Conceptual Modelling Issues in Web Applications Enhanced with Web Services

Sara Comai
Dipartimento di Elettronica e Informazione
P.zza L. da Vinci, 32 – I-20133 Milano, Italy
e-mail: comai@elet.polimi.it

Abstract

Several conceptual models have been proposed in the last years for designing and developing Web applications. This paper investigates the primitives that are necessary to make a Web modeling language capable of capturing the requirements of hypertextual applications interacting with Web services, and discusses some conceptual and implementation issues arising from this extension. In particular, the use of Web services as data sources, the storage of data obtained from Web services, and their management are discussed. These aspects are validated by means of some demonstrative examples.

1 Introduction

As the Web becomes more and more a platform for enterprise integration, the focus of Web application development is shifting from the mere publication of dynamic content (e.g., content coming from the enterprise Information System) to the integrated publication of content and services. For example, in a sell-side B2B extranet application a corporate may expose various business services, like the inspection of order status reports and credit notes, to its distributors, who in turn publish operations for assessing stock levels and sales reports. These services complement the content exchanged by the business partners and can be used to compose an integrated, multi-organization workflow, which supports the business processes of several partners.

From the technological point of view, the need arises of a widely accepted standard for service invocation, which complements the open standard of the Web for content publishing and management. Standards like [SOAP] have recently emerged for Web-oriented enterprise integration, thanks to their use of HTTP and XML as the vehicle of service invocation and information exchange.

However, enterprise integration challenges not only the technologies whereby systems are built but also the development methodology. 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, as advocated, for instance by the OASIS Web Services for Interactive Applications (WSIA) working group [OASIS].

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 of the proposal discussed in the paper is the Web Modeling Language (WebML), a simplified UML-compatible visual modeling language in use since 1998 for the high-level specification of data-centric Web application and the automatic generation of their implementation code. The paper investigates the primitives that are necessary to add to a Web modeling language to make it capable of capturing the requirements of hypertextual applications interacting with Web services, and discusses some conceptual and implementation issues arising from this extension.

The main contributions of the paper are:

- A high level study of the problems related to the invocation of a remote Web service in the definition of a Web-centric, hypertextual interface.
- A set of primitives that seamlessly extend WebML to cope with remote service invocations. Such primitives are not particular to WebML, but could be recast in the context of other conceptual languages providing the notion of operation invocation, like UML, W2000 [BGP01], etc.
- A brief overview of the implementation techniques adopted to integrate the proposed primitives within the architecture of a commercial CASE tool (called WebRatio [WebR]), which permits the visual modeling of data-centric Web applications and the generation of their code. Thanks to this integration,

the broader class of data- and service-centric Web application can benefit from the Web engineering development lifecycle supported in WebRatio. Similar techniques could be implemented in the context of other tools, like Rational Rose XDE [RationalRose].

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, Section 3.1 discusses how to use Web service as a data source; Section 3.2 focuses on the problem of storing data retrieved from Web services; Section 3.3 handles this problem in more depth by proposing a more effective and efficient management of 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, while in Section 5 related work is presented. Finally, Section 6 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) [CFB00, CF+02], 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. In particular, they allow specifying a set of attributes for an entity instance (*data* unit), all the instances for a given entity (*multidata* unit), list of properties, also called descriptive keys, of a given set of entity instances (*index* unit), and also scrolling the elements of a set (*scroller* unit). An additional unit represents a form for collecting input values into fields (*data entry* unit). 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.

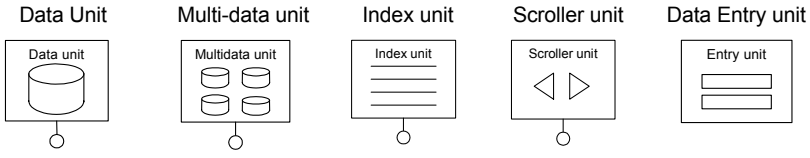

Figure 1 – WebML Content units.

As an example, Figure 2 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.

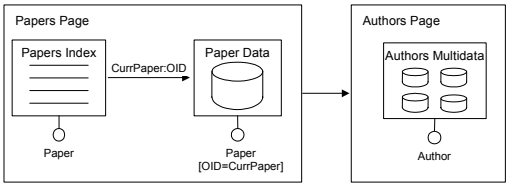

Figure 2 – Example of Hypertext model.

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, graphically represented in Figure 3.

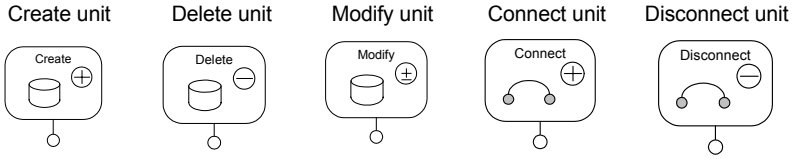

Figure 3 – WebML operation units.

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 [CF+02].

The language includes several other units and operations, and is *extensible*, allowing for the definition of customized operations and units. In [MB+03] this extensibility has been exploited to include Web services operations. In particular, 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 4. This collection of operation symbols corresponds to the WSDL [WSDL] 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).

