Ontology Driven Dynamic Web Interface Generation

Andréia Luna[1], Daniel Schwabe[1]
[1] Pontifícia Universidade Católica do Rio de Janeiro,
Rio de Janeiro, Brazil
{amluna, dschwabe}@ inf.puc-rio.br

Abstract. In this Web 2.0 era, the browsers perform ever-richer graphical interfaces. This paper discusses an approach to web applications interface design that employs the semantic web primitives and it is expressive enough to model most common Rich Internet Application functionalities. It proposes an abstract interface description language and a whole software environment that could make it possible to the application designer to automatically generate an executable interface from an abstract description.

Keywords: Semantic Web, Model-Driven Development, Rich Internet Applications, Web User Interface.

1 Introduction

In this Web 2.0 era, the browsers perform ever-richer graphical interfaces. Today, virtually every type of application can benefit from the ubiquity of Web browsers without compromising the user experience: social networking sites, online tickets reservation systems, shopping or auction sites and collaborative text editors are just some examples of applications built on rich Web clients.

The separation between logic and presentation in the software design is recommended by the widely used MVC (Model-View-Controller) pattern. Design methods for hypermedia applications are aimed at introducing models in different abstraction levels to capture the requirements and the operation of web applications in various perspectives. For example, conceptual modeling, navigation and presentation aspects are usually dealt with separately by Web development methods.

The goal of this work is to propose a model capable of describing interface elements as interaction objects, not only as presentation ones. In the Web 2.0 context, interface objects are provided with behavior and they react to events, making it insufficient to describe only their static attributes.

2 Abstract Interfaces Modeling

The Abstract Data View (ADV) design model was originally created to specify clearly and formally the separation of the user interface from the application component of a software system, and to provide a systematic design method that is independent of specific application environments, in order to lead to a high degree of reuse of designs for both interface and application components. [1] The user's interactions with the components of application, called Abstract Data Objects (ADOs), are modeled through the corresponding Abstract Data Views (ADVs), which are able to express both ADOs' perception properties and the events they can handle.

The ADV formalism is used by the Object-Oriented Hypermedia Design Model (OOHDM), during the Abstract Interface Design activity. [2] As a notation for the Semantic Hypermedia Design Method (SHDM) [1] [3] proposes an Abstract Widgets Ontology for the abstract interfaces design, a Concrete Widgets Ontology, which describes the interface elements available in the specific application environments, such as the elements of an HTML page, and a set of rules for mapping between the two ontologies. Thus, the designer of an application can automatically generate a concrete interface from an abstract interface description.

The Abstract Widgets Ontology provides a formalism to express both the structure and the layout of an interface. However, to express the interface objects' behavior in reaction to external events, the SHDM method does not yet have a notation. To fill this gap, the following describes an extended ontology with enough expressiveness to model the dynamic aspects of the interfaces present in Web 2.0 applications, and a compiler for this language: a tool that reads a high level description of the an interface's operation and generates the executable code in the target platform.

2.1 Extended Abstract Widgets Ontology

The OWL Abstract Widgets Ontology aims to describe how the data objects (ADOs) will be presented, detailing the visible elements (ADVs) that will be available to the user. Its main concepts are:

AbstractInterfaceElement: this class represents all the possible abstract widgets and is a generalization of the following subclasses:

ElementExhibitor: represents elements that exhibit some type of content, such as a text or an image;

SimpleActivator: represents an element capable of reacting to external events, such as a link or an action button;

IndefiniteVariable: represents elements that allow the user to enter data through the keyboard, for example, a text box on a form;

PredefinedVariable: represents elements that allow the selection of a subset from a set of predefined values, such as buttons and check boxes;

CompositeInterfaceElement: represents a composition of abstract elements;

[1] SHDM is an evolution of OOHDM method that employs the Semantic Web formalisms and primitives.

AbstractInterface: represents the final composition of all elements of the abstract interface.

