        <rdfs:domain rdf:resource="#EventualEvent"/>
        <rdfs:range rdf:resource="&owl-s;AtomicProcess"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="data">
        <rdfs:domain rdf:resource="#EventualEvent"/>
        <rdfs:range rdf:resource="&owl;Thing"/>
    </owl:ObjectProperty>

</rdf:RDF>
```

Figure 4: The portal's ontology (an excerpt).

4. PORTLET ONTOLOGY AND PORTAL ONTOLOGY

Portlet ontology. For the purpose of this work, a portlet is characterized by the set of processes that can occur along its life-cycle. We are only interested in what the portlet provides and what the portlet requests. To describe both input and output operations, *OWL-S Atomic Processes* is used as the baseline ontology [4]. OWL-S is an initiative of the Semantic Web community to facilitate automatic discovery, invocation, composition, interoperation and monitoring of Web services through their semantic description. OWL-S is an OWL ontology conceptually divided into three sub-ontologies for specifying what a service does (profile), how the service works (process) and how the service is implemented (grounding). This work focuses on the process side.

A portlet ontology includes a task ontology, along the lines of OWL-S, and a domain ontology to describe the parameters of the task ontology. As an example, consider the *bookFlight* portlet. This portlet comprises a set of fragments that realizes a multi-step process that ends with the booking of a flight. First, the first fragment collects the *departureAirport*, *flightDates* and so on. Available flights matching these criteria are rendered in the second fragment where the user is prompted to select one of these flights. And so on.

The portlet's ontology, *bookFlightOnto*, reflects this process as a collection of input and output OWL-S atomic process: *returnFlightsAvailable_OS*, *departureFlightChoice_IS* and the like. Figure 3 shows an excerpt of this ontology where the suffix *OS* (output service) and *IS* (input service) denote output and input Atomic Processes, respectively[3].

[3]It should be noted that for stable domains, this ontology can be standardized in the same way that EDI technologies force the standardization of document formats. The Open Travel Alliance,

Although it has not been implemented yet, this basic ontology can now be extended to specify the order in which processes proceeds or the relationships between their parameters. For instance, it can be stated that *departureFlightsAvailable_OS* should precede *departureFlightChoice_IS*, and that, at enactment time, the *departureFlightInput* parameter of the latter should be one of the values returned as the *departureFlightOutput* parameter of *departureFlightsAvailable_OS*. To this end, orchestration languages can be used [15].

Portal ontology. For the purpose of this paper, the role of the portal is restricted to be a mere mediator among the portlets. The portal is just a container for portlets with no content on its own. The portal acts as a controller. Based on this perspective, all that matters are the *events* that occur during portlet's enactment.

