MOVING WEB SERVICES DEPENDENCIES AT THE FRONT-END*

Juan J. Rodríguez, Oscar Díaz, Felipe Ibáñez
Dpto. de Lenguajes y Sistemas Informaticos
University of the Basque Country
Apdo. 649 - 20080 San Sebastian (Spain)
jibrojij@si.ehu.es, jipdigao@si.ehu.es, jipibanf@si.ehu.es

Abstract Most Web sites offer loosely-coupled services where the role of the site is almost restricted to be a front-end for service enactment. If the number of services is large, the designer should struggle to provide a sense of coherence and unity among the services that can be accomplished through the site. This endeavour is similar to business process automation, which are traditionally achieved through the use of workflow management systems. The difference stems from workflows working at the back-end (i.e. it normally implies database-level or API-based integration) whereas now the integration is achieved at the front-end. So far, these aspects are directly hard-coded and scattered around *HTML* pages. This hinders not only development, but most importantly, the maintenance and evolution of the site. This work investigates how traditional workflow concepts can be redefined in terms of front-end notions.

Keywords: **Web services, workflows, XML**

1. Introduction

Web services evolve the object-oriented vision of assembling software from component building blocks to the assembly of services that may or may not be built on object technology. While todays services are commonly delivered individually, it is most likely that value-added services will be provided by combining existing services, that might be offered by different companies [7] [8]. Indeed, distinct initiatives have been launched to provide XML languages for Web service composition (e.g. WSFL [4] or XLANG [9]) which facilitate the building of complex business processes. A process is realised as a flow

*This research was supported by the Secretaría de Estado de Política Científica y Tecnológica of the Spanish Government under contract TIC 1999-1048-C02-02. Founding was also received by the Departamento de Educación, Universidades e Investigación of the Basque Government under contract UE2000-32.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35614-3_21

of Web service operation calls, where each call involves taking data from the process state, creating an XML document based on the service's API, and invoking the API as a step in the flow (e.g. using SOAP). Similar to traditional workflow management systems, Web service integration occurs at the back end, underneath the presentation layer.

There is, however, another form of integration, one based at the front end. As an example, consider a car-retailer portal through which cars can be purchased and credits can be requested. These services could be supported by distinct providers. The portal designer needs to enforce a precedence rule so that a credit can not be requested till a car is bought during the same session. This control-flow dependency can be enforced in a "front-end" way by disabling the *"creditRequest"* button until the *"purchaseOrder"* button has been clicked on. The integration is not achieved at the backend (e.g. through some pre-conditions attached to the *creditRequest* service or some workflow engine enforcing the flow dependency) but via presentation-based mechanisms.

Unfortunately, when service dependencies are moved to the presentation layer, they end up being hard-coded, "melted" with the rest of the HTML page code. For instance, some of these dependencies can be enforced through page navigation. For example, the flow dependency *"a purchaseOrder can only be issued once during a session"* is commonly enforced by providing a separate purchase-order page that, once the service has been enacted, is no longer available. If more than one service can be issued from the very same page, flow dependencies are usually hard-coded in some obscure scripting code. This situation deteriorates if parameter passing between services is required. If this is the case, the programmer has to resort to cookies or other mechanism to overcome the stateless nature of the *http* protocol. This ends up in the flow dependencies being hard-coded, implicit and scattered around distinct pages. This hinders not only development, but most importantly, the maintenance and evolution of web applications.

1.1. Our contribution

Previous concerns are similar to those found in traditional workflow technology. The different stems from workflows working at the back-end whereas now the integration is pursued at the front-end.

Generally, traditional workflow systems provide a process engine that manages the processes and guides each one of them through the set of activities as specified by the process definition. The engine needs to perform the appropriate actions and evaluate the business rules to determine how the flow proceeds. A token (i.e. the common data structure throughout the process life cycle) is used as the context through which data is passed from one activity to the next. The challenge is how to re-interpret these notions in a front-end setting.

To this end, this work proposes the following mappings: activities are realised as Interactive Web Services (IWS)[5], activity dependencies are mapped to enabling conditions, and tokens to history logs. The result is termed "workview" to emphasize on one hand, the sense of coherence and unity among the distinct services offered by the Web site, and on the other hand, the fact that the integration is achieved at the front-end.

A **"Workview XML Language"** (*WorkviewXL*) is presented for the specification of workviews. *WorkviewXL* is to *IWSs* what *WSWL* or *XLANG* is to traditional Web services. Using *WorkviewXL*, the designer defines in a separate workview document all dependencies that harness the set of services which can be invoked from a site, regardless of the page from where the service can be enacted or its results rendered.