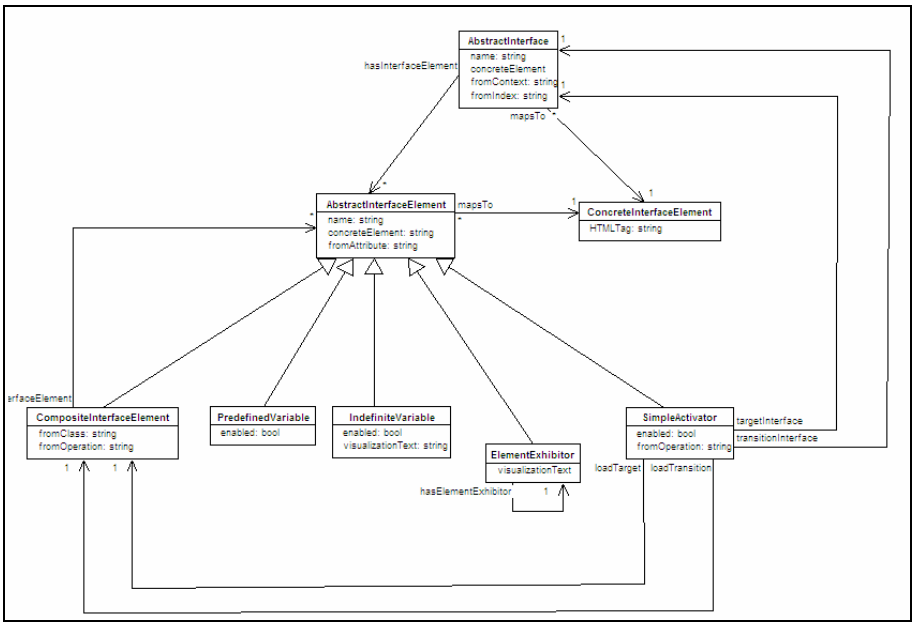

Fig. 1. Widgets Class Diagram

These are some properties defined for these classes:

hasElementExhibitor: It indicates the *ElementExhibitor* element that composes an element of the same type mapped as a *Link;*

hasInterfaceElement: It indicates which *AbstractInterfaceElement* instances compose the element;

loadTarget: It indicates the widget where the interface referenced by the property *targetInterface* should be loaded;

loadTransition: It indicates the widget where the interface referenced by the property *transitionInterface* should be loaded;

targetInterface: It indicates which *AbstractInterface* instance will be displayed when the element is activated;

transitionInterface: It indicates which *AbstractInterface* instance will be displayed during the operation referenced by the property *fromOperation;*

visualizationText: It represents a value that should be displayed by the element.

However, to capture the dynamic aspects of a widget, the ontology should be able to define some concepts present in the formalism of the ADV-Charts (a generalization of StateCharts [4] and ObjectCharts [5]), such as states, transitions and events. Thus, the Abstract Widgets Ontology was extended to include the following classes:

Transition: this class represents the changes of state in the interface;

RhetoricalStructure: represents a set of animations;

Animation: a change on the interface elements, which happens during a transition or in response to an event:

InsertElement: is an entrance animation, for the insertion of an element belonging to the destination state. It indicates the emergence of a new element during the transition;

RemoveElement: is an exit animation, for the removal of an element belonging to the current state. It indicates the absence of the element in the destination state;

MatchElements: is a maintenance animation, which performs transformations to match the parameters of an element in the current state and another in the destination state. It indicates the presence of the element in the two states of a transition;

TradeElements: is a substitution animation, which performs the transformation of an element in the current state into another in the destination state. It indicates a relationship between the elements;

EmphasizeElement: is an animation to highlight an element, altering it as a way to to emphasize it during the transition. It indicates that an action is being performed.

