Intelligent Matching for Public Internet Web Services – Towards Semi-Automatic Internet Services Mashup

Chen Wu, Tharam Dillon, Elizabeth Chang
Digital Ecosystems and Business Intelligence Institute
Curtin University of Technology, Perth, Australia
chen.wu@curtin.edu.au

Abstract

In this paper, we propose an Internet public Web service matching approach that paves the way for (semi-)automatic service mashup. We will first provide the overview of the solution, which requires a detailed review of two fundamental models – schema/graph matching and semantic space. Based on the conceptual model and the literature study, the complete service matching approach is then provided with four essential steps – semantic space, parameter tree, similarity measures, and WSDL operation matching. The system demonstration that proves the concept proposed in this approach is finally presented. The solution has the potential to facilitate the Internet services mashup.

1. Introduction

With the surge of SOA and Web services, there is an abundance of Web services in the global space. One way to fully utilise their capabilities is to organise them into small groups, which in turn can be composed into bigger service communities in order to serve various user requirements. Similar to the formation of a new couple and family, matching is considered a very effective "bottom-up" means of organising (or 'attracting') each individual into pairs, which can then constitute bigger communities, thus forming the society of service kind. Therefore, in this paper, we will discuss the motivation, the formulation, and solution to the Web services matching problem. This paper paves the way for the future research in realising service mashup, thereby building bigger service communities and societies for service consumers.

Service matching paves the way for the (semi-) automatic service integration, where a set of Web services works in cooperation to fulfil the requirements of the end users. Service matching also paves the way for realising the Service Mashup, in which service mediation, customisation, and combination are supported to deliver actual services to meet particular requirements demanded by various end users.

The rest of this paper is structured as follows. Section 2 briefly reviews the related work. Section 3 provides the detailed service matching approach. The prototype system and the evaluation are then summarized in Section 4. Section 5 concludes the paper with two future work directions.

2. Basic Schema Matching

Given the importance of Schema Matching in our solution, this section provides a brief review. Schema matching has been utilised in numerous applications [1]. The linguistics-based matching uses linguistic resources or simple string similarity function to obtain the distance between names of the schema elements and attributes. Giunchiglia et al. [2] have utilised the WordNet [3], in which various 'senses' (semantics) of the same words are organised in hierarchical forms that can be compared and reasoned. Different words with similar meanings can be identified based on their senses' position in the WordNet lexical hierarchy. A domain specific knowledge thesaurus has been used in [4], where the Cupid matching system relies on a thesaurus that has synonymy and hyponymy relationships to calculate the linguistic similarity coefficients between two schema elements. Examples of schema reuse can be found in [5], [6] and [7]. Equally important to the linguistic matching is the structural matching, in which the relations between schema elements are considered as "constraints" that restrict the matching patterns. Since many schemas can be represented as (labelled) graph-like structures, some research such as [8] uses graph-based techniques to compare the element positions within and graphs. As a combinatorial problem, graph matching can be computationally prohibitive. Therefore, a simplified graph – tree – has been used to measure the structural similarity between two schemas. Tree also captures the hierarchical containment relationship inherent in many schema definitions such as the XML schema. [4] and [5] utilise the tree elements structural relationship as

978-0-7695-3709-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICWS.2009.116

the key component for the XML schema similarity measure.

3. Matching Approach

In this section, we discuss the matching approach. Given a WSDL operation, the goal of service matching is to (1) obtain the matching similarity between input/output messages of this WSDL and output/input messages of all other operations in the WSDL corpus, and (2) select a number of WSDL operations with sufficient matching similarity scores calculated from (1). In this section, we further divide the similarity measure into three parts: semantic, syntax, and structure.

3.1. Service Semantic Space

In our previous work [9], we have crawled the Web and obtained some thousands Internet public Web services. In this paper, we will construct a semantic space for these public Web services. A Semantic Space is the *assignment of each word (i.e. term) in a language to a point in a real finite dimensional vector space* [10]. Therefore, the public Service Semantic Space is the result of assigning each term in the WSDL Corpus to a point in the reduced-rank vector space. The model incorporates the rank reduced matrix (produced by Latent Semantic Analysis [11]) as an important model element. Readers refer to our previous work in [12] for details of LSA on WSDL Corpus.

Lower [10] formally models a semantic space as a quadruple $\{A, B, S, M\}$, where B is a set of basis elements (b_1, b_2, $\ldots b_D$) that determines the dimensionality D of the space, A defines the mapping function that produces vector elements given the statistical co-occurrence frequencies of each word in both each b_i and the language, S represents the similarity measures which interpret pair-wise vector comparison results as semantic similarity in the form of quantity values, M is the mathematical or statistical model that can be used to transform one semantic space to another. In order to illustrate the instantiation from the semantic space model, we provide a mapping (Table 1) between the generic Semantic Space model and the Web services semantic space in the service matching approach.

Table 1. Mapping between model and services semantic space

Semantic Space Model	Web Services Semantic Space (LSA)
A	*log * entropy*
B	*n* WSDL documents / *k* factors
S	Cosine value
M	Singular Value Decomposition

The set of basis elements B are WSDL documents, thus each WSDL file representing one Web service corresponds to one b_i. Similarity measure S is the cosine value between two term vectors. The transform model M of the Web services semantic space is the Singular Value Decomposition (SVD [13]) used in LSA, which projects existing vectors in B into a linear subspace B' supported by k orthogonal "factors". Hence, the new B' consists of these k factors.

The Semantic Space can be utilised for multiple purposes. In our previous work, we discussed the term semantic similarity which is used for the service retrieval suggestion. In this paper, we will demonstrate that the retrieval suggestion mechanism is a representation (i.e. view) of the underlying service semantic space model. Therefore, the same model may correspond to different views.

