Web Services Composition by I/O Data Structure Correspondences

Qiushi Wang* and Osamu Shigo**
Graduate School of Information Environment*,
School of Information Environment**
Tokyo Denki University
qiushi_wang@hotmail.com, shigo@sie.dendai.ac.jp

Abstract

The Web service composition from existing web services to obtain the more suitable service for application still is a highly complex task. We propose a method to combine existing web services with list of results as their output into more functional Web service with nested list of results, which seems to be synthesis of the existing web service results, as its output. The method requires no workflow description, like flowchart, but only data correspondences among data structures of existing and combined Web services, as composition specification.

1. Introduction

Recently, SOA (Service Oriented Architecture) [1] that makes a new application by invoking Web services on the internet is paid much attention. In the internet, many Web services are already published, and more private services have been provided for SOA type Web applications. In Web application development, if no single Web service can satisfy the functionality required by the user, the developer should combine existing services together in order to fulfill the request. However, the Web service composition still is a highly complex task for the application developers.

To support such complex task, BPEL [5] is widely applicable in industry to compose Web services. BPEL seems to be helpful, but still too complex for business designer who has no enough knowledge in programming. In this paper, we propose a method to combine the Web services by providing the relation of the input and output data structure among existing and composed Web services. This requires no programming type effort, like to express flow chart, but only input and output data correspondence rules. The system for supporting the method includes acquiring the data structure of existing Web service from WSDL, and automatically generating the synthesized program in C#.

In the followings, section 2 describes the problems on current Web service composition methods, section 3 presents the specification of Web services composition, and how to generate the program from the specification is expressed in section 4. An example of Web service composition from three existing Web services is described in section 5 and the conclusion is in section 6.

2. Problems on current Web service composition methods

For automated Web service composition, several methods were proposed as in [2]. They can be classified into two types, data flow type including Golog[3] and SHOP2 planner[4] and control flow type including BPEL[5] and EFlow[6].

The data flow type method seems to be useful to compose many simple Web services. In the method, an output data of a Web service may be used as an input data of other Web service to connect the services. Thus, the composition rules are provided by output data to input data connections among existing Web services. This is simple enough for the user to describe the Web services composition. However, if the output of Web service has a list of elements, and each element data become the input of other Web service, the data flow method cannot be applicable anymore. Most of existing Web services produces the output as list of results. Therefore it is difficult to compose them by the data flow method.

As the control flow method, BPEL is widely applied in business system developments to compose Web services into more complex service. BPEL provides workflow description language to describe process flow including Web service call, data manipulation, trouble notification and exception handling, with control functions like condition branch,
loop, and parallel processing. The control flow method can be applied to combine those Web services with list output. However, this combination should have relatively complex flow descriptions with iterative control structures, similar to the usual programming.

For Web service composition with list structure, the data flow type is simple but not applicable well and the control flow type is applicable but not simple. In the following sections, a method to compose the Web services with list structure in a simple way similar to the data flow method is presented.

3. Composition specification

The Web service composition is specified by the relations among the data structures of the requested composed Web service and the existing Web services. The data structures may be obtained from WSDL (Web Services Description Language) descriptions. Figure 1 illustrates the outline of our Web service composition processes.

![Fig. 1 Outline of Web service composition processes](image)

The data structure of each existing Web service is obtained from its WSDL description. Using them, input and output data structure relation among the existing Web services and composed Web services is provided. Finally the program for the composed Web service is automatically generated.

3.1. Data structures of Web service

Data structures of Web service consist of input and output data structures, each includes data with data name, or field name, type and occurrence information. Type is either primitive, like integer and string, or non-primitive whose body is also defined in WSDL description. Occurrence shows a number of elements with min and max numbers. Three occurrence patterns, 0-1, 1-1 and 0-unbounded, are appeared in most cases, that mean optional, mandate and iterated elements, respectively. Figure 2 shows an example of WSDL description. The data structure diagram is generated from WSDL description, as shown in Fig. 3.

![Fig. 2 WSDL description](image)

![Fig. 3 Data structure diagram](image)

3.2. Web service composition specification

After defining existing Web service data structures, the Web service composition is specified through data structure correspondences.
3.2.1. Whole composition diagram
The Web service composition diagram, as shown in Fig. 4, is specified. The existing Web service diagrams are shown inside the composed service, and input and output data structures are outside the service. The input correspondence provides that each input of existing Web service is obtained from which input of composed service or other existing Web service output. The output correspondence provides that each output of composed Web service is obtained from which output of existing Web services. These correspondence relations permit the system to automatically generate the program.

3.2.2. The input correspondence.
The input correspondence specifies that from which data the input of existing Web service comes. The data of existing Web service input may be constant, input of composed service, output of other existing Web service, or calculation of these data. Figure 5 illustrates the input correspondence example. In the figure, boxes between the composed service input and existing service inputs specifies the correspondences. Here, "A>B" denotes that B, an existing Web service input, is obtained by A. During the input correspondence design, the necessary input of composed Web service may be defined.

