Towards quicker discovery and selection of web services considering required degree of match through indexing and decomposition of non-functional constraints

Chellammal Surianarayanan* and Gopinath Ganapathy

School of Computer Science and Engineering,
Bharathidasan University,
Tiruchirappalli, 620 024, India
Email: chelsrsd@rediffmail.com
Email: gganapathy@gmail.com
*Corresponding author

Manikandan Sethunarayanan Ramasamy

Department of Mathematics,
Bharathidasan University Constituent College,
Lalgudi, 621 601, India
Email: manirs2004@yahoo.co.in

Abstract: An approach is proposed for identifying best services for composition based on functional and non-functional characteristics of services with a special focus on computational optimisation of functional discovery and non-functional selection. Discovery is optimised using a unique indexing consists of two indices, one for outputs of services and the other for inputs of services. In either index, each key is mapped to its semantically related service categories. The fine split of semantic relations into eight categories assists in handling disparate similarity demands of service clients efficiently. Also, indexing eliminates semantic reasoning entirely during querying. Non-functional selection is optimised using local selection method in multithreaded fashion with a new method of decomposing non-functional constraints. Further, indexing is used to expedite the searching of finding best services during selection. Experimentation results are presented. The minimum time consumption of the method makes it more applicable to dynamic composition needs.

Keywords: quicker discovery; service selection; service indexing; constraint decomposition; optimisation; functional discovery; non-functional selection; performance optimisation.

Reference to this paper should be made as follows: Surianarayanan, C., Ganapathy, G. and Ramasamy, M.S. (2015) 'Towards quicker discovery and selection of web services considering required degree of match through indexing and decomposition of non-functional constraints', *Int. J. Computational Science and Engineering*, Vol. 10, Nos. 1/2, pp.45–69.

Biographical notes: Chellammal Surianarayanan is an Assistant Professor in Computer Science at Bharathidasan University Constituent College for Women, Orathanadu, India. She has 12 years of experience in research and academic services. She is a research scholar in the School of Computer Science and Engineering, Bharathidasan University, India. Her research interests include semantic web services, semantic web and data mining.

Gopinath Ganapathy is a Professor in Computer Science at Bharathidasan University, India. He has published around 30 research papers in international journals and conferences. He has 23 years of experience in academia, industry, research and consultancy services. He has around 8.5 years of international experience in the USA and UK. His research interests include semantic web, auto programming, natural language processing and text mining.

Manikandan Sethunarayanan Ramasamy is an Assistant Professor in Mathematics at Bharathidasan University Constituent College, Lalgudi. He holds a PhD in Graph Theory. He has 12 years of research and seven years of academic experience. He has published nine research papers in international journals and conferences. His research interests include graph theory and optimisation techniques.

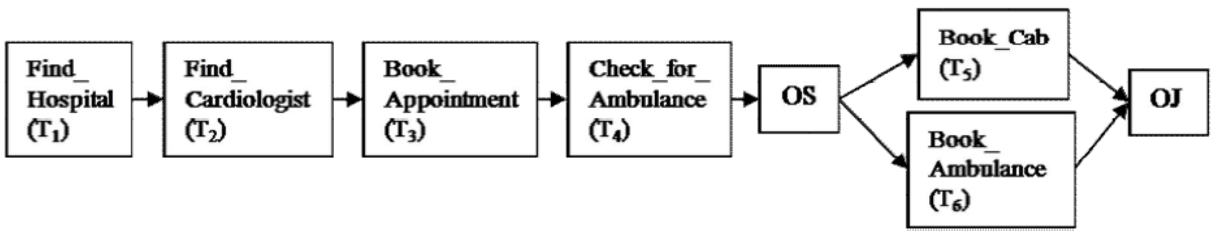
1 Introduction

Web service composition is a process in which more than one atomic service are combined in a specific pattern to accomplish a given business requirement. Any business requirement can be represented as a workflow consisting of a set of tasks combined in a specific pattern. A task represents an abstract function having inputs and outputs. The tasks are implemented by web services which are offered by service providers. During composition, at first the services which could implement the tasks are discovered and then they are combined in the mentioned pattern to produce the desired business goal. An example workflow which represents the query, *book an appointment with a cardiologist in a hospital located in a particular location and book a vehicle to reach the hospital* is given in Figure 1. The workflow given in Figure 1 contains tasks, namely, 'Find_Hospital', 'Find_Cardiologist', 'Book_ Appointment', 'Check_for_Ambulance', 'Book_Cab' and 'Book_Ambulance'. In Figure 1, 'OS' and 'OJ' indicate OR Split and OR Join of an OR execution pattern respectively. As shown in Figure 1, the tasks, T_1, T_2, T_3 and T_4 are executed in sequence. After the execution of T_4, either T_5 or T_6 will be executed. Like OR pattern, a workflow may contain other patterns such as AND, loop, etc. As in Figure 1 even services from different domains may be combined to meet a given business goal. Complex business goals need several services from different domains to be discovered and composed in relatively short time. To implement service composition efficiently, three critical aspects should be considered. They are automation in service discovery, incorporation of a particular level of similarity between tasks and their implementing services as demanded by client applications and non-functional aspects of services.

The prerequisite for composition is the discovery of services which implement various tasks of a given workflow. Discovery becomes a challenge owing to two conflicting aspects, namely, availability of many services and dynamic needs of business applications. These two aspects demand automation in discovery which has been brought by semantic service description languages such as Web Ontology Language for Services (OWL-S) (Guo et al., 2005). These languages allow services to be described with explicit semantics in the form of ontological (formal) constructs. Semantics makes services machine interpretable and brings automation in discovery. Corresponding to semantic description, during discovery, a matching component has to find the semantic matching between the query and each available service with the help of an ontology reasoner such as Pellet. The ontology reasoner interacts with ontologies and finds various semantic relations such as *equivalent*, *plug-in*, *subsumes and fail* (Paolucci et al., 2002) among the concepts of the query and the available services. Though semantics brings maximum automation in discovery, semantic reasoning with ontologies is time consuming (Mokhtar et al., 2006). In real applications, as many services from different domains have to be discovered and composed in a complex chain, the performance of semantic discovery becomes a crucial factor in deciding the feasibility of dynamic service composition. Hence, it is essential to optimise the performance of semantic discovery.

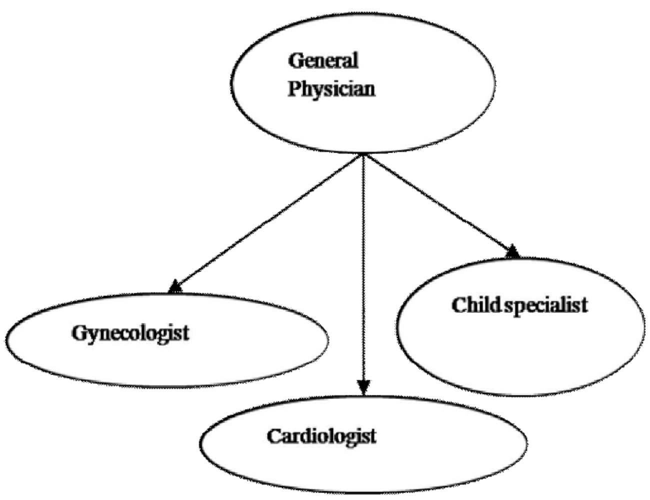
During discovery, the incorporation of a particular level of similarity between a task and its implementing service as demanded by clients should be considered. In practice, discovery approaches such as Skoutas et al. (2008), Zhag et al. (2009), Pathak et al. (2005) and Mokhtar et al. (2008) compute the similarity between a given task and the available services using four levels of degree of match (DoM), namely, *equivalent*, *plug-in*, *subsumes and fail* described in Paolucci et al. (2002). But these levels of DoM may not be suitable to handle all applications as the similarity needs of applications are disparate. The significance of incorporating required level of similarity is illustrated using two examples given below.

Figure 1 An example workflow



Consider the task T_2 in Figure 1. Let the function of T_2 be to find the details of a cardiologist in a given hospital. The input and the output of the task are *hospital* and *cardiologist* respectively. Let the fragment of an imaginary ontology given in Figure 2 describe the output of T_2. During discovery, approaches such as Skoutas et al. (2008) will find a service returning the details of *general physician* as a matched service of T_2 (as according to the given fragment of ontology, *general physician* is a super class of *cardiologist*). Also, methods such as Rozina et al. (2010) will find services returning details of *gynecologist and child specialist* as matched services of T_2 (as according to the given fragment of ontology, *gynecologist and child specialist* are siblings of *cardiologist*). In reality, T_2 cannot be implemented by services which return the details of general physician or gynecologist or child specialist, though general physician, gynecologist and child specialist are semantically related to cardiologist. The task, T_2 can be implemented only by its *equivalent* matches.

Figure 2 Fragment of an imaginary ontology



Let us consider another task which performs the function, *Book tickets in XYZ Airlines to travel from Singapore to London on 2nd October 2012*. Here the task may be implemented using a service offer from an airlines other than *XYZ Airlines* such as *ABC Airlines* (which provides the same function) when seats are not available with *XYZ Airlines*. In this case, though *ABC Airlines* is sibling of *XYZ Airlines*, the offer from *ABC Airlines* is acceptable.

Thus the required degree of match (*RDoM*) between a task and its implementing service is dependent on the nature of applications and it should be considered during discovery. We refer the minimum level of similarity that should exist between a task and its implementing service as demanded by a client as *RDoM*.

Selection of services for composition based on quality of service (QoS) characteristics (which refer to non-functional attributes such as response time, availability, latency, etc., of services) is essential as users tend to query such as *find a Malaysian tour planning service whose cost is <=\$500*. In this example, besides the functional requirements of tour planning, the cost of service should also be considered during composition. To accommodate the varying non-functional needs of service clients, service providers offer multiple services having same functional characteristics but different non-functional characteristics. The services which offer the function of same task with different QoS would form a *service class* (Mohammad et al., 2008). To accomplish the user's QoS requirements, it is essential to find the best service for each task from its corresponding service class based on QoS.

In view of the above analysis, a new approach which identifies the best service combination for composition is proposed in two stages, namely, functional discovery and non-functional selection. In functional discovery, services which implement various tasks of the workflow are discovered based on inputs and outputs of tasks according to the given *RDoM*. The functional discovery results in a set of service classes which will be given as input to non-functional selection. In selection the best service is selected for each task from its respective service class based on QoS.

The chief aim of the approach is to reduce the computation time of discovery and selection taking into account the demands of *RDoM* and QoS. The approach proposes a unique hash-based indexing mechanism to enhance the performance of discovery. The approach completely eliminates invoking of semantic reasoning during querying using two indices, namely, output index and input index. In the output index the services are indexed by their outputs (i.e., the output parameters of services are used as keys). In the input index, the services are indexed by their inputs (i.e., the input parameters of services are used as keys). In either index, each key is mapped to its eight different kinds of semantically related service categories, namely, *equivalent*, *direct-plugin*, *indirect-plugin*, *direct-subsumes*, *indirect-subsumes*, *sibling*, *partial-parent* and *grandparent*. The *RDoM* of client applications can be expressed using the above categories. As the approach uses hash-based indexing, it achieves time complexity of $O(1)$ while retrieving matched services of a query. The approach optimises the performance of selection using local selection in multithreaded fashion with a new method of decomposing QoS constraints. The method of decomposing constraints has a unique feature that it is independent of number of services present in a service class. Also, during selection the speed of searching for the best service is enhanced using hash indexing.

2 Literature survey

Recently clustering and indexing techniques are used to optimise the performance of semantic discovery. Clustering-based approaches such as Rozina et al. (2010), Liu et al. (2009) and Zhu et al. (2010) partition the available services into different groups of similar services. For example, the available services may be partitioned as 'education group', 'weather group', 'financial group', 'food group', 'travel group', etc. When a query arrives, a particular cluster which is most similar to the query is identified and semantic matching is employed only to that cluster, eliminating all other clusters as irrelevant. Hence, clustering-based methods achieve improved performance by reducing the number of services that require semantic reasoning. However clustering-based methods have to perform semantic matching during querying to those services which are present in the most similar cluster of the query to find matched services of the query.

Discovery approaches such as Zhou et al. (2009), Kuang et al. (2007) and Gao et al. (2009) identify a set of atomic or parallel combination of services called 'candidate services' which can deliver the outputs of a query using output-based indexing. The inputs of each candidate service are semantically matched with that of query by invoking semantic reasoning. Here, semantic reasoning is invoked during querying which is time consuming.

The methods (Skoutas et al., 2008; Qiu and Li, 2008) are close to our work with respect to discovery. The method (Qiu and Li, 2008) presents an approach for integrating semantic features of services into the conventional universal description discovery and integration (UDDI) and discovers matched services of a query based on semantic similarity of service properties. This method uses an ontology concept index and similarity data table to speed up the discovery. In the similarity data table each ontology concept is mapped to its related concepts which have similarity values higher than a minimum value. The ontology concept index maps each concept to its occurrence locations (service name, inputs and outputs) in published services. When a query arrives, the semantically related concepts for each concept of the query are obtained from similarity data table. Then, services which contain the query concepts and their related concepts are retrieved from the ontology index. Specifying minimum similarity as a numerical value while storing related concepts in similarity data table is difficult and compared to numeric values, categorical values are more suitable. In this method there is no inner split of matched services; but a split will help the clients to pick up the most desirable match. Further, the method does not present experimentation related to computation time of discovery. The method (Skoutas et al., 2008) prevents semantic reasoning during querying by indexing the subsumption relations among concepts in a numerically encoded format using two R-Trees, one for inputs and the other for outputs. Though the method mainly addresses the discovery of services based on functional aspects, it does not take into account the sibling or grandparent relations among concepts. Also, the methods (Skoutas et al., 2008; Qiu and Li, 2008) did not address selection of services based on QoS.