Hence, the portal's ontology includes two main classes: the *event* class and the *eventualEvent* class (see figure 4). The former describes a happening of interest, and its description includes the following properties: the *process* being enacted, which keeps an OWL-S Atomic Process; the *timestamp* at which this process was enacted whose range is *OWLTime Instant* [13]; and, the *data* of the process, which holds a *Thing*.

As for an *eventual event*, it represents a happening that might occur in the future. A portal offers a set of portlets where each portlet might display distinct course of action for the end user to follow. Eventual events capture the permitted range of actions an end user can click-on at a given moment. In our example, the booking of a flight may eventually lead to the booking of a hotel. The booking of a hotel is then an eventual event. Once the hotel is booked, it becomes an event. Section 6 describes the rationales behind the notion of eventual event.

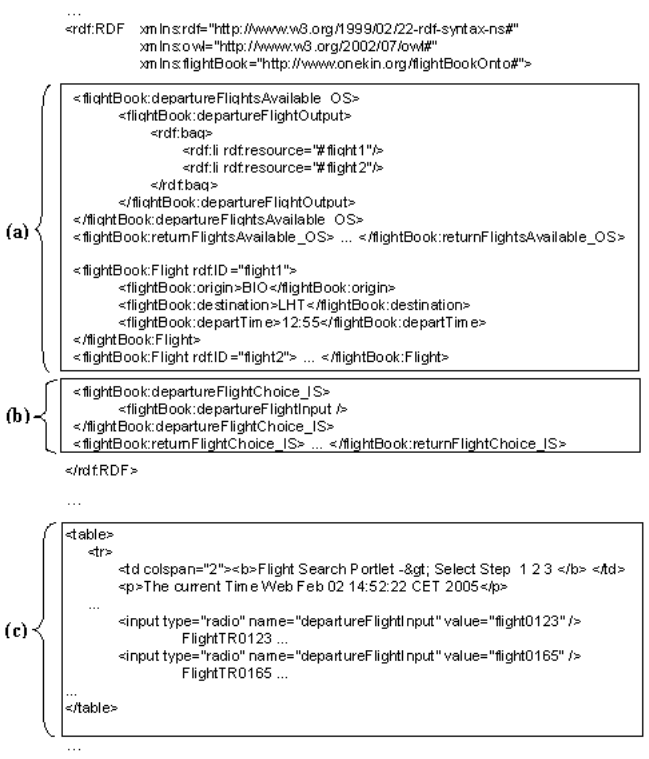
It is worth mentioning that the *data* property keeps a *Thing* (see figure 4). In our context, this "thing" stands for any of the domain classes of the portlet ontologies. For instance, a thing can be a flight, a city, a hotel, etc. As these domain classes come from distinct ontologies, the portal master must solve first potential mismatches and ontology mappings between the different portlet ontologies. Mapping may become necessary as distinct communities can have their own terms and regulations (e.g. the *bookFlight* portlet follows the *Open Travel Alliance* standards whereas *bookHotel* conforms to the normative of a different committee). Ontology mapping is a tough issue whose implications are outside the scope of this paper. But ontology mapping is a must to achieve portlet interoperability, no matter which approach is used.

5. FRAGMENT ANNOTATION

Broadly speaking, a portlet fragment is a chunk of XHTML code (or any other rendering language). So far, the portlet producer delivers this fragment with the only purpose of being readily rendered by the portal.

By contrast, deep annotation is a more demanding scenario where the very same portlet can play two roles. As a *backend owner*, fragments can additionally convey which output processes are used to obtain the content of the fragment. On the other hand, as a *querying actor,* fragments should indicate which kind of "queries" a fragment can pose. These queries correspond to widgets such as entry forms which, so far, can only be "answered" by the end-user. The ontological counterpart of these widgets are the input processes.

www.opentravel.org, is a case in point. This consortium defines XML Schemas and corresponding usage scenarios for messages that support business activities in the travel industry. This standard can be "OWL-ized", and used for deep annotating travel web sites.



