Model-driven development of Web Services and hypertext applications

M. Brambilla, S. Ceri, S. Comai, P. Fraternali (mbrambil|ceri|comai|fraterna@elet.polimi.it)
Dipartimento di Elettronica e Informazione P.zza L. da Vinci, 32
I-20133 Milano, Italy

Ioana Manolescu (Ioana.Manolescu@inria.fr)
INRIA Futurs, 78153 Le Chesnay, France

Abstract

This paper addresses the model-driven development of Web applications that integrate hypertextual navigation, content publishing and management, and interaction with remote Web Services. The proposed approach relies on an extension of the Web Modeling Language, a visual notation for the design of data-intensive Web applications, with primitives for capturing various forms of interaction with Web services, including one-way and request-response operations, asynchronous messaging, and long running conversations. The paper overviews some architectural issues raising from the integration of data-intensive Web applications and Web services.

1. Introduction

In the last years the Web has consolidated as the primary platform for application development. This success has required a revision of the pre-existing software engineering procedures and practices to adapt them to the specificity of Web development. The result of this adaptation effort is a new generation of Web engineering notations, methodologies, and tools [3, 4], which make the development of hypertextual interfaces accessed over the HTTP protocol the fulcrum of the software lifecycle. However, in recent times we have witnessed a shift of interest from "first generation" Web applications, mainly B2C and devoted to the dynamic publication of database content, to a "second generation" of Web-enabled systems, whose focus is primarily the B2B area.

From the technical standpoint, second generation Web application leverage Web Services and the related standards [6, 7] for achieving an open and universally available communication protocol, compatible with the underlying Web infrastructure. As a consequence, new proposals are flourishing that consider Web services as the building blocks of software systems and conceive new methods of composing them into applications for managing enterprise integration and distributed business processes [10, 1, 11]. A downside of several such proposals is that they neglect the fact that global business processes still involve the human user, who is often in charge of making the important decisions that make the process progress. To achieve this goal, new Web engineering methods and CASE tools are required, for analyzing, designing and implementing Web applications featuring both dynamic content publishing and Web service interaction. This paper presents a method for the conceptual modeling and automatic implementation of Web applications interacting with Web services, which extends an existing Web modeling language and is implemented in a commercial CASE tool. The starting point is the Web Modeling Language (WebML), a visual modeling language in use since 1998 for the high-level specification of data-centric Web applications and the automatic generation of their implementation code. The paper discusses the primitives that are necessary to extend a Web modeling language to make it capture the requirements of hypertextual applications interacting with Web services, and shows some conceptual and implementation issues arising from this extension.

The paper proceeds mainly by examples. After an overview of WebML and Web services (Section 2.), we introduce the various issues related to the invocation of Web services in the specification of a Web application (Section 3.). In particular, we discuss how to use Web service as a data source, the problem of storing data retrieved from Web services, and propose some primitives for effectively and efficiently manage the data to be sent or received from Web services. Then, Section 4. follows, briefly describing the implementation of the new proposed primitives in the architecture of the WebRatio CASE tool. Finally, Section 5. draws the conclusions.

2. Overview of WebML and Web services

Among the conceptual models for designing Web applications, our discussion will adopt the Web Modeling Language (WebML) [2, 3, 8], a visual language for the specification of data-intensive Web applications. However, the presented examples and results are independent of its particular notations and could be applied to other specifications languages as well. WebML allows specifying a Web site on top of a database. Such a conceptual Web specification consists of a *data model*, describing application data, and of one or more *hypertexts*, expressing the Web interface used to publish this data.

The data model. The WebML data model is the standard Entity-Relationship (E-R) model, widely used in general-purpose design tools. Its fundamental elements are therefore *entities*, defined as containers of data elements, and *relationships*, defined as semantic connections between

entities. Entities have named properties, called *attributes*, with an associated type. Entities can be organized in generalization *hierarchies* and relationships can be restricted by means of *cardinality constraints*.