The rest of the paper is structured as follows. Section 2 presents the running example. The constructs that provide the foundations for front-end integration, namely, Interactive Web Services, enabling conditions and history logs, are presented in sections 3, 4 and 5, respectively. *WorkviewXL* is introduced in section 6. Section 7 presents a set of JSP tags to bind workviews to HTML pages. Finally, conclusions are given. A car-retailer site is used as an example throughout the paper. This example can be found at *http://sipl68.si.ehu.es/wswl/examples/car-retailer/*. Besides the application itself, this site contains the complete specification of the services, the workview definition and an example of a hosting HTML page.

2. A running example

A car-retailer portal is used as a running example. The services offered, include, *"purchaseRequest"*, *"creditRequest"* and *"login"*, that initiates the obvious activities. These activities are bound together through a set of dependencies, namely:

- a customer can only apply once for a credit, this means, the *"creditRequest"* service is only enabled if the user has never applied for a credit in this or any previous session.
- the *"login"* service can only be successfully issued once during a session.
- the *"purchaseRequest"* service is only enabled if *"login"* has been successfully invoked within the session.
- the *"creditRequest"* service is only enabled if the user has made a purchase over $100, anytime.

Figure 1 displays the distinct control dependencies involved in the *car-retailer* workview. Nodes stands for services. An arc is drawn from node *S1* to node *S2*

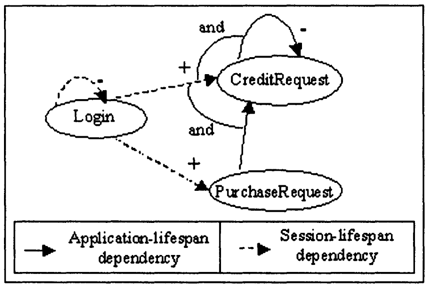


Figure 1. A dependency graph for the *car-retailer* workview.

to show that *S1* constrains somehow *S2*. A "+" indicates an existence condition whereas a "-" checks that the service has not yet been enacted.

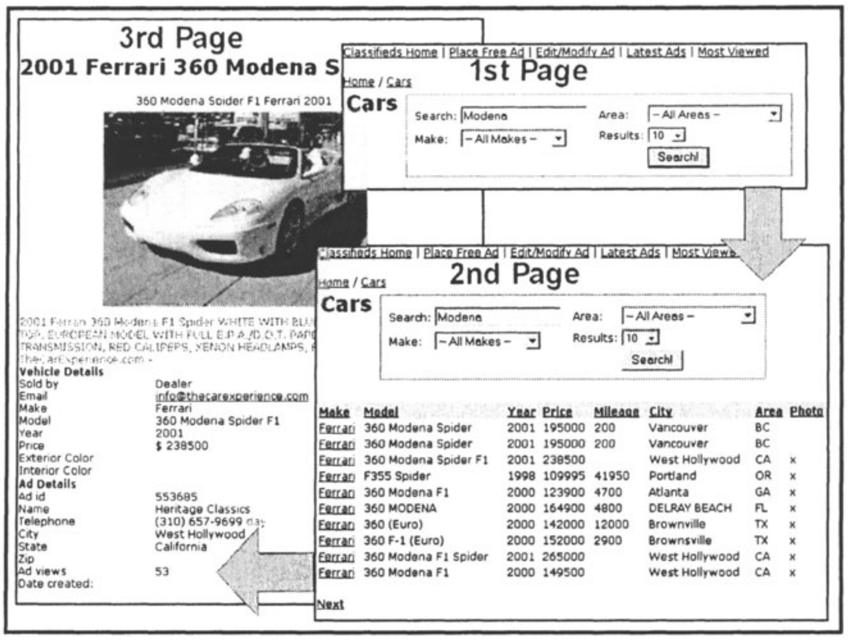
This dependency graph can be supported through a workview system. Traditional workflow systems provide a process engine that manages the processes and guides each one of them through the set of activities as specified by the process definition. The engine needs to perform the appropriate actions and evaluate the business rules to determine how the flow proceeds. A token (i.e. the common data structure throughout the process life cycle) is used as the context through which data is passed from one activity to the next. The challenge is how to re-interpret these notions in a front-end setting.

To this end, this work proposes the following mappings: activities are realised as Interactive Web Services, activity dependencies are mapped to enabling conditions, and tokens to history logs. Following sections look at the details.

3. Interactive Web Services as workview's activities

Traditional Web service standards facilitate the sharing of the business logic, and suggest that Web service consumers should write a new presentation logic on top of the business logic. This results in the "presentation logic" being hard coded time and again in each application requesting this Web service. Not only is development time increased, but also this re-coding of the presentation can also jeopardizes the company's image. The presentation logic can realize branding and customer experience strategies obtained after heavy investment and lengthy experimentations. Consequently, the company might be interested in maintaining this experience even if its services are offered through third-party portals. After all, usability practitioners have long realized that this presentation logic represents a main asset for the organization.