```
...
<rdf:RDF   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:owl="http://www.w3.org/2002/07/owl#"
           xmlns:flightBook="http://www.onekin.org/flightBookOnto#">

 <flightBook:departureFlightsAvailable_OS>
        <flightBook:departureFlightOutput>
              <rdf:bag>
                    <rdf:li rdf:resource="#flight1"/>
                    <rdf:li rdf:resource="#flight2"/>
              </rdf:bag>
        </flightBook:departureFlightOutput>
 </flightBook:departureFlightsAvailable_OS>
 <flightBook:returnFlightsAvailable_OS> ... </flightBook:returnFlightsAvailable_OS>

 <flightBook:Flight rdf:ID="flight1">
        <flightBook:origin>BIO</flightBook:origin>
        <flightBook:destination>LHT</flightBook:destination>
        <flightBook:departTime>12:55</flightBook:departTime>
 </flightBook:Flight>
 <flightBook:Flight rdf:ID="flight2"> ... </flightBook:Flight>

 <flightBook:departureFlightChoice_IS>
        <flightBook:departureFlightInput />
 </flightBook:departureFlightChoice_IS>
 <flightBook:returnFlightChoice_IS> ... </flightBook:returnFlightChoice_IS>

</rdf:RDF>

...

<table>
    <tr>
          <td colspan="2"><b>Flight Search Portlet -&gt; Select Step  1 2 3 </b> </td>
          <p>The current Time Web Feb 02 14:52:22 CET 2005</p>
    ...
          <input type="radio" name="departureFlightInput" value="flight0123" />
                     FlightTR0123 ...
          <input type="radio" name="departureFlightInput" value="flight0165" />
                     FlightTR0165 ...
...
</table>

...
```

Figure 5: The markup of the sample fragment of *bookFlight* (an excerpt).

Consider our sample fragment of the *bookFlight* portlet (see figure 1). A snippet of its markup is given in figure 5 where three distinct parts can be distinguished, namely:

- structure/context information markup (see figure 5 (a)). Specifically, for each "output" markup chunk (i.e. the one that renders a meaningful set of data), an additional markup is inlaid where the outcome is conceived as the result of a parameterless function. Our sample fragment conveys two output Atomic Processes (i.e. *departureFlightsAvailable_OS*, and *returnFlightsAvailable_OS*). Each Atomic Process comprises its actual parameters. Process parameters correspond to instantiations of the domain ontology of the portlet, i.e. *flightBookOnto*. The *flightBook* namespace is introduced with this purpose.

- query-oriented markup (see figure 5 (b)), which embeds the type of queries this portlet can make. Specifically, for each "input" widget (e.g. an entry form), an additional markup is introduced where the widgets are conceived as the realization of an input-only atomic process of the portlet's ontology. Our sample fragment includes two input Atomic Processes (i.e. *departureFlightChoice_IS* and *returnFlightChoice_IS*).

- rendering-oriented markup (see figure 5 (c)), whose purpose is to be interpreted by the browser.

This additional markup permits deep annotating, i.e. the process of mapping from the information structures found in the portlet's markup to the information structures of the portal. Here, the portal acts as the annotator which automatically produces a set of instantiations related to the portal's ontology, and referring to the portlet's

```
<ontopipe:Event>
  <ontopipe:timeStamp>
    <time:Instant>
        <time:inCalendarClockDataType rdf:datatype="&xsd;dateTime">
              2004-04-13T11:30:05
         </time:inCalendarClockDataType>
    </time:Instant>
  </ontopipe:timeStamp>
  <ontopipe:process>departureFlightSelected_OS</ontopipe:process>
  <ontopipe:data>
    <flightBook:Flight rdf:ID="flight1">
          <flightBook:origin>BIO</flightBook:origin>
          <flightBook:destination>MAD </flightBook:destination>
          <flightBook:departDate> 11/5/2004 </flightBook:departDate>
          <flightBook:departTime>12:55</flightBook:departTime>
          <flightBook:arriveTime>19:00</flightBook:arriveTime>
          <flightBook:flightCode>TR0123</flightBook:flightCode>
          <flightBook:adults>1</flightBook:adults>
          <flightBook:passenger> John Smith</flightBook:passenger>
    </flightBook:Flight>
  </ontopipe:data>
</ontopipe:Event>
```

Figure 6: Event instantiation generated as a result of the rendering of the sample fragment.

fragment. More specifically, the rendering of a fragment of a source portlet (i.e. a portlet that contains an output process) can cause the instantiation of the *event* class of the portal's ontology. These instances are kept as part of the portal state. And they will be used at query time for portlet feeding.

6. FRAGMENT QUERYING

In a traditional setting, deep annotation permits *querying parties* to interact with the background structure without the help of the HTML "surface". By contrast, we do not want to get rid of the HTML surface. One of the added-values of a portlet when compared with traditional Web Services is that it comprises the GUI, and we want to keep this interface.

The aim of our work is to use deep annotation for "feeding" fragments automatically. By "feeding" we mean the process of inlaying data into a current fragment. This data is obtained from other fragments through the event instantiations kept by the portal.

In this way, we do not do without the HTML surface. We want to interplay with the HTML surface, but with an enhanced HTML surface where entry forms are already filled up. In so doing, the end user interacts with a portlet but the effects span along multiple neighbouring portlets. Therefore, it should be stressed that "feeding" is not a substitution for end-user interaction. That is, it is always up to the end user to decide whether the hotel is booked with the parameters obtained from *bookFlight* or not.

To attain this goal, the portal should know the input processes being realized in the fragment's markup. Knowing the input processes, the portal annotates them as eventual events which, finally, are used to feed this fragment.

Implementation-wise, querying poses the following questions:

- how are instances of the eventual event class instantiated?
- when are instances of the eventual event class instantiated?
- how is a fragment fed with eventual event instances?

Next paragraphs address these questions.

