ANALYSIS OF DYNAMIC WEB SERVICES: TOWARDS EFFICIENT DISCOVERY IN CLOUD

Faisal Ahmad [1] and Anirban Sarkar [2]

[1] Tata Consultancy Services Limited, Edison, NJ, USA, 08837
[2] National Institute of Technology, Durgapur, India, 713209

Email: [1]faisal.nitdgp@gmail.com, [2]sarkar.anirban@gmail.com

ABSTRACT

Web services are very dynamic as they can be added, modified and deleted from the repository on the fly. Discovering web services for service composition have special requirements. In the service composition, service requirement is not only to satisfy its functional specification but it also needs to ensure that the selected composing services are compatible with respect to their input/output structure and its associated data types. Web Service community [1] organized the functionally similar services together in a group. But one important drawback of using web service community in the service composition is that it requires further analysis of the discovered service communities for selecting appropriate service which can be composed based on its input/output structure and data types. This paper proposes data structures, which can be used to create web service family [2] by filtering not only on the basis of functionality but also on the basis of its input/output structure, which are required for composing services with one another. In order to speed up the web service family creation process, the creation of web service family is divided in two phases. The first phase of the proposed approach introduces a novel concept of business family and dynamic business web service (DBWS) tree, which are used to organize all web services in the tree structure on the basis of their business context and input/output structure. In the second phase, the concept of DBWS tree is used further to create the web service families. The DBWS resembles the classification and organization of businesses in NAICS. The DBWS tree can be used very efficiently for creating web service families, which in turn can be used for selecting service efficiently in service composition.

Keywords: ***Web Services, Service Discovery, Web Service Composition, Web Service Orchestration, Dynamic Web Services, Repository, Web Service Community, Data Structure***

1.0 INTRODUCTION

Web services describe a standardized way of integrating Web-based applications. In recent days, the use of web services has dramatically increased due to the easiness, interoperability, and flexibility that web services offer to the software systems, which other software structures don't support or support poorly. The web services are composed to implements complex business requirements. In service composition more than one web services are glued together to achieve a new functionality. The process of service composition can be separated into the following tasks: (i) Requirements analysis and specification, (ii) Abstract designing, (iii) Formal representation (iv) validation (v) Execution. These steps are implemented manually [3, 4] or automatically [5, 6, 7]. In most of the approaches, web service composition design is represented graphically at a high level with help of an abstract representation of web services like web service family [2] or community of services [1]. These abstract representation of web service based business processes are further converted to formal languages, which are finally executed with help of execution engines [1, 3, 17].

Service discovery is a significant activity in service composition process. Efficient discovery plays a crucial role in decreasing service composition time. Although existing discovery techniques have produced promising results that are certainly useful, they may not very well aligned with the specific needs of web service composition. In most of the discovery approaches the focus is given to search web services based on only functional requirements. The concept of service community [1] is used as a solution to the problem of composing a potentially large number of dynamic Web services. Web service community is defined as a collection of Web services with a common functionality, although these Web services have distinct non-functional properties. The concept of grouping similar services in groups, called as communities [1] have helped to a large extent in improving the service selection process for service composition but the service communities are not organised by its input/output structures. Further processing of the selected communities is required to find suitable service for compositing. In composite web services, apart from checking the functional specification of services, it is also equally important that web services have some additional features for composing like, specific relationship between their input/output parameters, specific data type of their

input/output parameters and business context of participating web services. Service communities help in quick service selection as per the functional specification of the composing services. In such functional oriented groups, systems have to browse the community in detail to select the service which can be composed based on its input/output structure and associated data structures. A better organized and dynamic grouping at more granular level is required, which can speed up the composition process. Due to these drawbacks with the service composition, the concept of *web service family* [16] was proposed. Web service family not only group services on the basis of their functionality but also on the basis of Input/output structures, business context and associated data structures. Web service family can be used very efficiently in the service composition process as detail searching on the web service family is not required as it is required while using service communities. Once web service families are created, only searching appropriate web service family will suffice the need of service composition.

Creating a web service family from very large number of dynamic web services, which are hosted independently on clouds, is every challenging. To handle the challenge, this paper presents an efficient step by step filtering approach for creating web service families. The paper analyze dynamic web services in detail and propose an efficient novel data structures to reorganize existing web services of the repository in semantic groups by fetching web services metadata from the existing repositories. It presents an efficient discovery mechanism for creating web service families. The existing services are clustered in a tree on the basis of its business context and input/output parameters. The same tree structures are used further to fetch potential web services to create web service family by further filtering them. This will not only help in the faceted browsing of web services but also speed up the composition process, which are built using dynamic web services. The creation of web service family can resolve some of the challenges associated with web services composition. Under the proposed approach in this paper the web service family creation process is divided in two level tasks namely *web service family level* and *discovery level* tasks. *Web service family* level task used a novel data structure named as the *business family*, which organized web services in semantic groups based on their business context and input/output parameter relationships. The *discovery level* task used the novel *dynamic business web service* (*DBWS*) tree to further organized business families in a top-down approach on the basis of their business classifications. This organization of dynamic web services in the *business family* and the *DBWS* tree can help in representing, analyzing, discovering, and grouping dynamic web services. Different operations, algorithms possible on the *DBWS* tree are also illustrated. This paper discussed in detail not only usage of the *DBWS* tree but also its role in the efficient discovery process. A coding scheme for the nodes of the *DBWS* tree closely resemble coding scheme of *NAICS* [*18*], which enable browsing services with NAICS business code.

The rest of the paper is organized as follows. Section 2 discusses specific problems associated with service composition, which this paper targets to accomplish. Section 3 discussed some of important related works carried out before. Section 4 introduces some of the taxonomies related to the previous works on dynamic web services. Section 5 presents a two layered approach for the discovery process for web service family creation named as *web service family* and *discovery* layer. Section 6 presented the architecture of the web service family layer and discovery layer. It discusses the concepts of *business family* and DBWS tree data structure, which is used to organized the web services. Section 7 presents the concept of (dynamic business web service) *DBWS* tree in detail, which further organizes business families in a tree based data structure. This section also presents coding schemes, different operations, algorithms and advantages of the *DBWS* tree and presents its similarity with NAICS standards. Section 8 presents the *dynamic web service discovery (DWSD)* machine, which manages and executes all remaining activities required for the creation of web service families. Section 9 demonstrated an analytical approach for measuring discovery time and it further compares the results of discovery time of keywords based discovery with web service family based discovery. Section 10 concludes and proposed the possible future work.

2.0 PROBLEM FORMULATION

This section discussed the important challenges in the service composition process which increases the processing time of service composition.

2.1 Challenges in Service Composition

The requirement for standalone service discovery and service discovery for composite services are different. In the service composition, more than one service is composed with one another to form a new service. The gluing of one service with another service has special need for service selection. Some of the important challenges pertaining to service composition are as follow.

(1) *Discovering service as per the functional specification.* Discovering service as per required functionality is key to the successful service composition. If the service selected is not semantically same as per the specification then the whole composite service will fail. A single inappropriate selection of a service would spoil the complete composite service. The initial need is to select service as per their functional specification.

(2) *Importance of business context*- There are large number of web services available with diverse business objectives. There are possibilities of two services looking structurally same and also might have same description but might actually differ functionally. The functional objectives of similar looking web services might vary on the basis of their business context. Business groups or contexts are very important in differentiating web services.

For example in Table 1, web service w_1 and w_2 have same input, output parameters and also have similar description but their business objectives might be different. w_1 returns bank account detail like(saving account, current account or recurring account) whereas w_2 provide telecom's customer detail for a given customer account number whether customer have prepaid or post-paid account. w_1 belongs to banking domain and w_2 belongs to telecom domain. The ambiguity between w_1 and w_2 can be easily resolved using their business context. For banking domain based composite web services, w_1 will suits best and for telecom domain based composite web services, w_2 is the best.

(3) *Compatible Input/output structures of composing service*. In the service composition, output of one service is the input of another composing service. It is very important that the selected service can accept the output of incoming composing service.