As mentioned earlier, the model M is a rank reduced vector space, where term vectors are represented using orthogonal k singular vectors (i.e. factors). Capturing hidden relations between terms and WSDL documents, the semantic space model can be further mapped to various applications in the form of 'view'. An example of such a view has been demonstrated in our previous work as an application of 'term suggestion'. The associated terms are ranked based on their similarity value retrieved from the underlying semantic space model. To an end user, semantic space is merely a ranked list of related terms for a query.

Similarly, other views can be created based on user requirements in various applications. For example, a visualised representation of the semantic space is very helpful to gain a thorough understanding of the relations between all terms and to navigate users from one term to another during the query expansion. In this section, we will create the "Similarity Map" view for service matching. It is essential for the service matching process to efficiently obtain the semantic similarities between terms that are used in the WSDL operation parameters. Although the cosine similarity between vectors can be calculated and retrieved from the semantic space, a dedicated view that provides fast similarity retrieval would be far more desirable for service matching.

3.2. Generate Parameter Tree

Parameter is the WSDL Part element. Parameter Tree is a labelled unordered rooted tree, where each tree node represents an element constituting the data structure of the WSDL Part element. For each parameter, a corresponding parameter tree is generated to characterise its internal data structure that fits into the graph/tree model. The parameter tree is generated from the WSDL '<part />' element, which describes a logical abstract content of an IN/OUT message of a WSDL operation. This follows the WSDL1.1 specification, where <part /> is associated with a data type from some type system using a message-typing attribute.

Therefore, the parameter tree contains all essential data type information. In particular, the non-leaf tree nodes represent elements with complex data type, the leaf nodes are elements with base or simple data types. The label of each tree node describes the name of the element, and the root node of a parameter tree points to the '<Part />' WSDL element itself, with the label representing the value of the name attribute of the '<Part />'. The order of tree nodes at the same hierarchical level does not matter for the matching. In practice, the Parameter Tree is modelled as an XML document, where each XML element's label represents the Tree node label and the "type" attribute captures the data type.

The tree generation algorithm *G* takes as input a well-formed and valid WSDL document *W*, which contains *N* WSDL operations that have appropriate WSDL bindings. The algorithm generates as output *2N* parameter tree lists, each of which includes a list of parameters that constitute the IN or OUT message. Each parameter *P* encodes the hierarchical data structure of the type *T* defined in the type system, which can be either obtained from within this WSDL document or imported from another WSDL document located elsewhere. Since XML Schema is treated as the "intrinsic" [14] type system in WSDL, the main task becomes that of converting the XML schema definition into the hierarchical data structure encoded by *P*, i.e. the parameter tree. When thinking of representing XML Schema as a labelled unordered rooted tree, one needs to address several empirical problems. First of all, there exists no "root" node in an XML Schema definition. A generic W3C schema document often contains a number of data types (i.e. the *global schema components*) at the top level immediately under the element "<schema />". In this way, they can be referenced by other lower level data elements in the same schema or even imported to other schema documents for reuse purposes. On the other hand, a tree structure must have exactly one root, from which all other tree nodes can be traversed. Moreover, data types in an XML Schema document can arbitrarily "be referenced by" or "reference to" various complex or simple data types defined at any levels for the reuse purpose. This without doubt completely breaks the tree structure, where a "single parent – multiple children" relation is enforced at each level of the data elements hierarchy.

The parameter tree generation algorithm deals with these problems using several strategies. Firstly, the selection of root node is totally determined by the data type of the WSDL "<part />" element. In other words, a *global* schema data type automatically becomes the root node if it is directly referenced from the WSDL "<part />" element. Secondly, the 'multi-parents' problem is solved by duplicating the child whenever this child has more than one parent node. As a result, two parents will never share the same child, but each maintains a 'deeply-cloned' copy of the child. Thirdly, label the anonymous data types with names copied from their enclosing elements. Lastly, the algorithm will scan the schema prior to the actual tree generation. The cyclic definition can be detected, and recursive relations will be subsequently cut off in order to maintain the simple hierarchical structure. The rationale behind this is that the cyclic definition lies in the 'syntax' level of the problem, and does not significantly affect the semantics of the data type, and hence can be ignored. Future work can be carried out to investigate the impact of such a syntax level on the service matching problems. Figure 1 depicts the algorithm for parameter tree generation.

```
Input wsdlFile: String  // the location of the crawled WSDL file
3rd Party Library  parser // from WSDL2Java, Axis1.4 Open Source
Output paraTree: ParameterTree //each Part of each Operation has one paraTree

parser.run(wsdlFile);
symbolTable := parser.getSymbolTable()
definition := symbolTable.getDefinition()
FOR EACH binding in definition.getBindings()
  bdEntry := symbolTable.getBindingEntry(binding)
  portType := binding.getPortType()
  IF (portType has been processed)
          CONTINUE // since one portType can have more than one bindings
  FOR EACH operation in the portType
    parameters:Parameters := bdEntry.getParameters(operation)
    IF (parameters = null) CONTINUE
    FOR EACH parameter in parameters //each part
      paraTree := generateTree(parameter)
      Byte b = parameter.getMode()
      IF (b = IN || b = INOUT) paraTreeListIn.add(paraTree)
      ELSE IF (b = OUT || b = INOUT) paraTreeListOut.add(paraTree)
    IF (parameters.returnParam != null)
      paraTree := generateTree(parameters.returnParam)
      paraTreeListOut.add(paraTree)
    paraTreeListIn.serialiseToXML()
       paraTreeListOut.serialiseToXML()

ParaTree generateTree(parameter)
  type:TypeEntry := parameter.getType()
  typeDef : TypeDef := buildTypeDef(type)
  RETURN typeDef.generateTree()

TypeDef buildTypeDef(type)
  IF (type IS CollectionType OR type IS CollectionElement)
    RETURN new ArrayTypeDef(type)
  IF (NOT (type IS DefinedType OR type IS DefinedElement))
    RETURN new BaseTypeDef(type)
  IF (type.getComponentType() != NULL)
    RETURN new ArrayTypeDef(type)
  IF (type IS DefinedElement)
    IF (type IS BaseType)
      RETURN new BaseTypeDef(type)
    ELSE RETURN new ComplexTypeDef(type)
  IF (type IS SimpleType) //type is DefinedType
    RETURN new SimpleTypeDef(type)
  ELSE RETURN new ComplexTypeDef(type)

Class ComplexTypeDef implements TreeGen
  ParaTree generateTree()
    root := new ParaNode(getName(), getType())
    ptree := new ParameterTree(root)
    FOR EACH typeDef in getElements()
      addChildrenNode(root, typeDef)
    RETURN ptree

    void addChildrenNode(parent, typeDef)
      IF (typeDef IS ArrayTypeDef) // ignore array structure
        addChildrenNode(parent, typeDef.getElementType())
```