```
<ontopipe:EventualEvent>
   <ontopipe:process>searchHotel_IS</ontopipe:process>
   <ontopipe:data>
       <hotelBook:Hotel rdf:ID="hotel1">
           <hotelBook:entryDate>11/05/2004</hotelBook:entryDate>
           <hotelBook:cityName>Madrid</hotelBook:cityName>
           <hotelBook:hotelName>Plaza</hotelBook:hotelName>
           <hotelBook:duration>11</hotelBook:duration>
           <hotelBook:guest>John Smith</hotelBook:guest>
       </hotelBook:Hotel>
   </ontopipe:data>
</ontopipe:EventualEvent>
```

Figure 7: Eventual event instantiation generated after the event of figure 6.

6.1 How are eventual events obtained?

A portal is seen as a *collage* of portlet fragments. Each fragment can prompt the user for distinct courses of actions: the *bookFlight* fragment is waiting for the user to select a flight, the *bookHotel* fragment is prompting the user for the date of entrance, and so on. Eventual events capture the range of actions a user can click on at a given moment.

Since eventual events have not yet occurred, their parameters are obtained from past events. In our example, (some) data about the booking of a hotel can be obtained from the previous booking of a flight. This is, first, an event instance is obtained from the process *departureFlightSelected_OS* of the *bookFlight* portlet (i.e. an output process of the next fragment of the portlet) and, next, an eventual event can be instantiated from the *searchHotel_IS* process of *bookHotel* and its parameters are obtained from those of *departureFlightSelected_OS*. Figure 7 shows the *eventualEvent* instantiation generated after the *event* of figure 6. We said there exists a **pipe** from *bookFlight* to *bookHotel* (but not vice versa).

A pipe describes a data flow from the source portlet to the target portlet. More specifically, let *Ps* and *Pt* be two portlets which play the role of the source and the target, respectively. A pipe ***Ps—Pt*** is a mapping that specifies how parameters of an *input* Atomic Process at *Pt* can be obtained from the actual values of an event caused by an *output* Atomic Process at *P*s. In general, the source of the piping can be more than one event instance which can even come from different portlets. As a portlet's input processes are known in advance, the set of pipes are pre-established as part of the portal environment.

This piping is described *à la PROLOG* using *Jena* [8]. *Jena* is a Java framework for building Semantic Web applications. The framework includes both an RDF and OWL APIs as well as persistent storage for ontologies and statements. The specification of the *bookFlight—bookHotel* pipe using a Jena rule can be found in the appendix. The outcome of the piping process is a set of eventual events ready to feed the target portlets.

6.2 When are eventual events obtained?

Portals exhibit eclectic navigation styles from hypertext-based to totally constrained ones. The former *"lets users explore a body of information freely, by following the available links without obeying to predefined sequences of actions. The power of hypertext is in their feature-rich interfaces for navigating in a non-linear way a collection of related data."* [3]. This is in contrast with workflows, i.e. software systems for directing the work of users, by superimposing control over their activities and supplying only the data needed to accomplish the currently ongoing tasks. In workflow systems, the sequence of possible actions is predetermined and the user is accompanied through the activities according to the workflow specification. Depending on the task at hand, portals can be anyway in between these two extremes of the navigation spectrum.