If an address decomposer service with one input and one output needs to be composed with a weather service, which also have one input and one output. The requirement here is for a web service which provides climate information for one input i.e. zip code. This web service should have one inputs and one output. In Table 1 web service w_3, w_4 and w_5 provide temperature information of a place but w_5 have two inputs whereas w_3 and w_4 have only 1 input and 1 output. Thus potential web service will be w_3 and w_4, which can be filtered using the relationship types between input/output parameters.

(4) *Importance of Data type for composing service.* Not only input/output relationship is important between composing services but also the data structure is equally vital. If data type of the output of one composing service is different as required by the input of another composing service then the two services cannot be composed even if there functionality and input/output structures are compatible. Either the data type needs to be converted as per the input specification of the service or appropriate service with exact data type needs to be searched.

If a requirement is to search a web service, which provides temperature of a place for a given zip code (Integer) then in Table 1, w_3 and w_4 both provide temperature information but w_3 takes input zip code (integer) whereas w_4 takes city (String). So here w_3 is the suitable web service, which can be used directly for composition. Data types of the input/output parameters helps in selecting suitable web services, which can easily fit in the composition with other web services.

These above problems are potential areas which needs focus with respect to efficient and speedy composition. The service discovery for service composition should be carried out keeping these concerns in focus. The grouping in service community is built by focusing only on the functional perspective of services. Further analysis and filtering are required to be carried out in the selected community to select the appropriate service which can be composed with respect to its Input/output and associated data type structures. *Web service family* groups web services not only based on the functional specification but also on the input/output structure. The complete process is divided in two phase. In the first phase web services are organized in tree based structure, considering the business context and input/output structure. This phase is carried out in background. In the second phase, for a given web service request, the tree can be used to fetch potential candidate services and it is analyzed further to create web service family.

Table 1. Example of web services [2]

Web service	*Web service name*	*Business group*	*Operation*	*Input datatype*	*input*	*Output data type*	*Output*	*Description*
w_1	Account detail	Banking	GetActdetail	Integer	Account number	String	AccountDetail	Return detail of account like(saving, current), account opening date etc.
w_2	Account detail	Telecom	GetActdetail	Integer	Account number	String	Acvcountdetail	Return detail of account like (postpaid, prepaid), account opening date
w_3	Climate Info	Information	Temperature Bypostal Code	Integer	Postal Code	String	Temperature	Returns temperature for a given postal Code
w_4	Temperatue Info	Information	Temperature ByCity	String	City	String	Temperature	Returns temperature for a given city
w_5	Climate Info	Information	Temperature BySateAnd City	String	State, City	String	Temperature, rainfall	Returns temperature and rainfall for a given State and city

3.0 RELATED WORK

The web service discovery is a hot research topic in the past a few years. *Zhang et al*. [1] indicate that the service discovery and composition play the crucial role in the area of services computing. There are many approaches for discovering similar services. Some of the important work related to this work is discussed in this section.

One of the wide spread approaches used service discovery is clustering [8, 9, 10]. The clustering methodology is a technology that transforms a complex problem into a series of simpler ones, which can be handled more easily. Specifically, this technology reorganizes a set of data into different groups based on some standards of similarity. *Dong et al.* [8] puts forward a clustering approach to search web services where the search consisted of two main stages. A service user first types keywords into a service search engine, looking for the corresponding services. Then, based on the initial web services returned, the approach extracts semantic concepts from the natural language descriptions provided in the web services. In particular, with the help of the co-occurrence of the terms appearing in the inputs and outputs, in the names of the operations and in the descriptions of web services, the similarity search approach employs the agglomerative clustering algorithm for clustering these terms to the meaningful concepts. *Arbramowicz et al.* [9] proposes architecture for web services filtering and clustering. The service filtering is based on the profiles representing users and application information, which are further described through Web Ontology Language for Services (OWL-S). In order to improve the effectiveness of the filtering process, a clustering analysis is applied to the filtering process by comparing services with related the clusters. The objectives of the proposed matchmaking process are to save execution time, and to improve the refinement of the stored data. Our approach also divides the complete discovery process in two stages with the objective of improving the search time. Another similar approach is by *Nayak et al.* In [10] Nayak *et al*. concentrates on web service discovery with OWL-S and clustering technology, which consists of three main steps. The OWL-S is first combined with WSDL to represent service semantics before a clustering algorithm is used to group the collections of heterogeneous services together. Finally, a user query is matched against the clusters, in order to return the suitable services. *Constantinescu et. al.* [11] focuses on service discovery based on a directory where Web services are clustered into the predefined hierarchical business categories. In this situation, the performance of reasonable service discovery relies on both service providers and service requesters having prior knowledge on the service organization schemes.

Another approach used for service discovery is Indexing. To enable fast discovery of web services, available web services can be indexed using one of the indexing mechanisms such as inverted indexing and latent semantic indexing. *Huang et al.* [12] describe how inverted indexing can be used for quick, accurate and efficient web service discovery. *Aiello et al.* [13] describe VitaLab system which is web service discovery system based on indexing using hashtable. They have implemented indexing on WSDL descriptions which are parsed using Streaming API for XML (StAX). As centralized web service discovery approach has many disadvantages such as single point of failure and high maintenance cost. *Emekci et al.* [14] propose a peer-to-peer framework for web service discovery which is based on process behavior. Framework considers how service functionality is served. All available web services are represented using finite automaton. Each web service is defined as follows: A Web service p is a triple, *p= (I, S, R),* such that, *I* is the implementation of *p*

represented as a finite automaton, *S* is the service finite automaton, and *R* is the set of request finite automata. When user wants to search for web service, PFA of finite automaton of web service(*R*) is sent for matching. Matching is done against *S* by hashing the finite automata onto a Chord ring. Chord is a peer-to- peer system for routing a query on hops using distributed hash table. Regular expression of the queried PFA is used as the key to route the query to the peer responsible for that PFA.

Saadon et al. [19] proposed CMDis, an enhancement of cloud-based MWS discovery framework, with semantic-based matchmaking approach. In their work, semantic lightweight web service descriptions were used with a REST-based architecture. Accordingly, a preliminary prototype was developed on android-based to evaluate the applicability of the framework. *Hamza et al.* [20] propose a new algorithm of comparison between the request of the client and the description of the Web services. Their system is composed of two areas of Cloud. The first deals with Key words based research and second supports the filtering of the found web services. This discovery is performed by an algorithm based on the calculation of similarities between the request and the description of the web services.
Khan et. al. [21] proposes a framework for dynamic service discovery and mapping that supports the identification of service during its run time in Enterprise Cloud Bus (ECB) systems. They present a Web Service Relational Model (WSRM) for distributed service discovery that supports functional aspects of Web service and allow various trade-offs between the accuracy of discovery results and the efficiency of the service discovery process. The implementation aspects using SQL of WSRM are also discussed. *Zhang et al*. [22] presented a method for web service structural network construction based on web service description documents. Secondly, they propose a community discovery algorithm based on web service interaction relationships and analyze community structure of web service networks. Finally, we preform experiment on several real datasets, and the results show the efficiency and feasibility of community discovery algorithm.

An important approach is to put the similar services in groups. Such groups help in efficient searching services as it contains all functionally similar services in one group. Several previous works gather functionally similar web services into communities that are accessed via a common interface. Such an approach is proposed in SELF-SERV framework [1], Self-Serv leverages emerging Web services standards and an established modeling notation (state charts) to provide high-level support for defining composite web services involving a variable number of participants. Self-Serv enacts the resulting composite services in a P2P way within a dynamic environment. In addition, the system allows for monitoring and tracing the execution of composite services. *Benatallah et al.* [15], authors propose an approach that supports the concepts, architecture, operation and deployment of web service communities. The notion of community serves as an intermediary layer to bind to web services. A community gathers several slave web services that provide the same functionality. The community is accessed via a unique master web service. Users bind to the master web service that transparently calls a slave in the community. . This work details the management tasks a master web service is responsible for. Such tasks include among other things registering new Web services into the community, tracking bad Web services, and removing ineffective web services from the community.

In this work, web service family, which is similar to web service community in its approach of grouping similar services, is created. Web service family is richer concepts than web service community [1] as the service communities are not organised by input/output structures of its member so it needs more time in selecting suitable compatible services. In service composition the business context and input/output structures plays a very important role. The web service family organized the services not only on the basis of functionality but also on the basis of their business contexts and input/output structures. The discovery approach divides the approach in two stages. In first stage the existing web services are reorganized in a novel data structure with respect to its business context and input/output structure. In the second stages, further filters are applied on the output of first stages to put similar services in web service families.