Event: the occurrence of an action on an interface element:

onActivate: indicates that a *SimpleActivator* was activated;

onBlur: indicates that a data entry element lost focus;

onFocus: indicates that a data entry element got focus;

onHover: indicates that a mouse pointer was moved over an element;

onChange: indicates that a data entry element lost focus after its content has been modified;

onSelect: indicates that the text in a data entry element was selected;

Some properties defined for these new classes are:

afterAnimation: It states the sequence in which the animations occur within a rhetorical structure;

afterRhetoricalStructure: It states the order in which the rhetorical structures occur within a transition;

effect: It indicates the effect presented by an animation;

fromInterface: It indicates the initial interface of the transition;

hasAnimations: It indicates which animations compose a rhetorical structure;

hasElement: It indicates which element takes part in the animation;

hasRhetoricalStructures: It indicates which rhetorical structures compose a transition;

hasTargetElement: It indicates which element in the destination state takes part in the animation;

onElement: It indicates the element on which the event occurs;

onEvent: It indicates the event that triggers an animation sequence;

parameters: It indicates the animation effect parameters;

toInterface: It indicates the final interface of the transition.

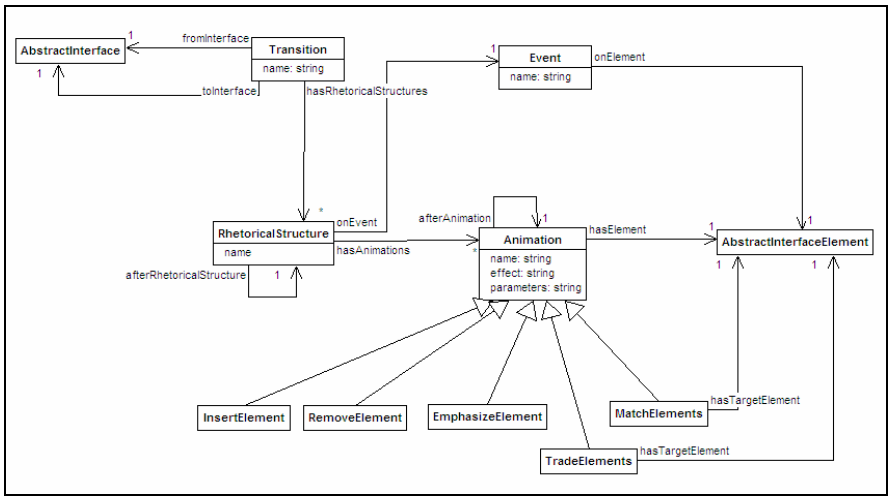

Fig. 2. Animations Class Diagram

2.2 Extended Concrete Widgets Ontology

This ontology describes classes of concrete interface elements, available in different application environments, namely:

ConcreteInterfaceElement: instances of this class represent the elements available in web applications interfaces, for example, the elements of an HTML page. It has the following subclasses:

Label: represents concrete elements that display text;
Image: represents concrete elements that display pictures;
Composition: represents compositions of concrete elements;
Table: represents concrete elements that display tabular data;
TableHeader: represents a table header;
TableBody: represents a table body;
TableFooter: represents a table footer;
TableRow: represents a table row;
TableCell: represents a table cell;
Form: represents compositions of data entry fields;
TextArea: represents a data entry field with n rows and m columns;
TextBox: represents a data entry field with one row and n columns;
CheckBox: represents a data entry field that can hold only two possible values: selected or not selected. When we use groups of elements of this kind, it is possible to select more than one at a time;
ComboBox: represents a data entry field that holds a list f options;
RadioButton: represents a data entry field that can hold only two possible values: selected or not selected. When we use groups of elements of this kind, it is possible to select just one at a time;
Link: represents web navigation links;
Button: represents concrete elements that can trigger an action.

The ontology defines the following properties for these classes:

HTMLtag: It indicates the HTML tag that will be used in the default translation of a specific element;

sortable: It indicates if the table is sortable.