As an alternative to the above approaches, the proposed approach uses a unique indexing scheme for discovery. It uses two indices, namely, output index and input index for output and input parameters of services respectively. In either index, each key is mapped to its eight different types of semantically related service categories. For each key, its service categories are computed using a semantic reasoner prior to querying itself. The keys and their service categories are archived. An in-memory indexing of pre-computed values is implemented using hash data structures to enhance the performance of discovery as well as to achieve constant time complexity. Maintenance of two indices eliminates semantic reasoning entirely during querying. Also, a service client is allowed to specify the *RDoM* in terms of the above categories and during querying matched services of a query are retrieved from the indices according to the given *RDoM*.

QoS-based selection is handled by two major methods, namely, global planning and local selection. In global planning, at first a composite service is formed by combining one service from each service class. Then the QoS attributes of composite service are computed and checked against user's QoS requirements. To reduce the exponential time complexity of global approach, local selection approaches such as Mohammad and Thomas (2009), Li et al. (2010), Jin et al. (2010) and Sun et al. (2010) are being used. Local selection methods treat the selection of service combination for a given workflow as a main problem and divide the main problem into *n* sub problems where *n* indicates the number of tasks in the workflow. The sub problems are constructed by decomposing the given user's QoS constraints called *global constraints* (workflow level) into task level constraints called local constraints and assigning the local constraints to individual tasks.

The method of decomposing constraints presented in Mohammad and Thomas (2009) is extensively used by other approaches such as Li et al. (2010), Jin et al. (2010) and Sun et al. (2010). Though Mohammad and Thomas (2009) describes an efficient way of decomposing constraints, it uses mixed integer programming (MIP) to identify local constraints. Hence, it suffers from poor time characteristics when number of services grows. Also, the computation time of this method is influenced by number of constraints. Further, this method assigns a particular level of a QoS attribute which is more frequent in a service class as local constraint of that service class. In such situation, there may be other service which may not be most frequent but has the maximum utility for a user. But such service which is capable of yielding the maximum utility is ignored by the method. But we propose a new method for decomposing QoS constraints based on extreme values (minimum and maximum) of QoS attributes of service classes. As the method of assigning constraints is based on extreme values, it is not affected by individual services present in service classes.

After construction, each sub problem is solved to select a suitable service from its service class subject to its local constraints. Here, the suitable service is identified based on the utility function which captures user's QoS preferences over different attributes. There are many approaches such as Zeng et al. (2003, 2004) and Hong and Hu (2009) which formulate the sub problem as a linear programming (LP) problem and find the optimal values for utility and QoS attributes (decision variables). Though solving sub problems using LP techniques helps in computing optimal values of utility and QoS attributes, the method does not help in identifying the service having the optimal values whereas identification (of service) is must for service composition. Hence, in the proposed approach, the service with maximum utility is determined using standard searching technique. Further, in the proposed approach, indexing of QoS values by *service ID* is used to speed up the searching.

3 Proposed approach

A new approach in two stages, namely, functional discovery and non-functional selection is proposed for identifying best services for composition with its main focus on computational optimisation of functional discovery and non-functional selection. The computation time of functional discovery is reduced by pre-computing various semantic relations among inputs and outputs of all services and maintaining the pre-computed semantic relations in two indices, namely, output index and input index. The computation time of non-functional selection is reduced by using local selection in multithreaded fashion with a new method of decomposing QoS constraints.

3.1 Functional discovery

The functional discovery is split into two tasks, namely, index creation and service retrieval. Index creation is performed prior to querying. During querying service retrieval is performed using the indices.

3.1.1 Index creation

In the proposed approach two indices, namely, output index and input index are used to optimise the performance of discovery. Output and input parameters of services are used as the keys of output index and input index respectively. In either index, each key is mapped to its semantically related service categories (values of the key). To realise the disparate similarity demands of clients, for each key, its value is categorised into eight different semantically related service categories, namely, *equal*, *direct-plugin*, *indirect-plugin*, *direct-subsumes*, *indirect-subsumes*, *sibling*, *partial-parent* and *grandparent*. The service categories for both indices are similar. The service categories with respect to output index are described below.

1 *Equivalent:* This category refers to the set of all services having at least one output whose type is exactly equivalent to the type of the key.

2 *Direct-plugin:* This category refers to the set of all services having at least one output whose type is direct super class (i.e., immediate super class) to the type of the key.

3 *Indirect-plugin:* This category refers to the set of all services having at least one output whose type is indirect super class (i.e., deeper super class) to the type of the key.

4 *Direct-subsumes:* This category refers to the set of all services having at least one output whose type is direct sub class (i.e., immediate sub class) to the type of the key.

5 *Indirect-subsumes:* This category refers to the set of all services having at least one output whose type is indirect sub class (i.e., deeper sub class) to the type of the key.

6 *Sibling:* This category refers to the set of all services having at least one output whose all parents' types are same as all parents' types of the key.

7 *Partial-parent:* This category refers to the set of all services having at least one output whose at least one parent's type is same as at least one parent's type of the key.

8 *Grandparent:* This category refers to the set of all services related to the type of the key in one of the following ways.
   - services having at least one output whose at least one grandparent's type is same as at least one grandparent's type of the key
   - services having at least one output whose at least one parent's type is same as at least one grandparent's type of the key
   - services having at least one output whose at least one grandparent's type is same as at least one parent's type of the key.

Towards the creation of indices, for each key its semantically related service categories with respect to all available services (present in a service repository) are pre-computed with the help of an ontology reasoner. The input and output service indices are constructed using the pre-computed semantically related service categories as discussed below.

Let $s_1, s_2, \ldots, s_m$ be the available services in a repository. Let s_i denote an i^{th} service. Let $out(s_i)$ denote the set of all outputs of s_i. Let $in(s_i)$ denote the set of all inputs of s_i. Let two sets, namely, *Input_index_keyset* and *Output_index_keyset* denote all the keys of input index and output index respectively. The keys of input and output indices are defined as follows.

$$Input\_index\_keyset = \bigcup_{i=1}^{m} in(s_i) \tag{1}$$

$$Output\_index\_keyset = \bigcup_{i=1}^{m} out(s_i) \tag{2}$$

For each key of the input index, i.e., for each $x \in$ *Input_index_keyset*, its various semantically related service categories, namely, *equal*, *direct-plugin*, *indirect-plugin*, *direct-subsumes*, *indirect-subsumes*, *sibling*, *partial-parent* and *grandparent* are denoted by $dom\_in(1, x)$, $dom\_in(2, x)$, $dom\_in(3, x)$, $dom\_in(4, x)$, $dom\_in(5, x)$, $dom\_in(6, x)$, $dom\_in(7, x)$ and $dom\_in(8, x)$

respectively. Each key is mapped to its service categories and the structure of the input index is given in Figure 3. In Figure 3, the left side denotes the keys of the input index (i.e., *x*) and the right side denotes the values of the keys (i.e., values of *x*). From Figure 3, one can understand how each key of the input index (i.e., each $x \in$ *Input_index_keyset*) is mapped to its semantically related service categories.

Similarly, for each key of the output index, $x \in$ *Output_index_keyset*, its various semantically related service categories, namely, *equal*, *direct-plugin*, *indirect-plugin*, *direct-subsumes*, *indirect-subsumes*, *sibling*, *partial-parent* and *grandparent* are denoted by *dom_out*(1, *x*), *dom_out*(2, *x*), *dom_out*(3, *x*), *dom_out*(4, *x*), *dom_out*(5, *x*), *dom_out*(6, *x*), *dom_out*(7, *x*) and *dom_out*(8, *x*) respectively. Each key of the output index is mapped to its semantically related service categories and the structure of the output index is given in Figure 4. In Figure 4, the left side denotes the keys of the output index (i.e., *x*) and the right side denotes the values of keys of output index (i.e., values of *x*). From Figure 4 one can understand how each key of the output index (i.e., each $x \in$ *Output_index_keyset*) is mapped to its semantically related service categories.

To show how a particular key entry will exist in an index, consider a key, $x \in$ *Input_index_keyset*. Let *x* be 'http://localhost/ontology/concept.owl#_recommended_price'. The entry of *x* and its mapped values in the input index is given in Figure 5. For each semantic service category, typical values are given in Figure 5. For example, from Figure 5, the equivalent service category for *x* is found to be $\{s_{23}, s_{24}, s_{25}\}$.

3.1.2 Service retrieval

Let *t* be a task for which matched services have to be retrieved using the indices. Let *Input*(*t*) denote the set of all inputs of *t*. Let *Output*(*t*) denote the set of all outputs of *t*. For each input $x \in$ *Input*(*t*), its semantically related service categories, namely, *dom_in*(1, *x*), *dom_in*(2, *x*), *dom_in*(3, *x*), *dom_in*(4, *x*), *dom_in*(5, *x*), *dom_in*(6, *x*), *dom_in*(7, *x*) and *dom_in*(8, *x*) are extracted from the input index. Similarly for each output $x \in$ *Output*(*t*), its semantically related service categories, namely, *dom_out*(1, *x*), *dom_out*(2, *x*), *dom_out*(3, *x*), *dom_out*(4, *x*), *dom_out*(5, *x*), *dom_out*(6, *x*), *dom_out*(7, *x*) and *dom_out*(8, *x*) are extracted from the output index.

At first, the *equivalent matched service category* of the given task denoted by *equal*(*t*) is computed by taking the intersection of the corresponding service categories of inputs and outputs of the task. Then, based on the results of intersection, we successively define *modified service categories* for inputs and outputs of the task and ultimately we define further matched service categories of *t*, namely, *direct_plugin*(*t*), *indirect_plugin*(*t*), *direct_subsumes*(*t*), *indirect_subsumes*(*t*), *sibling*(*t*), *partial-parent*(*t*) and *grandparent*(*t*).

Figure 3 Structure of input index

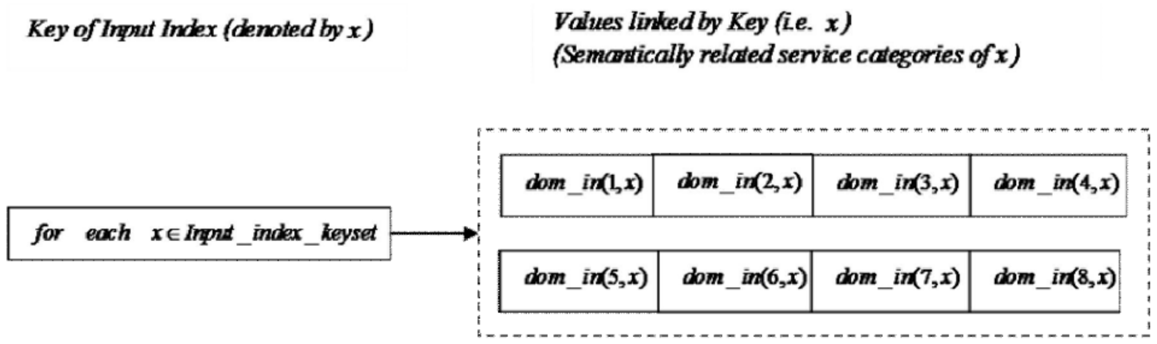

Key of Input Index (denoted by x)
Values linked by Key (i.e. x)
(Semantically related service categories of x)
for each x∈Input_index_keyset
dom_in(1,x)
dom_in(2,x)
dom_in(3,x)
dom_in(4,x)
dom_in(5,x)
dom_in(6,x)
dom_in(7,x)
dom_in(8,x)