Figure 1 Pseudo Code for Parameter Tree Generation

3.3. Structural Similarity

The overall similarity between two parameter trees (lists) determines the matching score between two WSDL operations. Definition the overall similarity as:

$$Sim_{overall} = \sum (Sim_{semantic}, Sim_{structure}, Sim_{Syntax})$$

where the semantic similarity determines the extent to which two tree nodes are conceptually related. The structural similarity examines the level of similarity between two parameter tree nodes in terms of their positions in the tree and the neighbourhood arrangement of the tree hierarchy. The syntax similarity considers the 'signature-level' proximity between two parameter tree nodes. These three types of similarities are then integrated into one composite similarity between two tree nodes – thus forming the overall similarity.

First, we define:

$$Sim_{structure} = \sum (LSim, GSim)$$

where *LSim* represents the local structural similarity and *GSim* represents the global structural similarity. Thus, the structural similarity can be seen as the composite of both local and global similarities. More specifically, the local structure refers to the structure neighbouring the current tree node. The global structure refers to the overall structure features in the whole tree. To calculate global structural similarity, we directly use the structural similarity provided in [15], which measures the difference in the tree depth at which the node appears. This is based on the observation that, in the parameter tree, each hierarchy level represents a grouping of related concepts, each of which in turn can be composed by a set of abstract data types. Such a depth-based scheme does reflect the position of the tree node in the tree. However, it neglects the local structure that might also play an important role in shaping the tree structure.

To work out the local structural similarity, we propose an alternative algorithm based on the sub-vector space model. The rationale is as follows. For a non-leaf node in a tree, there are two factors that distinguish its structure from others – its children and its sibling. Therefore, the number of child nodes and the number of sibling nodes play a central role in determining the structure of a node from a local neighbouring perspective. If we consider each tree node as a two-dimensional vector, the two dimensions are children and siblings respectively. And the component value of each dimension is the number of children or siblings, i.e. the extent to which this node performs on that particular dimension. Therefore, the structural similarity between two tree nodes can be calculated using the cosine similarity.

$$LSSim(A,B) = \frac{C(A)C(B) + S(A)S(B)}{\sqrt{C(A)^2 + S(A)^2}\sqrt{C(B)^2 + S(B)^2}}$$

where *C(A/B)* represents the number of children for node *A/B*, *S(A/B)* denotes the number of siblings of node *A* or *B*. For leaf nodes, only the sibling number is considered. Moreover, if two leaf nodes are compared, their syntactic similarities are also calculated as discussed later in this section.

Having obtained both the global and the local structural similarities, the overall structural similarity is defined as:

$$SSim(A,B) = \alpha \bullet LSSim(A,B) + (1-\alpha) \bullet GSSim(A,B)$$

where *0<=α<=1* represents the weight of local similarity in contributing to the overall structural similarity. An appropriate value of αcan be found after several rounds of experiments. It can also reflect the preferences of the matching requirements or the nature of the parameter tree lists. For example, if many complex data types have been defined at each level of the parameter hierarchy (e.g. the eBay or the Amazon Web services), the local structural similarity tends to be more important as the global structure becomes insufficient for capturing the rich data type definitions hidden in the "sub-trees". For this reason, we have set α = 0.7 in our prototype, which relatively favours the local structure similarity.

3.4. Syntax Similarity

The type similarity constitutes the sole part of the syntax similarity. The concept of data types is rooted in the theory of computer programming language (e.g. compiler) that helps to statically declare data used for different occasions. Such a "strong" type mechanism is well supported in the W3C schema and hence in the parameter tree list converted from the WSDL files. The basic idea of type similarity (syntax similarity in this paper) is to ascertain the closeness of all the available types in terms of their 'general purpose'. The assumption is that two parameters are considered similar if their data types are very similar, i.e. they might be used to serve similar purposes, to achieve parallel goals, or to fit into related contexts. The closeness between a pair of data types is determined by their position in the type hierarchy defined in the XML schema. The data type similarity comparison can be assessed using data type mapping table defined in [16]. Alternatively, in this paper, we have defined the similarity comparison result falls into four discrete values as shown in Equation 1.