The hypertext model. Upon the same data model it is possible to define different hypertexts (e.g., for different groups of users or for different publishing devices), called *site views*. A site view is a graph of pages, allowing users from the corresponding group to perform their specific activities. *Pages* consist of connected *units*, representing at a conceptual level atomic pieces of homogeneous information to be published. For example, they allow specifying a set of attributes for an entity instance (*data* units), all the instances for a given entity (*multidata* units), forms for collecting input values into fields (*entry* units), and so on. To determine the data to display, units are associated with an entity and a selector, testing complex logical conditions over the unit's entity. Units within a Web site are often related to each other through links carrying data from a unit to another (by means of *parameters*), needed to compute the hypertext. As an example, Figure 1 depicts a simple hypertext, consisting of two pages. Papers Page includes an index unit showing the list of all the instances of the entity Paper, from which one single paper can be selected and its details shown through the data unit. The link between the index unit and the data unit carries one parameter, containing the identifier (OID) of the paper selected from the index. The data unit uses this parameter for displaying the instance details: among all the instances of entity Paper, only those satisfying the selector [OID=CurrPaper] are retrieved, thus only the paper having attribute OID equal to the OID of the paper selected from the index is extracted. A further link allows instead navigating from Papers Page to Authors Page, where a multidata unit shows data about all the instances of the entity Author. This link does not carry any parameter: the content of Authors Page is independent from the content of Papers Page.

WebML allows specifying update operations on the data underlying the site too (e.g., the filling of a shopping trolley or the update of the users' personal information). Basic update operations are: the creation, modification and deletion of instances of an entity, or the creation and deletion of instances of a relationship. Unlike units, operations do not serve for displaying data (they are outside pages), but for updating it, or for performing other actions as we will see in the next sections. For further details about WebML the reader may refer to [3].

The language includes several other units and operations, and is *extensible*, allowing for the definition of customized operations and units. This extensibility can be exploited to include Web services operations.

In [5] a first proposal of such an extension is presented: both the data model and the hypertext model have been extended, to permit the storage of the meta-data about the Web services calls and the specification of the Web services calls themselves.

Web Services Data Model. All the interactions between the Web application and the Web services can be stored using a data schema for Web services, including the description of the service operations, their messages, and possibly the conversations of Web service operations fulfilling a common goal, e.g., browsing a product catalog followed by buying some product.

Web Services Hypertext Primitives. The interaction with Web service operations and the conversations of Web services have been represented by including six graphical primitives, depicted in Figure 2. This collection of operation symbols corresponds to the WSDL [7] type description of services. In the definition of the icons, we adopt two simple graphical conventions: (i) two-messages operations are represented as round-trip arrows; (ii) arrows from left to right correspond to input messages from the perspective of the service (i.e., messages sent by the Web application). Request-response and solicit-response operations can be defined also as asynchronous operations (represented by two superposed halves, one for each constituent message). In the case of a synchronous request-response, the user waits until the response message is received, while in the asynchronous case in-between operations are allowed. On the contrary, in the case of synchronous solicit-response operations the site constructs and sends right away the response, while in the asynchronous case the response may be sent later. Conversation creation and ending are designated by special markers attached to any Web service operation, as shown in Figure 2. Some examples of use of Web services operations are illustrated in the next section.

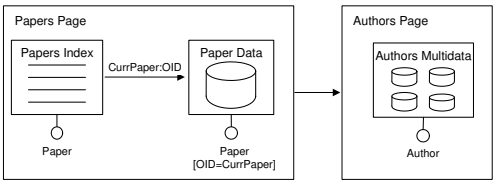

Fig. 1. Hypertext fragment specified using the WebML notation.

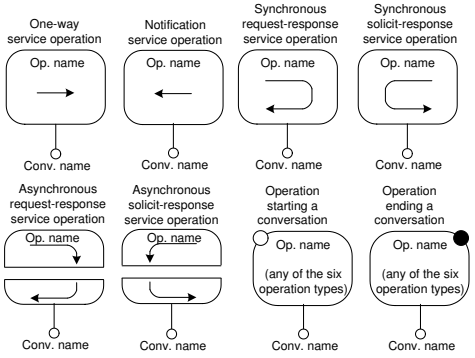

Fig. 2. WebML primitives for communicating with Web services.