Interactive Web Services (IWSs) address this problem. They strive to provide coarse-grained components that include both the business logic and the presentation logic. An IWS is a Web service (and thus, the standard WSDL



2001 Ferrari 360 Modena S

360 Modena Spider F1 Ferrari 2001

2001 Ferrari 360 Modena F1 Spider WHITE WITH BLU
TOP, EUROPEAN MODEL WITH FULL E.P.A./D.O.T. PAP
TRANSMISSION, RED CALIPERS, XENON HEADLAMPS,
TheCarExperience.com -

Vehicle Details	
Sold by	Dealer
Email	info@thecarexperience.com
Make	Ferrari
Model	360 Modena Spider F1
Year	2001
Price	$ 238500
Exterior Color	
Interior Color	
Ad Details	
Ad id	553685
Name	Heritage Classics
Telephone	(310) 657-9699 day
City	West Hollywood
State	California
Zip	
Ad views	53
Date created:	

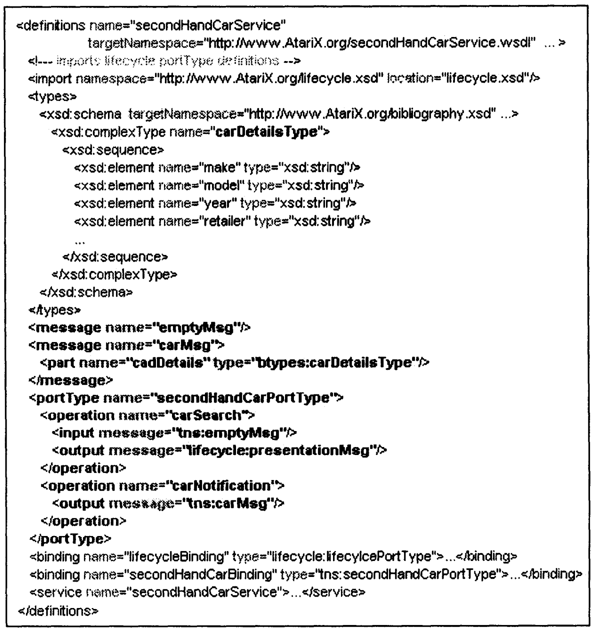
Make	Model	Year	Price	Mileage	City	Area	Photo
Ferrari	360 Modena Spider	2001	195000	200	Vancouver	BC	
Ferrari	360 Modena Spider	2001	195000	200	Vancouver	BC	
Ferrari	360 Modena Spider F1	2001	238500		West Hollywood	CA	x
Ferrari	F355 Spider	1998	109995	41950	Portland	OR	x
Ferrari	360 Modena F1	2000	123900	4700	Atlanta	GA	x
Ferrari	360 MODENA	2000	164900	4800	DELRAY BEACH	FL	x
Ferrari	360 (Euro)	2000	142000	12000	Brownville	TX	x
Ferrari	360 F-1 (Euro)	2000	152000	2900	Brownsville	TX	x
Ferrari	360 Modena F1 Spider	2001	265000		West Hollywood	CA	x
Ferrari	360 Modena F1	2000	149500		West Hollywood	CA	x

Figure 2. *carSearch* operation from *secondHandCarService*

[11] is used for its specification) where operations might not only return XML documents but also fragments of HTML (or another presentation language). An IWS operation is a cohesive, purposeful user interaction unit. In a Web setting, user interaction takes place via HTML pages. Hence, a main part of the realization of an IWS operation is a set of HTML fragments (or other rendering technology, e.g. WML). An IWS operation can be as simple as providing a form to collect some parameters, executing some business logic, and render back the results. However, the more intricate the interaction, the more make sense the notion of IWS.

For example, when invoking *carSearch* operation, it starts by prompting the user to provide the desired car model (see figure 2, first page); if more that one car matches the query criteria, the operation returns a list with these cars (second page); once a car has been univocally identified (either right at the beginning or after some further interactions), the IWS renders the corresponding car details and retailer's info (page 3). This set of pages supports a common goal, retrieving the available second hand cars of the desired car model, which is realized as an interactive operation .