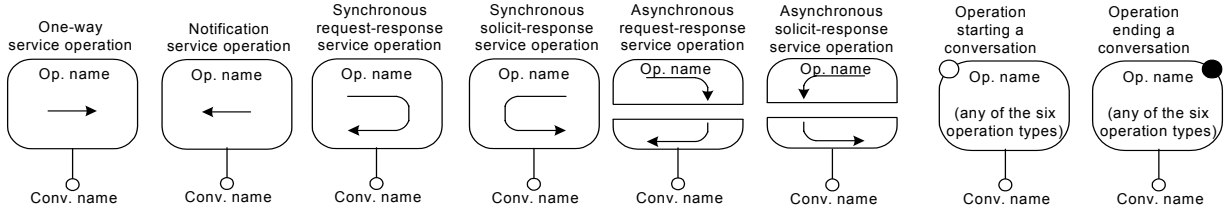

Figure 4 : WebML primitives for communicating with Web services.

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 4. Some examples of use of Web services are illustrated in the next section.

3 Web services at work

In this section we discuss some issues about the integration of Web services in an hypertextual 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. In particular, we will start by discussing how Web services can be used as operations or as data sources, and then we will focus on the storage of the data received by Web services and on their management. All the examples will be specified using request-response services, but the same concepts apply also to the other service operations.

3.1 Using Web services as operations or as data sources

The simplest way of using Web services in a hypertextual application consists of calling Web services operations that perform a particular task, and possibly using the data returned by the invoked Web service to define the content of other hypertext units. 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 operations available on the Internet. A Web application could incorporate such remote services, to 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*. The *adapter* unit, graphically represented as in Figure 5, takes in input N links carrying either HTTP-like parameters or XML parameters, and outputs another XML document with the desired schema and content or, possibly, HTTP-like parameters, obtained as a transformation of the input data.

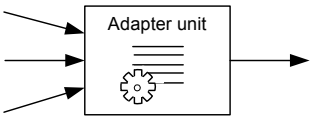

Figure 5 – Graphical notation for the adapter unit.

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.

Composing the input message. The hypertext in Figure 6 allows the user to compose an e-mail message and to invoke a remote Web service for sending the e-mail. The input message of the Web service has a complex schema including, for example, the *from*, *to*, *Cc*, *body* and *attachments* attributes. Some of these attributes, such as *to*, *Cc* and *attachments* consist of arrays of strings, thus permitting to send the same e-mail to different recipients, possibly with a set of attachments. The output message of the service is a simple string, which will be ignored in this preliminary example.

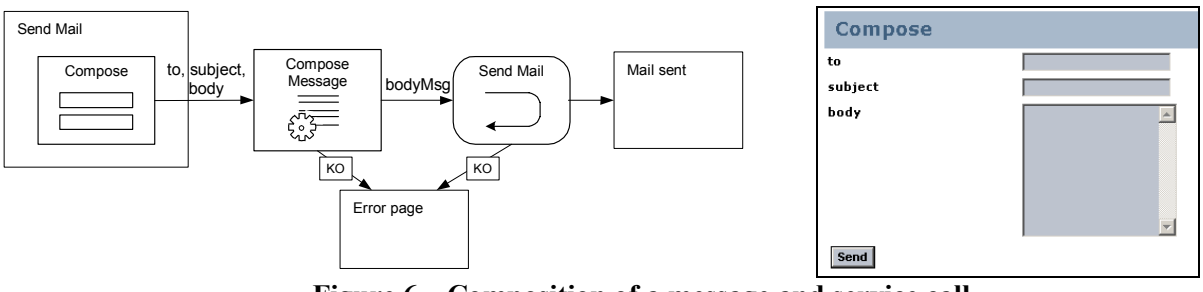

Figure 6 – Composition of a message and service call.

The hypertext in Figure 6 includes a page called Send Mail, which contains an entry unit (named Compose) for inputting the content of the mail message. A possible rendition of the Compose entry unit is shown on the right hand side in Figure 6. When the user clicks on the send button, a chain of operations is activated.
The first operation is an adapter operation (named Compose Message): it takes as input the set of values filled in by the user, bundled as a set of parameters. Each 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 mail Web service. The XML message body is then passed as a parameter to the Send Email request-response operation, which encloses it in the SOAP message actually delivered to the remote service.
If any of the two operations fails (for example, the Web service is unavailable), a KO link leads the user to an Error page; otherwise page Mail Sent is shown, displaying a static, unmodeled message confirming the delivery of the mail message.

Decomposing the output message. The opposite operation, transforming an XML document into a set of parameters, is exemplified in the hypertext fragment in Figure 7, where the user can enter the name of a given city and obtain from the call of a Web service (called CityToZip) the zip code of that city.

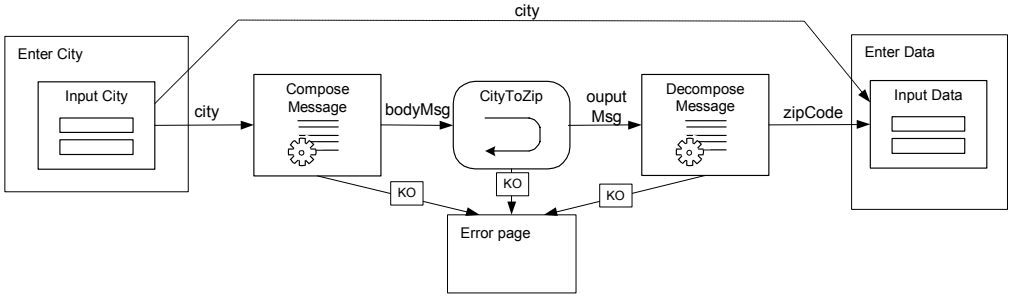

Figure 7 – Composition of a message, call of a Web service and decomposition of the output message.

