# Automatic Web Service Composition Based on Service Interface Description

**Jing-zhou Zhang**
College of Computer Science and Technology, Fudan University, Shanghai, China

**Shou-jian Yu**
College of Computer Science and Technology, University of Donghua, Shanghai, China

**Xiao-kun Ge**
Shanghai Development Center of Computer Software Technology, Shanghai, China

**Guo-wen Wu**
College of Computer Science and Technology, University of Donghua, Shanghai, China

**Abstract -** *Composition of Web services has received much interest to support business-to-business or enterprise application integration. In our work, we build the composition method based on service interface description. Compared to previous work, our approach for composition only uses the information that is already available in service interface definitions. It does not require service providers describe their interfaces with semantic markup. We have proposed datatype matching and service composition algorithm. We use the measure of linguistic similarity between two datatypes. The similarity score calculated by our algorithm allows for flexible Web service composition. Users can define different score threshold for either relax or accurate service composition. We have conducted experiments to evaluate the effectiveness of our datatype matching algorithm. The results demonstrate that the matching algorithm has good precision and recall.*

**Keywords:** Web service composition, Interface description, Interface matching, WSDL

## 1 Introduction

The emergence of Web services has led to more research into Web services composition. Service composition enables the ability to take existing services and combine them to form new services. Applications can be assembled from a set of appropriate Web services and no longer be written manually. Web service composition can be either static or dynamic. Dynamic composition is more suitable if the process has to dynamically adapt to unpredictable changes in the environment. Dynamic composition may involve run-time searching of registries to find services.

On the one side, the business world has developed a number of XML-based standards to formalize the specification of Web services, their flow composition and execution. On the other side, the Semantic Web community focuses on reasoning about web resources by explicitly declaring their preconditions and effects with terms precisely defined in ontologies. Our approach does not require that service providers describe their interfaces using semantic markup. Instead we attempt to perform composition based solely on the information that is already available in the service descriptions. We assume that semantic information is already implicitly captured in the WSDL (Web Service Description Language) interface specification. One of the possible composition mechanisms is to combine multiple services, such that the execution result of the former service is passed to the next service and accepted as its input, or combination of results from several services is passed to the next service and accepted as its input. This requires the ability to discover or detect pairs of services such that the output of one service is equal or equivalent to the input of another. This can be implemented by matching an input with an output interface definition. Once such pairs are identified, a composite service can be easily constructed.

The rest of this paper is structured as follows. Section 2 briefly lists the related works. Section 3 presents a detailed explanation for Web service interface matching. In section 4, we discuss the Web service composition algorithm based on interface matching. In section 5, we conduct a preliminary experiment on interface matching algorithm evaluation. Section 6 concludes this paper.

## 2 Related Work

Web services composition can be built statically or dynamically. [1] gives the first thorough survey of patterns for composition. On the one side, the business world has developed a number of XML-based composition languages, such as BPEL (Business Process Execution Language) [2], WSCL (Web Services Conversation Language) [3]. These languages only support static composition. On the other hand, for dynamic composition, the Semantic Web community uses semantic annotations to define preconditions and effects of services, e.g. in DAML-S [4]. In literature [5, 6], semantic composition is also discussed. But commonly, there aren't universal ontologies that all

This work has been partially supported by the Key Programs of Science and Technology Commission Foundation of Shanghai, China under contract 05DZ11C06, and the project: Visual Document Management System for Compoment Development.

partners will prefer to refer to, and partners will refer to different ontologies. This kind of composition is still in its infancy and impractical in real world.

eFlow [7] is a platform for specification, enactment, and management of composite e-services. A composite service is modeled by a graph, which defines the execution order of the nodes in the process. The *generic service node* in eFlow model supports dynamic process definitions for composite services. But the requestor must be familiar with the configuration parameters in it to promise the dynamic characteristics. SWORD Project developed by Stanford University also models services composition as a planning problem [8]. In this approach, each service is modeled as an action. Conjunction of all condition inputs is modeled as the precondition for the action and outputs for postcondition. State transitions are defined based on preconditions of actions and a transitions leads to new states. In [9], an AI planning system (SHOP2) is used with DAML-S Web service descriptions to automatically compose Web services. But in fact, the condition input and condition output are difficult to be determined comprehensively during service designing [10]. In [11], SCET (Service Composition and Execution Tool) allows for composing services statically using its designer and storing them as Web Service Flow Language (WSFL) based specifications. The problems of dynamic composition still remain.