2.3 Mapping Abstract Widgets into Concrete Widgets

The Abstract Widgets Ontology also provides vocabulary that allows you to map *AbstractInterfaceElement* and *AbstractInterface* instances into *ConcreteInterfaceElement* instances. This can be used to automate the generation of concrete interfaces.

In order to ensure that an abstract interface description will be translated into a piece of valid HTML code, some consistency rules were also defined. They constrain which concrete elements can represent an abstract one (*mapsTo* relationship), and which HTML tags can represent a concrete element. (*concreteElement* property).

2.4 Mapping Data Objects into Abstract Widgets

As we have already mentioned, the abstract interface design describes how users will perceive and interact with the application data objects. Therefore, the specification of abstract widgets must be able to express the mapping between corresponding elements in model and view layers.

Based on the SHDM primitives, we propose the extension of the Abstract Widgets Ontology with the following properties:

fromAttribute: It indicates which attribute in the SHDM navigational ontology corresponds to the abstract element;

fromClass: It indicates which class in the SHDM navigational ontology corresponds to the abstract element;

fromContext: It indicates which context in the SHDM navigational ontology the nodes displayed in the interface belongs to;

fromIndex: It indicates which index in the SHDM navigational ontology will be displayed in the interface;

fromOperation: It indicates which operation in the SHDM navigational ontology will be triggered by the abstract element.

2.5 An Abstract Interface Modeling Example

Consider a web application that is meant to help users compare prices from leading travel agencies. After users provide information about destination and departure date, the application queries a set of travel operators about packages prices and displays the results in a list ordered by price.

The concrete interface showing which partners are being queried as well the available results may look like Figure 3.

It displays two lists whose items are travel agencies' logos. An item is moved from one list to another as the query to a partner's web service is answered. Let us see how both interface's *structure* and *behavior* can be modeled using the described ontology. We use an XML syntax to write the abstract interface description.

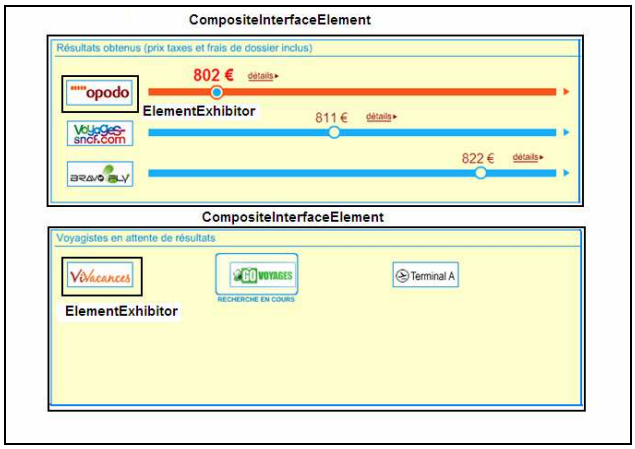

Fig. 3. Sample Concrete Interface[2]

The following listing is the abstract interface description. It defines two lists – *QueryResultsList* and *AvailableAgenciesList* – whose items are compositions of displaying widgets (*CompositeInterfaceElements*) that represent *PackagePrice* and *AvailableAgency* data objects respectively. But we also need to map these abstract elements into concrete ones, which is made by setting the *mapsTo* property.

```
<AbstractInterface name="PriceComparison"
mapsTo="Composition" >

  <CompositeInterfaceElement name="QueryResultsList"
mapsTo="Composition" >

   <CompositeInterfaceElement name="PackagePrice"
mapsTo="Composition" fromClass="TravelPackage" >

    <ElementExhibitor name="PackageAgencyLogo"
mapsTo="Image" fromAttribute="pckgAgencyLogo" />

    <ElementExhibitor name="PackageAgencyPrice"
mapsTo="Label" fromAttribute="pckgAgencyPrice" />

   </CompositeInterfaceElement>

  </CompositeInterfaceElement>

  <CompositeInterfaceElement
name="AvailableAgenciesList" mapsTo="Composition" >

   <CompositeInterfaceElement name="AvailableAgency"
mapsTo="Composition" fromClass="TravelAgency" >
```