3. Integrating Web services in data-centric Web applications

In this section we discuss some issues about the integration of Web services in a hypertext conceptual model. Various cases will be illustrated by means of examples expressed using the WebML language, and showing the basic building blocks needed for such integration. All the examples will be specified using request-response services, but the same concepts apply also to the other Web service operations.

Using Web services as data sources

The simplest way of using Web services in a hypertextual application consists of calling Web services operations that perform a specific task, and possibly using the data returned by the invoked Web service to define the content of Web pages. For example, services for sending e-mails, for converting a document into a different format (e.g., a postscript file into a pdf file), for getting the current temperature of a city, for doing searches on the Web and so on, are all examples of Web services available on the Internet. A Web application could incorporate such remote services, an offer them to the users without the need of implementing them directly or of storing the data required for computing them. The integration of a Web service into a Web application requires the management of the input and output messages of the Web service, which are in XML format. The following examples show how the marshalling of the data of the Web application into XML and vice versa can be handled, preserving both the use of ER as the abstract data model and the native parameter passing approach of WebML for propagating information among units. To this aim, a new unit for performing XML transformations is introduced, called *adapter*, taking in input N links carrying either HTTP-like parameters or XML parameters, and providing in output an XML document with the desired schema and content or, possibly, HTTP-like parameters, obtained as a transformation of the input data.

The adapter unit can be employed for: composing the input message of a Web service; decomposing the output message of a Web service; calling multiple Web services in sequence, adapting their input and output.

The hypertext in Figure 3 represents a complete example of invocation of a sequence of Web services. The user can enter the name of a city through an entry unit, depicted in page *Enter City*, and obtain its temperature, shown in page *Temperature*. Two Web services are used: the CityToZip allows to obtain the zip code of a given city; the ZipToTemp service provides the current temperature for the city having a given zip code.

When the user submits a request by inserting the name of the city, a chain of operation is activated. The first operation is an adapter operation (named Compose Message): it takes as input the value filled in by the user, bundled as a HTTP parameter. The parameter contains the value of a form field, and is transformed by the adapter unit into an element of the input XML message required by the CityToZip service. The XML message body is then passed as a parameter to the CityToZip operation, which encloses it in the SOAP message actually delivered to the remote service. The service returns in output a new XML message containing the zip code, which is transformed into the XML input message of the next service by means of another adapter unit (called Transform Message). Finally, the second service is invoked, which returns an XML document containing the temperature of the city having the zip code provided in the input message: the next adapter unit (Decompose Message) transforms it into an output parameter which is fed to an entry unit displaying its value. Notice that this example requires only the management of the messages of the Web service and their transformation.

Storing the data retrieved from Web services

In many applications, data retrieved from the Web services need also to be stored in the local data repository for being used after the request-response cycle has terminated: the lifetime of the data may require persistent or temporary storage. For example, the message provided by a Web service call could be stored permanently in the application database or temporarily until the user disconnects from the Web application.

Persistent storage. Persistent storage is achieved by means of operations updating the database (like creation, modification of instances, and so on). As an example, consider the hypertext in Figure 4, which stores the zip code obtained from the CityToZip service into the database. In this hypertext, the Create City unit is used to create a new instance of entity city in the underlying database. It receives two input parameters: the zip code obtained from the Web service and transformed into a parameter by the Decompose Message adapter unit, and the name of the city entered by the user into the Input City entry unit (the dash arc is used to "transport" parameters to other units). These two values are assigned to a new instance of entity City. When the user enters the City page, *possibly at a later time w.r.t. the service call*, a multidata unit (City Multidata) displays all the cities. Data in the repository are persistent; therefore, if the user disconnects from the Web application and accesses it again later, the data of the cities retrieved from the Web service are still available.

Temporary storage. In many cases, the data retrieved from a Web service have a shorter lifetime and need to be stored just for the time a user navigates the Web application (i.e., for the duration of a user's session), or for the time the user performs specific operations during navigation. This requires storing data temporarily in main memory instead of in the local database.

In WebML, it is possible to specify that an update operation be performed in main memory by adding a 'V' symbol to the unit to denote that a "virtual" entity instead of a database entity must be used. Instead, a "V" symbol added to a content unit indicates that the data to be shown to the user are retrieved from main memory. For example, if we add a "V" symbol to the Create City and City Multidata units in Figure 4 the zip codes would be stored and retrieved from main memory.