4.0 TAXONOMY FOR DYNAMIC WEB SERVICES

NAICS [18**]** - The North American Industry Classification System (*NAICS*) is the standard used by Federal statistical agencies in classifying business establishments for the purpose of collecting, analyzing, and publishing statistical data related to the U.S. business economy. *NAICS* was developed under the auspices of the Office of Management and Budget (*OMB*), and adopted in 1997 to replace the Standard Industrial Classification (*SIC*) system. *NAICS* is used by business and government to classify business establishments according to type of economic activity (process of production) in Canada, Mexico and the United States.

The *NAICS* numbering system employs six-digit code at the most detailed industry level. The first two digits designate the largest business sector, the third digit designates the sub sector, the fourth digit designates the industry group, the fifth digit designates the *NAICS* industries, and the sixth digit designates the national industries.

Web service Family [2] - There is many web service instances available in the cloud, which are managed independently. It is not possible to keep track of each and every web service, which are added or deleted from a registry without any prior notification. Families of web services are created to capture and manage the dynamic nature of web services in groups. Each web service family corresponds to one or many web service instances with a common business objective and same input /output structures. For example, Fig. 1 represents a web service family named as "***Fetch Weather of a Place***" (say W_1), which takes one input as zip code and returns temperature and humidity of that place. All such similar web services are placed in this family.

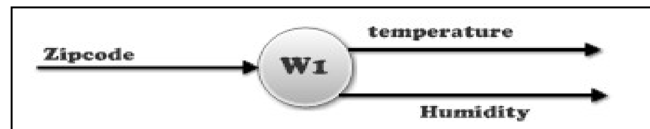

Fig. 1. Web-service family

5.0 WEB SERVICE DISCOVERY PROCESS FOR CREATING WEB SERVICE FAMILY

There can be many ways of discovering similar web services. Efficient discovery of dynamic web services largely depends on the data structures of the underlying repositories. In any discovery process many filters are required to search similar web services. These filters help in removing unwanted services. To improve the efficiency and discovery time of service composition in particular, all web services are reorganized in web service families. Once web service families are created, discovery process needs only to search appropriate web service family as per abstract family definition in composition design and any instances of selected web service family can be selected for composing. No further analysis and filtering are required. The main effort in this process is to create web service families using the existing web services of the repositories.

5.1 Web service discovery and Cloud environment

The cloud provides software and hardware resources via the Internet. The connections into the cloud are often referred to as application programming interfaces (APIs). These APIs use Web services, such as SOAP, REST, and JSON. The content sent over these APIs is usually XML or some form of name/value pairs. In general, cloud computing services that include Web hosting can be an alternative to other traditional kinds of Web hosting that are not based on cloud computing principles. One of the biggest differences could be called a "single client" versus "multi-tenant" approach. Cloud computing services that include Web hosting are usually multi-tenant. That means that the files and data resources of multiple clients are housed on the same server. This provides flexibility and on-demand services for individual clients, so that providers can scale up or scale down delivery easily.

Technological advances in Service Oriented Architecture (SOA) and cloud computing have led to a significant increase in the number of loosely coupled independent services available in the cloud. Discovery these cloud based web services is very complex. The complexity, in general, comes from the following sources. First, the number of web services available over the web increased dramatically during recent years and one can expect to have a huge web service repository to be searched. Second, web services can be created and updated on the fly, thus the composition system needs to detect the changes at run time and decisions should be made based on up-to-date information. Third, Web services can be developed by different organizations, which use different models to describe their services, however, there does not exist a unique language to define and evaluate web services in a consistent manner. There is a need for an architectural framework for discovering web service that can provide features and facilities for end-to-end web service discovery.

This paper presented a discovery model which manages the discovery of web services considering the challenges posed by the cloud. Though the model is designed considering the services hosted on the cloud in a multi-tenant approach but the model is equally compatible for discovering web services hosted using traditional

approach on single client. The proposed data structure business family and *DBWS* tree holds web services hosted across clouds.

5.2 Two Layers Approach for Discovery

One simple and trivial approach of discovery is to start each time by traversing the whole repository to filter out services. Clustering and Classification approaches also browse entire repository every time.

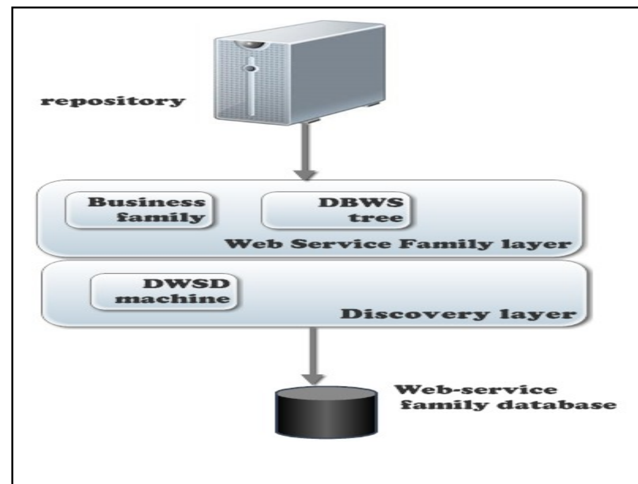

Fig. 2. Web service discovery process

This approach takes very long time in the discovery as each time it starts with traversing and filtering complete repository exhaustively. It does not store results of the filtering steps, which can be reused in other discovery processes. Also any updation or modification of services in repository is identified only when complete repository is traversed.

A novel approach is presented, which divides all filtering steps of the discovery process in layers, where each layer performs some steps of the filtering and its results (web services) are stored in data structures, which can be used later in other filtering steps. In this direction a new rich architecture is designed. A two layered approach for service discovery is proposed for creating web service family. Each layer not only carries out some filtering, which helps in reducing discovery space i.e. number of web services to be searched but it also adds some semantic information, which helps in efficient and error free discovery for creating web service families. Fig. 2 represents two layer approaches for discovery.

Web service family Layer- This layer interacts with the repository directly to fetch web services. This layer implements some of the filtering steps like business group filtering, input/output relationship types. This layer organizes and stores the results of this filtering step in a dynamic data structure named as the *business family*. This layer organized web services in groups and further store these groups in a tree structure named as the *DBWS tree*. The data structures of the web service family layer are reused to complete the discovery process.

The *web service family* layer helps in creating a proxy repository, which reorganized all existing web services in groups on the basis of different filtering steps i.e. business group filtering and input/output parameters relationship types filtering. Creation of this proxy repository helps in reducing the initial discovery space as all further discovery steps will start by searching a subset of the repository. The proxy repository contains web services organized in groups named as *business family* on the basis of their business context and relationship types between their input/output parameters. These business families are linked to each other on the basis of their business classification to form a *dynamic business web service (DBWS)* tree. The *DBWS* tree can be traversed and searched efficiently to find out suitable business families for further discovery.

Discovery Layer- The main purpose of discovery layer is to create web service family by fetching potential web services from the *web service family layer* for further filtering. The discovery layer does not interact directly with the repository. The discovery layer filters on the output of the *web service family layer* by executing remaining filters required for the creation of web service families i.e. Input/output data type filtering, semantic filtering of input/output parameters, semantic filtering on web service textual description.

The two layers of the discovery approach are managed independently and are loosely coupled, which helps in efficient organization and management of dynamic web services. The *web service family* layer always remains in sync with the repository and the *discovery layers* remains in sync with the *web service family* layer. This can provides a real time managing for dynamic web services.

6.0 ARCTITECTURE OF WEB SERVICE FAMILY LAYER

This section discussed the first layer and its associated data structures. All filtering steps required for the creation of web service family can be organized in a specific order, which helps in increasing the efficiency of the discovery process. Outputs of each filtering steps (web service) are stored in a dynamic data structures, which are reused in all further discovery process. Exhaustive traversing and filtering all web services of the repository is only one time process. Once web service families are created, it is required to keep it in sync with the repository.

6.1 Web Service Family Creation

For the given definition of the web service family (refer taxonomy section), filters can be determined, which are required to be executed in specific order to find its members (web service instance). For a given web service family following filtering steps are required to be carried out in the given order to find its members.