The OMG standard body is currently revising distinct proposals for IWS modeling under the "OASIS Web Service for Interactive Applications" initiative [5]. Distinct models have been proposed: the IWS model [12], the WSUI model



```
<definitions name="secondHandCarService"
            targetNamespace="http://www.AtariX.org/secondHandCarService.wsdl" ... >
  <!--- imports lifecycle portType definitions -->
  <import namespace="http://www.AtariX.org/lifecycle.xsd" location="lifecycle.xsd"/>
  <types>
    <xsd:schema  targetNamespace="http://www.AtariX.org/bibliography.xsd" ...>
      <xsd:complexType name="carDetailsType">
        <xsd:sequence>
          <xsd:element name="make" type="xsd:string"/>
          <xsd:element name="model" type="xsd:string"/>
          <xsd:element name="year" type="xsd:string"/>
          <xsd:element name="retailer" type="xsd:string"/>
          ...
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </types>
  <message name="emptyMsg"/>
  <message name="carMsg">
    <part name="cadDetails" type="btypes:carDetailsType"/>
  </message>
  <portType name="secondHandCarPortType">
    <operation name="carSearch">
      <input message="tns:emptyMsg"/>
      <output message="lifecycle:presentationMsg"/>
    </operation>
    <operation name="carNotification">
      <output message="tns:carMsg"/>
    </operation>
  </portType>
  <binding name="lifecycleBinding" type="lifecycle:lifecylcePortType">...</binding>
  <binding name="secondHandCarBinding" type="tns:secondHandCarPortType">...</binding>
  <service name="secondHandCarService">...</service>
</definitions>
```

Figure 3. IM-IWS WSDL definition

[1] or the WSXL model [3]). This paper is based on the Integration-Minded IWS (IM-IWS) model[2].

3.1. IM-IWS definition

Like any other Web service, an IWS is described by means of the Web Service Description Language (WSDL). Figure 3 gives an example of an IWS definition for the *secondHandCarService* Web Service using the IM-IWS model. The difference with other IWS models rests on the so-called "two-face" interfaces, and the notifications. The former indicates that Web service operations can be invoked either programmatically or through a form. For example, the *carSearch* operation has the following signature:

<operation name="carSearch">

<input message="tns:emptyMsg"/>
<output message="lifecycle:presentationMsg"/>

</operation>

When the form-based interface is used, no input parameters are provided at invocation time (indicated by *<input message="tns:emptyMsg"/>*) but the operation returns a form to prompt the user to introduce these parameters. It is user-oriented. In our example, *carSearch* returns a form similar to the one

```
<history>
 <login timestamp="1">
  <userName>Smith</userName>
  <password>*****</password>
  <alternative>correct</alternative>
  <userRole>Client</userRole>
 </login>
 <purchaseRequest timestamp="2">
  <buyer>Smith</buyer>
  <carInfo>
   <carPrice>14,400</carPrice>
   ...
  </carInfo>
 </purchaseRequest>
 <purchaseRequest timestamp="3">
  <buyer>Smith</buyer>
  <carInfo>
   <carPrice>18,700</carPrice>
   ...
  </carInfo>
 </purchaseRequest>
 <creditRequest timestamp="4">
  <userName>Smith</userName>
  <creditProvider>AK45W</creditProvider>
  <creditAmount>14,400</creditAmount>
 </creditRequest>
</history>
```

Figure 4. A history state.

shown in figure 2 (first page) to collect the author. This form realizes the first interaction. After this, other interactions can follow till the operation is finally fulfilled.

On the other hand, a programmatic interface mimics traditional function calling. In this case, the input parameter is a *carType* element which is used in an analogous way to traditional WSDL operations. Now the invoker is not the final user, but the container. At invocation time, the parameters should already be known, and the IWS no longer prompts the user but renders the next presentation fragment (see figure 2, second page). Each browsing operation can offer both types of interface, and it is up to the container to decide which one to use. For the purpose of this paper, we only consider form-based interfaces.

As for notifications, they signal meaningful states reached during the execution of an operation. These states are communicated through an XML document. An example of a notification operation follows,

```
<operation name="carNotification">
    <output message="tns:carMsg"/>
</operation>
```

Through this operation, the IWS indicates that a meaningful state has been reached in which the required car details are shown. Notice that notifications are the only way for the container to receive information about the landmarks reached during the execution of an IWS operation. The user is aware of such processes as his interaction is required to proceed with the IWS operation, but this process is transparent to the container.

For the purpose of this paper, the container is the workflow engine. The workflow engine regulates when an IWS can be activated. Once activated, the first step on the IWS activity is normally parameter gathering. To this end, a form is displayed which contains a button for IWS enactment. The reader should kept clear the distinction between activation (i.e. the workflow engine permits the IWS to be initiated) and enactment (the user triggers the IWS execution). All the interactions between the IWS and the user are transparent to the workflow engine. They are encapsulated within the IWS. The workflow engine becomes aware of the successful completion of an IWS enactment via a notification. The next section presents how these notifications are the backbone of the workflow token.