Figure 4 Structure of output index

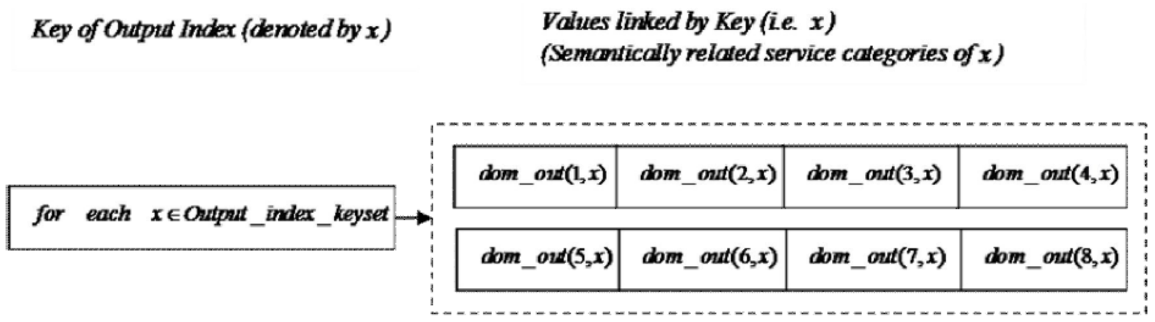
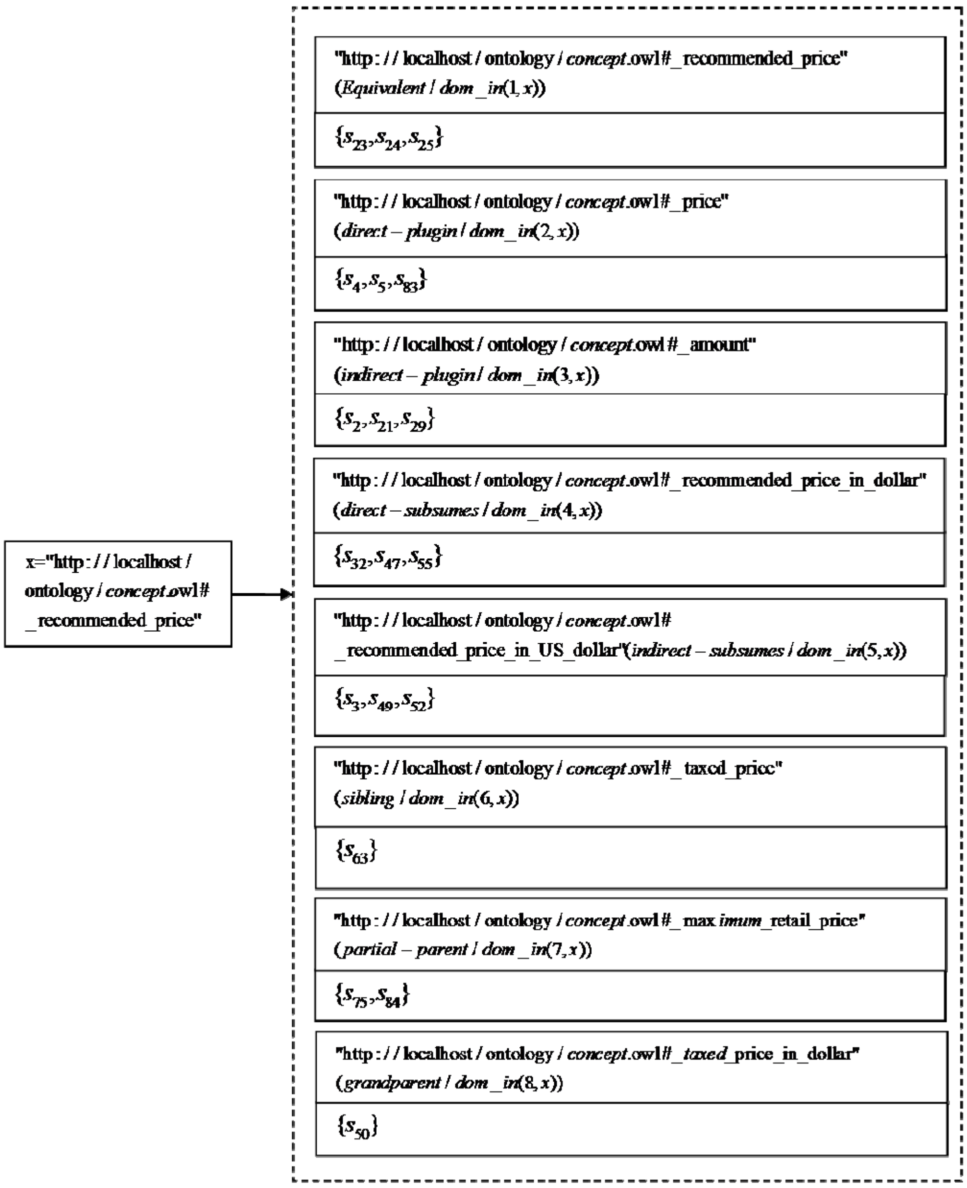

Key of Output Index (denoted by x)
Values linked by Key (i.e. x)
(Semantically related service categories of x)
for each x∈Output_index_keyset
dom_out(1,x)
dom_out(2,x)
dom_out(3,x)
dom_out(4,x)
dom_out(5,x)
dom_out(6,x)
dom_out(7,x)
dom_out(8,x)

Figure 5 Example of a particular key *x* and its service categories in input index


x="http://localhost/ontology/concept.owl#_recommended_price"
"http://localhost/ontology/concept.owl#_recommended_price" (Equivalent / dom_in(1, x))
{s23, s24, s25}
"http://localhost/ontology/concept.owl#_price" (direct – plugin / dom_in(2, x))
{s4, s5, s83}
"http://localhost/ontology/concept.owl#_amount" (indirect – plugin / dom_in(3, x))
{s2, s21, s29}
"http://localhost/ontology/concept.owl#_recommended_price_in_dollar" (direct – subsumes / dom_in(4, x))
{s32, s47, s55}
"http://localhost/ontology/concept.owl#_recommended_price_in_US_dollar" (indirect – subsumes / dom_in(5, x))
{s3, s49, s52}
"http://localhost/ontology/concept.owl#_taxed_price" (sibling / dom_in(6, x))
{s63}
"http://localhost/ontology/concept.owl#_maximum_retail_price" (partial – parent / dom_in(7, x))
{s75, s84}
"http://localhost/ontology/concept.owl#_taxed_price_in_dollar" (grandparent / dom_in(8, x))
{s50}

For each *i*, 1 ≤ *i* ≤ 8, and for each *x* ∈ *Input*(*t*), we define successively *m_dom_in*(*i*, *x*) and for each *i*, 1 ≤ *i* ≤ 8, and for each *y* ∈ *Output*(*t*), we define successively *m_dom_out*(*i*, *y*) as described below.

Case 1: i = 1

$$m\_dom\_in(1,x) = dom\_in(1,x), \quad \forall x \in Input(t) \tag{3}$$

$$m\_dom\_out(1,y) = dom\_out(1,y), \quad \forall y \in Output(t) \tag{4}$$

Case 2: i ≥ 2

Sub Case 2a:

$$\left(\bigcap_{x \in Input(t)} m\_dom\_in(i-1,x)\right) \cap \left(\bigcap_{y \in Output(t)} m\_dom\_out(i-1,y)\right) \neq \phi$$

$$m\_dom\_in(i,x) = dom\_in(i,x), \quad \forall x \in Input(t) \tag{5}$$

$$m\_dom\_out(i,y) = dom\_out(i,y), \quad \forall y \in Output(t) \tag{6}$$

Sub Case 2b:

$$\left(\bigcap_{x \in Input(t)} m\_dom\_in(i-1,x)\right) \cap \left(\bigcap_{y \in Output(t)} m\_dom\_out(i-1,y)\right) = \phi$$

$$m\_dom\_in(i,x) = dom\_in(i,x) \cup m\_dom\_in(i-1,x), \quad \forall x \in Input(t) \tag{7}$$

$$m\_dom\_out(i,y) = dom\_out(i,y) \bigcup m\_dom\_out(i-1,y), \quad \forall y \in Output(t) \tag{8}$$

Now we define various matched service categories for the given task as follows.

$$equal(t) = \left(\bigcap_{x \in Input(t)} m\_dom\_in(1,x) \right) \cap \left(\bigcap_{y \in Output(t)} m\_dom\_out(1,y) \right) \tag{9}$$

$$direct\_plugin(t) = \left(\bigcap_{x \in Input(t)} m\_dom\_in(2,x) \right) \cap \left(\bigcap_{y \in Output(t)} m\_dom\_out(2,y) \right) \tag{10}$$

$$indirect\_plugin(t) = \left(\bigcap_{x \in Input(t)} m\_dom\_in(3,x) \right) \cap \left(\bigcap_{y \in Output(t)} m\_dom\_out(3,y) \right) \tag{11}$$

$$direct\_subsumes(t) = \left(\bigcap_{x \in Input(t)} m\_dom\_in(4,x) \right) \cap \left(\bigcap_{y \in Output(t)} m\_dom\_out(4,y) \right) \tag{12}$$

$$indirect\_subsumes(t) = \left(\bigcap_{x \in Input(t)} m\_dom\_in(5,x) \right) \cap \left(\bigcap_{y \in Output(t)} m\_dom\_out(5,y) \right) \tag{13}$$

$$sibling(t) = \left(\bigcap_{x \in Input(t)} m\_dom\_in(6,x) \right) \cap \left(\bigcap_{y \in Output(t)} m\_dom\_out(6,y) \right) \tag{14}$$

$$partial\text{-}parent(t) = \left(\bigcap_{x \in Input(t)} m\_dom\_in(7,x) \right) \cap \left(\bigcap_{y \in Output(t)} m\_dom\_out(7,y) \right) \tag{15}$$

$$grandparent(t) = \left(\bigcap_{x \in Input(t)} m\_dom\_in(8,x) \right) \cap \left(\bigcap_{y \in Output(t)} m\_dom\_out(8,y) \right) \tag{16}$$

We define one more matched service category called *partial*(t) based on the value of *grandparent*(t).

Let

$$A = \left(\bigcap_{x \in Input(t)} m\_dom\_in(8,x) \right)$$

Let

$$B = \left(\bigcap_{y \in Output(t)} m\_dom\_out(8,y) \right)$$

If *grandparent*(t) ϕ, we define *partial*(t) as follows

Case 1: $A = \phi$; $B = \phi$

$$partial(t) = \phi \tag{17}$$

Case 2: $A = \phi$; $B \neq \phi$

$$partial(t) = B \tag{18}$$

Case 3: $A \neq \phi$; $B = \phi$

$$partial(t) = A \tag{19}$$

Case 4: $A \neq \phi$; $B \neq \phi$

$$partial(t) = A \bigcup B \tag{20}$$

The process of retrieving services is illustrated using a task t having two inputs denoted by *Input*(t) = $\{x_1, x_2\}$ and one output denoted by *Output*(t) = $\{y_1\}$. The inputs x_1 and x_2 of t will be matched with each key of the input index. Similarly, the output y_1 of t will be matched with each key of the output index. The values or (i.e., service categories) mapped by the matched keys of x_1 and x_2 are extracted from the input index and given in Table 1. Similarly, the values mapped by the matched key of y_1 are extracted from the output index and given in Table 2. Further, note that the data given in Tables 1 and 2 are typical values assumed to explain the method of service retrieval.

Table 1 Service categories extracted for x_1 and x_2 from the input index

x_i	$dom\_in(1, x_i)$	$dom\_in(2, x_i)$	$dom\_in(3, x_i)$	$dom\_in(4, x_i)$	$dom\_in(5, x_i)$	$dom\_in(6, x_i)$	$dom\_in(7, x_i)$	$dom\_in(8, x_i)$
x_1	$\{s_6, s_7\}$	$\{s_3\}$	$\{s_{11}, s_{12}\}$	ϕ	$\{s_{14}\}$	$\{s_{13}\}$	$\{s_{17}\}$	$\{s_{19}\}$
x_2	$\{s_8\}$	$\{s_7\}$	$\{s_{11}\}$	$\{s_{14}\}$	$\{s_7\}$	$\{s_{13}, s_{17}\}$	ϕ	$\{s_{19}, s_{23}\}$

Table 2 Service categories extracted for y_1 from the output index

y_i	*dom_out*$(1, y_i)$	*dom_out*$(2, y_i)$	*dom_out*$(3, y_i)$	*dom_out*$(4, y_i)$	*dom_out*$(5, y_i)$	*dom_out*$(6, y_i)$	*dom_out*$(7, y_i)$	*dom_out*$(8, y_i)$
y_1	$\{s_6, s_7\}$	$\{s_3\}$	$\{s_{11}, s_{13}\}$	ϕ	$\{s_{14}\}$	ϕ	$\{s_{17}\}$	ϕ

From Table 1, the service categories, namely, *equal*, *direct-plugin*, *indirect-plugin*, *direct-subsumes*, *indirect-subsumes*, *sibling*, *partial-parent* and *grandparent* mapped by the matched key of x_1, denoted by *dom_in*$(1, x_1)$, *dom_in*$(2, x_1)$, *dom_in*$(3, x_1)$, *dom_in*$(4, x_1)$, *dom_in*$(5, x_1)$, *dom_in*$(6, x_1)$, *dom_in*$(7, x_1)$ and *dom_in*$(8, x_1)$ are retrieved as

$$\begin{aligned} & dom\_in(1, x_1) = \{s_6, s_7\} \\ & dom\_in(2, x_1) = \{s_3\} \\ & dom\_in(3, x_1) = \{s_{11}, s_{12}\} \\ & dom\_in(4, x_1) = \phi \\ & dom\_in(5, x_1) = \{s_{14}\} \\ & dom\_in(6, x_1) = \{s_{13}\} \\ & dom\_in(7, x_1) = \{s_{17}\} \\ & dom\_in(8, x_1) = \{s_{19}\} \end{aligned} \tag{21}$$

Further, from Table 1, the service categories, namely, *equal*, *direct-plugin*, *indirect-plugin*, *direct-subsumes*, *indirect-subsumes*, *sibling*, *partial-parent* and *grandparent* mapped by the matched key of x_2, denoted by *dom_in*$(1, x_2)$, *dom_in*$(2, x_2)$, *dom_in*$(3, x_2)$, *dom_in*$(4, x_2)$, *dom_in*$(5, x_2)$, *dom_in*$(6, x_2)$, *dom_in*$(7, x_2)$ and *dom_in*$(8, x_2)$ are retrieved as

$$\begin{aligned} & dom\_in(1, x_2) = \{s_8\} \\ & dom\_in(2, x_2) = \{s_7\} \\ & dom\_in(3, x_2) = \{s_{11}\} \\ & dom\_in(4, x_2) = \{s_{14}\} \\ & dom\_in(5, x_2) = \{s_7\} \\ & dom\_in(6, x_2) = \{s_{13}, s_{17}\} \\ & dom\_in(7, x_2) = \phi \\ & dom\_in(8, x_2) = \{s_{19}, s_{23}\} \end{aligned} \tag{22}$$

Similarly, from Table 2, the service categories, namely, *equal*, *direct-plugin*, *indirect-plugin*, *direct-subsumes*, *indirect-subsumes*, *sibling*, *partial-parent* and *grandparent* mapped by the matched key of y_1, denoted by *dom_out*$(1, y_1)$, *dom_out*$(2, y_1)$, *dom_out*$(3, y_1)$, *dom_out*$(4, y_1)$, *dom_out*$(5, y_1)$, *dom_out*$(6, y_1)$, *dom_out*$(7, y_1)$ and *dom_out*$(8, y_1)$ are retrieved as

$$\begin{aligned} & dom\_out(1, y_1) = \{s_6, s_7\} \\ & dom\_out(2, y_1) = \{s_3\} \\ & dom\_out(3, y_1) = \{s_{11}, s_{13}\} \\ & dom\_out(4, y_1) = \phi \\ & dom\_out(5, y_1) = \{s_{14}\} \\ & dom\_out(6, y_1) = \phi \\ & dom\_out(7, y_1) = \{s_{17}\} \\ & dom\_out(8, y_1) = \phi \end{aligned} \tag{23}$$

Now how successively various levels of matched services are found out using the proposed method of service retrieval is given below. At first, the *first level modified service categories* of x_1, x_2 and y_1, namely, *m_dom_in*$(1, x_1)$, *m_dom_in*$(1, x_2)$ and *m_dom_out*$(1, y_1)$ are computed using (3) and (4) as Case 1 is satisfied.

$$\begin{aligned} & m\_dom\_in(1, x_1) = \{s_6, s_7\} \\ & m\_dom\_in(1, x_2) = \{s_8\} \\ & m\_dom\_out(1, y_1) = \{s_6, s_7\} \end{aligned}$$

Now, based on the result of intersection taken on the *first level modified service categories* of x_1, x_2 and y_1 the *second level modified service categories* of x_1, x_2 and y_1, namely, *m_dom_in*$(2, x_1)$, *m_dom_in*$(2, x_2)$ and *m_dom_out*$(2, y_1)$ are computed either using *Sub Case 2a* or *Sub Case 2b*.