[2] http://www.alibabuy.com/

```
    <ElementExhibitor name="AgencyLogo" mapsTo="Image"
fromAttribute="agencyLogo" />
   </CompositeInterfaceElement>
  </CompositeInterfaceElement>
 </AbstractInterface>
```

We will not discuss the data object mapping for this example, but focus on the dynamic aspects of such an interface. We need to specify the animation that must take place when a new item is inserted in the *QueryResultsList* composition. This is a maintenance animation (*MatchElements*) between the *AvailableAgency* (from) and *PackagePrice* (to) widgets, which presents a *Switch* effect to the user, i.e., the first widget is faded out while the last is faded in. This could be accomplished by the following Transition specification:

```
    <Transition name="QueryResponse"
toInterface="PriceComparison">
        <RhetoricalStrucure name="QueryResult">
            <MatchElements name="MovingItemsAnimation"
parameters="duration: 3.0" effect="switch">
               <AbstractInterfaceElement
name="PackagePrice" />
               <AbstractInterfaceElement
name="AvailableAgency" />
            </MatchElements>
        </RhetoricalStrucure>
    </Transition>
```

3 Ontology Driven Web-Based User Interface Generation

Once we have defined a language for abstract interfaces description, it is possible to build a compiler for this language, capable of generating the code for concrete interfaces. An abstract interface is a high level representation of the interactions between the user and application. The concrete interface is a representation of the interface widgets that takes into account the technology used. In the particular case of the *Web-Based User Interface Generator* described in this work, the concrete interfaces are represented in XHTML + RDFa[3]

The compilation of an abstract description generates concrete interfaces; however, there are some references that can only be resolved at runtime. For example, the specification of a *SimpleActivator* may define that the target interface (*targetInterface* property) should be loaded within a widget in the current interface (*loadTarget*

[3] RDFa stands for Resource Description Framework attributes, a set of extensions to the XHTML language, proposed by the W3C consortium, to express structured data.[6]

property). Rendering a concrete interface as an HTML page and resolving such references are functions of the *Concrete Interface Renderer*.

The concrete interfaces representation language is XHTML/RDFa. RDFa attributes are used to express the mapping between interface widgets and application components: *typeof* RDFa attribute translates *fromClass CompositeInterfaceElement* property, while *property* RDFa attribute translates *fromAttribute AbstractInterfaceElement* property.

Since most concrete widgets may not be supposed to display a constant value, but only hold the reference to an attribute in the application data model, the concrete interfaces need a runtime environment capable of resolving these references dynamically. The system component responsible for providing the execution environment for the concrete interfaces is called *Web User Interface Runtime* (WUI-Runtime).

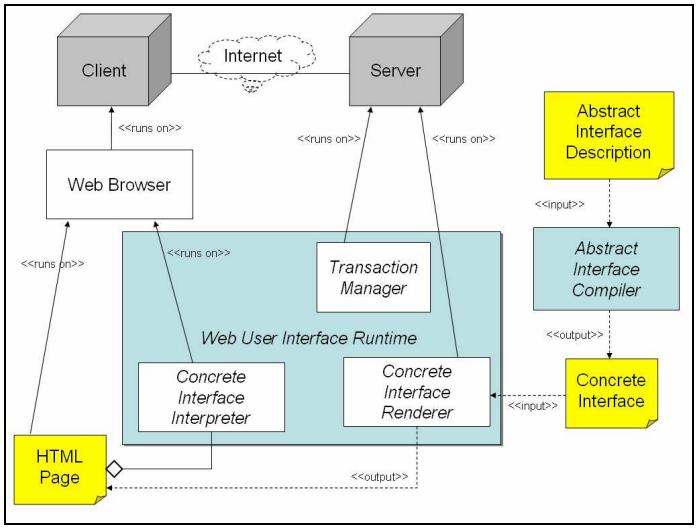

Fig. 4. System Architecture

The *Concrete Interface Interpreter* runs on the WUI-Runtime client side and it is responsible for interpreting the semantic annotations in the HTML pages rendered by the system. These annotations are provided by the RDFa code and they indicate, for instance, which data object should be displayed in the interface or which operation of the model should be triggered when an event occurs at the interface. Thus, after the HTML page is loaded on the client side, a set of Javascript functions should request the actual data and present them using the HTML annotated code as display template for the objects. This Javascript library is also responsible for performing animations in response to interface changes and events.