Page EnterCity contains an entry unit (Input City) for submitting the name of the city. Like in the previous example, the first invoked operation is an adapter unit (ComposeMessage), taking in input the value entered by the user. The adapter unit constructs the input message body of the Web service. The CityToZip operation sends a message to the Web service and returns in output a new XML message containing the zip code. The next adapter unit (Decompose message) transforms such XML document into an output parameter, which are fed to an entry unit (City and Zip) displaying both the city name inputted by the user and the retrieved ZIP code, and accepting further inputs from the user. The city parameter is passed to this entry unit by means of a transport (non navigational) link, represented with a dash arc.

Chains of Web services. The hypertext in Figure 8 represents an example of invocation of a sequence of Web services. The CityToZip service is used to obtain the zip code of a given city; the ZipToTemp service provides the current temperature for the city having a given zip code.

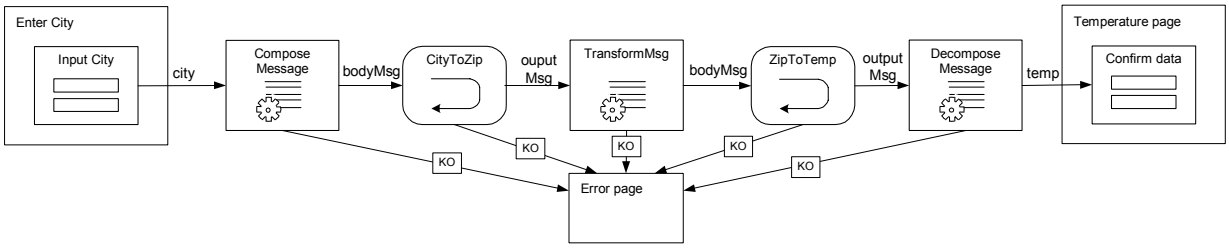

Figure 8 – Chains of Web services.

In this example the adapter unit placed between the two services is used for converting the XML output message of the first service unit into the XML input message of the next service unit. The decomposition of the output message and the composition of the input message can be merged into a single transformation.

In all these examples, Web service operations have been used to perform particular operations (e.g., sending an e-mail) or to retrieve data from remote Web services instead of keeping them in the database (e.g., the zip codes and the temperatures of certain city). Both cases require only the management of the messages of the Web service and their transformation: for integrating Web services operations in Web applications, WebML

adopts a mechanism based on the concept of parameter, which can be received in input or fed as output to other units, and introduces the concept of adapter for performing all the needed transformations.

3.2 Storing the data retrieved from Web services

In many applications, data retrieved from the Web services need also to be stored in the data repository: depending on the kind of information obtained from the service, the lifetime of the data may require persistent or temporary storage. For example, the message provided by a Web service could be stored permanently or 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 9, which stores the zip code obtained from the CityToZip service into the database.

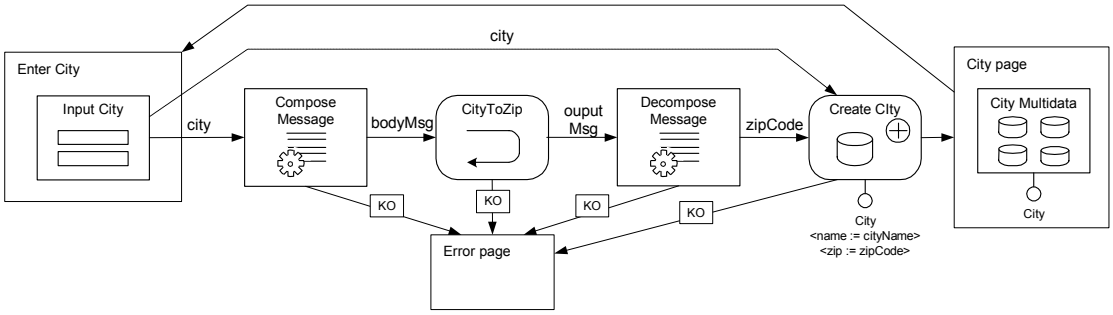

Figure 9 – Persistent storage.

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. These two values are assigned to a new instance of entity City. When the user enters the City page a multidata unit (City Multidata) retrieves all the instances from the City entity. 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 are still available.

Temporary storage. In many applicative 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 a user session), or for the time the user performs specific operations during navigation. A possible mechanism for storing data temporarily is to use main memory instead of the underlying database.

In WebML it is possible to specify that an update operation be performed into main memory entities by adding a 'V' symbol to the unit to indicate that a "virtual" entity instead of a database entity must be used, as shown in Figure 10: the Create City operation stores the data into a main memory entity called City and the City Mutidata unit retrieves the instances from such entity. If the lifetime of main memory entities is a user session, then, if the user disconnects from the Web application and accesses it later, the data of the cities entered during the previous sessions are lost. This solution allows an easy and efficient management of the data that need to be kept temporary, since no garbage collection need to be performed on them when the user disconnects.