In this example, the values of *m_dom_in*$(2, x_1)$, *m_dom_in*$(2, x_2)$ and *m_dom_out*$(2, y_1)$ are computed using (7) and (8) as *Sub Case 2b* is satisfied.

$$\begin{aligned} & m\_dom\_in(2, x_1) \\ & = dom\_in(2, x_1) \cup m\_dom\_in(1, x_1) \\ & = \{s_3, s_6, s_7\} \\ & m\_dom\_in(2, x_2) \\ & = dom\_in(2, x_2) \cup m\_dom\_in(1, x_2) \\ & = \{s_8, s_7\} \\ & m\_dom\_out(2, y_1) \\ & = dom\_out(2, y_1) \cup m\_dom\_out(1, y_1) \\ & = \{s_6, s_7, s_3\} \end{aligned}$$

The *third level modified service categories* of x_1, x_2 and y_1, namely, *m_dom_in*(3, x_1), *m_dom_in*(3, x_2) and *m_dom_out*(3, y_1) are computed using (5) and (6) as *Sub Case 2a* is satisfied.

$$m\_dom\_in(3,x_1) = dom\_in(3,x_1) = \{s_{11}, s_{12}\}$$
$$m\_dom\_in(3,x_2) = dom\_in(3,x_2) = \{s_{11}\}$$
$$m\_dom\_out(3,y_1) = dom\_out(3,y_1) = \{s_{11}, s_{13}\}$$

The *fourth level modified service categories* of x_1, x_2 and y_1, namely, *m_dom_in*(4, x_1), *m_dom_in*(4, x_2) and *m_dom_out*(4, y_1) are computed using (5) and (6) as *Sub Case 2a* is satisfied.

$$m\_dom\_in(4,x_1) = dom\_in(4,x_1) = \phi$$
$$m\_dom\_in(4,x_2) = dom\_in(4,x_2) = \{s_{14}\}$$
$$m\_dom\_out(4,y_1) = dom\_out(4,y_1) = \phi$$

The *fifth level modified service categories* of x_1, x_2 and y_1, namely, *m_dom_in*(5, x_1), *m_dom_in*(5, x_2) and *m_dom_out*(5, y_1) are computed using (7) and (8) as *Sub Case 2b* is satisfied.

$$\begin{aligned}&m\_dom\_in(5,x_1)\\&= dom\_in(5,x_1) \cup m\_dom\_in(4,x_1)\\&= \{s_{14}\}\\&m\_dom\_in(5,x_2) = dom\_in(5,x_2) \cup m\_dom\_in(4,x_2)\\&= \{s_{14}, s_7\}\\&m\_dom\_out(5,y_1)\\&= dom\_out(5,y_1) \cup m\_dom\_out(4,y_1)\\&= \{s_{14}\}\end{aligned}$$

The *sixth level modified service categories* of x_1, x_2 and y_1, namely, *m_dom_in*(6, x_1), *m_dom_in*(6, x_2) and *m_dom_out*(6, y_1) are computed using (5) and (6) as *Sub Case 2a* is satisfied.

$$m\_dom\_in(6,x_1) = dom\_in(6,x_1) = \{s_{13}\}$$
$$m\_dom\_in(6,x_2) = dom\_in(6,x_2) = \{s_{13}, s_{17}\}$$
$$m\_dom\_out(6,y_1) = dom\_out(6,y_1) = \phi$$

The *seventh level modified service categories* of x_1, x_2 and y_1, namely, *m_dom_in*(7, x_1), *m_dom_in*(7, x_2) and *m_dom_out*(7, y_1) are computed using (7) and (8) as *Sub Case 2b* is satisfied.

$$\begin{aligned}&m\_dom\_in(7,x_1)\\&= dom\_in(7,x_1) \cup m\_dom\_in(6,x_1)\\&= \{s_{13}, s_{17}\}\\&m\_dom\_in(7,x_2) = dom\_in(7,x_2) \cup m\_dom\_in(6,x_2)\\&= \{s_{13}, s_{17}\}\\&m\_dom\_out(7,y_1)\\&= dom\_out(7,y_1) \cup m\_dom\_out(6,y_1)\\&= \{s_{17}\}\end{aligned}$$

The eighth level modified service categories of x_1, x_2 and y_1, namely, *m_dom_in*(8, x_1), *m_dom_in*(8, x_2) and *m_dom_out*(8, y_1) are computed using (5) and (6) as *Sub Case 2a* is satisfied.

$$m\_dom\_in(8,x_1) = dom\_in(8,x_1) = \{s_{19}\}$$
$$m\_dom\_in(8,x_2) = dom\_in(8,x_2) = \{s_{19}, s_{23}\}$$
$$m\_dom\_out(8,y_1) = dom\_out(8,y_1) = \phi$$

Now, various matched service categories of *t*, namely, *equal*(*t*), *direct_plugin*(*t*), *indirect_plugin*(*t*), *direct_subsumes*(*t*), *indirect_subsumes*(*t*), *sibling*(*t*), *partial-parent*(*t*) and *grandparent*(*t*) are computed using (9) to (16) as follows.

$$equal(t) = \phi$$

$$direct\_plugin(t) = \{s_7\}$$

$$indirect\_plugin(t) = \{s_{11}\}$$

$$direct\_subsumes(t) = \phi$$

$$indirect\_subsumes(t) = \{s_{14}\}$$

$$sibling(t) = \phi$$

$$partial\text{-}parent(t) = \{s_{17}\}$$

$$grandparent(t) = \phi$$

As *grandparent*(*t*) = ϕ, we define *partial*(*t*) using (19) (as Case 3 is satisfied).

$$partial(t) = \{s_{19}\}$$

Thus, service retrieval results in nine different kinds of matched service categories for a given task. Further, *RDoM* can take the values, *equal*, *direct_plugin*, *indirect_plugin*, *direct_subsumes*, *indirect_subsumes*, *sibling*, *partial-parent* and *grandparent* (here *partial* or *none* can also be included). Based on the strictness of similarity requirements of applications, the value of *RDoM* is specified and accordingly matched results up to the given *RDoM* are returned. For example, if *RDoM* is *direct_subsumes*, then matched results up to *direct_subsumes* (including *direct_subsumes*) are returned.

3.2 Non-functional selection

For each task in the workflow, its respective service class is discovered using functional discovery. In non-functional selection, the best service for each task is selected from its respective service class based on non-functional attributes of services. As mentioned earlier, to reduce the time complexity of global planning methods, it is proposed to optimise the non-functional stage using local selection method. Local selection method divides the problem of selecting best service combination for a given workflow into *n* sub problems where *n* denotes the number of tasks present in the workflow. Besides QoS constraints, service consumers may also specify their preferences over different QoS attributes as in the query: *find a travel plan service to plan Malaysian Package trip such that the cost of service is <=\$500 and response time is <10 seconds with 80% preference to response time and 20% preference to cost.* The user-defined preferences are captured in the form of a utility function which is used to select the best service of a task.

The non-functional selection is carried out in two steps, namely, decomposition of constraints and service selection. In decomposition of constraints, the sub problems are constructed by decomposing the given global QoS constraints into local constraints and assigning the local constraints to individual tasks. In service selection, each sub problem is resolved to find a service of maximum utility as the best service for a particular task from its respective service class subject to its local constraints.

3.2.1 Decomposition of constraints

In decomposition of constraints each given global constraint is decomposed into local constraints and the local constraints are assigned to the tasks present in the workflow. The proposed method of decomposing global constraints into local constraints is based on the extreme (minimum and maximum) values of QoS attributes of the service classes and the global constraints.

The method of decomposing constraints is explained using sequential workflows and it can be employed to any combinational workflow after the conversion of combinational workflow into sequential workflow. To show the implementation of the proposed method of decomposition to combinational workflows, a typical case, namely, *decomposing constraints to a combinational workflow having parallel units* is presented at the end of this sub section. As negative attributes (negative attributes are attributes whose values should be minimised) such as response time and cost gain much significance in service-based applications, the process of decomposition is described using any k^{th} negative attribute denoted by q_k and its global constraint denoted by G_k.

A service class contains many functionally similar services with varying QoS attributes. So, the value of a QoS attribute of a service class ranges from a minimum to a maximum. Let $Q_{\max}(j, k)$ denote the maximum value of q_k of j^{th} service class. Let $Q_{\min}(j, k)$ denote the minimum value of q_k of j^{th} service class. Now the QoS attributes of workflow are computed based on the QoS attributes of service classes. Let $Q'_{\max}(k)$ and $Q'_{\min}(k)$ represent the maximum and minimum value of q_k of the given workflow respectively. The values of $Q'_{\max}(k)$ and $Q'_{\min}(k)$ are computed using

$$Q'_{\max}(k) = \sum_{j=1}^{n} Q_{\max}(j,k) \tag{24}$$

$$Q'_{\min}(k) = \sum_{j=1}^{n} Q_{\min}(j,k) \tag{25}$$

For a given global constraint, the local constraints are computed in an iterative manner. The iteration starts with initial values assigned to local constraints and it continues till the sum of local constraints is within 90% to 100 % of the global constraint (i.e., the given global constraint is utilised to its 90% to 100%). The iteration process is given below.

Let *constraint*(j, k) denote the local constraint of q_k of j^{th} service class.

The initial values of constraints of q_k for all service classes are computed using

$$\begin{aligned} &\textit{constraint}(j,\ k) \\ &= \frac{Q_{\min}(j,k) + Q_{\max}(j,k)}{2}, \quad 1 \le j \le n \end{aligned} \tag{26}$$

After computing the initial values of constraints, the values are summed and the sum is compared against the condition, $0.9G_k \le \sum_{j=1}^{n} \textit{constraint}(j,\ k) \le G_k$. The comparison results in three cases.

Case 1: $\sum_{j=1}^{n} \textit{constraint}(j,\ k) < 0.9G_k$

In this case, for each j^{th} service class, the value *constraint*(j, k) is increased by 10% of its current value. Then, the newly computed constraint values are summed and the sum is checked against the condition, $0.9G_k \le \sum_{j=1}^{n} \textit{constraint}(j,\ k) \le G_k$. This process is repeated until $\sum_{j=1}^{n} \textit{constraint}(j,\ k)$ satisfies the above condition.

Case 2: $\sum_{j=1}^{n} \textit{constraint}(j,\ k) > G_k$

In this case, for each j^{th} service class, the value *constraint*(j, k) is decreased by 10% of its current value. Then, the newly computed constraint values are summed and the sum is checked against the condition, $0.9G_k \le \sum_{j=1}^{n} \textit{constraint}(j,\ k) \le G_k$. This process is repeated until $\sum_{j=1}^{n} \textit{constraint}(j,\ k)$ satisfies the above condition.

Case 3: $0.9G_k \le \sum_{j=1}^{n} \textit{constraint}(j,\ k) \le G_k$

In this case, the initial values of constraints satisfy the condition, $0.9G_k \le \sum_{j=1}^{n} \textit{constraint}(j,\ k) \le G_k$ and there is no further change in the values of constraints.

The pseudo code describing the above cases is given below.

```
If ($\sum_{j=1}^{n} constraint(j, k) < 0.9G_k$)
{
    while ($\sum_{j=1}^{n} constraint(j, k) < 0.9G_k$)
    {
        for ($j = 1; j \le n; j = j + 1$)
        //add 10% of constraint(j, k) to constraint(j, k)
        $constraint(j, k) = constraint(j, k) + 0.1 \times constraint(j, k)$
    }
}
Elseif ($\sum_{j=1}^{n} constraint(j, k) > G_k$)
{
    while ($\sum_{j=1}^{n} constraint(j, k) > G_k$)
    {
        for ($j = 1; j \le n; j = j + 1$)
        //deduct 10% of constraint(j, k) from constraint(j, k)
        $constraint(j, k) = constraint(j, k) - 0.1 \times constraint(j, k)$
    }
}
else
{//no change in the initial values of the constraints}
```

At last one can verify that the condition $0.9G_k \le \sum_{j=1}^{n} constraint(j, k) \le G_k$ is satisfied.

Thus the given global constraint of q_k is decomposed into local constraints and the local constraints are assigned to all service classes present in the workflow.

Further, as mentioned earlier, how the proposed method of decomposing constraints is implemented to a combinational workflow having parallel units is described as a typical case.