4. The history as the workview's token

In the workflow terminology, a token is the common data structure throughout the process life cycle. More specifically, the token is used as the context through which data structures are passed from one activity to the next. We use the notion of history for this purpose. The history records the ordered flow of service enactments (i.e. service occurrences). Each service occurrence causes an entry into the history log. As an example, consider a session where the user first logs in, then, invokes two *"purchaseRequest"* services, and finally, issues a *"creditRequest"*. The resulting state of the history is shown in figure 4. The log is supported by an XML document which has <*history*> as its root element and, as sub-elements, the name of the enacted services and its actual parameters. Additionally, an entry can have a session or an application life-span. The former is removed as soon as the session ends. By contrast, application entries are persistent and thus, are shared among distinct sessions of the same application. Whether an occurrence is temporal or persistent depends on the semantics of the related dependencies (see below).

5. Enabling conditions as the workview's dependencies

A common situation is for a service to require some *necessary condition* to be available, and once available, it is let to the user the freedom to invoke it (e.g. once an item has been added to the cart, the *"sending the order"* service becomes available). In our opinion, this approach naturally fits the event-driven nature that characterize GUI applications. Besides it accounts for flexibility and

Table 1. Control-flow dependencies for the *car-retailer* site (the bold ones have been implemented in this example)

Control dependency	**The dependency as a predicate on the history**	**Lifespan**
creditRequest is enabled ... only once during a session	not(history/creditRequest)	session
creditRequest is enabled... provided the same user has bought a car during this session	history/purchaseRequest [buyer=history/login/user]	session
creditRequest is enabled... no more than 100 times during the application lifetime	count(history/creditRequest) < 100	application
creditRequest is enabled... only once a day	not(history/creditRequest [atarix:day(@timeStamp)= atarix:day(history(login/@timeStamp)]	application session
***creditRequest* is enabled... only twice by the same user**	**count(history/creditRequest [userName = history/login/user])< 2**	application session
***login* can only be successfully issued once**	**not(history/login)**	session
***purchaseRequest* is enabled.. if the user has logged in**	**history/login**	session
***creditRequest* is enabled.. . if the user has made a purchase over $100 . during the application life**	**history/purchaseRequest [number(carPrice)>100]**	application

maintainability at the price of complex debugging[1]. We term such dependencies as enabling conditions.

Enabling conditions are stated as predicates on the history log. As this log is an XML file, these predicates are expressed as XPath 1.0 expressions[10]. Some possible dependencies on the *"creditRequest"* service are listed in table 1.

Besides, the predicate as such, three futher aspects should be considered: the binding policy, the selection policy and the consumption policy. Next subsections look at the issues, while section 6 focus on the specification.

5.1. The binding policy

Service parameters can be obtained by querying the user directly. Alternatively, these parameters can be retrieved from previous service occurrences kept in the history. For example, the *"creditRequest"* service has three parameters: *creditProvider*, *creditAmount,* and *userName.* While the former should be provided directly by the user, *creditAmount* and *userName* can be obtained from previous occurrences of the *login* and *purchaseRequest* services, respectively. These requirements state a data-flow dependency.

[1]Whereas a procedural description provides a clear picture of the event flow at compile time, this is not the case for enabled-based descriptions where a global scheduler is responsible for keeping each service informed about its availability. A similar approach has also been proposed for deductive information system modeling [6].

The binding policy indicates whether the mapping is *"compulsory"* or *"optional"*. In our example, the *userName* is compulsory obtained from the *login*; no option is given to the user to change it. Alternatively, the *creditAmount* value is also obtained from the history but it is not fixed, which means, that it is presented to the user as an option, and the user is free to change it.

5.2. The selection policy

As a motivating example, consider that *creditRequest* takes its *creditAmount* parameter from the *purchaseRequest* task. First, *purchaseRequest* is enacted more than once which is reflected in the history as a sequence of occurrences*: { purchaseRequest(...100...), purchaseRequest (...200...), purchaseRequest(...300...)}.* If now the *creditRequest* is issued, which of the available *purchaseRequest* occurrences should be used to obtain the *creditAmount* parameter? The selection policy resolves this ambiguity by explicitly indicating which occurrences of the history log should be selected, if more than one is available.

Two selection policies are introduced to avoid any ambiguities: *"mostRecent"* which selects the lastest occurrence, and *"oldest"* which selects the occurrence which appears first in the history. In our example, defining a *"mostRecent"* selection policy will make the system to request a credit of $300 while the *"oldest"* alternative would go for $100. These keywords are shortcuts for easy access to the history (i.e. they are expanded to XPath predicates based on the occurrences' time-stamps). If the designer needs to enforce more complex policies, he can always result to stating directly the XPath predicate himself. For instance, if the *creditAmount* is not obtained from individual purchases, but from the sum of the prices of distinct items bought so far, the designer can express such policy as follows: *sum(history/purchaseRequest/carPrice).*