Querying, i.e. the process of making the data flow along one of the pre-established pipes, serves navigation. The time at which querying is enacted can be tuned to the navigation style that better fits the task at hand. Two options are possible, namely:

- *forward style*. By triggering piping rules in a forward mode, the target portlet is fed by the source portlet as soon as the source portlet is enacted. As soon as an event is risen, this happening is piped to all neighbouring portlets. In so doing, you are conducting the user towards the next task to be fulfilled, i.e. the portlets at the end of the pipe,
- *backward style*. Triggering piping rules in a backward mode implies the data-flow occurring on demand. Here, the happening of an event is not immediately propagated to the piped portlets. There is no update on the fragments of the target portlets. The end user is not distracted, and he or she can feed the target portlet on demand. Implementation-wise, this is achieved by extending the portlet decorator with an extra icon.

Jena2 includes a general purpose rule-based reasoner which is used to implement the OWL reasoner. This reasoner supports rule-based inference over RDF graphs, and provides forward chaining, backward chaining and a hybrid execution model[4]. The designer should be aware that the triggering mode can influence not only the moment at which the derived data is obtained but the data being derived as well. This stems from event occurrences being inserted in the Jena database continuously as the user interacts with the portlets.

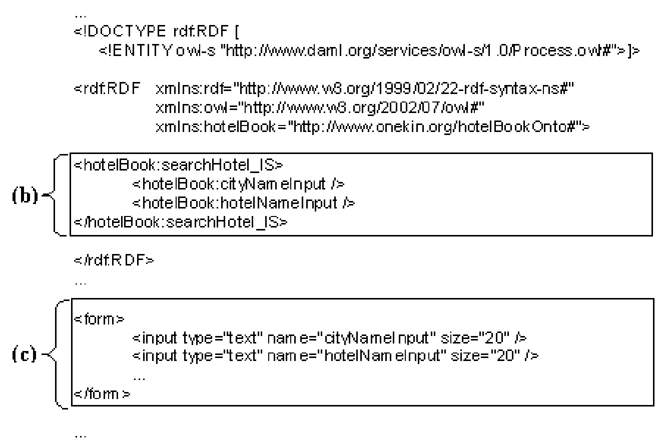
6.3 How is a fragment fed with an eventual event?

Feeding is an operation on a fragment which contains an entry widget, e.g., an entry form. This operation fills up the widget from the parameters of an eventual event instance. To this end, a convention is needed to identify which widget obtains the value of which process property. This is achieved by identifying the widget from the process property of the ontology.

Figure 8 shows a snippet of a fragment of the *bookHotel* portlet (its rendering can be seen in figure 1). The form inputs are identified from the process properties (e.g. *cityNameInput*). Feeding is then implemented as an XSLT stylesheet for the selected eventual event. A template locates the corresponding *<input>* element in the XHTML markup, and introduces a *value* attribute whose content is obtained from the corresponding parameter of the chosen eventual event. In the current implementation, this process is fulfilled by the portlet producer.

To this end, the *getMarkup()* operation has been extended with an eventual-event parameter. On reception, the provider proceeds to feed the current fragment with this parameter, and returns the result to the portal. It is worth noticing that the WSRP two-phase protocol enforces that an interaction in any portlet should cause *getMarkup()* to be invoked on <u>all</u> the portlets being syndicated. For target portlets, *getMarkup()* will now convey an additional parameter: the eventual event. For these portlets, *getMarkup()* will return the very same markup (provided no data sharing causes a state change) but with the values of the form already filled up. Now, it is up to the user to accept these values or provide her own.

[4]For clarity sake, the example uses a forward rule, although we are currently investigating a backward approach.