# 3 Web Service Interface Matching

Web service has emerged as an important mechanism for developing distributed applications in the dynamic e-business environment. Much functionality can be contained in one Web service, and each is implemented by an operation. A Web service can be expressed as a set of operations. An operation is specified by its name, its input and output message types, i.e. o: =<name, $t_{in}$, $t_{out}$>, which is the interface of Web service. Our service composition method is aimed at enabling composition in a more flexible and automatic manner, utilizing the information already provided by WSDL document. This relies on the automatic matching of inputs/outputs of operations in Web services, i.e. interface matching. First we will describe the structure of WSDL. Then we will discuss the interface matching procedure in this section.

## 3.1 Structure of WSDL

WSDL is the current standard of Web service description. The syntax of WSDL is defined in XML Schema. A WSDL document describes the location of a Web service, its available operations and their associated messages and data types as well as the format of their result values. The XML Schema types of message parameters may be defined using a <types> element. A <message> element is needed to compose such data types into messages. Messages need to be grouped into operations, which may define an <input>, an <output> and a <fault> message. The simplified structure of WSDL is shown in Figure 1.

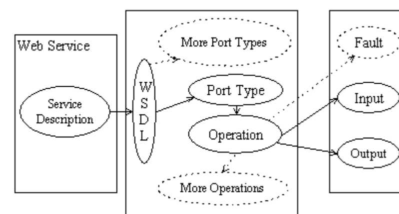



Figure 1. Simplified WSDL structure

## 3.2 Operation's Data Type Matching

Web service interface matching involves the comparison of operations in Web service, which is based on the comparison of operation's input and output messages. This is in fact the comparison of the data types of the object communicated by these messages. The degree to how two messages are similar is decided on the basis of how similar their parameter lists are, in terms of the data types they contain and their organization. The data types of Web services specified in WSDL are XML elements. Types in XML schema can be primitive, simple or complex. Primitive are built-in ready to use objects, like numbers, logical values (boolean) or characters. Simple and complex data types are objects that anyone can derive from the built-in ones using methods provided by the schema language. In this section, we will first introduce a Web service interface description example. Then we will discuss the simple data type matching and complex data type matching.

### 3.2.1 Web Service Interface Description Example

For Web service interface matching illustration, we introduce two specific WSDL documents: one for the source service description and the other for target service description, as shown in Figure 3 (a) and (b). The source Web service contains one operation, *getData* and the target Web service contains one operation, *getProduct*. In WSDL specification (see Figure 3), all operations are grouped under the <portType> tag, with each unique operation specified under an <operation> tag. In Figure 3, operations are highlighted in boxes with bold, dashed lines. Operations' request and response messages are specified as input and output messages under <message> tags (highlighted in boxes with bold, solid lines). Data types are grouped under the <types> tag. Complex data types are defined under a <complexType> tag. Data type definitions are highlighted using solid line boxes. *Item* in Figure 3 (a) is a complex data type defined for the source Web service and it consists of two simple data types i.e. *product* (string) and *quantity* (int).

#### 3.2.2 Simple Data Type Matching

A simple data type is associated with element name and value type attributes. The element name describes the actual semantic of the element. The value type should be one of the primitive data types. Thus the simple data type matching can be done from element name and its value type aspects.