5.3. The consumption policy

Consider again that *purchaseRequest* has been enacted more than once: *{purchaseRequest (..100..), purchaseRequest (..200..), purchaseRequest (..300..)}.* If the *creditRequest* service is issued using the *"mostRecent"* selection policy, a credit of $300 will be requested. If the user clicks on *creditRequest* again (if this is possible), should the system select the very same *creditAmount*? The specification should indicate whether the service occurrence is "consumed" (i.e. removed from the history log) when invoking the *creditRequest* or not. For instance, *purchaseRequest* takes the value of its parameters *userName* and *creditAmount* from previous occurrences of *login* and *purchaseRequest*, respectively. A common situation could be to keep the *login* occurrence in the history: *userName* is obtained repeatedly from the very same *login* occurrence, regardless of how many times *creditRequest* has been enacted. By contrast, we might

Table 2. Data-flow dependencies for the *car-retailer* site.

	dependecy 1	dependecy 2	dependecy 3
Mapping parameter	*purchaseRequest. buyer*	*creditRequest. creditAmount*	*creditRequest. userName*
Mapped parameter	*login.userName*	*purchaseRequest. carPrice*	*login.userName*
lifespan	session	application	session
Selection policy	mostRecent	mostRecent	mostRecent
Consumption policy	none	mostRecent	none
Binding policy	compulsory	compulsory	compulsory

be interested in "consuming" the *purchaseRequest* occurrences once they have been used by *creditRequest*; in this case, each *creditRequest* requires a different *purchaseRequest* occurrence in order to be enacted.

Table 2 sums up the additional requirements which have been ascertained after facing these concerns. We argue that such issues should be surfaced during analysis/design time rather than leaving them to the implementator.

6. Workview specification: the Workview XML Language

This section proposes an XML language for workview definition. Figure 5 gives an example for the *car-retailer* site. The identifier and description of the workview are provided first. Next followed by the services that participate in the workview. For each service, its selector and the WSDL file holding its definition are provided. The list of dependencies ends the definition of the workview.

The element *<dependency type= "aType" service= "aService">* indicates the type of the constraint (i.e. *"controlFlow"* or *"dataFlow",*) and the service being constrained. Moreover control flow dependencies include a *<controlFlow>* sub-element which holds an XPath expression[2] on the history log. This expression has to be satisfied (i.e. some elements must be retrieved) for the associated service to be available. It is the enabling condition. On the other hand, data flow dependencies add a *<dataFlow>* sub-element which contains one or more mapping requirements. An example follows:

```
<dependency service= "creditRequest" type= "dataFlow">

    <dataFlow>

        <mapping parameter= "userName"
                select= "login/userName/text()"
                selectionPolicy= "mostRecent"
                bindingPolicy= "compulsory" />
        <mapping parameter= "creditAmount"
```

[2]For simplicity, XPath expressions do not need to include the root of the history document. The system automatically completes it.

```
<workview id="car-retailer">
 <title>car-retailer Workview</title>
 <description>car-retailer Workview</description>
 <services>
  <service id="login" definition="loginService.wsdl"/>
  <service id="purchaseRequest" definition="purchaseService.wsdl"/>
  <service id="creditRequest" definition="creditRequestService.wsdl"/>
 </services>
 <dependencies>
  <dependency service="login" type="controlFlow">
   <controlFlow enabledWhen="not(login[alternative='correct'])"/>
   <historyManagement>
    <on service="login">
     <register lifeSpan="session"/>
    </on>
   </historyManagement>
  </dependency>
  <dependency service="purchaseRequest" type="controlFlow">
   <controlFlow enabledWhen="login[alternative='correct']"/>
   <historyManagement>
    <on service="login">
     <register lifeSpan="session"/>
    </on>
   </historyManagement>
  </dependency>
  <dependency service="purchaseRequest" type="dataFlow">
   <dataFlow>
    <mapping parameter="buyer" select="login/userName/text()"
             selectionPolicy="mostRecent" bindingPolicy="compulsory"/>
   </dataFlow>
   <historyManagement>
    <on service="login">
     <register lifeSpan="session"/>
    </on>
   </historyManagement>
  </dependency>
  <dependency service="creditRequest" type="controlFlow">
   <controlFlow enabledWhen="login[alternative='correct'] and
    not(creditRequest[userName=most-recent(login[alternative='correct']/userName/text())])"/>
   <historyManagement>
    <on service="login">
     <register lifeSpan="session"/>
    </on>
    <on service="creditRequest">
     <register lifeSpan="application"/>
    </on>
   </historyManagement>
  </dependency>
  <dependency service="creditRequest" type="dataFlow">
   <dataFlow>
    <mapping parameter="userName" select="login/userName/text()"
                selectionPolicy="mostRecent" bindingPolicy="compulsory"/>
    <mapping parameter="creditAmount" select="purchaseRequest/carInfo/carPrice/text()"
                selectionPolicy="mostRecent" bindingPolicy="optional"/>
   </dataFlow>
   <historyManagement>
    <on service="login">
     <register lifeSpan="session"/>
    </on>
    <on service="purchaseRequest">
     <register lifeSpan="application"/>
     <remove service="purchaseRequest" consumptionPolicy="mostRecent"/>
    </on>
   </historyManagement>
  </dependency>
 </dependencies>
</workview>
```