(1) *Business group filtering*- The complete repository is traversed and filtered to search all web service instances that belongs to specific business group for the given web service family definition. As discussed in section 2, the functional objectives of similar looking web services mostly vary on the basis of their business context. Business groups or contexts are very important in differentiating web services.

(2) *Input/output relationship types filtering*- After business group filtering, web services need to be filtered on the basis of the relationship types between their input/output parameters (i.e. *1-k, 1-n, k-1, k-n, n-m, 1-1*). Relationship types based filtering is very important as web services, which are searched and used in composite web service largely depends on their number of input/output parameters.
.

(3) *Input/output data type filtering*- Web services to be discovered may have pre-requisite requirements of their data type of input/output parameters. Even if their business objectives matched but if their input/output data types do not match then it cannot be used in the composition with other web services. Web services needs to be filtered on the basis of their input/output data type after filtering on the basis of relationship between input/output parameters.

(4) *Semantic filtering of input/output parameters*- After input/output data type filtering, it is also important to establish the semantic similarity between their input/output parameters. This helps in establishing similarity in their business objectives and validates correctness of the web services to be used.

For example w_4 and w_6 both have same business objectives, which can be established by semantically analyzing their input parameters (city, town). City and town are semantically same so both w_4 and w_6 can be used.

(5) *Semantic filtering of web service description*- After semantic filtering of input/output parameters, it is also important to establish similarity in their objectives on the basis of their descriptions. Descriptions of web services help in establishing and verifying objectives of web services.

For example in Table 1*,* w_4 and w_6 descriptions can be semantically analyzed to established similarity in their business objective.

Fig. 3 represents all the filters required to be executed to create web service family. If all the above 5 filtering steps are executed in the given order then a web service family will be created. Execution of each step reduced

the number of web services to be searched in the consecutive filter. If these steps are not followed and web service are directly searched based only on the keyword then it required to search the complete repository exhaustively every time, which is time consuming and expensive. These 5 steps can help not only in the creation of web service families but also in the process of efficient service composition.

6.2 Data Structures of Web Service Family Layer

Efficiency of discovery process largely influence by the data structure in which web services are organized. In the *web service family layer* two data structures name as *business family* and *DBWS* tree are used to store and organized web services, which results from different filtering steps.

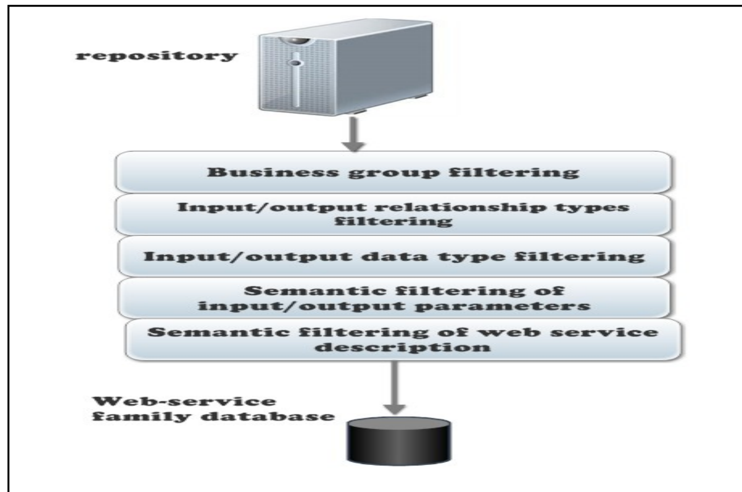

Fig. 3. Filters required for the creation of Web service family

6.2.1 Business Family

Each web service is associated with some business. Objectives of web services are largely influenced by its business contexts. There is high possibility for two web services having same input/output structure will differ in their business objectives if their business contexts are different. The relationships between input/output parameters also play a very important role in the discovery process for composite services. Applying two filters (business group filter and input/output relation filter) on web services of existing repository, services are grouped in families called as *business family*.

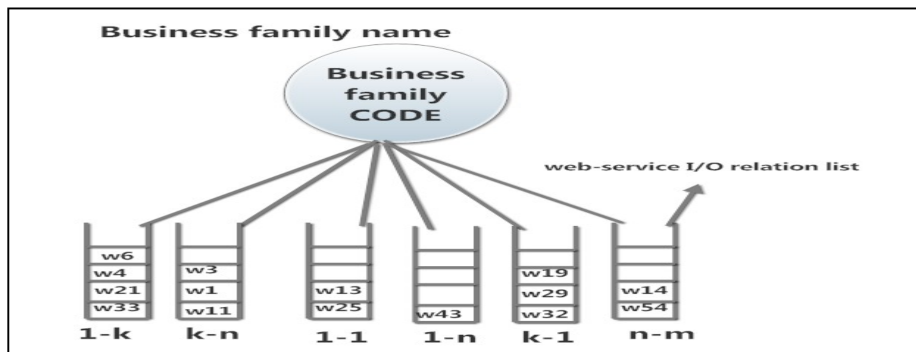

Fig. 4. Business family

The *business family* is a group of one or more web service instances, which belongs to same business group. Business context can be fetched from *Yellow pages* of the web service description file, which provide a classification of services or businesses, based on standard taxonomies like *NAICS* and *SIC*. The web service

instances in the *business family* are grouped in separate lists named as *web-service I/O relation* list. Each *web service I/O relation* list contains web services belonging to a specific I/O relationship type. All possible relationship between input/output parameters can be divided into 6 types (e.g. *1-k, 1-n, k-1, k-n, n-m, 1-1*). Corresponding to each relationship type there exist one *web-service I/O relation* list. There can be maximum 6 web service I/O relation lists in a business family. All web service instances in lists are unordered and belong to a single business group. Business family is graphically represented in the Fig. 4.

A *business family* can be defined formally as $BF = \{key, B, R_L\}$ where
Key- Is the unique key, which identified a business family uniquely.
B- Is the business group to which the business family belongs.
R_L- Is the mapping between input/output relationship type I/O_R to list of web service instances i.e. $I/O_R \rightarrow W$.

6.2.2 Others Related Definition

(1) *Concrete business family*- A business family is said to be concrete if it cannot be further divided on the basis of business groups and its web service I/O relation lists (I/O_R) are not derived from business families of its sub groups.
(2) *Abstract business family*- A business family is said to be abstract if it can be further divided on the basis of business group and its *web service I/O relation* lists is derived from business families of its sub groups.
(3) *Empty business family*- A business family is said to be empty if it belongs to a business group but its all *web service I/O relation* lists is empty.
(4) *Homogeneous Business family*- Business family is homogeneous if all its web services belongs to a single business family and have only one type of web service I/O relation list.
(5) *Heterogeneous Business family*- Business family is heterogeneous if its web services belongs to different business families and have more than one type of web service I/O relation lists.

6.2.3 Different Operations on the Business Family

(1) *Union operation*- Union operation on two non-empty business families BF_i and BF_j results in a heterogeneous business family BF_k such that BF_k contains all types of web-service I/O relation lists and each list contains web services, which results from the union of the corresponding web-service I/O relation lists(I/O_R) of BF_i and BF_j .
(2) *Unary union operation*- Unary union operation is a special case of union operation where union operation is applied only one type of I/O_R list. *Unary union* operation on two non-empty business families BF_i and BF_j on basis of a relationship type I/O_{Ri} results in a homogeneous business family BF_k such that BF_k contains all web services of BF_i and BF_j having I/O relationship types as I/O_{Ri}.
(3) *Intersection operation*- Intersection operation on two non-empty business families BF_i and BF_j results in a business family BF_k such that BF_k contains all common web services of BF_i and BF_j in their respective type of web service I/O relationship lists.

6.2.4 Advantages of the Business Family

Business family organized the services on the basis of similar functionality and input/output structures which helps in selecting potential services for the composing services for a given definition of the services. Business family works as the starting point from where further filtering is carried out for creating the web service family.