Lifetime of main memory entities. A meaningful lifetime of main memory entites is a user session; however, in many cases data need to be stored for shorter time, for

example, only between two calls of the same Web service. For this reason different policies may be applied to fine tune the behavior of update operations applied to main memory objects. Two policies are the most useful:

- **Flush**: each time an entity is updated the new data replace any pre-existing data.
- **Preserve**: when an entity is updated the new data is added to any pre-existing data.

A significant example of the flush policy will be shown in the next subsection.

Managing Web services data

Since Web services are not developed by the creator of the hypertext, the content of their input and output messages may not exactly correspond to the schema of the Web application data repository. As a consequence:

- The data for composing the input message required by a Web service may be spread across different application entities;
- The data obtained from an output message coming from a Web service may be stored into application entities having a structure that differs from the output of the Web service;
- To build a single entity instance several Web services calls may be needed.

In general, the above requirements result into the need of creating instances of complex data sub-schemas, involving several related entities, from the output of a service call and, dually, of assembling the input message of a service from the instances of a complex data sub-schema. To avoid the need of expressing the sequence of operations needed to create a data sub-schema from a complex output message and the patterns of content units necessary to extract the data for producing a complex input message, we introduce the notions of materialization and dematerialization of entities and relationships instances as a syntactic sugar for marshalling and unmarshalling data sub-schemas into/from XML messages in one step.

Managing the creation of new instances by multiple service calls. A Web service may be called multiple times to create instances of the same data sub-schema. For example, in the hypertext in Figure 4 the user may insert the same city name repeatedly; more interestingly, different services may retrieve different pieces of information of the same object or sub-schema (for example, in Figure 3 the first Web service retrieves the zip code of a city, while the second one retrieves its temperature), or also the same Web service may be used at different times to retrieve different pieces of information for the same object. This requires policies for managing the cumulative update of data, because the insertion of the data retrieved from a Web service in some cases may require the creation of novel instances, while in other situations may require the modification of existing instances. Given an object instance, whose unique identity is specified by a set of *key attributes*, examples of possible behaviors are:

- If no other instance having the same key exists, a new instance is inserted.
- If an instance having the same key already exists, then:
  - **No action**: no action is performed; the content of the existing instance is held.
  - **Overwrite**: the content of the new instance overwrites the existing one; all the attributes of the previous instance are deleted.
  - **Extend**: the content of the new instance overwrites the existing one, for all the attributes of the new instance.

Materialization and dematerialization. In order to facilitate the retrieval of the desired content from the database or from main memory to compose a complex input message, and the update of the database or of main memory with the data obtained from a complex output message of a Web service, the *dematerializer* and the *materializer* operations are introduced.

Both operations exploit an intermediate XML representation of the ER instances, structured according to a *canonical XML schema*, where all the instances of entities, their relationships, and the selected attributes are enclosed between XML tags (for example, the XML schema generated by a relational DBMS marshalling XML data could be used).

The *dematerializer* unit allows the selection of a set of objects belonging to a ER sub-schema from main memory or from the database: the objects and relationships are provided in output as an XML file conforming to the canonical XML schema.

Symmetrically, the *materializer* unit performs the opposite operation: it accepts as input an XML file conforming to the canonical XML schema and describing a set of instances of an ER sub-schema, and inserts them into the database or into main memory.

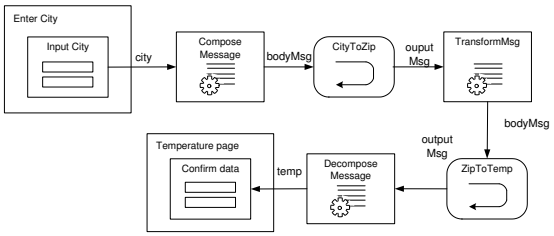

Fig. 3. Chains of Web services.

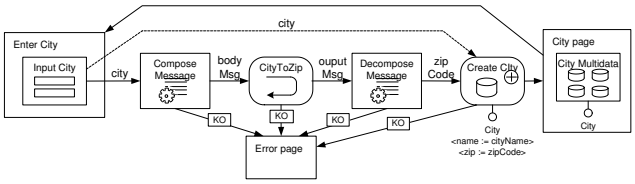

Fig. 4. Persistent storage.