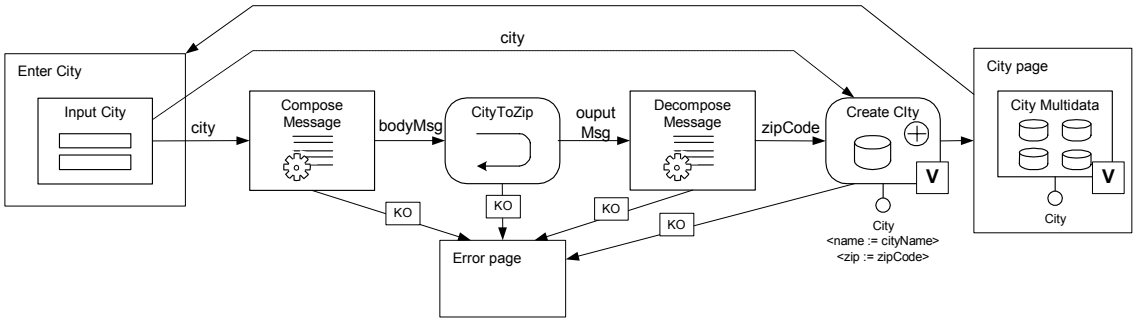

Figure 10 – Temporary storage in main memory (virtual) entities.

Lifetime of main memory entities. In the previous example we have assumed that the lifetime of main memory entities is a user session, which is meaningful for many Web applications. 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 the behavior of update operations into main memory. We have identified two useful policies:

- **Flush policy**: each time an entity is updated its content is emptied and the new data replace the previous ones.
- **Preserve policy**: when an entity is updated the previous content is kept.

If we apply these to policies to the Create City unit in the example in Figure 10 the hypertext produces different results: with the *flush policy*, each time the user inserts a new city, only the last inserted value and its zip are stored in the City main memory entity and therefore the City Multidata unit shows only one city value; with the *preserve policy*, all the cities inserted by the user are kept in main memory until the end of the user session, and therefore the City Multidata unit shows all the cities inserted by the user during the session. These two policies may be useful in many cases; a more significative example of use of the flush policy will be shown in the next subsection.

3.3 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 Web application data designed for the hypertext. As a consequence:

- the data for composing an input message may be spread in different application entities;
- the data obtained from an output message may be stored into application entities, different from the structure defined by the Web service;
- to build a single entity instance several Web services calls may be needed.

The concepts introduced in the previous sections permit to cope with all these requirements. However, from an analysis of the characteristics of the Web services data and of the possible operations on such data, we have identified some common semantic behaviors and actions that can be encapsulated into the semantics of traditional update operations or into new operation units, thus optimizing the management of such data and simplifying the hypertextual conceptual model. At this aim, we extend the concept of creation of new instances and introduce the notions of materialization and dematerialization of entities and relationships instances.

Managing the creation of new instances. A Web service may be called multiple times, possibly for the same data (e.g., in the hypertext in Figure 9 the user may insert the same city name repeatedly); different services may retrieve different pieces of information of the same object (for example, in Figure 8 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 in different instants to retrieve different pieces of information for the same object. It may therefore happen, that the insertion of the data retrieved from a Web service, both in main memory and into the database, in some cases requires a creation, while in others an update of existing instances.

Different semantics to the create operation may be applied. Given an instance identified by a key, 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 null**: only the null attributes of the existing instance are overwritten by the attributes of the new instance.
  - **Extend by overwriting**: the content of the new instance overwrites the existing one, for all the attributes of the new instance.

Consider, for example, the hypertext in Figure 11, extending the example shown in Figure 10 (for simplicity KO links are omitted): the user may either ask the zip code for a city or its temperature. In the former case, the Web service CityToZip is called, in the latter case the Web service CityToTemp is invoked. In both cases, the data are stored in main memory into the City entity. This entity contains only one instance for each city. City page shows all the cities inserted by the user with the requested zip codes/temperatures. Depending on which service is called first, a creation or a modification operation need to be performed on the city entity. For example, suppose that the user first asks the zip code of a city, and then requests its temperature twice. If the *extend by overwriting* option is chosen for both create operations, the hypertext behaves as follows: initially, the zip code is stored; then when the CityToTemp service is invoked for the first time the instance of the city is extended with the temperature value; then, when the CityToTemp service is invoked for the second time the new temperature replaces the existing value.