(1) *Organizes web services on basis of business context*- Business families helps in organizing and grouping web services in context of their businesses. This helps not only in efficient discovery but also provide provision of efficient browsing of web services on basis of their business groups and contexts.
(2) *Reduce the discovery space*- Business families divide the complete repository in many groups on the basis of their business context and input /output parameter relationship types. Discovery process starts with searching in one or more appropriate business families rather than searching exhaustively complete repository. Reducing the discovery space has direct impact on the efficiency of the discovery.
(3) *Helps in real time managing*- Business family organized web services in small manageable groups, which have potential to be managed efficiently. It can modify its member on the fly with help of appropriate agents if any changes occurred in the repository. Business families remain in sync with the repository to supports real time updation.
(4) *Building block of the DBWS tree*- Business family is the building block of the *DBWS* tree which organized business families in a tree structure on the basis of business relationships among business family.

6.3 Dynamic Business Web Service (DBWS) Tree

All web service instances are part of one or other *business families* on the basis of their business context and relationship types between their input/output parameters. These *business families* can be further organized in a top-down hierarchy structure with respect to business relations among them. One *business family* is related to other *business family* on the basis of their business hierarchy. For instance if business family BF_1 belongs to business group of rice farming and business family BF_2 belongs to wheat farming then both BF_1 and BF_2 is related to each other as they both belong to business group of crop production. Similarly all business families have some or other relationship with each other. Such organization (see section 7) of business families can help not only in efficient and dynamic organization of web services but also can help in efficient discovery.

7.0 DYNAMIC BUSINESS WEB SERVICE (DBWS) TREE

The *DBWS* tree is an unordered general tree, which have only one type of node called as *business node*. A business node represents a *business family*. Leaves nodes of this tree represent concrete business family, which cannot be further sub classed on the basis of business groups. Each non leave nodes can have any number of child nodes. Each node has a value (code), level (business group name) and a list of references to other child nodes (its children). The height of the *DBWS* tree can be of any value. All nodes except leave nodes are abstract business families as they do not have their own *web service I/O relation list*s but have lists derived from their child nodes. All non-leaves nodes (abstract business family) derived their web service I/O relation lists by executing union operation on all possible leaves nodes, which can be reached starting from that node.

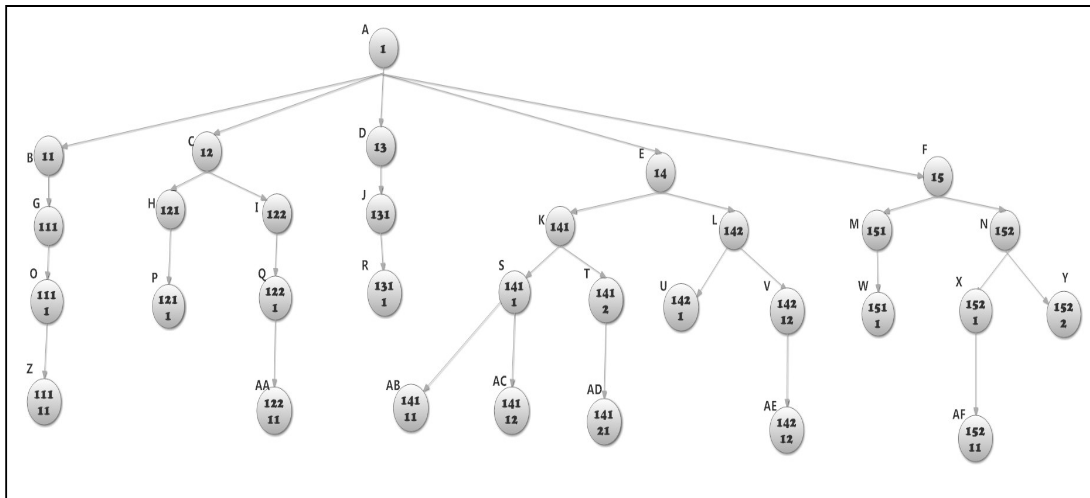

Fig. 5. The DBWS tree

Formally a dynamic business web service (*DBWS*) tree can be defined as tuple *DBWS*= {*BN, SN, H, C*} where

BN= set of all business nodes (i.e. abstract business family).

SN= set of all leaves nodes (i.e. concrete business family).

H= is the height of the tree.

C= is the maximum number of child node a business node can have.

Fig. 5 represents a *DBWS* tree. The *DBWS* tree represents a complete organization of web services in a tree structure using top-down approach. The *DBWS* tree organizes business families on basis of the business relationships among business families. The *DBWS* tree is very generic and can easily be extended in the future to include more number of different web service instances belonging to diverse business groups. This organization closely resembles with *NAICS* business classification. The *DBWS* tree helps in capturing and managing large number of dynamic web services. The *DBWS* tree helps in putting similar web services together with respect to their business context. The *DBWS* tree also helps in the web service discovery process.

7.1 Labeling and Coding of Nodes of the DBWS Tree

Nodes of the *DBWS* tree needs to be identified uniquely. Each node of the *DBWS* tree has a value, a label, links to child nodes and direct or indirect derived collection of web service I/O relation lists. The label of a node is the textual name of the business family, which closely resembles with the business group of the family. The value of a node is a special code named as *business family (BF)* code, which uniquely identifies a business family in the tree. The *BF* code is similar to codes used in *NAICS* classification of business.

BF code for a node of the *DBWS* tree depends on the *BF* code of its parent and on the position of the node from its left most sibling. Its position from the left most sibling is appended to the *BF* code of its parent node to generate its *BF* code. If the *BF* code of the parent node is "*0356*" and the node is at the 2nd position from the leftmost sibling then the *BF* code for this node will be "03562" i.e. its position from the left most sibling is appended to *BF* code of its parent. The *BF* code of the root node is "0'. The length of the *BF* code depends on the level of the node. If the root node starts at level "0" then size of the *BF* code for nodes at level *L* is *L+1*. The maximum size for the *BF* code in the *DBWS* tree depends on the maximum level. If the number of child node is more than 9 then 10th child is represented as "*A*", 11th child as "*B*" and so on i.e. if a node is at 11th position from its leftmost sibling and its parent BF code is "*0524*" then its BF code will be "*0524B*".

7.2 Searching Node in the DBWS Tree

Searching is an important operation for any data structure. The *DBWS* tree has nodes, which are uniquely identified by its *BF code*. The sequence and position of *BF code* digits have special significance. The sequence of digits in the *BF code* indicates the position of the node in the *DBWS* tree. An algorithm can be design to search a specific node in the *DBWS* tree for a given *BF code*.

ALGORITHM 1. SearchNodeInDBWS // searching node in *DBWS* tree for given *BF code*

```
Input: BF code.
Output: Node of DBWS tree.
Read the BF code;
Array BFCodeArray = parse (BF code);   //Parse the BF code and placed it in an
array.
node_Pointer = Root node;
ArraySize= size (BFCodeArray)
For i 1 to ArraySize
{
Read BFCodeArray [i];
BF_value = BFCodeArray [i];
If BF_valueth left child of the node represented by node_Pointer does not exist
  Print (node does not exist) and break;
else
node_Pointer = BF_valueth left child of the node represented by node_Pointer
}
Result = node at node_Pointer.
End
```

The efficiency of the search algorithm is very high. Number of search required is equal to number of digits in the BF code. In the other word searching algorithm is directly depends on the height of the *DBWS* tree. The maximum number of comparison required to search any node in the *DBWS* tree is equals to the height of the *DBWS* tree. As per *NAICS* classification, *DBWS* tree will have maximum height of 6, so it time complexity will be O (6) which is equivalent to order of constant *C*.

7.3 Traversing the DBWS Tree

Traversing is an important operation which is very often used in a tree. It is used either to traverse complete tree starting from the root node or to traverse part of the tree starting from a specific node. Traversing the *DBWS* tree from a node means visiting all its possible reachable leaves nodes. Traversing helps non-leaves nodes (abstract business family) in fetching *web service I/O relationship lists* from their leaves nodes.

Traversing the *DBWS* tree, which is a general tree, is complex and time consuming process .To improves the efficiency of traversing, the *DBWS* tree is first converted to a binary tree. The converted binary tree is traverse from a given node. The *DBWS* tree can be converted to a binary tree using any standards algorithms.