Case (1) of Equation 1 represents those types that are actually equivalent to each other. For example, type "int" and type "integer" refer to the same concept. The only difference lies in their format, which is ignored during the type similarity comparison. In case (2) two types originates from the same 'ancestor' in the type hierarchy. For example, based on the XML Schema [17], type "token" and type "normalizedString" both derive from the type "string" but they process special characters (e.g. spaces) in a slightly different way. Case (3) resembles the concept 'cast' widely used in most advanced strong type-based programming languages. Typical examples are numeric data types such as "integer" and "positiveIngeter", "short" and "byte", "decimal" and "float", etc. Time-related data types often fit in this category: such as "time" and "dataTime", "date" and "gYearMonth", etc. Moreover, "QName" and "NCName" also reflect such a 'part-of' relation.

It should be noted that the 'cast' operation might lead to the loss of data for either data types, which differentiates case (3) from case (1) and (2). Many pairs of types belong to different type hierarchy. However, they sometimes can be converted from/to each other with good reasons. Take the type pair "token" and "Name" as an example. Each one is from different type hierarchy, but generally "token" can be converted to "Name" without seriously changing the intention or context for the usage of type "Name", and vice versa. We thus define case (4) for such data types. In the last case (5), two types are in different type hierarchies and there appears little justification and grounding to support any kinds of explicit conversion between these two. For example, the type "boolean", which indicates the two states ('true' or 'false') of a concept, cannot be converted to type "anyURL" whatsoever. Hence, they do not share positive syntax similarity values.

$$Sim_{type} = \begin{cases} \mathit{1.00}\text{, equivalent types (1)} \\ \mathit{0.75}\text{, the same basic types but with minor restrictions (2)} \\ \mathit{0.50}\text{, the same basic types that maintains } \textit{part-of} \text{ relations (3)} \\ \mathit{0.25}\text{, different basic types, but conversion can be justified (4)} \\ \mathit{0.00}\text{, different basic types, and conversion is not usual (5)} \end{cases}$$

Equation 1. Four possible values of the type similarity result

Applying Equation 1, each simple data type is compared with all other simple types, and the result is then fed to the similarity comparison matrix, which contains pair-wise comparisons between any two types. During the run-time parameter tree similarity calculation, this type comparison matrix will be used as an in-memory hashtable dictionary that is responsible for a fast, simple type of similarity lookup. If either parameter tree node is of a complex type, the result of the syntax similarity comparison is assigned value '0'. In other words, syntax similarity is applicable only for parameter tree nodes with simple (primitive) data types in our solution.

3.5. WSDL Operation Matching

Given two WSDL operations (A and B), service matching is able to produce two matching scores. The forward matching score examines how well A's output message matches B's input message. Likewise, the backward matching score measures the degree to which B's output message matches A's input message. The basic idea of matching scores is to measure the 'complementary degree' between two Web service operations, which can be seen as two tails of an information (i.e. message) transaction (or exchange).

The outputs of this algorithm are two matching scores as two message similarity measures. Since the Service messages can be modelled, the problem is thus reduced to comparing two parameter tree lists. This boils down to the parameter tree matching problem, which is solved using the Maximum Weighted Bipartite Matching (MWBM) algorithm given the available pair-wise comparison matrix. Next, two corresponding post order lists (i.e. parameter trees) are generated to fit in the MWBM model. Lastly, the tree node similarity comparison is obtained through the combination of semantic, structural, and syntactic similarities. While structural and syntactic similarities are calculated based on their associated metadata, the semantic similarity requires the MWBM routine to get the maximum matching score between two token lists given a token weights matrix. This is because the label of each tree node consists of a list of English words after the tokenisation process. For example, one of the parameter tree nodes of eBay auction service has the label "Get Feedback Response". Therefore, the semantic similarity between two tree labels equates to comparing two token lists, which can be solved using the MWBM routine.

The MWBM routine has been employed three times in order to (1) calculate the semantic similarity between two token lists (two nodes), (2) calculate the overall similarity between two trees (i.e. node lists), and (3) calculate the similarity between two messages (i.e. tree lists). The Hungarian Method [18] has been utilised. The implementation of MWBM in this paper, however, has been optimised based on the detailed algorithm provided in [19]. The basic idea of the MWBM algorithm remains the same: to start with any empty matching, and repeatedly discover 'augmenting' paths that can maximise the overall matching weight. Interested readers can refer to graph theory, and in particular the network flow problem [20], for a comprehensive understanding of the rationale behind the MWBM algorithm. In what follows, we will focus solely on the maximum matching weight normalisation.

Finding the maximum weight matching is one thing; normalising the matching score is another. Intuitively, the graph (tree) with more vertices (nodes) is bound to have higher weights than those with smaller number of vertices, which leads to fewer matching weights. Therefore, it is essential to normalise the matching weight such that all graphs are treated equally regardless of their vertices numbers. Formally, given a matching *M* between sub-graph *A* and *B* with total maximum weight *W*, Definition:

$$score_M = W \big/ \max(|V(A)|, |V(B)|)$$

Equation 2

Where *|V(A)|* and *|V(B)|* represent the number of the vertices in graphs A and B respectively. This normalised score scheme takes into account the size of the tree nodes and thus penalises those parameter trees that receive higher weight only because the absolute number of their enclosing tree nodes is bigger than average. Figure 2 is the algorithm for parameter tree generation.

```
Input: oA, oB : WSDLOperation Output: two matching scores between oA and oB
    //PT represents ParameterTree
forwardMatchingScore := getMsgSim(oA.allOutputPT(), oB.allInputPT())
backwardMatchingScore := getMsgSim(oA.allInputPT(), oB.allOutputPT())

float getMsgSim(paraTreeListA, paraTreeListB)
    FOR EACH paraTreeA in paraTreeListA
    FOR EACH paraTreeB in paraTreeListB
      treeWeights[i][j] := getTreeSim(paraTreeA, paraTreeB)
  RETURN bipartiteMatch(paraTreeListA, paraTreeListB, treeWeights)

float getTreeSim(paraTreeA, paraTreeB)
  nodeListA := postOrderTraverse(paraTreeA)
  nodeListB := postOrderTraverse(paraTreeB)
  FOR EACH nodeA in nodeListA
    FOR EACH nodeB in nodeListB
        nodeWeights[i][j] := getNodeSim(nodeA, nodeB)
  RETURN bipartiteMatch (nodeListA, nodeListB, nodeWeights)

float getNodeSim(nodeA, nodeB)
  structSim := getStructSim(nodeA, nodeB) //refer to structure similarity
  typeSim := getTypeSim(nodeA, nodeB) //refer to syntax similarity
  tokenListA := tokenise(nodeA.text)
  tokenListB := tokenise(nodeB.text)
  FOR EACH tokenA in tokenListA
    FOR EACH tokenB in tokenListB
      tokenWeights[i][j] := getSemanticSim(tokenA, tokenB)//use semantic space
  semanticSim := bipartiteMatch (tokenListA, tokenListB, tokenWeights)
  RETURN compositeNodeSim(structSim, typeSim, semanticSim)//refer to sim

float bipartiteMatch(List A, List B, float[][] weightMatrix) //normalise as well

```

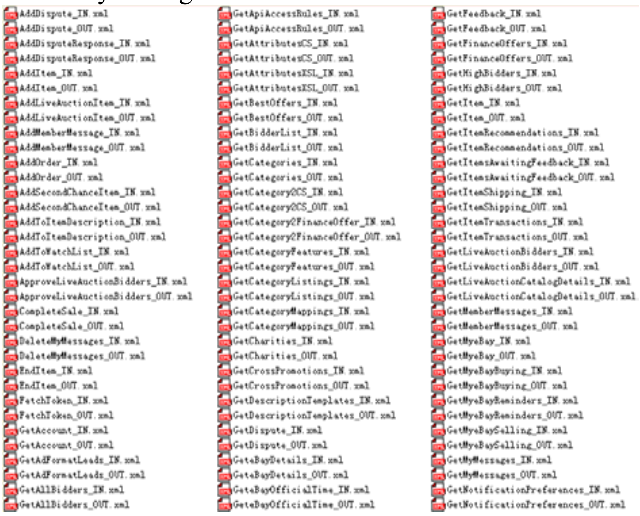
Figure 2 Algorithm of operation matching based on MWBM

A service chain is initially started by a 'central' Web service, from which both 'up' and 'down' stream Web services are gradually identified using the service operation matching algorithm discussed above. The service chain needs a central Web service as the starting point and a list of Web services as the potential searching space. By default, each operation in the central Web service corresponds to a service chain. However, users can also narrow down the starting point to a particular operation. For each chain, we have set the maximum length as the threshold to end the chain generation process. Interesting future work can be carried out to propose alternative thresholds. For example, one possible condition would be to check if the two tails of the chain have reached close enough to some parameters specified in the input and the output as a complex user service discovery request. A list of integrated Web services (vs. a single Web service) is returned as a service chain in response to user queries with input and output constraints. During the chain generation, the last and the first operations are compared with all Web service operations in the searching space in order to find the two that have the highest forward and backward matching scores respectively.

4. Prototype Demonstration

In this section, we will provide the service matching demonstration. Firstly, we will examine the parameter tree generation, and then we will present the service matching result.

Applying the parameter tree algorithm to the eBay Web service [1] will yield the following results. As shown in Figure 3, the algorithm generates a total of 220 Parameter Tree Lists (only 102 are shown) out of one WSDL file, which contains 110 WSDL operations. That is, each operation produces two parameter tree lists: IN and OUT. From the names of these lists, one can easily distinguish between IN and OUT.



```
AddDispute_IN.xml
AddDispute_OUT.xml
AddDisputeResponse_IN.xml
AddDisputeResponse_OUT.xml
AddItem_IN.xml
AddItem_OUT.xml
AddLiveAuctionItem_IN.xml
AddLiveAuctionItem_OUT.xml
AddMemberMessage_IN.xml
AddMemberMessage_OUT.xml
AddOrder_IN.xml
AddOrder_OUT.xml
AddSecondChanceItem_IN.xml
AddSecondChanceItem_OUT.xml
AddToItemDescription_IN.xml
AddToItemDescription_OUT.xml
AddToWatchList_IN.xml
AddToWatchList_OUT.xml
ApproveLiveAuctionBidders_IN.xml
ApproveLiveAuctionBidders_OUT.xml
CompleteSale_IN.xml
CompleteSale_OUT.xml
DeleteMyMessages_IN.xml
DeleteMyMessages_OUT.xml
EndItem_IN.xml
EndItem_OUT.xml
FetchToken_IN.xml
FetchToken_OUT.xml
GetAccount_IN.xml
GetAccount_OUT.xml
GetAdFormatLeads_IN.xml
GetAdFormatLeads_OUT.xml
GetAllBidders_IN.xml
GetAllBidders_OUT.xml
GetApiAccessRules_IN.xml
GetApiAccessRules_OUT.xml
GetAttributesCS_IN.xml
GetAttributesCS_OUT.xml
GetAttributesXSL_IN.xml
GetAttributesXSL_OUT.xml
GetBestOffers_IN.xml
GetBestOffers_OUT.xml
GetBidderList_IN.xml
GetBidderList_OUT.xml
GetCategories_IN.xml
GetCategories_OUT.xml
GetCategory2CS_IN.xml
GetCategory2CS_OUT.xml
GetCategory2FinanceOffer_IN.xml
GetCategory2FinanceOffer_OUT.xml
GetCategoryFeatures_IN.xml
GetCategoryFeatures_OUT.xml
GetCategoryListings_IN.xml
GetCategoryListings_OUT.xml
GetCategoryMappings_IN.xml
GetCategoryMappings_OUT.xml
GetCharities_IN.xml
GetCharities_OUT.xml
GetCrossPromotions_IN.xml
GetCrossPromotions_OUT.xml
GetDescriptionTemplates_IN.xml
GetDescriptionTemplates_OUT.xml
GetDispute_IN.xml
GetDispute_OUT.xml
GeteBayDetails_IN.xml
GeteBayDetails_OUT.xml
GeteBayOfficialTime_IN.xml
GeteBayOfficialTime_OUT.xml
GetFeedback_IN.xml
GetFeedback_OUT.xml
GetFinanceOffers_IN.xml
GetFinanceOffers_OUT.xml
GetHighBidders_IN.xml
GetHighBidders_OUT.xml
GetItem_IN.xml
GetItem_OUT.xml
GetItemRecommendations_IN.xml
GetItemRecommendations_OUT.xml
GetItemsAwaitingFeedback_IN.xml
GetItemsAwaitingFeedback_OUT.xml
GetItemShipping_IN.xml
GetItemShipping_OUT.xml
GetItemTransactions_IN.xml
GetItemTransactions_OUT.xml
GetLiveAuctionBidders_IN.xml
GetLiveAuctionBidders_OUT.xml
GetLiveAuctionCatalogDetails_IN.xml
GetLiveAuctionCatalogDetails_OUT.xml
GetMemberMessages_IN.xml
GetMemberMessages_OUT.xml
GetMyeBay_IN.xml
GetMyeBay_OUT.xml
GetMyeBayBuying_IN.xml
GetMyeBayBuying_OUT.xml
GetMyeBayReminders_IN.xml
GetMyeBayReminders_OUT.xml
GetMyeBaySelling_IN.xml
GetMyeBaySelling_OUT.xml
GetMyMessages_IN.xml
GetMyMessages_OUT.xml
GetNotificationPreferences_IN.xml
GetNotificationPreferences_OUT.xml
```