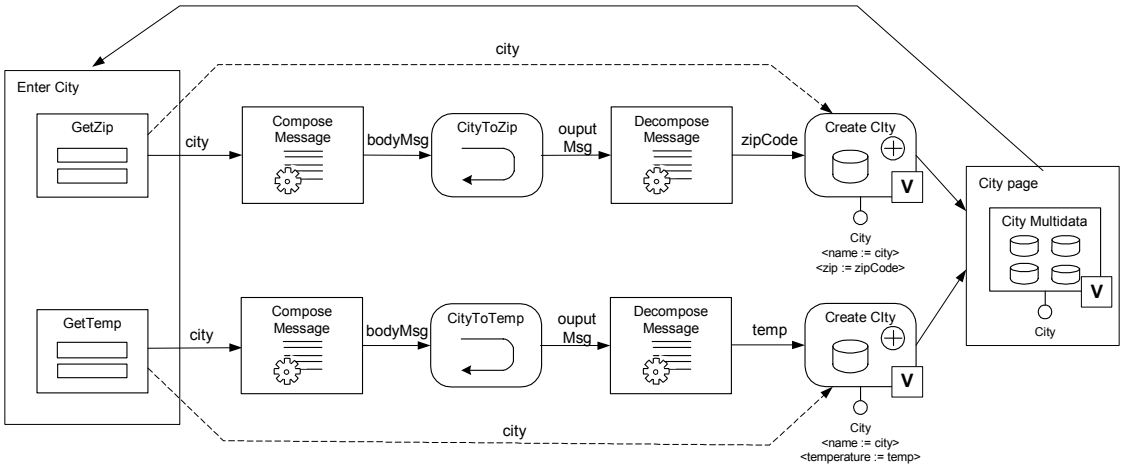

Figure 11 – Using Web services for getting different information for the same object.

Materialization and dematerialization. In order to facilitate the retrieval of the desired content from the database (or main memory) entities to compose a complex input message, and the update of the database (or 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 manage an XML version of the data stored in relational databases, conforming to a *canonical XML schema*. For the purpose of this paper we can assume that the XML version of the database has a format like the one shown in Figure 12, where all the instances of entities, their relationships, and the selected attributes are enclosed between intuitive tags[1].

[1] Alternatively, the XML schema generated by a relational DBMS marshalling XML data could be used.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <entity id="Ent1">
    <instance>
      <attribute id="oid">item1</attribute>
      <attribute id="att1"> ... </attribute>
    </instance>
    <instance>
      <attribute id="oid">item2</attribute>
      <attribute id="att1"> ... </attribute>
    </instance>
  </entity>
  <entity id="Ent2">
    <instance>
      <attribute id="oid">object1</attribute>
      <attribute id="att2"> ... </attribute>
    </instance>
    <instance>
      <attribute id="oid">object2</attribute>
      <attribute id="att2"> ... </attribute>
    </instance>
 </entity>
```