```
...
<!DOCTYPE rdf:RDF [
    <!ENTITY owl-s "http://www.daml.org/services/owl-s/1.0/Process.owl#">]>

<rdf:RDF   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:owl="http://www.w3.org/2002/07/owl#"
           xmlns:hotelBook="http://www.onekin.org/hotelBookOnto#">

<hotelBook:searchHotel_IS>
        <hotelBook:cityNameInput />
        <hotelBook:hotelNameInput />
</hotelBook:searchHotel_IS>

</rdf:RDF>
...

<form>
        <input type="text" name="cityNameInput" size="20" />
        <input type="text" name="hotelNameInput" size="20" />
        ...
</form>

...
```

Figure 8: The markup of the sample fragment of *bookHotel* (an excerpt).

7. RELATED WORK

Portlet interoperation has been addressed in [17] where the authors propose the use of a custom JSP tag library in order to enable portlets to be a source of data. Moreover, the target portlet is defined in a WSDL file with a custom extension to describe the actions which can consume data transferred from other portlets. At execution time, a click-able icon is inserted into the portlet fragment. By clicking on this icon, the user enacts the flow of data from the source portlet to the target portlet. Hence, this approach follows a "backward style" of navigation, and piping information is described in the WSDL file. By contrast, our approach uses the fragment markup to convey this information, and uses ontologies to facilitate portlet interoperation. Additionally, the use of inference rules enables sophisticated ways of piping that are "declaratively" described using Jena rules.

This work also relates to Web service composition and orchestration. In the SELF-SERV architecture [2], the composition of Web Services is encoded using statecharts. With the statechart, the service deployer generates the post-processing and precondition tables, and this information is distributed among the participating services. During the definition of the composite service, the producer decides if the value of the input of a component is obtained from the output of another component or requested from the user.

By contrast, our approach is centralized (i.e. all flow information, the piping rules, are kept in a single place, the portal), and it is always up to the user to accept the values suggested by the piping flow. This is akin to the portal manners where content is centralized, and freely browsed by the user.

Paolucci et al. [14] and Sirin et al. [18] use a semantic approach for Web service location and composition. DAML-based ontologies are used to describe the inputs and outputs of the services. The semantic match between a service's outputs and another service's input are determined by the minimal distance between concepts in a taxonomy tree. This is similar to our piping in which "matching" between portlets is achieving through the help of the ontology.

Agarwal et al. [1] describe the use of deep annotation for Web Service integration. WSDL files are extended with an ontology which is used to describe input and output parameters. The service consumer acts as a querying party by mapping the Web Service ontology with its own. A framework, *OntoMat-Service*, generates the mapping rules between the consumer ontology and the ontologies referred to in the WSDL documents. At enactment time, the data for the Web Services are retrieved automatically from the client's ontology. From this perspective, our work explores the use of a rule-based approach where the flow is based not just on the matching between parameters but in richer flow policies.

8. CONCLUSION

Enhancing the user experience is one of the hallmarks of portals. This implies for the user to perceive the distinct offerings of a portal as an integrated workplace where data flow smoothly among the distinct portlets being framed by the portal. The controlled and cooperative environment that characterizes the portal facilitates the use of deep annotation to portlet interoperation.

This paper describes such an approach by using a piping mechanism. A pipe basically describes how events of a source portlet can be mapped into eventual events of a target portlet. Distinct navigation styles can be supported by triggering piping rules in either a backward or forward way. Another aspect is event consumption. In a backward mode, distinct events (e.g. flight books) can happen before an eventual event (e.g. hotel book) is issued. This raises the question of which flight reservation to consider to feed the hotel booking. Distinct policies can be possible (e.g. FIFO, LIFO) which will presumably depend on the application semantics.