3.2.1.1 Typical case: decomposing constraints to a combinational workflow having parallel units

Consider a typical combinational workflow denoted by *W* having parallel units (or AND units) as given in Figure 6. A parallel unit basically consists of more than one path of tasks and all paths will be executed simultaneously. As in Figure 6, *W* contains *x* number of sequential tasks, denoted by $t_1, t_2, t_3, ..., t_x$ and *y* number of parallel units denoted by $u_1, u_2, u_3, ..., u_y$. Further, in Figure 6, 'AS' denotes AND split and 'AJ' denotes AND join. The decomposition of constraints to all tasks of the given workflow is performed using three steps, namely, conversion of combinational workflow into its equivalent sequential workflow, decomposition of constraints to the converted sequential workflow and derivation of constraints to all tasks of the given combinational workflow.

Step 1: Conversion of combinational workflow into its equivalent sequential workflow

The given workflow is converted into its equivalent sequential workflow by converting each parallel unit into its equivalent sequential task. The conversion of a parallel unit into its sequential task is based on the method of computing QoS attributes for parallel unit.

Consider a typical parallel unit, *u* as given in Figure 7. Let *l* denote the number of paths in *u*. Let m_i, $1 \le i \le l$ be the number of tasks in the ith path. Let 'p_i' denote the ith path in *u*. Let t_{ij} denote the jth task of p_i. Let denote q_k denote kth negative attribute. Let $\min\_q_k(p_i)$ and $\max\_q_k(p_i)$ denote the minimum and maximum value of q_k of p_i. Let $\min\_q_k(t_{ij})$ and $\max\_q_k(t_{ij})$ denote the minimum and maximum values of q_k of t_{ij}.

The values of $\min\_q_k(p_i)$ and $\max\_q_k(p_i)$ are calculated using

$$\min\_q_k(p_i) = \sum_{j=1}^{m_i} \min\_q_k(t_{ij}), \quad i = 1, 2, \ldots, l \tag{27}$$

$$\max\_q_k(p_i) = \sum_{j=1}^{m_i} \max\_q_k(t_{ij}), \quad i = 1, 2, \ldots, l \tag{28}$$

Let $\min\_q_k(u)$ and $\max\_q_k(u)$ denote the minimum and maximum value of q_k of *u*. The values of $\min\_q_k(u)$ and $\max\_q_k(u)$ are computed using

$$\min\_q_k(u) = \max\left\{\min\_q_k(p_i) \middle| 1 \le i \le l\right\} \tag{29}$$

$$\max\_q_k(u) = \max\left\{\max\_q_k(p_i) \middle| 1 \le i \le l\right\} \tag{30}$$

Figure 6 Typical combinational workflow having parallel units (W)

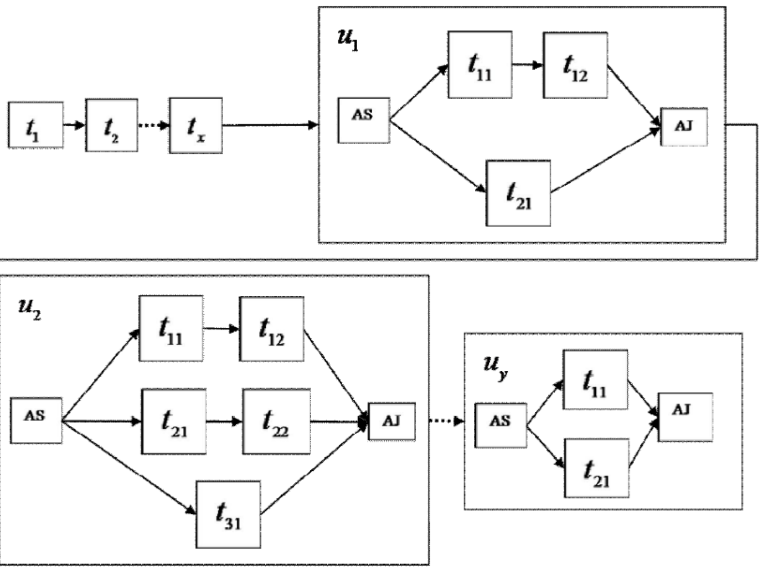


Figure 7 Typical parallel unit (u)

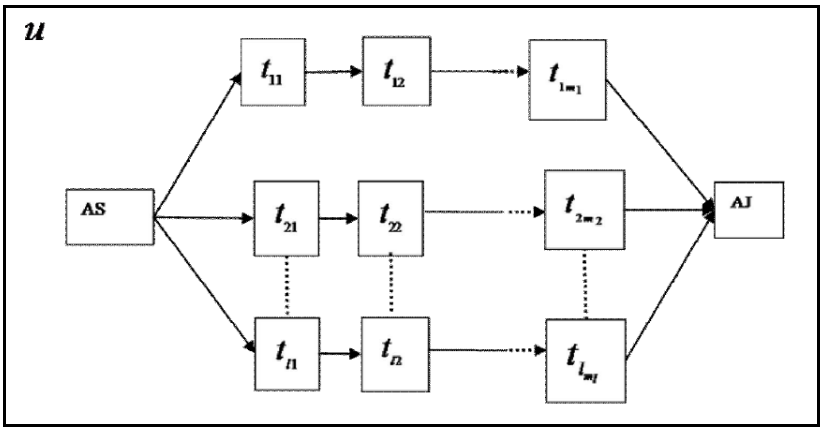
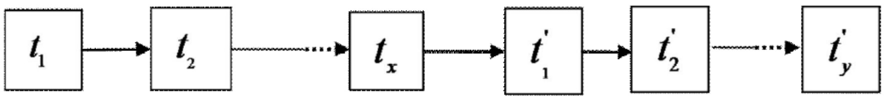


Figure 8 Converted sequential workflow (W′)



Now, the parallel unit u is replaced by its equivalent sequential task called converted-task whose minimum and maximum values of q_k are equal to min_$q_k(u)$ and max_$q_k(u)$ respectively given by (29) and (30).

Step 2: Decomposition of constraints to the converted sequential workflow

The previous step results in a sequential workflow, denoted by W' which is equivalent to the given combinational workflow. As shown in Figure 8, W' contains a set of original sequential tasks present in the given workflow and a set of converted-tasks. Let t_1, t_2, t_3, ..., t_x denote the original sequential tasks and t_1', t_2', t_3', ..., t_y' denote the converted-tasks in W'. In this step, the given global constraint of q_k, denoted by G_k is decomposed into local constraints and assigned to tasks and converted-tasks present in W' using the proposed method of decomposition.

Step 3: Derivation of constraints to all tasks of the given combinational workflow

Each converted-task is associated with its respective parallel unit. Let us consider a converted-task t' in W'. Let u denote the parallel unit associated with t'. Let *constraint_*$q_k(t')$ denote the constraint of q_k of t'. Let *constraint_*$q_k(u)$ denote the constraint of q_k of u. As u is equivalent to t', one can write

$$constraint\_q_k(u) = constraint\_q_k(t') \qquad (31)$$

Now from the *constraint_*$q_k(u)$, the constraints of paths present in the unit are computed. As all the paths of tasks present in a parallel unit are executed in parallel, the constraint of q_k of each i^{th} path, is same as *constraint_q*$_k$(*u*). Let *constraint_q*$_k$(p_i) denote the constraint of q_k of p_i which is computed as

$$constraint\_q_k\left(p_i\right) = constraint\_q_k(u),\ \ i = 1, 2, \ldots, l$$

Now, the constraints of q_k of all tasks present in p_i are computed from *constraint_q*$_k$(p_i) using the proposed method of decomposition.

Thus, the proposed method of decomposition can be implemented to combinational workflows having parallel units.

3.2.2 Service selection

After assigning local constraints to individual tasks of the workflow, each sub problem is formulated to select a service of maximum utility as the best service for a particular task from its respective service class subject to its local constraints. In our work, it is proposed to adopt the method presented in Mohammad and Thomas (2009) for the computation of utility. In this method, preferences of users over different QoS attributes are captured as a utility function using simple additive weighting (SAW) technique (Paul and Ching-Lai, 1995). The method computes the utility of say an i^{th} service of j^{th} service class which satisfies the local constraints of its service class (i.e., j^{th} service class) using

$$U\left(s_{ji}\right) = \sum_{k=1}^{r} \frac{Q_{\max}(j,k) - q_k\left(s_{ji}\right)}{Q'_{\max}(k) - Q'_{\min}(k)} \times w_k \tag{32}$$

In (32), $Q'_{\max}(k)$ represents the maximum value of q_k of the given workflow (the value of $Q'_{\max}(k)$ is computed as sum of maximum values of q_k of all service classes) and $Q'_{\min}(k)$ represents the minimum value of q_k of the given workflow (the value of $Q'_{\min}(k)$ is computed as the sum of minimum values of q_k of all service classes). Further, in (32), $Q_{\max}(j, k)$ represents the maximum value of q_k of j^{th} service class, $q_k(s_{ji})$ represents the value of q_k of i^{th} service of j^{th} service class and w_k represents the weight of q_k. The utility function is subject to the condition, $\sum_{k=1}^{r} w_k = 1$ where r denote the number of negative attributes.

The utility of each service of a service class which satisfies the local constraints of that service class is computed using (32). For each task, the service which produces the maximum utility is found out as the best service from the utility values of all services of that service class using searching technique. Further, indexing of QoS values based on *service ID* is used to speed up the searching while finding the best service.

4 Experimentation

4.1 Implementation setup

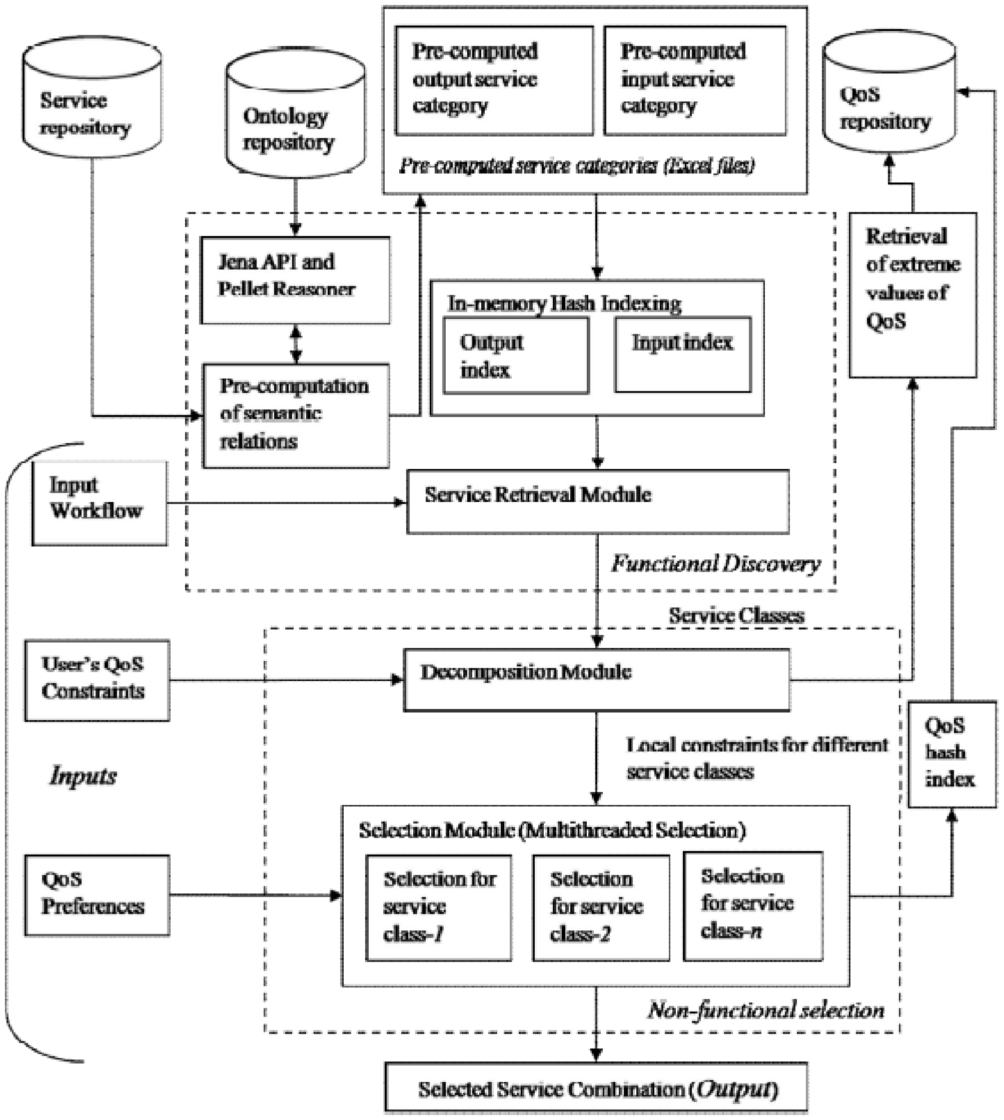
The proposed approach is implemented as in Figure 9. The two stages of the proposed approach are shown as two major parts (functional discovery and non-functional selection) in Figure 9. In functional discovery, prior to querying (i.e., prior to the arrival of query/workflow), semantic relations among input and output parameters of all services in the *service repository* are computed using *Jena API and Pellet reasoner*. Semantic relations among the input parameters of all services are computed as *input service category* and the semantic relations among the output parameters of services are computed as *output service category*. The input and output service categories are archived in Excel files. At the start up of concerned composition-based application itself, two hash-based indices, namely, *output index* and *input index* are created using the pre-computed output and input service categories. The output and input parameters of services are used as keys of output index and input index respectively. During querying, the input workflow for which discovery has to be done is given as input to *service retrieval module*. This module interacts with the indices and retrieves a set of service classes.

The service classes obtained in functional discovery are given as inputs to *decomposition module* of non-functional selection along with the global constraints of users. While decomposing constraints, the extreme values of QoS attributes of different service classes are retrieved from *QoS repository* and retrieval of QoS is an external process to service selection. As the selection of best service of each task is independent, the selection of best services for all tasks of the workflow is executed simultaneously using multiple threads by the *selection module*. Ultimately the best services selected for all tasks are combined and returned as the best service combination for the given workflow.