Figure 5. The *car-retailer* workview definition.

```
                        select= "purchaseRequest/carInfo/carPrice/text()"

                            selectionPolicy= "mostRecent"
                            bindingPolicy= "optional" />
        </dataFlow>
           <historyManagement>...</historyManagement>
    </dependency>
```

According to this specification, *"creditRequest"* must obtain its *userName* parameter from the *"login"* service, while the *creditAmount* is initially obtained from the *carPrice* of the *"purchaseRequest"* service, though it can be changed by the user (i.e. *bindingPolicy* holds 'optional'). The other parameters of *"creditRequest"* are obtained by directly prompting the user. Furthermore, the selection policy has been set to *"mostRecent"* which indicates that the last purchase enactment is taken, should the user accomplish more than one purchase.

Finally, the *<historyManagement>* element indicates how the history is dealt with. As this management can reflect important semantic aspects of the dependency, a separate history log is kept for each dependency. To realize the consumption policy (see section 5.3), each dependency indicates what service occurrences should be recorded (the *<register>* tag) and, when and how these occurrence should be removed from the history log (the *<remove>* tag). Occurrences can last till the session ends (a *"session"* lifespan) or outlive the session (an *"application"* lifeSpan). Continuing with our example, its history management specification is:

```
        <historyManagement>
            <on service= "login">
                <register lifeSpan= "session"/>
          </on>
            <on service= "purchaseRequest">
                <register lifeSpan= "application"/>
                <remove service= "purchaseRequest" consumption-
              Policy= "mostRecent"/>
          </on>
    </historyManagement>
```

As *"creditRequest"* collects parts of its parameters from previous enactments of *"login"* and *"purchaseRequest"*, the occurrences of these services are recorded in the history log. The login is recorded only during the current session *(lifeSpan= "session")* while the *"purchaseRequest"* occurrence persists beyond the current session. Besides, only the last *"purchaseRequest"* occurrence is kept in the history log as the consumption policy is set to *"mostRecent"*. We are currently investigating to which extent the management of the history can be derived from the dependency specification.

Since a separate history log is kept for each dependency, a question arises about space consumption. If the life-span is *"session"*, this does not pose any storage problem, as the number of service enacted within a session is not too large, and the log lifespan is that of the session. However, if the life-span is *"application"*, the history log persists between sessions, and hence, the designer should carefully check that an appropriate consumption policy is in place so that history logs are prevented from growing too large. Some timeout mechanism that limits the lifespan of an occurrence could be useful in this context but has not yet been investigated.

7. Binding workview with HTML pages

Separation of concerns eases the stringent demands put on web application development and maintenance. This work strives to differentiate between two dimensions throughout web site development: content management and service management. So far, we have focused on workview elicitation and definition. But, the workview sits between the site's pages (the outcome of the content team) and the services themselves (provided by external sources or developed in house). This section addresses how this binding is realized.

Turning back to our previous example, consider the content team has come up with an e-brochure on car information. The user is smartly guided through a set of HTML pages to find an appropriate car. Simultaneously, the team in charge of the service side provides a workview[3]. The question is how to integrate the workview into an HTML page. If a page renders car data, how is this page "extended" to include services (e.g. *purchaseRequest*) which belong to this workview? This in turn, poses three questions: (1) how is a workview managed? (2) how is a service enacted? and (3), where are the service views rendered on the hosting page? In terms of implementation, the answers to these questions have been realized as three functions, more concretely, three JSP custom tags: *<wxl:use>*, *<wxl:include-activator>* and *<wxl:include-viewer>*. The next paragraphs discuss these questions in more detail.

The notion of workview has been realized in a first implementation. Although space limitation prevent us from describing this implementation in detail, three functions are provided to interact with this environment:

1 *<wxl:use workviewId= "aWorkview">*, which initializes the session state, if necessary. A session state includes an instance of a *SessionWorkview-Manager* that interprets the definition of *"aWorkview"*, and maintains the session history log. If the session state has already been initialized,

[3]The reader is strongly recommended to access *http://sipl68.si.ehu.es/wswl/examples/car-retailer/*. Besides running the application, this site contains the complete specification of the services, the workview and a hosting HTML page.

the *<wxl:use>* function retrieves the *REQUEST* parameters and informs the workview runtime to take the appropriate actions (e.g. enacting a service or updating the history log).