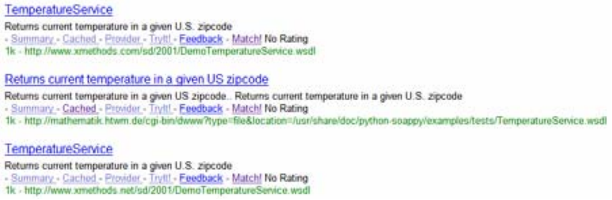
Figure 3. 102 out of 220 parameter tree lists generated from the eBay Web service

An example of a generated parameter tree list from "GetFeedback_OUT.xml" is illustrated in Figure 5. In both samples, the leaf tree nodes are all kept with primitive XML schema data type. The eBay Web service sample also suggests that the crucial role of the cyclic checking, without which the "StackOverFlow" runtime error is reported due to the infinite type reference recursion during the tree generation process.

We now demonstrate the service matching process and result. Suppose a system developer needs to constantly check the weather condition in the software

[1] http://developer.ebay.com/webservices/latest/ebaySvc.wsdl

application in order to provide real-time weather forecasting services to the end users. A user's physical location is stored in the user profile as the name of the city and the suburb. The developer is then looking for a Web service in the weather-related domain syndication, trying to find a Web service that takes as input the user location (i.e. city name), and produces as output the weather condition. He might put 'weather' or 'weather report' in the service retrieval user interface, which then returns a list of Web services ranked according to their relevance to the theme 'weather'. Suppose the first Web service in the ranking list provides comprehensive weather condition (i.e. wind, humidity, temperature, etc.) that well suffices for an end user's requirement. Unfortunately, while perusing this Web service's specification (i.e. WSDL), the system developer finds that the most relevant WSDL operation 'getWeatherByZipCode' does not support city/suburb names as the default input. In other words, the developer needs another Web service that can convert a geographic location into a postal code before using the desired weather Web service. This implies that another round of Web service retrieval is necessary for the developer, who then needs to manually compare the output of the first Web service with the input of the second Web service in order to check if they can be integrated as a whole to provide the weather forecasting services to the end users.



TemperatureService
Returns current temperature in a given U.S. zipcode
- Summary - Cached - Provider - TryIt! - Feedback - Match! No Rating
1k - http://www.xmethods.com/sd/2001/DemoTemperatureService.wsdl

Returns current temperature in a given US zipcode
Returns current temperature in a given US zipcode.. Returns current temperature in a given U.S. zipcode
- Summary - Cached - Provider - TryIt! - Feedback - Match! No Rating
1k - http://mathematik.htwm.de/cgi-bin/dwww?type=file&location=/usr/share/doc/python-soappy/examples/tests/TemperatureService.wsdl

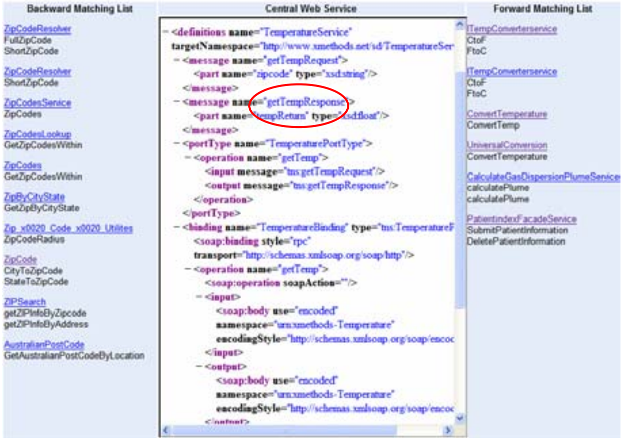
TemperatureService
Returns current temperature in a given U.S. zipcode
- Summary - Cached - Provider - TryIt! - Feedback - Match! No Rating
1k - http://www.xmethods.net/sd/2001/DemoTemperatureService.wsdl