We use the measure of the linguistic similarity between two element names. Various name and string matching algorithms like NGram, semantic matching, abbreviation expansion etc are used for element name matching. The *NGram* algorithm calculates the similarity by considering the number of qgrams that the names have in common [12]. Experiments testify that 3-grams excels 2-grams in our work and 3-grams is used. The *SemanticMatch* algorithm uses WordNet [13] to find semantic matching between two element names. A custom abbreviation dictionary is used for the *CheckAbbreviations* algorithm. If any of these algorithms return a full match i.e. 1 on scale of 0 to 1, then a match score of 1 for linguistic similarity is returned. If all the match algorithms give a match value of zero, then the linguistic similarity score is 0. If on the other hand, none of the match algorithms give a match score of 1 i.e. an exact match, then the average of all non-zero match scores is taken. Equation 1 explains all these cases.

$$\mathit{ElementName\ Match} = \begin{cases} 1 & \text{if } (ms_1 \cup ms_2 \cup ms_3 = 1) \\ ms_2 & \text{if } ((0 < ms_2 < 1) \cap (ms_1 = ms_3 = 0)) \\ 0 & \text{if } (ms_1 = ms_2 = ms_3 = 0) \\ (ms_1 + ms_2 + ms_3)/3 & \text{if } (ms_1 \neq 1, ms_2 \neq 1, ms_3 \neq 1) \end{cases} \quad \mathbf{(1)}$$

Where $ms_1$=MatchScore (NGram), $ms_2$=MatchScore (SemanticMatching), $ms_3$=MatchScore (Abbreviation)

The *SemanticMatch* algorithm uses WordNet [13] to find semantic matching between two element names based on the semantic similarity. Figure 2 lists the algorithm.

```
1  double SemanticMatch (term1, term2) {
2  maxScore = 1;
3  if (term1 is identical to term2)
4          score = maxScore;
5  else if (term1 and term2 are synonymous)
6          score = 0.8* maxScore;
7  else if (term1 and term2 have hierarchical relations)
8          score = (0.6* maxScore) / N;
           // N is the number of hierarchical links;
9  else score =0;
10 return score;}
```

Figure 2. Semantic matching algorithm based on WordNet

As mentioned before, for each element, there is a type for its value. The value type should be one of the primitive data types. In our approach, we make modest use of value type. Two primitive data types can be compatible, semi-compatible, or incompatible. Examples of compatible data types include identical data types and data types that can be adapted to each other at little cost such as long and float, etc. Examples of semi-compatible data types are types that can be adapted to each other at some cost such as int and float, etc. The idea of matching data types with different compatible levels is to allow the ideas of relaxed matching. The equation 2 illustrates its use. If two element names return a zero match, their value type matching score should be zero also.

$$\mathit{ValueType\ Match} = \begin{cases} 0 & \text{if ElementNameMatch} = 0 \\ 1 & \text{if two datatypes have compatible value type} \\ 0.8 & \text{if two datatypes have semi-compatible value type} \\ 0 & \text{if two types are incompatible} \end{cases} \quad \mathbf{(2)}$$

The matching score of two simple data types is calculated by combination of *ElementNameMatch* and *ValueTypeMatch*: $S_{simpleDatatype} = w \times S_{ElementNameMatch} + (1-w) \times S_{valueTypeMatch}$, where the constant *w* is in the range of 0 to 1. In our work, *w* is set as 0.8. The optimized value for *w* can be determined experimentally by constantly changing its value to get the best precision and recall.

#### 3.2.3 Complex Datatype Matching

Complex data types are user-defined data types that contain child elements or attributes. It uses different group style to group the sub elements and attributes. We match complex data types from these two aspects. In our approach, we treat attribute as element for simplicity. Because one of the data type is complex, we need to match all sub elements in it to find corresponding data types pairs with the maximal sum matching score. Considering the example in Sect. 3.2.1, in order to calculate the matching score between two complex data types: *Item* and *ProductParts*, we need to construct a 2⊗2 matrix, as shown in Table 1. Each cell in the matrix is filled with a value that indicates the matching score between the two simple data types corresponding to the row and the column of the cell. From matching result in Table 1, we can see that *amount* in *productParts* should match with *quantity* in *Item* with matching score of 0.8, and *productName* should match with *product* with matching score of 0.6. If a complex data type also includes another complex data type, the matching process should be proceeded recursively.