2 *<wxl:include-activator workviewId="aWorkview" componentId="aComponent">*, which checks whether "aComponent" is enabled, and if so, provides some HTML to enact this component (e.g. an anchor, a button and the like).

3 *<wxl:include-viewer workviewId="aWorkview" componentId="aComponent">*. If *"aComponent"* is enacted (i.e. the activator has been clicked on), this function indicates the place where the outcome is rendered.

Another aspect to be addressed is how to pass data from the hosting page to the workview. For example, The *purchaseRequest* service can be enacted from the page which renders car data. If this service is invoked, its *carId* parameter should refer to the car which is currently on display. If the car data is dynamically generated, this data can only be known at runtime.

Parameter passing between the page and the hosted workview can be seamlessly achieved through a **notification service**. As far as the workview is concerned, a notification service behaves like any other service: it should be included in the *"services"* tag and can be stored in the history log. The only difference is that it cannot be subject to any dependency since its enactment is outside the control of the workview. It is the hosting page that decides when to produce a notification service. This is achieved through the **<wxl:notify>** function. For instance, when included into an HTML page the following function notifies the car being displayed:

```
<wxl:notify workviewId="car-retailer" componentID="carSelection">

    <carSelection>
        <carId> <%=carInfo.getCarID()%> </carId>
        <carModel> <%=carInfo.getCarModel()%>
        </carModel>
    </carSelection>

</wxl:notify>
```

The function notifies the workview about the enactment of *"carSelection"* together with its actual parameters. Notice that the enactment of this service is a necessary condition for the availability of *"carConfiguration"* (see figure 5). If enacted, *"carConfiguration"* retrieves its *carId* and *carModel* parameters from the last notification of *"carSelection"*.

8. Conclusions

This paper present the notion of workview. The *WorkviewXL* language has been presented, and an interpreter, implemented. Workviews strive to achieve at the front-end what workflows fulfill at the back-end: smooth, loose-coupled integration of independent services. In our opinion, the notion of workview brings the following advantages:

1 It alleviates the application server workload as some of the flow dependencies can be enforced at the Web server side

2 It eases development and maintenance. The declarative and modular description of flow dependencies facilitates the addition and removal of flow dependencies during the site lifespan. As the site incorporates new services or additional business policies need to be enforced, flow dependencies can be added/removed with little or no impact on the rest of the site. Furthermore, site evolution is an increasing demand among practitioners due to the dynamic and competitive business environments that characterized most Web applications.

3 It improves coherence and thus, usability of the site. Most sites offer loosely-coupled services where the role of the Web site is almost restricted to be a front-end for invoking services. However, if the number of services is large, it is fundamental to struggle for the site to appear to function as a single whole.

4 It promotes separation of concerns, a well-known strategy for speeding up complex software projects. This work strives to separate content presentation from service delivery on the grounds that the underlying technologies and requirements demand distinct skills. This notion allows to distinguish service dependencies from how and where these services are enacted. In this way, it gives service and page designers the freedom to work independently. The latter are able to focus on content presentation and navigation without being involved in the intricacies of service dependencies. The page designer only has to decide where a service can be enacted and rendered, but she is not burdened with flow dependency enforcement and parameter passing concerns. These issues fall within the realm of the workview runtime.

References

[1] E. Anuff, M. Chaston, and D. Moses. Web Services User Interface: WSUI 1.0 at http://www.wsui.org/doc/20010717/wd-wsui-20010717.html, 2001.

[2] O. Diaz and J. J. Rodriguez. Interactive web services:definition and realisation. *Submited to ACM Transactions on Internet Technology*, 2002.

[3] A. Diaz et al. Web Services Experience Language (WSXL), January 2001. http://www-106.ibm.com/ developerworks/webservices/library/ws-wsxl/.

[4] F. Leyman. Web Services Flow Language (WSFL 1.0), May 2001. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[5] OASIS. Web Service for Interactive Applications, 2002. http://www.oasis-open.org/commitees/wsia/.

[6] A. Olivé. A comparison of the operational and deductive approaches to conceptual information systems modelling. In *Proc. IFIP-86*, pages 91–96, Dublin, 1986.

[7] W. Rajput. *E-Commerce Systems Architecture and Applications*. Artech House Publishers, 2000.

[8] R. Kalakota M. Robinson. *e-Business: Roadmap for Success*. Addison-Wesley, 1999.

[9] S. Thatte. XLANG: Web Services for Business Process Dessing, 2001. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.

[10] W3c. XML Path Language (XPath) Version 1.0, 1999. http://www.w3.org/TR/xpath.html.

[11] W3c. Web Services Description Language(WSDL) 1.1, 2001. http://www.w3c.org/TR/wsdl.

[12] WebCollage. Interactive web services (IWS), 2002. http://www.oasis-open.org/committees/ wsia/presentations/webcollage_iws_draft_0.7.pdf.