4.2 Objectives and test collection

There are five objectives of experimentation. The first one is to find the time taken by the proposed method for functional discovery and compare it with conventional approaches. The second one is to find the time taken by the proposed method for non-functional selection and compare it with conventional approaches. The third objective is a special case which tests the computation time taken for decomposing constraints to combinational workflows having parallel units. The fourth one is to test the quality of results produced in functional discovery and non-functional selection of the proposed approach against the standards. The fifth objective is to compare the results obtained using proposed approach with existing similar approaches.

Figure 9 Implementation setup of the proposed method


Service repository
Ontology repository
Pre-computed output service category
Pre-computed input service category
Pre-computed service categories (Excel files)
QoS repository
Jena API and Pellet Reasoner
Pre-computation of semantic relations
In-memory Hash Indexing
Output index
Input index
Retrieval of extreme values of QoS
Input Workflow
Service Retrieval Module
Functional Discovery
Service Classes
User's QoS Constraints
Decomposition Module
Local constraints for different service classes
QoS hash index
Inputs
Selection Module (Multithreaded Selection)
Selection for service class-1
Selection for service class-2
Selection for service class-n
QoS Preferences
Non-functional selection
Selected Service Combination (Output)

Two test collections, namely, Test_Collection_1 and Test_Collection_2 are constructed to conduct experiments related to functional discovery. Test_Collection_1 is constructed using 800 services collected from the publicly available OWL-S service retrieval test collection Version-3(OWLS-TC3). Using Test_Collection_1 as base, Test_Collection_2 containing 10,000 services is constructed. The above test collections contain services from 7 different domains, namely, education, medical care, food, travel, communication, economy and weapons. Semantic relations among input and output parameters are pre-computed and service categories are archived.

To conduct experiments for selection stage, QoS dataset from http://www.uoguelph.ca/~qmahmoud/qws/index.html/ is used. This dataset contains nine QoS attributes of 2,500 real web services. The QoS attributes include response time, availability, throughput, likelihood of success, reliability, compliance, best practices, latency and documentation. Using this dataset as base, QoS data have been created for a collection of 10,000 services. The QoS data are archived in Excel.

Experiments are performed on a laptop with Intel Pentium(R) Dual-Core, 2.20 GHz CPU, 3.0 GB memory and Windows 7 Ultimate Operating System. The proposed approach is implemented using J2EE environment.

4.3 Results and discussions

4.3.1 Computation time of functional discovery

The computation time of functional discovery of the *proposed method* (*hash index-based*) is analysed by varying the number of services. To study the influence of indexing in computation time, the computation time of functional discovery of the proposed method is compared with two other methods, namely, *sequential method and sequential method with pre-computed semantic relations*. Three experiments are conducted to study the computation time of functional discovery using the above methods.

In an experiment computation time of functional discovery is analysed using *sequential method* by varying the number of services. To find matched services of a given task, the inputs and outputs of the task is matched with that of each available service in the service repository. During matching the semantic relations among the concepts of the task and available services are found out using Pellet reasoner. In this method, the semantic reasoning and matching is performed in online with the query. This experiment is tested using Test_Collection_1. The computation time of functional discovery of sequential method with respect to number services is given in Table 3 as well as in Figure 10. From Table 3 and Figure 10, the computation time of functional discovery is found to increase linearly with increase in number of services. Further, the average computation time involved in single semantic match is found to be 9.7239 seconds. In real business applications, as several services have to be discovered and composed in complex chain, invoking semantic reasoner in online with the query becomes impractical. Hence, we suggest avoiding semantic reasoning during querying.

Another experiment is conducted using sequential method with pre-computed semantic relations. In this method, two pre-computed service categories, namely, *output service category* and *input service category* are created prior to querying. In the *output service category*, against each output parameter, its various semantically related matches, namely *equal, direct_plugin*, *indirect_plugin*, *direct_subsumes*, *indirect_subsumes*, *sibling*, *partial-parent* and *grandparent* are stored. Similarly, in the *input service category*, against each input parameter, its various semantically related matches are stored. The pre-computed service categories may be archived in different formats such text files, Excel files, database files, etc. In our experiment, the pre-computed values are stored in Excel file. Time taken for discovering matched services by *sequential method with pre-computed semantic relations* is split into two components, namely, time required to load the pre-computed semantic relations and time taken to retrieve the matched services. The computation time of *sequential method with pre-computed semantic relations* with respect to number of services (using Test_Collection_2) is given in Table 4. From Table 4, both time taken for loading the semantic relations and time taken for retrieving matched services are found to increase linearly with number of services. The influence of loading time can be prevented by loading the semantic relations prior to querying itself. The more influential part is retrieval time which is found to increase with number of services.

Table 3 Time taken for functional discovery by sequential method (using Test_Collection_1)

# of services	*Time taken for discovery (in seconds)*
100	482
200	1,195
300	2,396
400	3,755
500	5,275
600	7,378
700	9,185
800	10,923

Figure 10 Time taken for functional discovery by sequential method

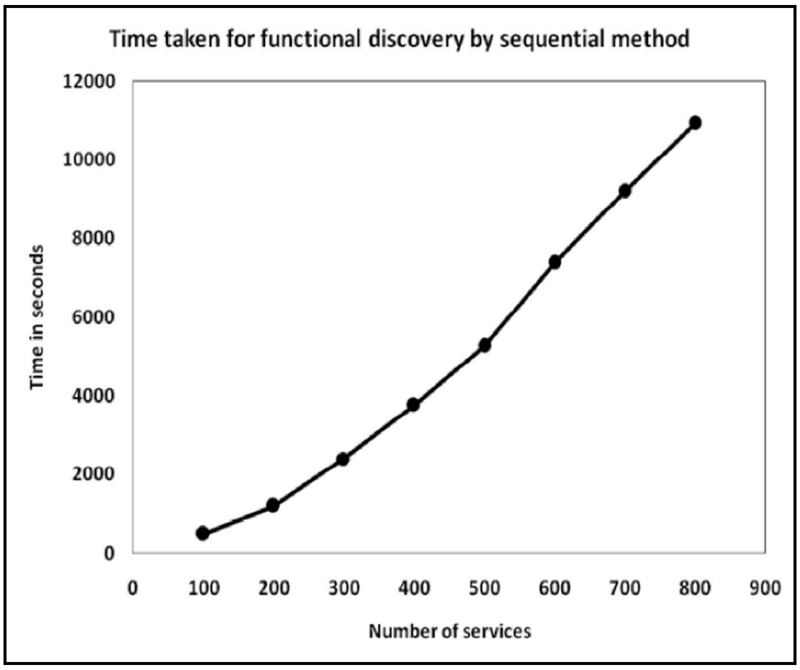


Table 4 Time taken for functional discovery by sequential method with pre-computed semantic relations (using Test_Collection_2)

# of services	*Loading time of pre-computed service categories (in milli seconds)*	*Retrieval time (in milli seconds)*
1,000	2,926	75.872
2,000	4,327	90.522
3,000	5,415	102.82
4,000	7,855	121.954
5,000	9,257	134.968
6,000	11,357	149.102
7,000	12,660	165.314
8,000	13,957	187.418
9,000	15,073	204.342
10,000	17,125	221.665

Another experiment is conducted to find the computation time of discovering services using the proposed method. In this method, immediately after loading the pre-computed service categories, in-memory hash-based input index and output index are created with input and output parameters as keys. In either index, each key of the index is mapped to its semantically related service categories. During querying matched services are retrieved from indices using *service retrieval* described in Subsection 3.1.2. The time taken for functional discovery by the proposed method is split into three components, namely, time required to load the pre-computed service categories, time required to create hash-based indices and time required to retrieve matched services using indices. The time taken for functional discovery by the proposed method with respect to number of services is given in Table 5. From Table 5, the time taken to load the pre-computed service categories and time taken to create hash indices are found to increase with respect to number of services. As loading of pre-computed service categories and creation of indices can be performed prior to querying their effect on computation time can be eliminated. Out of the three components, the most influential part is the time taken for retrieval of services using indices. From Table 5, the time taken to retrieve matched services using indices is found to remain almost constant with respect to number of services. This is an interesting feature achieved with hash-based indices. Also, for comparison, the time taken for retrieval of matched services by the *sequential method with pre-computed semantic relations* and the *proposed method* with respect to number of services is presented in Figure 11. From Figure 11, the retrieval time taken by the proposed method is found to be very low when compared to that of sequential method with pre-computed semantic relations. Further, the retrieval time of the proposed method remains constant with respect to number of services. Here the hashing of indices helps in achieving time complexity of $O(1)$ irrespective of number of services.

4.3.2 Computation time of non-functional selection

The time characteristics of non-functional selection are evaluated by studying the time characteristics of decomposition of constraints and service selection. The proposed methods for decomposing constraints and selecting services are implemented in Java and a series of experiments have been conducted. The time taken for decomposing constraints is independent of number of services present in a service class as the method is based on the extreme values of QoS attributes of service classes. Hence, the time taken for decomposing constraints is analysed by varying the number of service classes in a workflow. Time taken for decomposing a given global constraint to a sequential workflow by varying the number of service classes from 10 to 100 in steps of 10 is given in Table 6 and Figure 12. From Table 6 and Figure 12, the time taken for decomposing constraints with respect to number of service classes is found to increase very slowly. For example, when the number of service classes is increased from 10 to 100, the time taken for decomposing a constraint is found to increase from 376 micro seconds to 726 micro seconds.

Table 5 Time taken for functional discovery by proposed method (hash index-based) using Test_Collection_2

# of services	*Loading time of pre-computed service categories (in milli seconds)*	*Time required to create hash indices (in milli seconds)*	*Retrieval time using indexing (in milli seconds)*
1,000	2,926	1,338	1.025
2,000	4,327	2,066	1.009
3,000	5,415	3,072	1.121
4,000	7,855	3,301	0.988
5,000	9,257	3,736	1.045
6,000	11,357	4,174	1.110
7,000	12,660	4,944	0.974
8,000	13,957	5,415	1.09
9,000	15,073	5,997	0.935
10,000	17,125	6,529	0.992

Figure 11 Time taken for retrieving services (during functional discovery) by proposed method and sequential method with pre-computed semantic relations

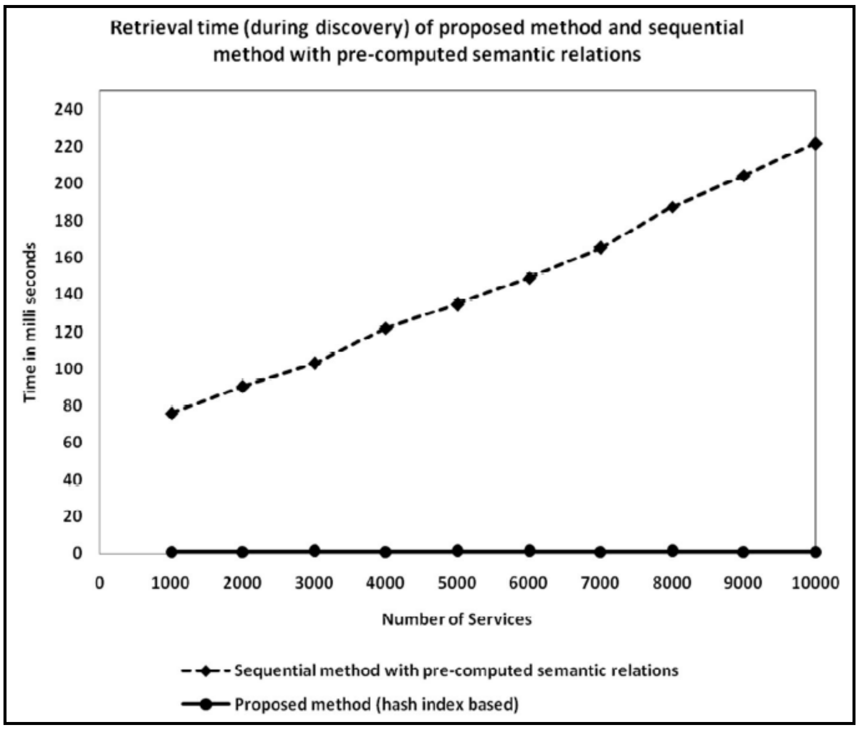


Figure 12 Time taken for decomposing constraints to a sequential workflow with respect to number of service classes

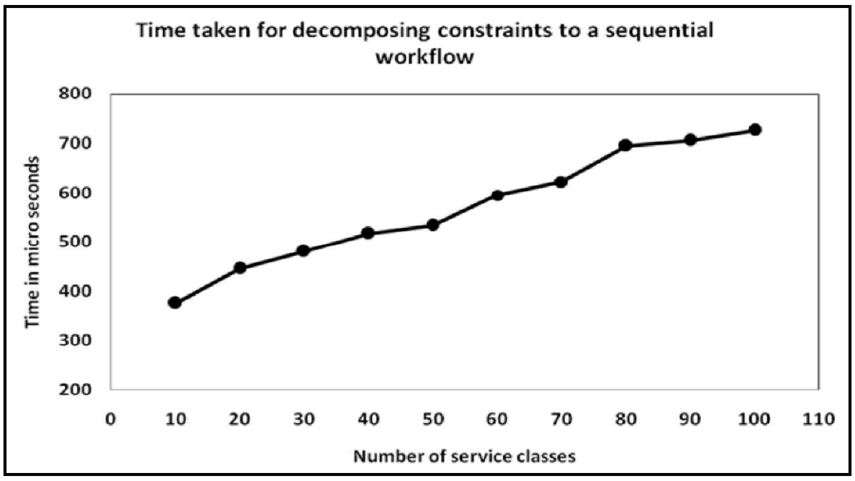


Table 6 Time taken for decomposing constraints to sequential workflow with respect to number of service classes

Number of service classes	*Time taken for decomposition (in micro seconds)*
10	376
20	445
30	480
40	517
50	533
60	595
70	622
80	695
90	706
100	726

The time taken for selecting the best service for a service class depends on the number of services present in that service class (as selection is performed by checking the QoS attributes of individual services against local constraints). As service selection is performed simultaneously for all service classes in multithreaded fashion, the time taken for service selection will be equal to the time taken for selecting the best service for a single service class.