Table 1. Complex data types matching example

| ProductParts / Item | productName: str | amount: int |
|---|---|---|
| quantity: int | 0 | 0.8 |
| product: str | 0.6 | 0 |

In order to be with the same measure of matching score as simple data types, the sub elements matching score of complex data types is computed as the average score of its each sub simple data types, as shown in Equation 3.

$$subDatatypeMatch = \frac{\sum_{i=1}^{n} S(subSimpleDatatype)}{n} \qquad (3)$$

Where n is the number of sub simple data types in the source complex data type

There are three styles for grouping elements and attributes together as follows. If two data types have the same internal grouping style, their matching score is increased by a bonus score: $S_{groupStyle}$.

- sequence: must contain the elements listed in the exact order
- all: must contain the elements listed in any order
- choice: must contain one of the elements listed

The matching score of two complex datatypes is calculated by: $S_{complexDatatype} = w \times S_{subDatatypeMatch} + (1\text{-}w) \times S_{groupStyle}$, where the constant $w$ is in the range of 0 to 1. In our work, $w$ is set as 0.8.

Considering the example in Figure 3, there are two sub elements in complex data types *Item*. Both *Item* and *ProductParts* have the same group style: all. According to Equation 3, the matching score between *Item* and *ProductParts* should be calculated as follows.

$$M = 0.8 \times \frac{(0.8 + 0.6)}{2} + (1 - 0.8) \times 1.0 = 0.76$$

```
<definitions> <types> <schema .... >
<element name="id" type="string"/>
<element name="name" type="string"/>
<element name="items">
   <complexType>
      <all>
         <element name="item" type="tns:Item"
         minOccurs="0" maxOccurs="unbounded"/>
      </all>
   </complexType>
</element>
<complexType name="Item">
<all>
   <element name="quantity" type="int"/>
   <element name="product" type="string"/>
</all>
</complexType>
</schema> </types>
<message name="getDataRequest">
   <part name="id" type="string"/>
</message>
<message name="getDataResponse">
   <part name="id" type="id"/>
   <part name="name" type="name"/>
   <part name="items" type="items"/>
</message>
<portType name="Data_PortType">
   <operation name="getData">
      <input message="tns:getDataRequest"/>
      <output message="getDataResponse"/>
   </operation>
</portType> </definitions>
```

Figure 3. (a) WSDL document of source Web service

```
<definitions> <types> <schema .... >
<element name="id" type="int"/>
<element name="name" type="string"/>
<element name="price" type="float"/>
<element name="part" type="productParts"/>
<complexType name="productParts">
<all>
   <element name="productName" type="string"/>
   <element name="amount" type="int"/>
</all>
</complexType>
</schema> </types>
<message name="getProductRequest">
   <part name="id" type="int"/>
</message>
<message name="getProductResponse">
   <part name="id" type="id"/>
   <part name="name" type="name"/>
   <part name="price" type="price"/>
   <part name="part" type="part"/>
</message>
<portType name="Product_PortType">
   <operation name="getProduct">
      <input message="getProductRequest"/>
      <output message="getProductResponse"/>
   </operation>
</portType>  </definitions>
```

Figure 3. (b) WSDL document of target Web service

### 3.3 Web Service Interface Matching

A set of input and output definitions describes the message formats that a Web service can accept or generate. After evaluating the data type matching scores, the service interface matching can be proceeded by comparison source service messages against the target service message. Several data types can be enclosed in a message. For message *getDataResponse* in Figure 3 (a), there are three parts in it, i.e. *id*, *name* and *items*. The objective of interface matching is to identify the parameter correspondence that maximizes the sum of their individual data type matching score. Figure 4 illustrates the algorithm.

Now we will discuss the algorithm *matchInterface*. We use a matching scores matrix to store matching score of all pair-wise combinations of source and target data types. This procedure identifies all possible matches between two lists of data types and calculates the overall matching score matrix between these two data type lists.