ALGORITHM 2. TraverseDBWS //traversing from a node N_i having BF code as BF_i

```
Input: node Ni, BF code of Ni i.e. BFi
Output: List of leaves node.
Read BFi, Ni
DBWSBinaryTree = Convert DBWS tree to Binary Tree.
ResultArray=null
Node_Pointer = null
Node_Pointer = SearchNodeInDBWSBinaryTree (BFi) // search the node Ni in the
converted binary tree.
Start from the node present at Node_Pointer
Inorder traversal on binary tree.
while traversing
 {
if nodes have no child
  add the node in the ResultArray.
}While end
Return ResultArray
End
```

ALGORITHM 3. SearchNodeInDBWSBinaryTree

```
Input:  BF code of Node i.e. BFi
Output: Node of DBWSBinary tree.
Read BFi
Array BFCodeArray = parse (BF code) //Parse the BF code and placed it in an array.
node_Pointer = Root element
ArraySize= size (BFCodeArray)
For i 1 to ArraySize
{
Read BFCodeArray [i]
If (i=1){
If BFCodeArray[i]>1
 Then node_Pointer= node after traversing in right direction (BFCodeArray[i]-1)
times
 Else if BFCodeArray[i]=1
 Then node_Pointer = node after traversing in left direction once
} else if (i>1){
If BFCodeArray[i]>1
 Then node_Pointer = node after traversing left once and traverse in right
direction (BFCodeArray[i]-1) times
 Else if BFCodeArray[i] =1
 Then node_Pointer = node after traversing in left direction once
}
Result = node at parent_Pointer.
}
End
```

The time complexity for searching and traversing is *O (n)* where *n* is the total number of nodes in the tree.

7.4 Operations on the DBWS Tree

The *DBWS* tree has collections of concrete business families and abstract business families. In the web service family creation or in the discovery process, a specific business family needs to be processed to fetch all its web service instances. If the processing node is leaves node (i.e. concrete business family) then its web services can be directly fetched. If the processing node is non-leaves node (i.e. abstract business family) then further processing is required, which includes traversing of all its possible leaves down in the hierarchy to fetch web services instances and combine all results. Also some operations are required for fetching web services belonging to a business family that have a specific type of input/output relation type I/O_R.

Relation(R) operation- The *R* operation on a node N_i returns all its web service instances belonging to all types of web-service I/O relation lists requested. If the node is concrete business family (leave node) then the list can be fetched directly but if the node represents an abstract business family (non-leave nodes) then web-service I/O relation list of all requested types is combined for all possible reachable leaves nodes from N_i to form the result.

ALGORITHM 3. R (Node Ni, BF Code, Relationship I/O$_R$ List)

```
Input: Node Ni, BF code, I/O Relationship list I/OR
Output: List of nodes
Read the BF code;
Convert DBWS tree to Binary tree
Node_Pointer = SearchNodeInDBWSBinaryTree (BF Code)
Node Ni = node at Node_Pointer
  LeavesNodesArray = TraverseDBWS (Ni, BF code)
For each nodes in the LeavesNodesArray
 For each Relationship types in I/OR List in the node
Appends corresponding lists
Return lists
End
```

The time complexity for the above algorithm is $O(n^2)$ where n is the total number of nodes in the tree.

7.5 NAICS and the DBWS Tree

The *DBWS* tree has a very close resembles with *NAICS* approach of business classification. The value of *business family (BF)* code is same as *NAICS* code for business group. Each web services are part of one or other business node. In *NAICS* all business are parts of some business group. This helps in efficient managing large number different businesses. Similarly, concept of *web service family* and *business family* is used to organized large number of dynamic web services instances. The *DBWS* tree represents complete organization of web services in a tree structure using top-down approach. If the height of the *DBWS* tree is six then it represents exact 6 digit code of *NAISC*. Coding schemes of *DBWS* tree and *NAICS* are similar. In *NAICS* also, code of a business group is dependent on code of its parent group and one additional digit is added to create *NAICS* code.

7.6 Use of DBWS Tree in Discovery

The *DBWS* tree can help in the efficient discovery. The *DBWS* tree organized all web services in business families in a top-down approach on the basis of their business context and input/output relationships. This tree can be searched efficiently and quickly to find out lists of relevant web services in context of their business and input/output relationships. This tree can help in reducing the discovery space by arranging web services belongings to similar business groups and input/output relationships. The tree can be traversed to return lists of web services, which have same input/output relationship and have same business context. This lists needs to be filtered further to discover required web services. The *DBWS* tree helps in getting initial list of web services which have potential candidates without traversing or searching complete repository.

7.7 Advantages of the DBWS Tree

The *DBWS* tree helps in efficient searching and traversing business families to select potential services for creating web service family and for service composition.

(1) *Organized web services in top-down tree structure*- The *DBWS* tree organized the business families in a top down hierarchy with more generic business families placed in the top. Top down approach helps in the efficient searching by eliminating the search options in the early stage.

(2) *Business oriented tree structure*- Web services are used in businesses. Functionalities/objectives of web services largely depend on their business context. The *DBWS* tree organized web services on basis of their business context. These helps in efficient discovery.

(3) *Faceted Browsing*- The *DBWS* tree has rich faceted data structure, which is groups in more generic (abstract) business classes and builds a deep hierarchy to support browsing. The *DBWS* tree can be used to browse web services at different level of business hierarchy and within each business further browsing can be done on the basis of different parameters like relationship between input/output parameters, data type of input/output parameters.

(4) *Resembles NAICS*- Similar to the classification and organization of businesses in *NAICS*, the *DBWS* tree also classify and organizes business families with respect to their business context. The coding standards of nodes of the *DBWS* tree are same as codes for business groups in *NAICS*.

(5) *Searching with NAICS code*- The BF code of the DBWS tree is same as that of the NAICS code. It can be used to search the potential web services from the *DBWS* tree.

(6) *Focused at group label rather than at instance level*- The number of web services in the repository are increasing exponentially. Managing large number of web services at its instance level will not be efficient. The *DBWS* tree organized web services at an abstract level rather than at its instance level. This helps in managing large number of web services.

(7) *Potential to assist in efficient web service discovery*- The *DBWS* tree organizes the web services of business families on the basis of their business context and input/output relationships. The *DBWS* tree can be used to fetch lists of web services having specific business context and specific input/output parameters relationship. These lists can be further processes to discover similar web services.

(8) *Open ended to include future business hierarchies and web services*- The *DBWS* tree is very flexible and generic in design, which can easily be extended to include more business groups and web services in the future

(9) *Filters on business context and input/output relationship*- The *DBWS* tree filter the original repository on the basis of web services business context and their input/output parameters. These filters are basic filters which are required in the discovery process. These filters save times in the discovery process as it is carried out in the back end.

8.0 DISCOVERY LAYER

This section discussed the implementation of the discovery layer of the two layered discovery architecture, which focused on the creation of the web service family using the *DBWS* tree. Web service families are created with the help of the *dynamic web service discovery (DWSD)* machine, which manages and executes all remaining activities required for the creation of web service families. This section also discussed a novel concept of *pull wave* operation and *push wave* operation to keep the web service families sync in the real time.

8.1 Synchronization Operations

For the real time updation of web services, it is important to keep web service families in sync with the repository. For this purpose two important operations are introduced, which helps in keeping web service families in sync with the repository.

Pull Wave Operation- This operation is initiated by the *DWSD* module to fetch web services, which falls under a specific business context. The *Pull wave* requests contain details of web service business type and I/O relationships type (I/O_R) between input/output parameters. The *DWSD* machine decides when to initiates a *pull wave* request and also selects an appropriate external *business agent* to which the pull request is passed. The external *business agents* fetch the business context information from the incoming pull requests. It searches business nodes in the *DBWS* tree to find appropriate nodes having requested business group and returns all web services of that node. The external business agent returns *pull wave* response to the requesting *DWSD* machine. The response contains list of web services having requested business context and input/output relationships. Pull wave have both request and response part.

Push Wave Operation- External business agents maintained list of all interacting *DBWS* machines and list of all web services, which are send to different *DWSD* machines as part of the pull wave operation. The push wave operation has only response part. This is initiated by the external *business agent* to notify its registered *DWSD* machine partners about any changes in web services in the repository. This operation is triggered whenever any web services, which are used by the *DWSD* machine is modified in registries. The business agents keep all business families of its *DBWS* tree in sync with the repository. The response contains deletion command if the associated web service is deleted from the repository, updation command if it is updated, or addition command if new similar web service is added. The *push wave* response helps in notifying *DWSD* machines related to the dynamic behavior of independently developed web services.