Also for the materializer operation, it is possible to define the lifetime of the data (in case of main memory storage) and the behavior in presence of cumulative updates (e.g., no action, extend, overwrite, etc.). Figure 5 shows the graphical notation of the materializer and dematerializer units. These two units can be exploited in combination with Web service operation units as shown in the following example.

Materialization. The hypertext in Figure 6 calls the Google Web Search service, which requires in input the keywords to search plus some parameters like the maximum number of results and possible search filters, and returns a list of search results, possibly classified into categories. The Web Search page in Figure 6 contains an entry unit (Search the Web) for inputting some search keywords; a multidata unit (All results) displays the results and contains the links to the corresponding Web pages; finally, an index unit (Categories) lists the Google categories of the results. When the user selects a category, page Results of Category is loaded, listing the results belonging to the selected category. The implementation of this example is available at http://www.webratio.com/webservicedemo. When the user submits the keywords through the entry unit, the output link passes them as a parameter to the adapter unit (Compose message), which builds the XML message for the request-response unit (Google Search). While the structure of the input message is very simple, the output message has a more complex structure. For simplifying the specification of the mapping of its data into main memory we use a materialization unit. Therefore, in the hypertext in Figure 6 the output message of the Google Search service is supplied to an adapter unit (Transform into XML-ER) transforming it into the canonical XML representation of the instances to create; the canonical XML is fed to a materializer unit (Materialize), which creates the new instances into main memory. Specifically, search results are stored into an entity called GoogleResultElements, while categories are stored into the GoogleDirectoryCategory entity; moreover, for all the elements associated with a category, a relationship between the two entities instances is built.

As far as the main memory policies are concerned, the *flush* policy is appropriate for this example, because only the results of the most recent search must be shown. Among the possible cumulative update semantics, any option can be chosen, because the main memory objects collecting the results are emptied every time a new search is performed: for example, with the *no action* option, at each service call all the distinct elements of the result are copied into main memory.

After the materialization operation, the Web Search page is reloaded. The multidata unit (All results), defined over the GoogleResultElements virtual entity, retrieves all the results stored by the materialization unit. The index unit (Categories) defined over the GoogleResultDirectories main memory entity shows the categories stored by the materialization unit.

Notice that when the user enters this page for the first time, the multidata and index units are empty, since no data have been stored yet. Every time the service is invoked, the data in main memory and the content of these two units are updated accordingly.

Dematerialization. The dematerializer unit can be used when the input message of a service is complex and must be composed by combining objects coming from an ER sub-schema. To this end, the dematerializer unit allows selecting an ER sub-schema and transforms its instances into the canonical XML representation, which can then be transformed by the adapter unit into the input message of the Web service.

Transferring data between main memory and database. Notice that by combining a dematerializer unit with a materializer unit, it is also possible to transfer portions of data from main memory entities into the database (and, vice versa, from the database into main memory).

Indeed, some applications may require that the data retrieved by Web services be temporarily stored in main memory and that at some point during the application (typically as a consequence of a particular operation, like the checkout of a shopping cart) be persistently stored.

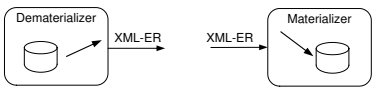

Fig. 5. Graphical notation for the materializer and dematerializer units.

4. Implementation

The hypertext conceptual model and its extensions with Web services have been implemented inside the WebRatio CASE tool [WebR], an integrated environment for the visual specification of Web applications and the automatic generation of code for the J2EE and Microsoft .NET platforms.

Several real examples using the concepts introduced in this paper and implemented with WebRatio are available at the site http://www.webratio.com/webservicedemo.

Adapter unit. As shown in Section 3.1 the adapter unit supports three coupling modes among units: it allows

- Building an XML document from a set of HTTP or server-side parameters;
- Building a set of HTTP or server-side parameters from an XML document;
- Transforming input XML documents into output XML documents.