```
<relationship id="rel_Ent1_Ent2">
    <instance>
      <source-oid>item1</source-oid>
      <target-oid>object2</target-oid>
    </instance>
    <instance>
      <source-oid>item2</source-oid>
      <target-oid>object2</target-oid>
    </instance>
  </relationship>
</root>
```

Figure 12 – Fragment of document conforming to a canonical XML schema.

The *dematerializer* unit allows the selection of a sets of objects belonging to different entities and/or relationships 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 some entities and/or relationships, and copies them into the database or into main memory.
Since the materialization operation consists of a set of updates into the database or into main memory, the lifetime of the entities (in case of main memory storage) and the behavior of each update operation (e.g., no action, extend, overwrite, etc.) must be chosen.
Figure 13 shows the graphical notation of the materializer and dematerializer unit.

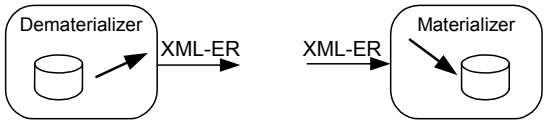

Figure 13 – Graphical notation for the materializer and dematerializer units.

These two units can be effectively exploited in combination with Web service operation units as shown in the following example, demonstrating the typical use of the materializer unit.

Materialization. The hypertext in Figure 14 calls the Google Web Search service, which requires in input the keywords to search plus some parameters like, for example, the maximum number of results to supply and possible search filters, and provides in output a list of search results and their categories.

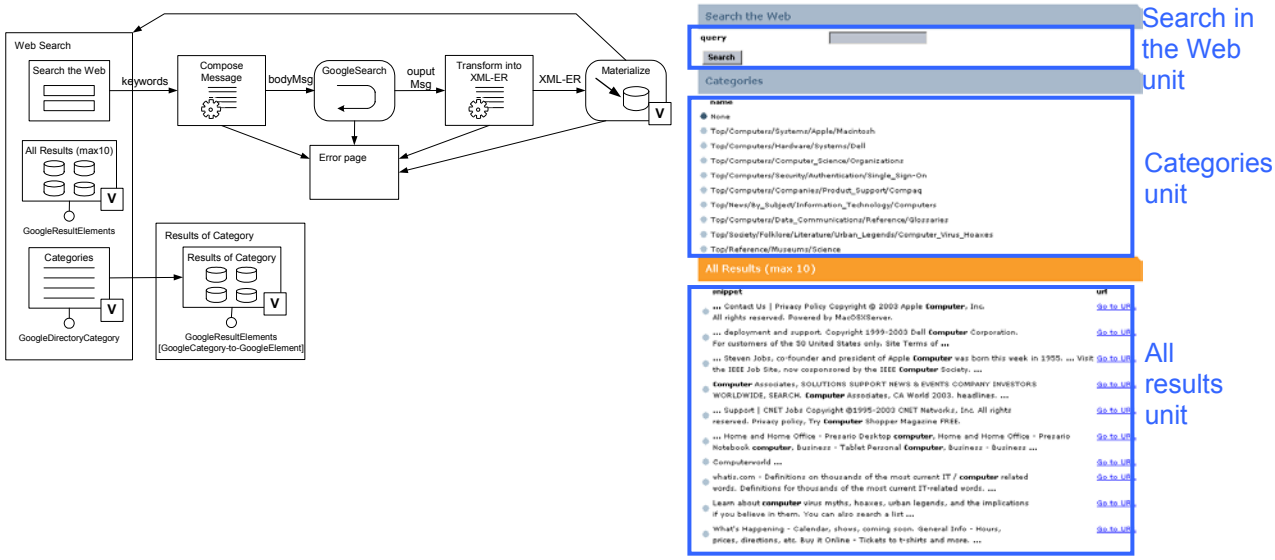

Figure 14 –Google Search service and use of materialization.

The Web Search page in Figure 14 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/directories of the results. When the user selects a category, page Results of Categories is loaded, listing all the results belonging to the selected category.

An example of snapshot of the Web Search page is shown on the right hand side in the same figure. When the user submits the keywords through the entry unit, its output link passes them as a parameter to the adapter unit (Compose message), which composes the XML message to be passed to the request-response service unit (Google Search). Figure 18 and Figure 19 show an example of the body of XML input message and an example of XML output message of the Google search service, respectively. These examples point out that the structure of the input message is very simple, while the output message presents a more complex structure. For simplifying the specification of the mapping of its data into main memory we can use a materialization unit. Therefore, in the hypertext in Figure 14 the output message of the Google Search service supplied to the next adapter unit (Transform into XML-ER) is transformed into an XML document describing the instances and relationships to create, according to the XML canonical schema; this file can then be fed to a materializer unit (Materialize), which creates the new instances into main memory. In particular, 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 created. Notice that this operation is not primitive (a set of elementary updates could be specified), but permits a clear and compact specification at the conceptual level.

As far as the main memory policies are concerned, the *flush* policy is appropriate for this example, since only the search results of the most recent keywords must be shown. Instead among the possible behaviours for creating the new instances, any solution could be chosen, since the main memory entities 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 main memory 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 this page is entered by the user for the first time, the multidata and index units are empty, since no data have been stored into main memory. Every time the service is invoked the data in main memory and the content of these two units are updated accordingly.

Dematerialization. Symmetrically to the use of the materializer, the dematerializer unit can be efficaciously used when the input message of a service is complex and its content must be composed by combining objects coming from several entities, bound one to each other by relationships. Indeed, the dematerializer allows

selecting a portion of database (or of main memory objects) and supplies its XML canonical representation. This XML representation can then be transformed by the adapter unit into the input message of the service operation.

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) be persistently stored.

Notice that materialization and dematerialization operations may occur very frequently in the specification of Web applications integrated with Web services, thus justifying the introduction of non-primitive concepts.

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.
Figure 15 depicts its architecture, consisting of two layers: a design layer, providing functions for the visual editing of specifications (internally represented by XML descriptors), and a runtime layer implementing a Model View Controller Web application framework [CFA03], an architecture widely adopted by Web development platforms. These layers are connected by the WebRatio code generator, mapping the XML descriptors of the visual specifications of the design layer into application code executable within the runtime layer.

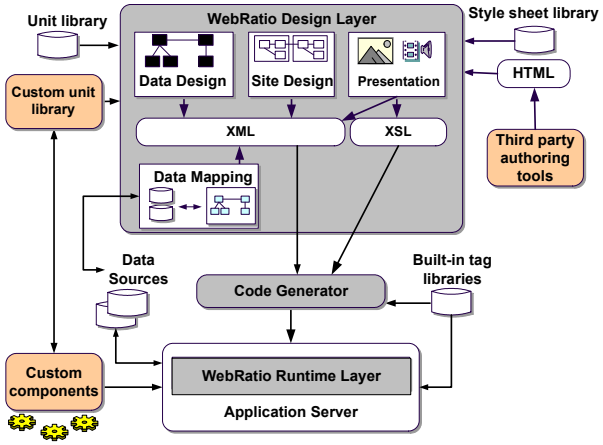

Figure 15 - Architecture of WebRatio CASE tool.

This framework has a plug-in architecture: new components can be defined as custom WebML units, the code generator can be extended to produce code wrapping the plug-in components, and the components themselves can be deployed in the runtime application framework. These extensibility features have been exploited to embed Web services in the Web development tool suite, in particular for introducing the Web service operation units, the adapter, the materializer and the dematerializer units. Some comments about the implementation of the units introduced in this paper follow.

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

- including sets of parameters into an XML document;
- extracting sets of parameters from an XML document;
- transforming generic XML documents into other XML documents.

In WebRatio this unit has been implemented by means of XSL [XSL], a language for defining transformations of XML documents[2]. The transformations are specified by means of templates, i.e., pattern-matching rules to be applied to XML elements. While the XML-XML transformation is very general, here we show an example of parameter-XML coupling realized by means of an XSL template. The XML-parameter coupling works analogously, but producing parameters as output.
Consider the Compose adapter unit in Figure 7, exposing the *to*, *subject* and *body* input parameters received from the entry unit. A possible XSL document for this transformation is shown in Figure 16: it contains the three input parameters and a rule (template) matching the entire document and defining a fixed XML markup: only the actual values of the input parameters varies at each call and at run-time are inserted into the fixed markup 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="to"/>
  <xsl:param name="subject"/>
  <xsl:param name="body"/>

  <xsl:template match="/">
      <!-- other elements composing the message body -->
      <to xsi:type="xsd:string"><xsl:value-of select="$to"/></to>
      <subject xsi:type="xsd:string"><xsl:value-of select="$subject"/></subject>
      <body xsi:type="xsd:string"><xsl:value-of select="$body"/></body>
      <!-- other elements composing the message body-->
  </xsl:template>

</xsl:stylesheet>
```

Figure 16 – XSL document for transforming input parameters into a message body.