3.2.3. The output correspondence.
The output data structure of the composed Web service is derived by the synthesis of the existing Web service output data structures. First of all, among the existing Web services, a main Web service that does not use the output of other existing Web service as its input is chosen. In Fig. 4, WebServiceA becomes main Web service. The output data of other existing Web service is added into the output data structure of the main Web service to synthesize the data structure. Repeat the synthesis process as required, and finally delete unnecessary data fields and add necessary new data field that does not exist in the existing Web services. The hierarchical data structure is obtained through the above synthesis process, as shown in Fig. 6.

The top level of the hierarchy indicates the class name that the composed service returns. For each output field of the hierarchy, the data source should be specified after "<" mark, as the figure 6 shown, which may be constant, like service name, or a part of existing Web services.

It should be notified that above input and output correspondences among Web service data structures.
4. Program generation

The target composed Web service program is automatically generated by using the data correspondences. The program includes classes appeared in the data structure of composed Web service and the class with public static method for composed Web service execution. The programs to access the existing Web services may be obtained from their WSDL descriptions.

4.1. Classes for the data structure of the composed service

The class definitions with only properties in C# are generated from the data structure Description of the composed Web service. In Fig. 3, three classes, NClass, TAB and TB, exist as types in the output data structure. NClass has two fields, N1 of string and A2 of TAB, TAB has five fields, four primitive type fields and field of TB, and TB has five primitive fields.

4.2. Class for composed Web service

The composed program is generated along the output structure specification of the composed service. At beginning, the class header is generated. Then the method header is generated for the top element of the output data structure. After that, program codes for each data element in the output are generated.

4.2.1. Skeleton of the class The composed Web service class and method skeleton is generated with class name and method name appeared in composed Web service box in Fig. 4. The skeleton of the class becomes as follows.

```csharp
public class ClassName
{
    public static OutputDataClass MethodName(InputParameters) {
        OutputDataClass x0 = new OutputDataClass();
        - - -
        return x0;
    }
}
```

The above program is generated by getting the composed input and the top level of the output structure. Before proceeding to process the rest of output, the level number which is used for subscription of variable names should be zero as an initial value.

4.2.2. Primitive field process If the output data field is primitive, then simple assignment code is generated. If the data source is constant,

```csharp
xi.FieldName = constant;
```

is generated, where i is level number.

For example, in Fig. 4, N1's data source is constant "ABWebService", and level number is 0,

```csharp
x0.N1 = "ABWebService";
```

is generated for N1 data field.

If the data source is from existing Web service,

```csharp
xi.FieldName = yi.DataSourceFieldName;
```

is generated, where yi is data source variable name defined for non primitive data process, described later.

4.2.3. Non-primitive field process If the data field is non primitive, its data source should be a part of the existing Web service output. Thus the generated program code should include the existing Web service call at the beginning of iteration process. At the beginning of non-primitive process, level number is increment by 1, and at the end, it is decrement by 1.
(1) Web service call

The existing Web service call is generated when the data source is Web service output. This is done when the new existing Web service output is appeared. For example in Fig. 4, WebServiceA call is generated at "A2 TAB *" field. The input correspondence for the Web service, described in 3.2.2, is used to generate the parameter of the call. For example in Fig. 4, WebServiceA call becomes

(new WebSV1()).MethodX(P1, P2, P3)

and WebServiceB call becomes

(new WebSV2()).MethodY(P4 - y1.A4, y1.A5, P5)

where WebServiceA.A2 is replaced by y1 which is the loop variable of the WebServiceA.A2 list iteration process described in the next.

(2) Iteration process

If the non primitive field denotes iteration mark ":", the iteration codes are generated, as follows;

List<OutputClass> zi = new List<OutputClass>();
foreach( InputClass yi in WebServiceCall . part )
{
  OutputClass xi = new OutputClass();
  - - -
  zi.add( xi );
}  
xi-1.fieldName = zi;

In the above loop body, assignment like "xi.field = yi.field;" as described in 4.2.2 is appeared.

For example in Fig. 4, "A2 TAB * < WebServiceA. A2" part generates following codes;

List<TAB> z1 = new List<TAB>();
foreach( TA y1 in (new WebSV1()).MethodX(P1, P2, P3) . A2 )
{
  TAB x1 = new TAB();
  - - -
  z1.add(x1);
}  
x0.A2 = z1;

(3) No iteration process

If non primitive is not iteration field, no List<> is used. The generated codes are,
transformation type. The target composed Web service combines these services and outputs a list of hotel information with restaurants near the hotel with possible routes from the hotel to the restaurant.

5.1. Input correspondences

For each input of original Web services, input correspondences should be given. The hotel search service has three inputs, location, roomType and number, which are obtained from hotelLoc, roomType and number of the composed service input, respectively. The restaurant search service has three inputs, location, foodType and aveBudget, which are obtained from restLoc, foodType and “budget – HotelWS.Hotels.Fare,” respectively, where restLoc, foodType and budget are in the composed service input and HotelWS.Hotels.Fare is from the Hotels.Fare, the output of hotel search service HotelWS. The input of restaurant aveBudget shows an example of calculation usage in input correspondence. The route search service has three inputs, depart, destine and searchWay, which are obtained from HotelWS.Hotels.Station of hotel search service output, RestWS.Rests.Station of restaurant search output and searchWay of the composed service input. Table 1 shows the above input correspondences.