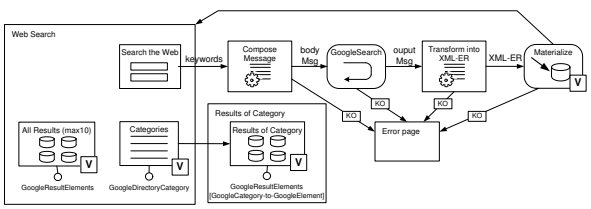

Fig. 6. Google Search service and use of materialization.

The unit has been implemented by means of XSL [XSL], a language for defining transformations of XML documents . The transformations are specified by means of templates, i.e., pattern-matching rules applied to XML elements. While the XML-XML transformation is very general, here we show an example of parameter-to-XML coupling, realized by means of an XSL template. The XML-to-parameter coupling works in a similar way, but produces parameters as output.

Consider the Compose adapter unit in Figure 3, exposing the *city* input parameter received from the entry unit. A possible XSL stylesheet for this transformation is shown below: it contains the input parameter and a rule template matching the entire document and defining a fixed XML markup: only the actual value of the input parameter varies at each call and at run-time its value is inserted into the output document through the <xsl:value-of ... /> construct.

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
 <xsl:param name="city"/>
 <xsl:template match="/">
  <!-- other elements of the message body -->
  <city xsi:type="xsd:string">
      <xsl:value-of select="$city"/>
  </city>
  <!-- other elements of the message body-->
  </xsl:template>
</xsl:stylesheet>
```

Dematerializer and materializer units. The dematerializer unit marshals relational data out of databases (or main memory) according to a pre-defined XML schema. The materializer unit performs the opposite operations and maps XML data into relational (or main memory) data. Currently, they have been implemented using the generic canonical XML schema described in the preceding section as an intermediate data format.

Temporary entities. Internally, the WebRatio runtime framework is organized according to the model-view-controller (MVC) architecture: each request is sent to the Controller, which dispatches it to the model action in charge of serving it. If the request is an HTTP request for a page, a page action is invoked, which extracts the content from the data sources and builds the content objects (unit beans) necessary for the page templates of the View to assemble the HTML response. In this architecture all the business objects are implemented as beans, i.e., server-side components. Temporary entities can therefore be stored into entity beans: the implementation of virtual units defined over main memory entities can access such data instead of the database, implementing the updates in main memory with the semantics of the corresponding updates in relational databases.

5. Conclusion

In this paper we have presented by means of simple examples the requirements and the primitives needed to model Web applications interacting with Web services, with the aim of providing an automating implementation of such integration. The proposed approach has been shown on the WebML language, which is based on the Entity-Relationship model. However, all the issues discussed in the paper apply also to other conceptual models.

With the help of some simple examples we have shown the building blocks for the management of the data to be sent to and received from Web services:

- Supporting the transformation of application data into Web services data, and vice versa.
- Providing different storage alternatives, for an efficient management of data obtained from Web services.
- Providing a few specific operations for representing at the conceptual level common processes deriving from the integration of Web services.

As future work we plan to study these problems in more depth and to propose also a design methodology for the specification of traditional Web applications enhanced with Web services.

References

1. Business Process Execution Language for Web Services Available at: http://www.ibm.com/developerworks/webservices
2. S. Ceri, P. Fraternali, A. Bongio Web Modeling Language (WebML): a Modeling Language for Designing Web sites. *WWW Conference, 2000*.
3. S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, M. Matera. Designing Data-Intensive Web Applications. *Morgan-Kaufmann*, Dec. 2002.
4. J.Conallen. Building Web Applications with UML . *Addison Wesley Professional*: 1999.
5. I. Manolescu, M. Brambilla, S. Ceri, S. Comai, P. Fraternali. Exploring the combined potential of Web sites and Web services. *WWW'03* (poster), Budapest, 2003.
6. SOAP. Available at http://www.w3.org/TR
7. WSDL. Available at http://www.w3.org/TR
8. The WebML Project: http://www.webml.org.
9. The WebRatio Tool Suite: http://www.webratio.com
10. Web Service Flow Language. http://www-4.ibm.com/software/solutions/Webservices/pdf/WSFL.pdf
11. Web Services for Business Process Design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c