8.2 Dynamic Web Service Discovery (DWSD) Machine

The *DWSD* is a machine model, which accepts requests for creation of web service families and carries out all activities and interaction with external interfaces required to create web service families. The *DSWD* machine also helps in keeping service instances of web service families in sync with the repository in real time. The *DSWD* machine interacts with external business agents to fetch relevant web services required for the creation of families. The *DWSD* machine has many components. Each component has fixed objectives and all components are loosely coupled, which enables scope for the future enhancement. Some of the important components of the WSDM machine are as follows.

Business Agents- The business agents manages and executes all activities of the first layer. These are developed and managed by third party vendors. These agents play a very crucial role in the construction of web service families. Each business agents are specialized in one or many business domains. They maintain lists of all web services instances falling in their domains. Lists of services are maintained in the form of a *DBWS* tree, which provides quick updation, deletion and modification operations. These agents keep browsing and listening repositories. When any new web service instance is added or any existing web service's *WSDL* files are modified, it updates appropriate nodes of its *DBWS* tree and triggers *push wave* to notify all *DWSD* machines, which are using that particular web service instances. The *Business agents* received requests from the *DWSD* machine for fetching list of all web services, which have same input/output relationship type and belongs to same business context. If requests are triggered from *DWSD* machine to *business agents* then it is called *pull wave*. If *Agents* initiate requests to send any notification to *DWSD* machines then it is called *push wave* operation.

Agent Interface- The agent interface is responsible for all communication to or from business agents. Agent interface establish and negotiate *SLA* with external business agents to get lists of web services on the basis of the business domain it is serving. Once SLA agreement is finalized, *business agents* assign a unique identification number to the *DWSD* machine, which will be used by the external business agents for future communications like sending *push wave* response.

Web Service Execution Engine- This is the engine which accepts the abstract high level design of the composite service either in a form of graphs or as a formal language. It analyzes the composite service to determine the sequence of the flow. It also determines details of the web services corresponding to each abstract service used in the composite service. Fig. 6 represents the Dynamic Web services Discovery (*DWSD*) machine.

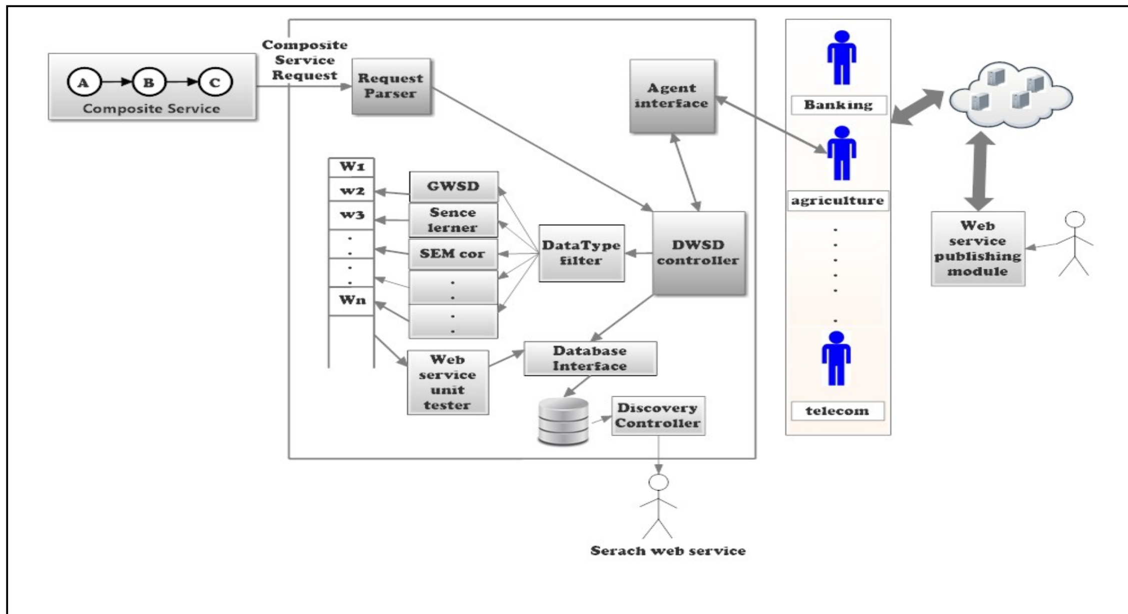

Fig. 6. Dynamic Web Services Discovery (DWSD) Machine

DWSD Controller- This is very important part of the *DWSD* machine. This component controls all process of discovery and web service family creation. As per the incoming web service family request it checks the existing web service database to confirm if it is present , if it is available then it selected and returned to execution engine for its execution otherwise process of creating a web service is initiated. It analyzes incoming web service family requests to determine the types of *business agents* required. It creates and manages SLA, which are used by the *agent interface*. It is the controller, which decides when to send a *pull wave* request to business agents. If the controller receives a *push wave* response containing deletion command then it sends the response directly to the web service family database through the web service family database (*WSDB*) interface

to delete the web service. If *push wave* contains updation or addition command then it sends to the semantic analyzer stack for further filtering and validation.

DataType Filter Module- This module is responsible for filtering web services on the basis of data types of their input/output parameters. It works on the list of web services received from *business agents*.

Semantic Analyzer Stack- This is the stack containing different semantic analyzer tools and algorithms, mostly word sense disambiguation tools like *GWSD*, *senceLerner*, *semCor*, which are used to established semantic similarity of input/output parameters by analyzing ambiguity in their input /output parameters. This also semantically analyses descriptions of web service. They work on the output of the *DataType filter* module. Each semantic analyzer tools analyze independently and its outputs are passed to *web service unit tester*.

Web Service Unit Tester- This module is responsible for unit testing of the web service instances. All outputs (web service) of the *Semantic Analyzer* is executed with sample inputs and it outputs are analyzed to determine whether it full fills requirements as per the request like, data type of outputs, business type of the web service, etc

Web service family database- This is the database where all similar web services are stored in different families. The modification in the database is carried out by a database interface.

Discovery controller- This is responsible for querying the *web service family database* to search out appropriate web service family. When a user searches a web service by specifying the web service family properties then the *discovery controller* uses these properties to search appropriate web service family. Any instance of the selected family can be used as they all are semantically similar.

8.3 Working of the DWSD Machine

A composite service is created and submitted to web service execution engine. Web service execution engine analyze the design and determine the details of the web service family required for its execution. Requests for creating a web service family are received by the *request analyzer,* where requests are validated. It passed the request to the *DWSD* controller. The *DWSD* controller further analyses and prepares *pull wave* request containing business type and input/output relationship type (I/O_R) and invokes the *agent interface*. It also prepares the *SLA*, which is used for establishing connection with *business agents*. The agent interface established connections with appropriate business agents and negotiates as per the SLA. If negotiation is successful then connection is established and the *pull request* is passed to the *business agent*. The agent prepares response, which contains list of all web services having same input/output relation type and business context as per *pull request* and send the response back to the *DWSD* machine's agent interface. The agent interface forward the response to the *DWSD* controller, which analyzed the response and sends it to the *DataType filter* modules to filter out those web services, which have same input/output relationships but their data type are different as per initial web service family request definition. The output of the *DataType* filter is forwarded to all word sense disambiguation tools and algorithms to filter out semantically similar input/output parameters and semantically similar descriptions. The outputs are collected in an array and send to *unit-tester* module to carry out unit testing before saving them in the *web service family database*. In the unit test, each web service is checked against the web-service family definition and also real time request is send to the web service using dummy inputs. Once *web service family database* is ready, the *discovery controller* can query the web service family database to search out the appropriate web service family. As all instances of the web service family hold semantically similar services, any service can be used in the business process flows or in composite web services.

8.4 Advantages of the DWSD Machine

The *DWSD* machine helps in creating and managing web service families. It incorporates some of the remaining filters. It is very flexible and fetches the potential services from the *DBWS* tree.