The concrete interfaces runtime machine also introduces a message queue-based protocol for asynchronous communication between the model and view layers. Using this architecture, the client is able to consume the partial results of a request processing. This is achieved by the implementation of a Transaction concept, which replaces the traditional request/response cycle between client and server.

In a conventional web application, when the client makes a request, we feel the effect of a synchronous call, i.e., the user has to wait for the page to be refreshed before being able to interact with the application again. On the other hand, Ajax technology allows asynchronous communication between client and server, which prevents the user from having to hold on at each request. However, using transactions, it is possible to achieve an even higher degree of asynchronicity.

A transaction occurs inside the context of an HTTP session, and is represented by a unique identifier, with a FIFO data structure attached. Responses generated by the execution of a query or operation in the model are queued there. The runtime environment needs, therefore, a server side component for the management of transactions: the generation of identifiers and the insertion/removal of elements in the response queue.

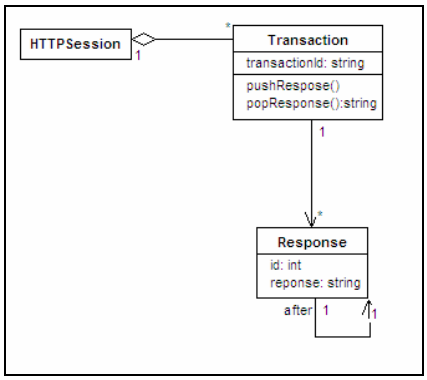

Fig. 5. Transaction Conceptual Scheme

Sending a request characterizes the beginning of a new transaction. However, the method executed by the application does not send a response directly to the client. In fact, it will store the results of its execution in the transaction response queue. In addition, after sending the request, the client starts a loop to consume the results as soon as they become available in the response queue. This makes it possible to track the execution of an action by its partial results – without having to wait until the whole processing has been completed before receiving any return. The format for the data interchange follows the JavaScript Object Notation (JSON).

3.1 A Concrete Interface Generation and Execution Example

Let us return to our previous airfare comparison web site example. We have already discussed how we can model the *structure* and the *behavior* of an interface that displays the results of a query for travel packages prices. Now it's time to consider the *data object mapping* for the interface widgets. Since the lists' items in Fig. 3 represent travel agencies and travel packages (*AvailableAgency* and *PackagePrice*), they have to be mapped into the TravelAgency and TravelPackage classes respectively (*fromClass* property). The *ElementExhibitor* widgets are mapped into the corresponding classes attributes (*fromAttribute* property).

Compiling the abstract interface description will generate a piece of XHTML+RDFa code that is just a display template for the agencies and packages:

```
    <div id='QueryResultsList'>
        <div id='PackagePrice' style='display: none'
typeof='TravelPackage'>
            <img id='PackageAgencyLogo'
property='pckgAgencyLogo' src='#'
alt='pckgAgencyLogo'/>
            <p id='PackageAgencyPrice'
property='pckgAgencyPrice'> pckgAgencyPrice </p>
        </div>
    </div>
    <div id='AvailableAgenciesList'>
        <div id='AvailableAgency' style='display: none'
typeof='TravelAgency'>
            <img id='AgencyLogo' property='agencyLogo'
src='#' alt='agencyLogo'/>
        </div>
    </div>
```