Just prior to selection, the QoS data of services which are archived in Excel are brought to the concerned service-based application. To enhance the performance of search process, the QoS values of services are hash-indexed and this index is called as *QoS hash index* as in Figure 9. In *QoS hash index*, the *service_ID* is used as the key and each key is mapped to its QoS values. During non-functional selection, the QoS values of each service present in a service class are checked for their compliance with the local constraints. If a service is found to satisfy the local constraints of its service class, then its utility is computed. By checking the utilities of all services, the service having the maximum utility is found out as the best service for a service class.

To provide an insight on how hash indexing helps in achieving constant time while accessing QoS values of a service, two experiments are conducted. In one experiment, time taken for loading the QoS values into memory and time taken for retrieving a service using *sequential search* (i.e., without hash index) are analysed. The time taken for retrieving first, middle and last services by varying the number of services from 1,000 to 10,000 in steps of 1,000 is given in Table 7 along with time taken for loading QoS values. In another experiment, the time taken for loading QoS values into memory, time taken for creating QoS hash index and time taken for accessing a service from the QoS hash index with respect to number of services are analysed. The time taken for retrieving first, middle and last services by varying the number of services from 1,000 to 10,000 in steps of 1,000 using *QoS hash index* (*index-based search*) is given in Table 8 along with time taken for loading QoS values and time taken for creating indices. While comparing Tables 7 and 8, QoS hash is found to help in achieving almost constant access time with respect to number of services. For comparison, the time taken for accessing an element (say, last element) with respect to number of services, using sequential method and index-based method is given in Figure 13. From Figure 13, it is found that access time of sequential method increases with respect to number of services whereas the access time of index-based method remains constant with respect to number of services.

Table 7 Time required for retrieving a service using sequential search (without QoS hash)

# of services	*Time to load QoS (in milli seconds)*	*Retrieval time (in micro seconds)*		
		First service	*Middle service*	*Last service*
1,000	254	31	103	181
2,000	273	32	194	298
3,000	287	33	363	705
4,000	280	37	489	967
5,000	285	38	579	1,225
6,000	298	31	568	1,213
7,000	304	32	655	1,389
8,000	307	31	670	1,335
9,000	329	31	754	1,576
10,000	331	35	1,035	2,134

Table 8 Time required for retrieving a service using QoS hash index

# of services	*Time taken to load QoS values (in milli seconds)*	*Time taken to create hash (in micro seconds)*	*Time taken to retrieve a service from hash (in micro seconds)*		
			First service	*Middle service*	*Last service*
1,000	254	28,293	5	7	4
2,000	273	33,679	3	7	3
3,000	287	57,233	4	6	5
4,000	280	56,639	4	6	4
5,000	285	54,282	4	6	5
6,000	298	50,584	3	8	5
7,000	304	55,028	4	8	4
8,000	307	58,722	5	7	4
9,000	329	63,897	6	7	3
10,000	331	85,311	4	7	4

Figure 13 Time taken for accessing an element using hash index-based method (QoS hash index) and sequential method

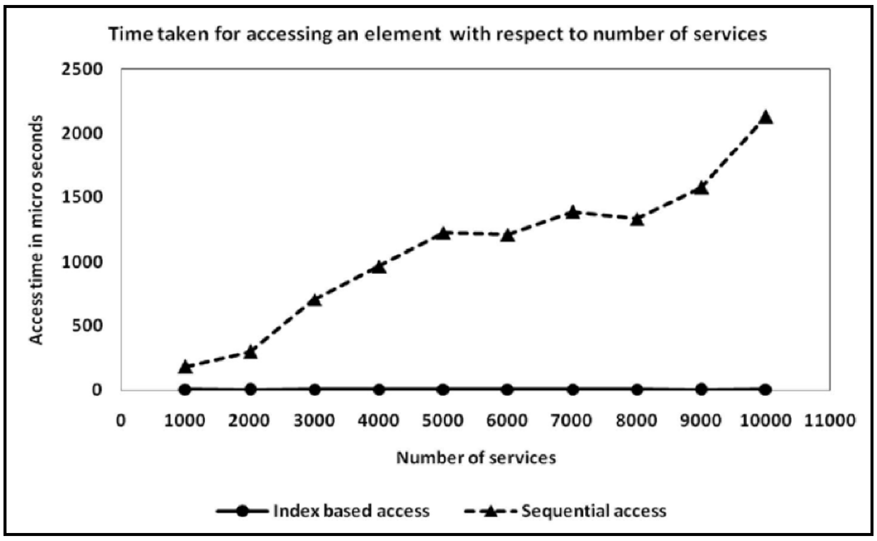
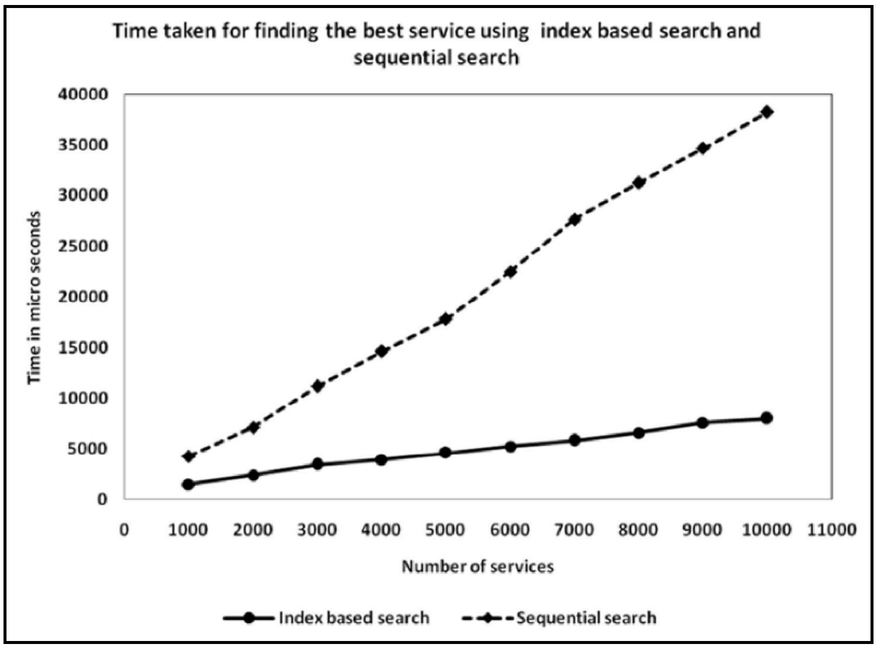


Table 9 Time taken for finding the best service by sequential search and index-based search (using QoS hash)

# of services	*Time to load QoS (in milli seconds)*	*Time to create hash (in micro seconds)*	*Time taken to select the best service*	
			QoS hash index-based search (in micro seconds)	*Sequential search (in micro seconds)*
1,000	254	28,293	1,489	4,275
2,000	273	33,679	2,383	7,161
3,000	287	57,233	3,403	11,201
4,000	280	56,639	3,907	14,642
5,000	285	54,282	4,586	17,752
6,000	298	50,584	5,238	22,468
7,000	304	55,028	5,831	27,661
8,000	307	58,722	6,589	31,217
9,000	329	63,897	7,588	34,625
10,000	331	85,311	7,991	38,197

Figure 14 Time taken for finding the best service using index-based search (QoS hash index) and sequential search



Also, the time taken for finding the best service from a service class with respect to number of services using sequential search and index-based search is given in Table 9 and Figure 14. Sequential search involves two time components, namely, time taken to load the QoS values and time taken to search whereas index-based search involves three components, namely, time taken to load the QoS values, time taken to create QoS hash and time taken to search using index. Both the methods have loading component in common and as loading of QoS is done prior to querying, it will not affect the time taken for selecting the best service during querying. Similarly in the case of index-based search, as index is created prior to querying, it will not affect the time taken for selecting the best service. The influential component is the time taken for searching and finding the best service based on constraints and utility. From Table 9 and Figure 14, the time taken to find the best service using index-based search is found to be very low when compared to sequential search. From Table 9, when the number of services is increased from 1,000 to 10,000, the time taken for selecting the best service using index-based search is found to increase very slowly (from 1.489 milli seconds to 7.991 milli seconds) when compared to that of sequential search (from 4.275 milli seconds to 38.197 milli seconds). As the variation in time taken for selecting the best service using the index-based search is very slow with respect to number of services, it is suggested to use hash-based index while searching for the best service.

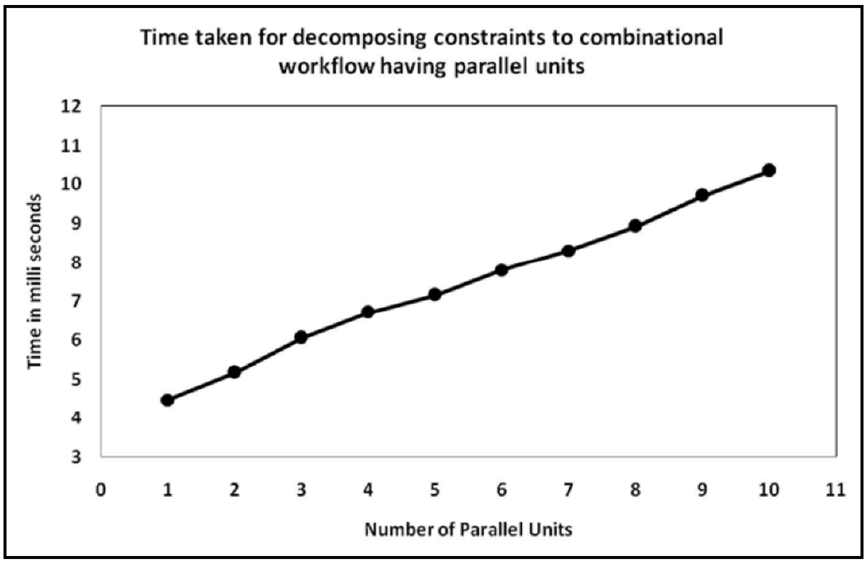
4.3.3 *Computation time of decomposing constraints to combinational workflows having parallel units*

To study the time variation in decomposing constraints to a combinational workflow having parallel units, the number of parallel units in the workflow is increased from 1 to 10. Each parallel unit contains two paths of tasks and each path contains five tasks/service classes. The variation in time taken for decomposing constraints to a combinational workflow having parallel units with respect to number of parallel units is given in Table 10 and Figure 15. From Table 10 and Figure 15, the computation time of decomposing constraints with respect to number of parallel units is found to increase very slowly. For example, when the number of parallel units is increased from 1 to 10, the time taken for decomposing constraints is found to increase from 4.4835 milli seconds to 10.3435 milli seconds.

Table 10 Time taken for decomposing constraints to a combinational workflow having parallel units

# of parallel units	*Time taken (in milli seconds)*
1	4.4835
2	5.1725
3	6.0535
4	6.712
5	7.1605
6	7.8105
7	8.3105
8	8.9225
9	9.7155
10	10.3435

Figure 15 Time taken for decomposing constraints to a combinational workflow having parallel units



4.3.4 Testing the accuracy of results of proposed approach

The accuracy of results obtained with functional discovery of the proposed approach is compared with standard sequential method using two evaluation measures, namely, precision and recall. A set of test queries has been chosen and for each test query its matched services are discovered using sequential method and proposed method of functional discovery. For a query, let N_{rr}^{s} and N_{r}^{s} denote the number of relevant services retrieved and the number of services retrieved using sequential method. Similarly, for a query, let N_{rr}^{p} and N_{r}^{p} denote the number of relevant services retrieved and the number of services retrieved using proposed method respectively. Further, for a query, let N_{tr} denotes the actual number of relevant services present in the test collection. For each query, let P_s and P_p denote the precision of sequential and proposed methods respectively. For each query, let R_s and R_p denote the recall of sequential and proposed methods respectively. The values of P_s, R_s, P_p and R_p are computed as follows.

$$P_s = \frac{N_{rr}^{s}}{N_{r}^{s}} \times 100 \tag{33}$$

$$R_s = \frac{N_{rr}^{s}}{N_{tr}} \times 100 \tag{34}$$

$$P_p = \frac{N_{rr}^{p}}{N_{r}^{p}} \times 100 \tag{35}$$

$$R_p = \frac{N_{rr}^{p}}{N_{tr}} \times 100 \tag{36}$$

Eight test queries have been chosen. Matched services for these queries have been found out using both the methods. The values of N_{rr}^{s}, N_{r}^{s}, N_{rr}^{p}, N_{r}^{p} and N_{tr} obtained for different test queries are given in Table 11 along with details of queries. Out of eight queries, the precision and recall for queries Q2, Q7 and Q8 are found to be zero for both methods and for remaining queries, the precision and recall are found to be 100% for both methods. Hence the precision and recall of discovery are not altered by indexing. Further, the recall and precision of both the methods are same because the way of computing *DoM* among service concepts is same for both methods.