An unsolved issue is whether this approach can be used to specify complex transactions among several portlets. In the current scenario, booking a hotel is completely detached from booking a flight, i.e. failing to book a hotel does not invalidate the flight booking. However, if both tasks were defined as a transaction then, the impossibility of booking a hotel would have resulted in canceling the flight. The notion of transaction implies recoverability. Being portlets independent components, rollback of a transaction that expands among distinct portlets rests on the existence of contingency actions provided by the portlet to undone state changes. Otherwise, there is not much to be done by the portal, since its role is restricted to be a container that keeps track of the events risen during portlet interaction. And it is not always possible to recover a past state from events, unless contingency actions are provided.

So far, portlet interaction is limited to data flow. More complex interactions can be envisaged which involve a portlet influencing the control of another portlet. This is left to further research.

9. ACKNOWLEDGEMENTS

This work is partially funded by the Spanish Science and Technology Ministry (MCYT) under contract TIC2002-01442 with the co-support of the European Social Fund, and the Industry Department of the Basque Government under contract UE02-A16. Our gratitude to Sergio Fernández, Iñaki Paz, Fernando Bellas and Dave Hollingworth for their support throughout.

10. REFERENCES

[1] S. Agarwal, S. Handschuh, and S. Staab. Surfing the Service Web. In *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 211–226. Springer, October 2003.

[2] B. Benatallah, M. Dumas, Q. Z. Sheng, and A. H. H. Ngu. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In *18th International Conference on Data Engineering (ICDE'02)*. IEEE, 2002.

[3] M. Brambilla, S. Ceri, S. Comai, and P. Fraternali. Specification and Design of Workflow-driven Hypertexts. *Journal of Web Engineering*, 1(1), 2002.

[4] W3C Consortium. OWL-S: Semantic Markup for Web Services, 2004. at http://www.w3.org/Submission/OWL-S/.

[5] S. Handschuh and S. Staab (eds.). *Annotation for the Semantic Web*. IOS Press, 2003.

[6] S. Handschuh, S. Staab, and R. Volz. On Deep Annotation. In *WWW2003*. ACM, May 2003.

[7] S. Handschuh, R. Volz, and S. Staab. Annotation for the Deep Web. *IEEE Intelligent Systems*, 18(5):42–48, September/October 2003.

[8] Hewlett-Packard. Jena: a Java framework for writing Semantic Web applications, 2003. at http://www.hpl.hp.com/semweb/jena.htm.

[9] R. Jasper and M. Uschold. A Framework for Understanding and Classifying Ontology Applications. In *IJCAI99 Workshop on Ontologies and Problem Solving Methods KRR5*, August 1999.

[10] Java Community Process. JSR 168 portlet specification, October 2003. at http://www.jcp.org/en/jsr/detail?id=168.

[11] G.P. Marquis. Application of traditional system design techniques to web site design. *Information and Software Technology*, 44(9):507–512, 2002.

[12] OASIS. Web Service for Remote Portlets Specification Version 1.0, 2003. http://www.oasis-open.org/commitees/tc_home.php?wg_abbrev=wsrp.

[13] F. Pan and J. R. Hobbs. Time in OWL-S. In *1st International Semantic Web Services Symposium*, March 2004.

[14] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *1st International Semantic Web Conference*, pages 333–347. Springer-Verlag, June 2002.

[15] C. Peltz. Web Service Orchestration and Choreography. A look at WSCI and BPEL4WS. *WebServices Journal*, 03(7), July 2003.

[16] E. Reshef. Building Interactive Web Services with WSIA & WSRP. *Web Services Journal*, pages 2–6, December 2002.

[17] A. Roy-Chowdhury, S. Ramaswamy, and X. Xu. Using Click-to-Action to Provide User-Controlled Integration of Portlets, December 2002. at http://www7b.software.ibm.com/wsdd/library/teacharticles/0212_roy/roy.html.

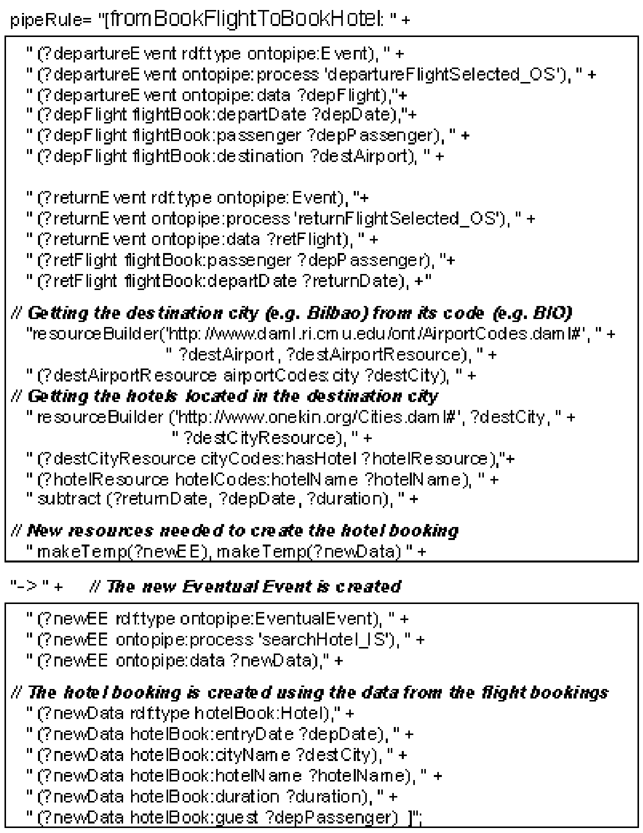
[18] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. In *1st Workshop on Web Services: Modeling, Architecture and Infrastructure. In conjunction with ICEIS 2003*, pages 17–24. ICEIS Press, April 2003.

[19] The Delphi Group. Portal Lifecycle Management: Addressing the Hidden Cost of Portal Ownership, 2001. at http://www.mongoosetech.com/downloads/portal_ownership.pdf.

APPENDIX

Figure 9 shows the *bookFlight—bookHotel* pipe. A *Rule* object is defined which includes a name, a list of premises, a list of conclusions, and an optional direction. The premise includes triples, that check the existence of RDF statements in the Jena repository, built-in user-defined functions (e.g. *subtract*), and a set of predefined functions (e.g. *makeTemp*).

In the sample, a *searchHotel_IS* eventual event is obtained from a pair of *departureFlightSelected_OS* and *returnFlightSelected_OS* events. Specifically, the rule checks the existence of a pair of departure and return events associated to the same passenger, uses a