Table 1. Input correspondences

<table>
<thead>
<tr>
<th>ServiceName</th>
<th>DataClass</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HotelWS</td>
<td>Hotels</td>
<td>HotelWS.Hotels.Station</td>
</tr>
<tr>
<td>RestaurantWS</td>
<td>Rests</td>
<td>RestWS.Rests.Station</td>
</tr>
<tr>
<td>RouteWS</td>
<td>Routes</td>
<td>RouteWS.Routes</td>
</tr>
</tbody>
</table>

5.2. Output correspondences

The output correspondence specifies that each output of the composed service comes from constant or the original Web service output. Table 2 shows those output correspondences. The top level name HRSInfo is a class name used as return type of the composed service. ServiceName field is set by constant "HRSWebService", and SearchTime field is set by DateTime.Now, provided as standard function in C#. HotelsRests field, which is non primitive, is obtained from HotelWS.Hotels that is the Hotels part of the output of hotel search service HotelWS. This requests the HotelWS Web service call. For each field of HotelsRests, its corresponding field of Hotels from HotelWS is specified as data source. Then RestsRoutes field of Hotels, which is also non primitive, is obtained from the Rests part of the output of restaurant search Web service RestWS, which request the RestWS Web service call.

Table 2. Output correspondences

<table>
<thead>
<tr>
<th>ServiceName</th>
<th>DataClass</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cHotelRestInfo</td>
<td>HotelWS.Hotels.Station</td>
</tr>
<tr>
<td></td>
<td>cRestRouteInfo</td>
<td>RestWS.Rests.Station</td>
</tr>
<tr>
<td></td>
<td>cRouteInfo</td>
<td>RouteWS.Routes</td>
</tr>
</tbody>
</table>

5.3. Composition diagram

The whole composition diagram, including the original Web services with input and output data, composed Web service input data with input correspondences and composed Web service output data structure with output correspondences, is illustrated in Fig. 8.

5.4. Generated program

The program generated from Fig. 8 contains four data class, HRSInfo, cHotelRestInfo, cRestRouteInfo and cRouteInfo, and one service class HRSWS. HRSWS contains a method HRSSearch with return type HRSInfo and input parameters appeared as whole input. The method includes three nested foreach loops for Hotels of HotelWS, Rests of RestWS and Routes of RouteWS.

Figure 9 show a part of program codes generated from the composition specification in Fig.8 by the above processes.
6. Conclusion

A Web service composition method from existing web services each of which provides a list of result as its output is presented. Here, each data in the list generated by a Web service becomes the input of other Web service to combine them. This requires a loop process to invoke the second Web service. BPEL can be applicable to describe such Web service composition by workflow description with loop process, which seems to be similar to programming, and highly complex task. Our method does not require the workflow type description, instead the correspondence rules among data structure of existing and synthesized Web services is provided to specify the composition. The example in the paper, combining hotel search, restaurant search and route search services to generate the synthesized data, seems to illustrate the feasibility of the method.

The automated system of the method is under development, which contains the data structure diagram generator using the WSDL description of existing Web services, data structure diagram editor to synthesizing composed service output data diagram from existing ones and drawing arrows between the data elements to specify the data correspondences, and program generator to generate programs in C# and other necessary configuration files.

As our future work, we need to evaluate the feasibility and effectiveness of the method and the system by applying them into other relatively complex Web service compositions.
public class HRSWS
{
    public static HotelInfo HRSSearch
    (string hotelLoc, string roomType, int Number,
     string foodType, int Budget, string searchWay)
    {
        HotelInfo x0 = new HotelInfo();
        x0.ServiceName = "HRSWebService";
        x0.SearchTime = DateTime.Now;
        List<cHotelRestInfo> z1 = new List<cHotelRestInfo>();
        foreach (HotelInfo y1 in
            (new HotelWS()).HotelSearch
            (hotelLoc, roomType, number).Hotels)
        {
            cHotelRestInfo x1 = new cHotelRestInfo();
            x1.Name = y1.Name;
            x1.State = y1.State;
            x1.Address = y1.Address;
            x1.Station = y1.Station;
            x1.RoomType = y1.RoomType;
            x1.Number = y1.Number;
            x1.Fare = y1.Fare;
            List<cRestRouteInfo> z2 = new List<cRestRouteInfo>();
            foreach (RestInfo y2 in
                (new RestWS()).RestSearch
                (restLoc, foodType, Budget - x1.Fare).Rests)
            {
                cHotelRestInfo x2 = new cHotelRestInfo();
                x2.Name = y2.Name;
                z2.add(x2);
            }
            x1.Restaurants = z2;
            z1.add(x1);
        }
        x0.Hotels = z1;
        return x0;
    }
}

Fig.9 A part of generated program codes of the sample

7. References