As can been seen in Figure 4, the algorithm takes as input two lists of data types: *sourceList*, which contains *m* data types, and *targetList*, which contains *n* data types. Using these two lists, it constructs a m⊗n matrix, whose rows correspond to the source data types and columns correspond to the target data types. Each cell in the matrix is eventually filled with a value that indicates the matching score between the two data types corresponding to the row and the column of the cell. For each two data type from the source list and the target list, if they are both simple data types, the procedure *matchSimpleTypes* uses the algorithm in Sect. 3.2.2 to compute their match scores and the score is stored in the corresponding cell of the matrix. If one (or both) of the data types being compared is (are) complex, then the procedure *getCompositeDataElements* collects all sub elements of the complex data structure(s) to form new data type list(s) to be further matched recursively. After the matrix is filled, the algorithm forms all possible matches between the two lists represented by the matrix and returns the highest matching score between two lists of data types.

```
1 int matchInterface (sourceList(m), targetList(n)){
2 matrix = construct a m⊗n matrix;
3 //exhaustive matching
4 for (int i=0; i<m; i++) {
5     for (int j=0; j<n; j++) {
6         sourceType = sourceList(i)
7         targetType = targetList(j)
8         if (both sourceType and targetType are primitive)
9             matrix[i, j] =
                  matchSimpleTypes(sourceType,
  targetType);
10        else if (either sourceType or targetType is complex)
  {
11            newSourceList =
                  getCompositeDataElements(sourceType);
12            newTargetList =
                  getCompositeDataElements(targetType);
13            matrix[i, j] = matchInterface
                 (newBaseList, newTargetList)
                      +Bonus(sourceType, targetType); } } }
14 return the matches with the maximum score;}
```

Figure 4. Web service interface matching algorithm

We use the example in Sect. 3.2.1 to illustrate the interface matching procedure. We match output message *getDataResponse* of operation *getData* in Figure 3 (a) with output message *getProductResponse* of operation *getProduct* in Figure 3 (b). There are three data types in *getDataResponse* message and four data types in *getProductResponse* message. Thus we construct a 3⊗4 matrix as shown in Table 2. Each cell in the matrix is filled with the matching score between the two data types corresponding to the row and the column of the cell. The matching score between complex data types *items* and *part* is 0.76 as calculated in Sect. 3.2.3. We assume that the matching result is shown as Table 2. The last work we have to do is to find corresponding data type for each data type in source interface. Each data type in source interface can only have one correspondence in the target interface, and this is the same with data type in target interface. Our objective is to get the maximal sum matching score for source service interface. This can be done by a nested loop accessing to each cell in the matrix. In this example, the pair-wise correspondence is obvious from results in Table 2. The sum matching score is 0.8 + 1.0 + 0.76 = 2.56 between messages *getDataResponse* and *GetProductResponse*.

Table 2. Web service interface matching example

| getProductResponse / getDataResponse | id: int | name: str | price: float | part: productParts |
|---|---|---|---|---|
| id: str | 0.8 | 0 | 0 | 0 |
| name: str | 0 | 1.0 | 0 | 0.5 |
| items: item | 0 | 0 | 0 | 0.76 |

# 4 Web Service Composition

A service is composed of several operations. The input/output of each operation tells us what type of parameter needs to be provided in order to execute it, as well as the types of output that will be returned. This gives us information on possible compositions of services. For example, if a particular service has an operation *buyBook*, which takes as input an *isdnCode*, and another operation *getISDN* outputs values of the same type, then we know that the latter operation is a composition of the former operation. This process can be implemented automatically and recursively.

In order to express service requestor' need, the service requestor may provide some local information. The *inputList* represents the local information that the requestor provides. *goal* represents the desirous output that the requestor wants. For simplicity, the operations candidate (*canOperList*) of services is represented by two sets of parameters: *canInList* represents the parameter lists of input messages in operations, and *canOutList* represents the parameter lists of output messages in operations. We assume that there are *k* candidate operations.

For illustration, we give a definition as follows. The match score threshold can be defined by the user.