The sequence diagram in Figure 6 shows the interaction between the WUI-Runtime and the web application. The user submits a form and, in response, the *Concrete Interfaces Interpreter* initiates a transaction for the "List Packages Prices" request. Then, the application provides the available travel agencies list and starts to query each of them ("Query Package Price" loop). As the queries are answered, the results are queued, instead of sent to the client. Within the "Pop Response" loop, the client-side WUI-Runtime component will use the semantic annotations in the concrete interface code (*typeof* and *property* RDFa attributes) as a mapping between the actual data objects and the interface widgets. Due to the asynchronous communication and since our sample interface is meant to present both the *progress* and the *results* of an action performed by the application, the Javascript library will create one HTML image and one HTML paragraph element for every travel agency that answers the query as soon as the response is available. Otherwise, the user would not see any result until all the agencies had answered the queries.

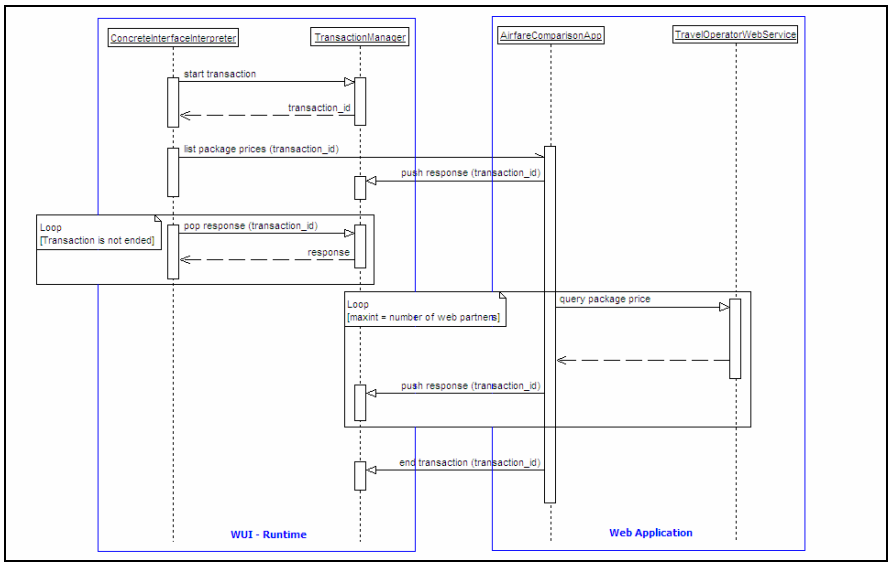

Fig. 6. Transaction Sequence Diagram

4 Conclusion

The Abstract Widgets Ontology proposed by [3] provides a formalism to express both the structure and the layout of an interface, and which data object properties should be presented. This work contribution is to extend that ontology with a vocabulary capable of modeling the interface *behaviour*, not only its *structure*. It also introduces the *transaction* concept which relies on asynchronous communication, but can also enhance it.

References

1. Cowan, D.; Lucena, C.: Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. In: IEEE Transactions on Software Engineering Volume 21, Issue 3 (1995)
2. Schwabe, D.; Rossi, G.: The Object-Oriented Hypermedia Design Model (OOHDM), http://www.telemidia.pucrio.br/oohdm/oohdm.html
3. Moura, S.: Desenvolvimento de Interfaces para Aplicações Hipermídia na Web Semântica., http://www.tecweb.inf.puc-rio.br/autoria/space/Arquivos/Interface+Abstrata+%28Sabrina%29.ppt
4. Antunes, D. C.: Statecharts. Bate Byte Magazine, Issue 36 (1994)
5. Coleman, D.; Hayes, F.; Bear, S.: Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. In: IEEE Transactions on Software Engineering Volume 18, Issue 1 (1992)
6. Adida, B.; Birbeck M.: RDFa Primer. Bridging the Human and Data Webs, http://www.w3.org/TR/2008/NOTE-xhtml-rdfa-primer-20081014