Dematerializer and materializer units. The dematerializer unit acts as a converter for marshalling 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 a generic canonical XML schema.

The architecture of the Web application generated by WebRatio is shown in Figure 17. The Web front end contains an HTTP server extended with a SOAP listener to support both human users posing regular HTTP requests with a browser, and remote applications sending SOAP messages. 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. Notice that in this architecture all the business objects are implemented as beans, i.e., server-side components. If the request is a SOAP message or an HTTP request for an operation, the Controller sends it to an operation action, which performs the required function and returns a status code to the Controller, who decides what to do next. In particular, the operation action may trigger a business object that interacts with the remote Web service. Further details about the architecture of Web applications can be found in [CFA03].

In this architecture, temporary entities are stored into entity beans in the model part of the MVC architecture: entity beans wrap business data and expose them as persistent objects, allowing also database-related operations. Therefore the implementation of virtual units defined over main memory entities access such data instead of the database. The semantics of updates into main memory is the same of the corresponding updates in relational databases.

[2] Any other transformation language could be used.

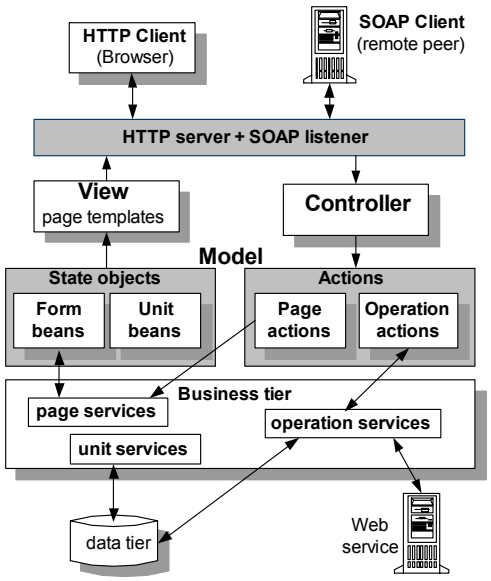

Figure 17 - Architecture of WebRatio supporting Web service composition.

5 Related work

Several platforms and languages have been developed for describing data-intensive Web applications (a survey can be found in [F99]). Recent projects, like WebML [CPB00], W2000 [BGP01], OO-HMETHOD [GCP01] (based on UML interaction diagrams), and OO-HDM [RLS99], allow the conceptual specification of complex Web application. So far, only WebML has been extended with the use of Web services; however, most of the considerations discussed in this paper apply also to the other conceptual models.

Among the commercial tools, also Oracle9i Designer [OracleDesigner], RationalRose [RationalRose], ArcStyler [ArcStyler], CodeCharge [CodeCharge] allow model-driven Web development: however, they do not address the problem of integrating Web services and traditional Web applications at the conceptual level.

On the other hand, several other tools (like, e.g., Novell SilverStream [SilverStream], BEA WebLogic Workshop suite [BEA], Microsoft .NET [NET], Bowstreet [Bowstreet], Avinon [Avinon], WebCollage [WebCollage]) focus on Web services production/development, providing sets of wizards, shortcuts and mechanisms to avoid programming, but do not expose a complete and coherent conceptual model.

The concepts exposed in this paper apply to conceptual models but take into account the characteristics of Web services: they try to combine the need for representing Web application by means of conceptual models with the possibility of automatically generating the code for integrating Web services.

6 Conclusions

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

[ArcStyler] ArctStyler. http://www.arcstyler.com

[Avinon] Avinon. http://www.avinon.com

[BGP01] L. Baresi, F. Garzotto, P.Paolini: From Web Sites to Web Applications: New Issues for Conceptual Modeling. ER Workshops 2000: 89-100.

[Bowstreet] Bowstreet. http://www.bowstreet.com

[CodeCharge] CodeCharge. http://www.codecharge.com

[CFA03] S. Ceri, P.Fraternali, R. Acerbis, A. Bongio, S. Butti, F. Ciapessoni, C. Conserva, R. Elli, C. Greppi, M. Tagliasacchi, G. Toffetti: "Architectural Issues and Solutions in the Development of Data-Intensive Web Applications", CIDR2003, Asilomar, January 2003.

[CFB00] S. Ceri, P. Fraternali, A. Bongio: Web Modeling Language (WebML): a Modeling Language for Designing Web sites. WWW Conference, 2000.

[CF+02] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, M. Matera: Designing Data-Intensive Web Applications, Morgan-Kaufmann, Dec. 2002.

[CS01] F.Casati, M.Shan. Dynamic and adaptive com-position of e-services. Information Systems 26(3), 2001.

[Divine] Divine Inc, Opermarket. http://www.divine.com

[F99] P. Fraternali. "Tools and Approaches for Developing Data-Intensive Web Applications: A Survey". ACM Computing Surveys 31(3):227-263, 1999.