**Definition.** Let $DT_1$ ($O_1$, $O_1$, …, $O_n$) and $DT_2$ ($S_1$, $S_2$, …, $S_m$) represent two sets of data types. If $\forall$ $O_i \in DT_1$, $\exists$ $S_j \in DT_2$, that match score of ($O_i$, $S_j$) ≥ threshold, then we call $DT_1$ **satisfies** $DT_2$.

```
1  comSerList (x) serCompose (inputList, goal,
    canInList (k), canOutList (k)){
2  for (int i= 0; i <k; i++) {
3     outputMatchScore(i) = matchInterface (goal,
       canOutList (i))}
     //reorders the canOutList with descending matching
   score and returns DescOutList
4     DescOutList = descendSort (canOutList)
5     for (i= 0; i <l; i++){ //l≤k, some messages are
          omitted with a matching score threshold
6        {if (DescOutList (i) satisfies goal)
7           if (inputList satisfies canInList (i))
8              comSerList.add (DescOutList (i))
9           else//get the unmatched parts
10 unMatched = getLeftParts(canInList (i), inputList)
11          comSerList.add serCompose (inputList,
                unMatched, canInList (k), canOutList (k))
12       else
13 unMatched = getLeftParts(goal, DescOutList (i))
14 comSerList.add serCompose (inputList,
          unMatched, canInList (k), canOutList (k)) } } }
```

Figure 5. Web service composition algorithm

Now we discuss the algorithm *serCompose* in Figure 5. First, the algorithm takes the *goal* as input to be achieved and calculates the interface matching score between the *goal* and each candidate output message (*canOutList*) (line 3), using the interface matching algorithm described in Sect. 3. The output message lists are reordered with descending matching score, which is *DescOutList* (line 4). (step 1)

Second, the algorithm checks each output message if it *satisfies* the *goal* (line 6). If not, this means that individual operation is not enough for the *goal* and composition is necessary. Then the algorithm searches the outputs of other operations for the left unmatched part of the goal (line 13, 14). This is implemented by the procedure calling itself recursively. The times of recursive calling can be set by user. (step 2)

Third, if the output message satisfies the goal, whether the input data types list *satisfies* that output message' input part should also be checked (line 7). If not, the procedure *getLeftParts* gets the unmatched data types in that input message to form new data types to be further composed recursively (line 10, 11). This procedure proceeds recursively the same as in step 2. (step 3)

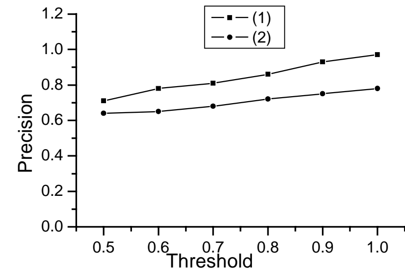
After the matching between the goal and each candidate output message in step 1, some output message with small matching score can be omitted (line 5). User can use a matching score threshold to filter some irrelevant messages, and the system burden is alleviated. Also user can use different threshold for flexible service composition. With a big threshold, the composition exactness is relatively higher than with a small threshold. If one of the output message *satisfies* the *goal* and the input data types list *satisfies* that message' input part, that operation for this message is added to the composite services list (*comSerList*) (line 8). In the end, the algorithm returns the composite services list *comSerList*.

# 5 Implementation and Experiments

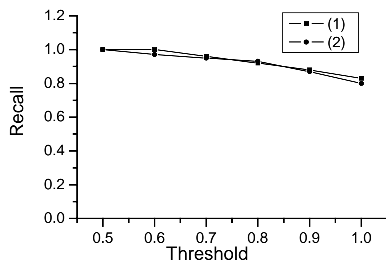
We have implemented an initial prototype to illustrate the composition method we have proposed. WordNet 2.0, an on-line lexical database for the English language, is embedded into the system through APIs for semantics interpretation of data types matching.