Figure 4. Service matching UI

First, a service consumer attempts to find temperature-related Web services, the service retrieval provides the following three Web services as shown in Figure 4. This GUI is the extended version of our previous service discovery system in [21]. Suppose that after the service selection, the service consumer prefers the second one. So s/he decides to use the service that "*returns current temperature in a given US zipcode*". However, as stated in the requirement, this Web service does not accept address (e.g. city/suburb names) as the default input but US zipcode. Due to the limited knowledge of zipcode, the service consumer cannot use this service 'as is'. This is where service matching comes into the help. The service consumer can launch the service matching by clicking the "*Match!*" link as shown in Figure 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<GetFeedback type="ParaTreeList">
        <GetFeedbackResponse type="GetFeedbackResponse">
                <FeedbackDetailType type="FeedbackDetailType">
                        <CommentingUser type="string"/>
                        <CommentingUserScore type="int"/>
                        <CommentText type="string"/>
                        <CommentTime type="dateTime"/>
                        <CommentType type="token"/>
                        <FeedbackResponse type="string"/>
                        <Followup type="string"/>
                        <ItemID type="string"/>
                        <Role type="token"/>
                        <ItemTitle type="string"/>
                        <ItemPrice type="token"/>
                        <FeedbackID type="string"/>
                        <TransactionID type="string"/>
                        <CommentReplaced type="boolean"/>
                        <ResponseReplaced type="boolean"/>
                        <FollowUpReplaced type="boolean"/>
                        <any type="any"/>
                </FeedbackDetailType>
                <FeedbackDetailItemTotal type="int"/>
                <FeedbackSummary type="FeedbackSummaryType">
                        <FeedbackPeriodType type="FeedbackPeriodType">
                                <PeriodInDays type="int"/>
                                <Count type="int"/>
                                <any type="any"/>
                        </FeedbackPeriodType>
                        <FeedbackPeriodType type="FeedbackPeriodType">
                                <PeriodInDays type="int"/>
                                <Count type="int"/>
                                <any type="any"/>
                        </FeedbackPeriodType>
                        <FeedbackPeriodType type="FeedbackPeriodType">
                                <PeriodInDays type="int"/>
                                <Count type="int"/>
                                <any type="any"/>
                        </FeedbackPeriodType>
......
```

Figure 5. eBay feedback response parameter tree list



Backward Matching List

ZipCodeResolver
FullZipCode
ShortZipCode

ZipCodeResolver
ShortZipCode

ZipCodesService
ZipCodes

ZipCodesLookup
GetZipCodesWithin

ZipCodes
GetZipCodesWithin

ZipByCityState
GetZipByCityState

Zip_x0020_Code_x0020_Utilities
ZipCodeRadius

ZipCode
CityToZipCode
StateToZipCode

ZIPSearch
getZIPInfoByZipcode
getZIPInfoByAddress

AustralianPostCode
GetAustralianPostCodeByLocation

Central Web Service

```
- <definitions name="TemperatureService"
  targetNamespace="http://www.xmethods.net/sd/TemperatureSer
  - <message name="getTempRequest">
      <part name="zipcode" type="xsd:string"/>
    </message>
  - <message name="getTempResponse">
      <part name="tempReturn" type="xsd:float"/>
    </message>
  - <portType name="TemperaturePortType">
    - <operation name="getTemp">
        <input message="tns:getTempRequest"/>
        <output message="tns:getTempResponse"/>
      </operation>
    </portType>
  - <binding name="TemperatureBinding" type="tns:TemperatureP
      <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    - <operation name="getTemp">
        <soap:operation soapAction=""/>
      - <input>
          <soap:body use="encoded"
          namespace="urn:xmethods-Temperature"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoc
        </input>
      - <output>
          <soap:body use="encoded"
          namespace="urn:xmethods-Temperature"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoc
        </output>
```

Forward Matching List

ITempConverterservice
CtoF
FtoC

ITempConverterservice
CtoF
FtoC

ConvertTemperature
ConvertTemp

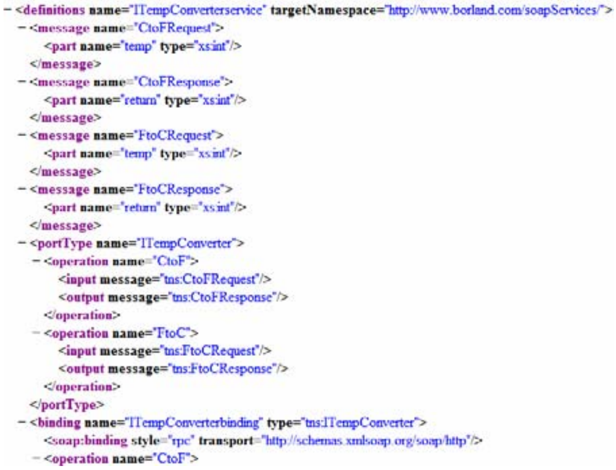
UniversalConversion
ConvertTemperature

CalculateGasDispersionPlumeService
calculatePlume
calculatePlume

PatientIndexFacadeService
SubmitPatientInformation
DeletePatientInformation

Figure 6. Service Matching Result

The service matching process uses the matching algorithm discussed in Section 4 to enumerate the WSDL collection in order to conduct pairwise matching score calculation. For each pair, both forward and backward matching scores are obtained. Each type of matching scores is ranked in a descending order, thus forming forward matching list and backward matching list as shown in Figure 6. In the middle is presented the WSDL file of the selected Web service that takes zipcode and generates the temperature. This service is considered as the 'central Web service'. On the left is the backward matching list, which includes a list of Web services that can produce 'zipcode' as service output message. On the right side is the forward matching list, which includes a list of Web services that can take ‘temperature’ as service input message. Both lists are ranked based on their matching scores. Perusing the first Web service ‘*ITempConvertservice*’ in the forward matching list as shown in Figure 7, one can find that it provides the temperature conversion function so that Celsius can be converted to Fahrenheit through the WSDL operation ‘*CtoF*’. The input message ‘*CtoFRequest*’of this operation matches the output message ‘*getTempResponse*’ of the central Web service.