The accuracy of non-functional selection of the proposed method is evaluated using a measure called *optimality_factor*. We define *optimality_factor* as the ratio of utility obtained using the proposed method to the utility obtained using global selection. In our work, to find the utility of global selection, we adopt the method presented in Mohammad and Thomas (2009) where utility of a composite service is computed as

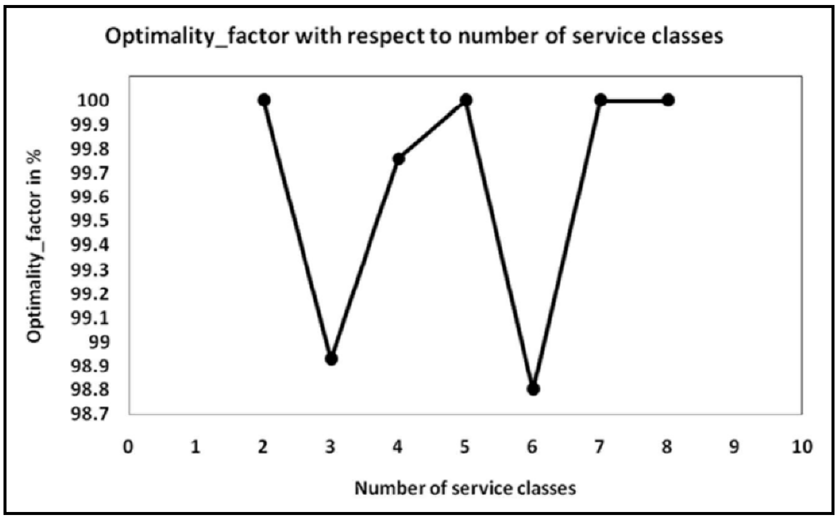
$$U_{CS} = \sum_{k=1}^{r} \frac{Q'_{\max}(k) - q_k(CS)}{Q'_{\max}(k) - Q'_{\min}(k)} \times w_k \tag{37}$$

In (37), U_{CS} denotes the utility of a composite service (*CS*) obtained using global approach, $Q'_{\max}(k)$ denotes the maximum value of *k*th attribute of the given workflow (this is computed as the sum of maximum values of *k*th attribute of all service classes present in the workflow) and $Q'_{\min}(k)$ denotes the minimum value of *k*th attribute of the given workflow (this is computed as the sum of minimum values of *k*th attribute of all service classes present in the workflow), $q_k(CS)$ denotes value of *k*th attribute of *CS* and w_k represents the weight of *k*th attribute.

Table 11 Values of N_{tr}, N_{r}^{s}, N_{rr}^{s}, N_{r}^{p} and N_{rr}^{p} for different test queries

Query ID	*Outputs of query*	*Inputs of query*	*Ontologies of query*	N_{tr}	N_{r}^{s}	N_{rr}^{s}	N_{r}^{p}	N_{rr}^{p}
Q1	Book	Author	Books.owl	2	2	2	2	2
Q2	Price	camera	Extendedcamera.owl, concept.owl	0	0	0	0	0
Q3	Price	Three_wheeled_Car	Concept.owl, my_ontology.owl	3	3	3	3	3
Q4	Drinks	Price	Concept.owl, mid_level_ontology.owl	1	1	1	1	1
Q5	Whiskey	price	Concept.owl, mid_level_ontology.owl	4	4	4	4	4
Q6	Hotel	city	travel.owl	1	1	1	1	1
Q7	Weapon	Geopolitical entity	portal.owl, SUMO.owl, mid_level_ontology.owl	0	0	0	0	0
Q8	PatientTransport Acknowledgement	PatientTransportProfile, PatientTransport_chosen hospital	Hospitalphysicianontology.owl	0	0	0	0	0

Figure 16 *Optimality_factor* with respect to number of service classes



In the proposed method, the utility of any i^{th} service of j^{th} service class is computed using (32). Using this formula, utility of the best service of j^{th} service class can be found out. In a similar manner, the utilities of the best service of all service classes of the given workflow are computed and total of all utilities is taken as the utility obtained using proposed method. Let U_p denote the utility obtained using proposed method. Now, we define *optimality_factor* as the ratio of U_p to U_{CS}. The value of *optimality_factor* is computed by varying the number of service classes (keeping number of services per service class as 50) and the results are given in Table 12 and in Figure 16. From Table 12, the average *optimality_factor* is found to be 99.64%.

Table 12 *optimality_factor* with respect to number of service classes

# of service classes	U_{CS}	U_p	*optimality_factor (%)*
2	0.969802	0.969802	100
3	0.964901	0.954574	98.929735
4	0.960232	0.957941	99.761412
5	0.970998	0.970998	100
6	0.979542	0.967806	98.801889
7	0.981256	0.981256	100
8	0.966363	0.966363	100

4.3.5 Comparison of proposed approach with existing approaches

To compare the performance of functional discovery of the proposed approach, the method presented in Skoutas et al. (2008) has been chosen. The time taken for functional discovery by the proposed method with respect to number of services is given in Table 5. From Table 5, the retrieval time of the proposed method to find all matched services of a query is found to be constant and its average value is 1.0289 milli seconds. The retrieval time of proposed method is compared to Skoutas et al. (2008). Though Skoutas et al. (2008) saves the *processing time* significantly while finding top-*k* matches where $k \in \{1, 50, 500\}$, when all matches of a query is required to be discovered, then there is a significant increase in processing time ranging from around 40 milli seconds to 150 milli seconds (interpreted from the centralised approach of Skoutas et al.'s method). Whereas in our proposed method, in a single retrieval itself, all categories of matched services of a query are obtained with the help of unique indexing. With a typical test collection, the average retrieval time is found to be 1.0289 milli seconds to retrieve all matches of a query.

To compare the performance of non-functional selection of the proposed approach, the method proposed in Mohammad and Thomas (2009) has been chosen. The time involved in the non-functional selection of the proposed approach with respect to number of service classes is compared to that of Mohammad and Thomas (2009). In our approach the time involved in non-functional selection is computed by finding the sum of time involved in decomposition of constraints and time involved in service selection. By keeping number of services as 500 and number of constraints as 3, the time taken by non-functional selection is computed by varying the number of service classes from 10 to 100. The variation in computation time of non-functional selection of the proposed approach is compared to that of Mohammad and Thomas (2009) as given in Table 13. From Table 13, the variation in computation time of proposed approach is found to be very small and negligible when compared to that of Mohammad and Thomas (2009).

Table 13 Time taken for non-functional selection – proposed approach versus Mohammad and Thomas (2009) approach with respect to number of service classes

# of service classes	*# of constraints*	*# of services*	*Computation time (in milli seconds)*	
			Mohammad and Thomas (2009) approach	*Proposed approach*
10–100	3	500	500–20,000	1.146–1.557

5 Conclusions

This paper presents a better approach for identifying best services for composition based on functional discovery and non-functional selection. The paper presents a unique indexing of pre-computed semantic relations for input and output parameters of all services in a repository. More specifically the proposed method can handle disparate similarity demands of client applications in terms of *RDoM* and deliver matched services accordingly. *RDoM* can take nine different categorical values. The provision of nine different matches gives better flexibility to service clients in feeding a desirable value for *RDoM*. This is another unique feature of the proposed approach. Further, the usage of two indices fully eliminates the invoking of semantic reasoning during querying. A new method for service retrieval using indices is also presented. The method is efficient in finding all possible matches according to the given *RDoM* in constant time.

The non-functional selection is optimised using a local selection in multithreaded fashion with a new method of decomposing QoS constraints. The usage of local selection, multithreading and QoS hash index enhances the performance of non-functional selection significantly. The method of decomposing constraints has a unique feature that the method is independent of number of services present in a service class. This is very desirable as any real time business transaction involves several services from different domains to be composed quickly.

The proposed method has been implemented and from a series of experiments the method is found to yield excellent time characteristics. Also, the accuracy of the method is evaluated by comparing the results with that of standard approaches using the evaluation measures, precision, recall and *optimality_factor*. The accuracy is found to be not affected by the optimising techniques. The minimum time consumption of the method makes it more applicable to dynamic composition needs.

References

Gao, T., Wang, H., Zheng, N. and Li, F. (2009) ‘An improved way to facilitate composition-oriented semantic service discovery’, in *International Conference On Computer Engineering and Technology – ICCET ‘09*, Singapore, pp.156–160.

Guo, R., Le, J. and Xia, X. (2005) ‘Capability matching of web service based on OWL-S’, in *Proceedings of the 16th International Workshop on Database and Expert Systems Applications*, DEXA, IEEE Computer Society, Copenhagen, Denmark, pp.653–657.

Hong, L. and Hu, J. (2009) ‘A multi-dimension QoS based local service selection model for service composition’, in *Journal of Networks*, Vol. 4, No. 5, pp.351–358, Academy Publisher.

Jin, J., Zhang, Y., Cao, Y. and Zhou, R. (2010) ‘An enhanced QoS decomposition approach for efficient service composition’, in *5th International Conference on Computer Science & Education (ICCSE)*, Beijing, China, pp.1680–1684.

Kuang, L., Li, Y., Deng, S. 0and Wu, Z. (2007) ‘Inverted indexing for composition-oriented service discovery’, in *IEEE International Conference on Web Services*, Salt Lake City, Utah, USA, pp.257–264.

Li, J., Zhao, Y., Liu, M., Sun, H. and Ma, D. (2010) ‘An adaptive heuristic approach for distributed QoS-based service composition’, in *IEEE Symposium on Computers and Communications*, Beijing, China, pp.687–694.

Liu, P., Zhang, J. and Yu, X. (2009) ‘Clustering-based semantic web service matchmaking with automated knowledge acquisition’, in *Proceedings of the International Conference on Web Information Systems and Mining (WISM ‘09)*, Shanghai, China, pp.261–270.

Mohammad, A. and Thomas, R. (2009) ‘Combining global optimization with local selection for efficient QoS-aware service composition’, in *Proceedings of the 18th International Conference on World Wide Web*, ACM, New York, pp.881–890.

Mohammad, A., Thomas, R., Peter, D. and Wolfgang, N. (2008) ‘A scalable approach for QoS-based web service selection’, in *Service-Oriented Computing ICSOC 2008 Workshops*, Springer-Verlag, Berlin, pp.190–199.

Mokhtar, S.B., Kaul, A., Georgantas, N. and Issarny, V. (2006) ‘Towards efficient matching of semantic web service capabilities’, in *International Workshop on Web Services Modeling and Testing (WS-MaTe)*, Palermo, Italy, pp.137–152.

Mokhtar, S.B., Preuveneers, D., Georgantas, N., Issarny, V. and Berbers, Y. (2008) ‘EASY: efficient semantic service discovery in pervasive computing environments with QoS and context support’, in the *Journal of Systems and Software*, Elsevier Science Inc., New York, NY, USA, Vol. 81, No. 5, pp.785–808.

Paolucci, M., Kawamura, T., Payne, T.R. and Sycara, K. (2002) ‘Semantic matching of web service capabilities’, in *International Semantic Web Conference*, Springer Verlag, LNCS, Sardinia, Italy, Vol. 2342, pp.333–347.

Pathak, J., Koul, N., Caragea, D. and Honavar, V.G. (2005) ‘A framework for semantic web services discovery’, in the *Proceedings of 7th ACM International Workshop on Web Information and Data Management (WIDM-2005)*, Bremen, Germany, ACM, pp.45–50.

Paul, Y.K. and Ching-Lai, H. (1995) ‘Multiple attribute decision making: an introduction’, in Sage University Paper Series on *Quantitative Applications in the Social Sciences*, ISBN 0-8039-5486-7.

Qiu, T. and Li, P. (2008) 'Web service discovery based on semantic matchmaking with UDDI', in the *Proceedings of 9th International Conference for Young Computer Scientists (ICYCS 2008)*, IEEE Computer Society, Hunan, China, pp.1229–1234.

Rozina, C.V., Bianca, P.C., Ioan, S., Mihaela, D., Vlad, A. and Tudor, D. (2010) 'an ant-inspired approach for semantic web service clustering', in *Roedunet 9th International Conference (RoEduNet) Sibiu*, Romania, pp.145–150.

Skoutas, D., Sacharidis, D., Kantere, V. and Sellis, T. (2008) 'Efficient semantic web service discovery in centralized and P2P environments', in the *Proceedings of 7th International Semantic Web Conference (ISWC)*, Karlsruhe, Germany, pp.583–598.

Sun, S.X., Zhao, J. and Wang, H. (2010) 'A negotiation based approach for service composition', in *Proceedings of the 5th International Conference on Global Perspectives on Design Science Research GPIC (DESRIST '10)*, St. Gallen, Switzerland, pp.381–393.

Zeng, L., Benatallah, B., Marlon, D., Kalagnanam, J. and Sheng, Q.Z. (2003) 'Quality driven web services composition', in *Proceedings of the 12th International Conference on World Wide Web*, ACM Press, Budapest, Hungary, pp.411–421.

Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J. and Chang, H. (2004) 'QoS-aware middleware for web service composition', in *IEEE Transactions on Software Engineering*, Vol. 30, No. 5, pp.311–327.

Zhag, J., Yu, X., Liu, P. and Wang, Z. (2009) 'Research on improving performance of semantic search in UDDI', in the *Proceedings of WRI GCIS, Global Congress on Intelligent Systems*, IEEE Computer Society, Xiamen, China, pp.572–576.

Zhou, B., Huang, T., Liu, J. and Shen, M. (2009) 'Using inverted indexing to semantic WEB service discovery search model', in *Proceedings of 5th International Conference on Wireless Communications, Networking and Mobile Computing, (WiCom '09)*, IEEE Press, Beijing, China, pp.4872–4875.

Zhu, Z., Yuan, H., Song, J., Bi, J. and Liu, G. (2010) 'WSSCAN: a effective approach for web services clustering', in *Proceedings of International Conference on Computer Application and System Modeling (ICCASM)*, Taiyuan, China, Vol. 5, pp.618–622.