```
pipeRule= "[fromBookFlightToBookHotel: " +
  " (?departureEvent rdf:type ontopipe:Event), " +
  " (?departureEvent ontopipe:process 'departureFlightSelected_OS'), " +
  " (?departureEvent ontopipe:data ?depFlight),"+
  " (?depFlight flightBook:departDate ?depDate),"+
  " (?depFlight flightBook:passenger ?depPassenger), " +
  " (?depFlight flightBook:destination ?destAirport), " +

  " (?returnEvent rdf:type ontopipe:Event), "+
  " (?returnEvent ontopipe:process 'returnFlightSelected_OS'), " +
  " (?returnEvent ontopipe:data ?retFlight), " +
  " (?retFlight flightBook:passenger ?depPassenger), "+
  " (?retFlight flightBook:departDate ?returnDate), +"

// Getting the destination city (e.g. Bilbao) from its code (e.g. BIO)
  "resourceBuilder('http://www.daml.ri.cmu.edu/ont/AirportCodes.daml#', " +
                        " ?destAirport, ?destAirportResource), " +
  " (?destAirportResource airportCodes:city ?destCity), " +
// Getting the hotels located in the destination city
  " resourceBuilder ('http://www.onekin.org/Cities.daml#', ?destCity, " +
                         " ?destCityResource), " +
  " (?destCityResource cityCodes:hasHotel ?hotelResource),"+
  " (?hotelResource hotelCodes:hotelName ?hotelName), " +
  " subtract (?returnDate, ?depDate, ?duration), " +

// New resources needed to create the hotel booking
  " makeTemp(?newEE), makeTemp(?newData) " +

"-> " +    // The new Eventual Event is created
  " (?newEE rdf:type ontopipe:EventualEvent), " +
  " (?newEE ontopipe:process 'searchHotel_IS'), " +
  " (?newEE ontopipe:data ?newData)," +

// The hotel booking is created using the data from the flight bookings
  " (?newData rdf:type hotelBook:Hotel)," +
  " (?newData hotelBook:entryDate ?depDate), " +
  " (?newData hotelBook:cityName ?destCity), " +
  " (?newData hotelBook:hotelName ?hotelName), " +
  " (?newData hotelBook:duration ?duration), " +
  " (?newData hotelBook:guest ?depPassenger) ]";
```

Figure 9: A pipe from *bookFlight* to *bookHotel*.

user-defined function, *subtract*, to calculate the duration of the stay at the hotel.

The final part of the premise uses the predefined function, *make-Temp*, to indicate the creation of two new instances, *newEE* and *newData*, whose properties are assigned in the conclusion of the rule. The former is an *eventualEvent* of type *searchHotel_IS* whose *data* property corresponds to an instantiation of *hotelBook*. The properties of this instance are in turn obtained from the variables which have been instantiated in the premise of the rule.