```
- <definitions name="ITempConverterservice" targetNamespace="http://www.borland.com/soapServices/">
  - <message name="CtoFRequest">
      <part name="temp" type="xs:int"/>
    </message>
  - <message name="CtoFResponse">
      <part name="return" type="xs:int"/>
    </message>
  - <message name="FtoCRequest">
      <part name="temp" type="xs:int"/>
    </message>
  - <message name="FtoCResponse">
      <part name="return" type="xs:int"/>
    </message>
  - <portType name="ITempConverter">
    - <operation name="CtoF">
        <input message="tns:CtoFRequest"/>
        <output message="tns:CtoFResponse"/>
      </operation>
    - <operation name="FtoC">
        <input message="tns:FtoCRequest"/>
        <output message="tns:FtoCResponse"/>
      </operation>
    </portType>
  - <binding name="ITempConverterbinding" type="tns:ITempConverter">
      <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    - <operation name="CtoF">
```

Figure 7. the First Web Service in the Forward Matching List

5. Conclusions and Future Work

In this paper, we propose a service matching approach that paves the way for (semi-)automatic service mashup. We reviewed two fundamental models – schema/graph matching and semantic space. Based on the conceptual model and the literature study, the complete service matching approach is then provided with four essential steps – semantic space, parameter tree, similarity measures, and WSDL operation matching. The system demonstration that proves the concept proposed in this approach is finally presented. In the future, we aim to achieve two important goals: (1) To develop a service matching/mashup benchmark dataset, which needs intensive human labelling, and (2) to carry out more quantitative experiment in order to test the matching performance.

7. References

[1]P. Shvaiko and J. Euzenat, "A Survey of Schema-based Matching Approaches," *Journal on Data Semantics*, 2007.

[2]F. Giunchiglia, P. Shvaiko, and M. Yatskevich, "S-Match: an algorithm and an implementation of semantic matching," presented at European Semantic Web Symposium, 2004.

[3]A. G. Miller, "A lexical database for English," *Communication of the ACM*, vol. 38, pp. 39 - 41, 1995.

[4]J. Madhavan, P. A. Bernstein, and E. Rahm, "Generic Schema Matching with Cupid," presented at 27th VLDB Conference, Roma, Italy, 2001.

[5]H. H. Do and E. Rahm, "COMA - A system for flexible combination of schema matching approaches," presented at 28th VLDB Conference, Hong Kong, China, 2002.

[6]E. Rahm, H. H. Do, and S. Mabmann, "Matching large XML schemas," *SIGMOD Record*, vol. 33, pp. 26 - 31, 2004.

[7]J. Madhavan, P. A. Bernstein, K. Chen, A. Halevy, and P. Shenoy, "Corpus-based schema matching," presented at International Conference on Data Engineering, 2005.

[8]S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching," presented at 18 International Conferences on Data Engineering, 2002.

[9]C. Wu and E. Chang, "Searching services “on the Web”: A public Web services discovery approach," presented at THE THIRD INTERNATIONAL CONFERENCE ON SIGNAL-IMAGE TECHNOLOGY & INTERNET–BASED SYSTEMS, Shanghai, China, 2007.

[10]W. Lowe, "Towards a Theory of Semantic Space," 2002.

[11] S. Deerwester, S. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshamn, "Indexing by Latent Semantic Analysis," *Journal of the American Society for Information Science*, vol. 41, pp. 391 - 407, 1990.

[12]C. Wu, V. Potdar, and E. Chang, "Latent Semantic Analysis - The Dynamics of Semantic Web Service Discovery," in *Advanced Web Semantics*, vol. 4891. LNCS-IFIP, 2008, pp. pp. 346–373.

[13]D. M. Berry, T. Do, G. W. O'Brien, V. Krishna, and S. Varadhan, "SVDPACKC (Version 1.0) User's Guide," Computer Science Department, Univeristy of Tennessee 1993.

[14]E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," 2001.

[15]D. Caragea and T. Syeda-Mahmood, "Semantic API Matching for Automatic Service Composition," presented at WWW2004, New York, USA, 2004.

[16]A. Sheth and J. Cardoso, "Semantic e-workflow composition," *Journal of Intelligent Information Systems (JIIS)*, vol. 21, pp. 191 - 225, 2003.

[17]D. C. Fallside, "XML Schema Part 0: Primer," W3C, 2001.

[18]D. Konig, "Graphs and matrices," *Mat. Fiz. Lapok (Hungarian)*, vol. 38, pp. 116 - 119, 1931.

[19]D. S. Johnson and C. C. McGeoch, "Network flows and matching: first DIMACS implementation challenge," *Series in DIMACS*, vol. 12, 1993.

[20]L. R. Ford and D. R. Fulkerson, *Flows in Networks*: Princeton University Press, 1962.

[21]C. Wu and E. Chang, "Aligning with the Web: An Atom-based Architecture for Web Services Discovery," *Service Oriented Computing and Applications*, vol. 1, pp. 97 - 116, 2007.