We evaluate our composition method by evaluating the interface matching algorithm. We found a collection of Web services from SALCentral.org and XMethods.com. We extract 394 pairs of input messages and output messages from their WSDL service descriptions. To evaluate the quality of the matching algorithm, we compare the manually determined real matches(R) for those messages with the matches P returned by our matching algorithm. We determine the correctly identified matches as I. The following quality measures are computed:

- $Precision = \frac{|I|}{|P|}$ estimates the reliability of the match predictions;
- $Recall = \frac{|I|}{|R|}$ specifies the share of real matches that is found;




(a) Precision




(b) Recall

Figure 6. Web service interface matching algorithm Evaluation

We vary the threshold form 0.5 to 1. We compare the matching results in two circumstances: (1) element name, value type and group style are all used for data types matching; (2) only element name is used for data types matching. The experiments results are shown in Fig. 4. We can see that value type and group style used for matching can greatly improve the matching precision about 15%. It has no effect on matching recall. The threshold of 0.86, where the precision and recall interconnect, should be used for service composition. With the threshold of 0.86, it has the best results for both precision and recall above 90%.

# 6 Conclusions and Future Work

In this paper, we have proposed a Web service composition model based on interface matching. The intuition behind our work is that the semantic information is already implicitly captured in the current WSDL interface specification. We have proposed data type match algorithm for both simple types and complex types. The interface matching score calculated by our algorithm allows for flexible Web service composition. Users can use different score threshold for both relaxed and accurate service composition. Experiments conducted to evaluate the effectiveness of our data type matching algorithm shows that value type and group style used for matching can greatly improve the matching precision.

While good results are obtained using our method, there is room for improvement. First, the weight value used in simple data type and complex data type matching can be determined experimentally by constantly changing its value to get the best precision and recall. Second, by interface matching, only a sequential composite service can be constructed. We will develop workflow techniques facilitated complex composition in the future work.

# 7 References


[1] B. Benatallah, M. Dumas, M.-C. Fauvet, and F. Rabhi, "Towards Patterns of Web Services Composition. Patterns and Skeletons for Parallel and Distributed Computing", Springer Verlag, UK, November 2002.

[2] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana, "Business Process Execution Language for Web Services, Version 1.0", http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, July 2002.

[3] Arindam Banerji, Claudio Bartolini, "Web Services Conversation Language (WSCL) 1.0", Hewlett-Packard Company, http://www.w3.org/TR/wscl10/, March 2002.

[4] DAML-S Coalition, "DAML-S: Web Service Description for the Semantic Web", Proceedings of the 1st International Semantic Web Conference (ISWC), 2002.

[5] E. Sirin, J. Hendler, B. Parsia, "Semi-automatic composition of Web Services using Semantic Descriptions", Web Services: Modeling Architecture and Infrastructre workshop in conjunction with ICEIS2003, 2003.

[6] N. Milanovic, V.Stantchev, J. Richling, M. Malek, "Towards Adaptive and Composable Services", Proceedings of IPSI2003, 2003.

[7] Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., and Shan, "Adaptive and Dynamic Service Composition in eFlow", Proceedings of the International Conference on Advanced Information Systems Engineering, 2000.

[8] Shankar, P. and Fox, A, "SWORD: A Developer Toolkit for Web Service Composition", Proceedings of the Eleventh International World Wide Web Conference, 2002.

[9] D. Wu, E. Sirin, J. Hendler, D. Nau & B. Parsia, "Automatic Web Services Composition Using SHOP2", International Conference on Automated Planning & Scheduling, ICAPS 2003.

[10] Carman, M., and Serafini, L, "Planning for Web services the hard way", Workshop on Service Oriented Computing, International Symposium on Applications and the Internet (SAINT-2003), 2003.

[11] Senthilanand Chandrasekaran, "Composition, Performance Analysis and Simulation of Web Services", Diploma Dissertation, University of Georgia, 2002.

[12] R. C. Angell, G. E. Freund, "Automatic spelling correction using a trigram similarity measure", Information Processing and Management 19(4), 255-161, 1983.

[13] WordNet 2.0, http://www.cogsci.princeton.edu/~wn/.