[GCP01] J. Gómez, C. Cachero, O. Pastor: Conceptual Modeling of Device-Independent Web Applications. IEEE MultiMedia 8(2): 26-39 (2001).

[MB+03] 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.

[OASIS] Oasis Web Services for Interactive Applications: http://www.oasis-open.org/committees/wsia/

[OracleDesigner] Oracle 9i Designer. http://otn.oracle.com/products/designer/content.html

[RationalRose] Rational Rose XDE. http://www.rational.com/products/xde/index.jsp

[RSL99] G. Rossi, D. Schwabe, F. Lyardet. Improving Web Information Systems with Navigational Patterns. WWW Conference, 1999.

[SilverStream] SilverStream workbench. http://www.silverstream.com/Website/app/en_US/Workbench

[SOAP], [WSDL] Available at http://www.w3.org/TR

[WebCollage] WebCollage. http://www.webcollage.com

[WebML] The WebML Project: http://www.webml.org.

[WebR] The WebRatio Tool Suite: http://www.webratio.com

[Xaware] XAware's XA-Suite, http://www.xaware.com/

```
<?xml version="1.0"?>
    <ns1:doGoogleSearch xmlns:ns1="urn:GoogleSearch"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
         xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
         xmlns:xsd="http://www.w3.org/1999/XMLSchema">
      <key xsi:type="xsd:string">...</key>
      <q xsi:type="xsd:string"> shrdlu winograd maclisp teletype </q>
      <start xsi:type="xsd:int">0</start>
      <maxResults xsi:type="xsd:int">10</maxResults>
      <filter xsi:type="xsd:boolean">true</filter>
      <restrict xsi:type="xsd:string"></restrict>
      <safeSearch xsi:type="xsd:boolean">false</safeSearch>
      <lr xsi:type="xsd:string"></lr>
      <ie xsi:type="xsd:string">latin1</ie>
      <oe xsi:type="xsd:string">latin1</oe>
</ns1:doGoogleSearch>
```

Figure 18 – Example of input message of the Google Search service.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance" xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:doGoogleSearchResponse xmlns:ns1="urn:GoogleSearch"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="ns1:GoogleSearchResult">
        <documentFiltering xsi:type="xsd:boolean">false</documentFiltering>
        <estimatedTotalResultsCount xsi:type="xsd:int">3</estimatedTotalResultsCount>
        <directoryCategories xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
           xsi:type="ns2:Array" ns2:arrayType="ns1:DirectoryCategory[0]">
        </directoryCategories>
        <searchTime xsi:type="xsd:double">0.194871</searchTime>
        <resultElements xmlns:ns3="http://schemas.xmlsoap.org/soap/encoding/"
                        xsi:type="ns3:Array" ns3:arrayType="ns1:ResultElement[3]">
          <item xsi:type="ns1:ResultElement">
            <cachedSize xsi:type="xsd:string">12k</cachedSize>
            <hostName xsi:type="xsd:string"></hostName>
            <snippet xsi:type="xsd:string">  . . . </snippet>
            <directoryCategory xsi:type="ns1:DirectoryCategory">
              <specialEncoding xsi:type="xsd:string"></specialEncoding>
              <fullViewableName xsi:type="xsd:string"></fullViewableName>
            </directoryCategory>
            <relatedInformationPresent xsi:type="xsd:boolean">
                             true</relatedInformationPresent>
            <directoryTitle xsi:type="xsd:string"></directoryTitle>
            <summary xsi:type="xsd:string"></summary>
            <URL xsi:type="xsd:string">
                 http://hci.stanford.edu/cs147/examples/shrdlu/</URL>
            <title xsi:type="xsd:string">&lt;b&gt;SHRDLU&lt;/b&gt;</title>
          </item>
          <item xsi:type="ns1:ResultElement">
            <cachedSize xsi:type="xsd:string">12k</cachedSize>
            <hostName xsi:type="xsd:string"></hostName>
            <snippet xsi:type="xsd:string">  . . </snippet>
            <directoryCategory xsi:type="ns1:DirectoryCategory">
              <specialEncoding xsi:type="xsd:string"></specialEncoding>
              <fullViewableName xsi:type="xsd:string"></fullViewableName>
            </directoryCategory>
            <relatedInformationPresent xsi:type="xsd:boolean">
                true</relatedInformationPresent>
            <directoryTitle xsi:type="xsd:string"></directoryTitle>
            <summary xsi:type="xsd:string"></summary>
            <URL xsi:type="xsd:string">http://hci.stanford.edu/winograd/shrdlu</URL>
            <title xsi:type="xsd:string">&lt;b&gt;SHRDLU&lt;/b&gt;</title>
          </item>
        </resultElements>
        <endIndex xsi:type="xsd:int">3</endIndex>
        <searchTips xsi:type="xsd:string"></searchTips>
        <searchComments xsi:type="xsd:string"></searchComments>
        <startIndex xsi:type="xsd:int">1</startIndex>
        <estimateIsExact xsi:type="xsd:boolean">true</estimateIsExact>
        <searchQuery xsi:type="xsd:string">shrdlu winograd maclisp teletype</searchQuery>
      </return>
    </ns1:doGoogleSearchResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 19 – Example of output message of the Google Search service.