(1) *Real time discovery*– It supports real time discovery of web services. The *push wave* operations helps in notifying the *DWSD* machine related to any addition of new web services. If any new web services are added to the repository, an appropriate business agent adds them to its data structure (*DBWS* tree) and notifies all *DWSD* machines registered with that agent.
(2) *Real time management of dynamic web service*- It supports real time managements of web services. Whenever any existing web services is deleted or its definition is changed in the registry, appropriate agent, update its details in its data structure and notify all registered *DWSD* machines, which analyze the change request and make necessary changes in their database.

(3) *Validation*- It supports special features of performing validation before adding the web service in a family. As most of participating modules are outsourced and managed by third party, it is very important to carry out unit testing of selected web services to cross check its functionality and validates it before adding in a family.
(4) *Flexible and loosely coupled*-The whole *DWSD* machine is very loosely coupled and flexible, which enable easily future extension and enhancement. Any new modules can be easily integrated in the future
(5) *Works on the existing repository system*- It is based on the existing *UDDI* repository. Information in the existing repositories is processed and fetched to create families.

9.0 EXPERIMENTS AND ANALYSIS

Creations of web service families require fetching and processing information like input/output data type, business context, descriptions etc. This information is analyzed to create web service families by filtering the complete repository. This is a one-time process. In the web service family based discover mechanism, discovery is made by searching a pool of web service families whereas in the traditional keywords based discovery, whole repository needs to be traversed. This section compared the execution time for both approaches.

Web service discovery time largely depends on the time required in processing web services to fetch its relevant information. More the information required to be fetched, more the discovery time. Discovery time also depends on the size of the repository to be traversed. In the cloud environment, network and data communication also have large impact on the discovery time.

To establish and compare the execution time following notations are used.

p = represent properties of web service like data type of input/output parameters, business context of the service, descriptions of the services.

C_{pi} = cost of fetching data related to the property p from a web service w_i.

$\sum_{p=1}^{q} Cpi$ = cost of fetching data related to q number of properties from a web service w_i.

λ = data transfer cost. $\propto$ = network cost. n= number of web services in the repository.

m= number of web service family. q= number of properties of web service under consideration.

FC_{Time} = Total time for creating a web service family

$DT_{keyword}$ - Discovery time for keyword based mechanism.

DT_{family} - Discovery time for family based mechanism.

TCi = Total cost of fetching and processing a web service w_i for q properties.

$$\text{Where } TC_i = \sum_{p=1}^{q} \text{Cpi} + \lambda + \propto \; \ldots\ldots\ldots (1)$$

9.1 Analysis of keyword Based Discovery

In the keyword based discovery mechanism, all web services of the repository are traversed and its description are fetched and analyzed to establish the similarity. In this approach, total discovery time depends on the size of the repository and the time required in fetching description from each web services. In keywords based discovery only one property (keyword) are used. $\sum_{p=1}^{1} C_{Pi}$

$$\text{DTkeyword} \propto \text{n} \qquad \text{DTkeyword} \propto \text{Cpi}$$

$$\text{DTkeyword} = \sum_{i=1}^{n} \text{Cpi} \quad where\ P = 1 \ldots\ldots\ldots (2)$$

Where n= maximum number of web services in the repository.

The keywords based discovery process cannot be used in the compositional based business flows. In the composition based business flows further filtering is required like data type of the input/output parameters, business context etc. The additional filtering will increase the discovery time. More the filters required, more information needs to be fetched and more will be the discovery time. A general formula for calculating discovery time with k filters can be written as below

$$\mathrm{DT} = \sum_{i=1}^{n_1} C_{p_1 i} + \sum_{i=1}^{n_2} C_{p_2 i} + \sum_{i=1}^{n_3} C_{p_3 i} + \sum_{i=1}^{n_4} C_{p_4 i} + \cdots \sum_{i=1}^{n_k} C_{p_n i} \ldots\ldots (3)$$

Where n_1, n_2, n_3, n_4, n_k are size of the web services needs to be traversed. k is the number of filters applied. P_1, P_2, P_3, $P_4 \ldots P_n$ are different properties on which filters are applied. Subsequent filter is applied on the output of the previous filter result i.e. $n_1 \geq n_2 \geq n_3 \geq n_4 \geq \ldots n_k$

In keyword based discovery, the major factor on which the discovery time depends is the size of the repository. In cloud based environment where size of the repository in increasing every day, the discovery time will increase exponentially. Further, results of keywords based discovery are not stored, which can be used in other similar discovery process to improve the discovery time.

9.2 Analysis of Web Service Family Based Discovery

The most time taking process is the web service family creation as it involves many filtering steps and needs to be applied on the complete repository. This web service family creation process is executed only once in the back ground and in the future only needs to keeps it in sync with the repository. Web service family creation time can be calculated using equation 3. Total 5 filters are required in this process.

If all 5 filters are applied in the equation (3), which is used in the creation of web service family then equation can be written as below-

$$FC_{Time} = \sum_{i=1}^{n_1} C_{p_1 i} + \sum_{i=1}^{n_2} C_{p_2 i} + \sum_{i=1}^{n_3} C_{p_3 i} + \sum_{i=1}^{n_4} C_{p_4 i} + \sum_{i=1}^{n_5} C_{p_5 i} \ldots\ldots\ldots\ldots (4)$$

In the web service family based discover most of the filtering activities are carried out in the background and results are organized in groups called as *web service family*. Each *web service family* is very well defined and holds semantically similar web services. Each web service family has a description, which not only describes its common functionality but also specify briefly all other properties required for discovery. In this discovery process, it is not required to traverse the complete repository rather needs to traverse only web service families. The number of web service family will always be less than number web services of the repository.

$$\mathrm{DTfamily} \propto m \qquad\qquad \mathrm{DTfamily} \propto \mathrm{Cpi}$$

$$\mathrm{DTfamily} = \sum_{i=1}^{m} \mathrm{Cpi} \ldots\ldots\ldots (5)$$

9.3 Experiment

A simulation environment was set up using matlab to compare the discovery time of keywords based discovery and web service family based discovery. A random number of web services are created in the matrix with all its relevant information. Random time (millisecond) required for fetching different properties of each web services are also maintained. This matrix, which simulate like a repository is used for keywords based discovery. Discovery time is calculated using equation (2). For calculating discovery time of *web service family* based discovery, same web services of the matrix are grouped in families with each family having some specific number of web services and have a common description. This experiment is repeated for 10 times with different number of web services and the graph is drawn. Fig. 7 represents outputs, when number of web service family increase with the increase in number of web services in the repository. Fig. 6II represents outputs, when number of filters used in the discovery process is increased.

9.4 Analysis of the Result

The outcome of the Fig. 6*I* clearly depicts efficiency of the web service family based discovery as compared to keywords based discovery. With the increase in the number of services in the repository, discovery time of the keywords based process increases very steeply whereas family based discovery time increases gradually. For any number of instances, the family based discovery time is always less than the keyword based discovery time.

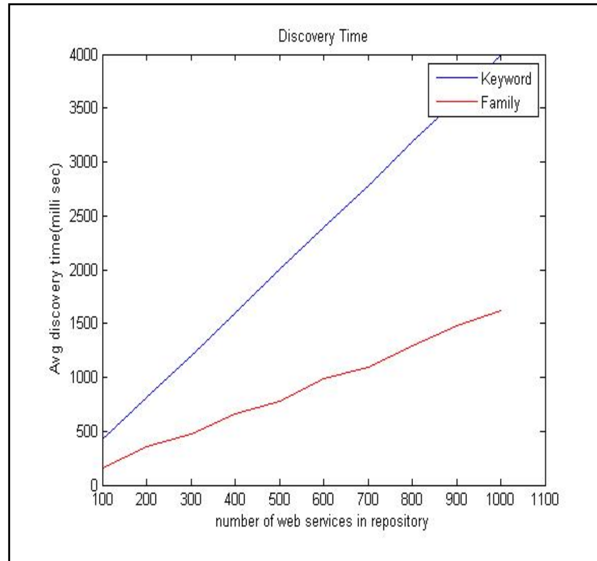

Fig. 7 Comparison of keyword and family based discovery

The output of the Fig. 6*II* illustrates that the impact of increasing number of filters on the keywords based discovery and family based discovery. The increasing numbers of filters have no impacts on the family based discovery whereas discovery time of keywords based discovery processes increases exponentially. The most important reason of this immunity of family based discovery towards the filters is that these filers are applied in the background during the creation of web service families, which is one time process.

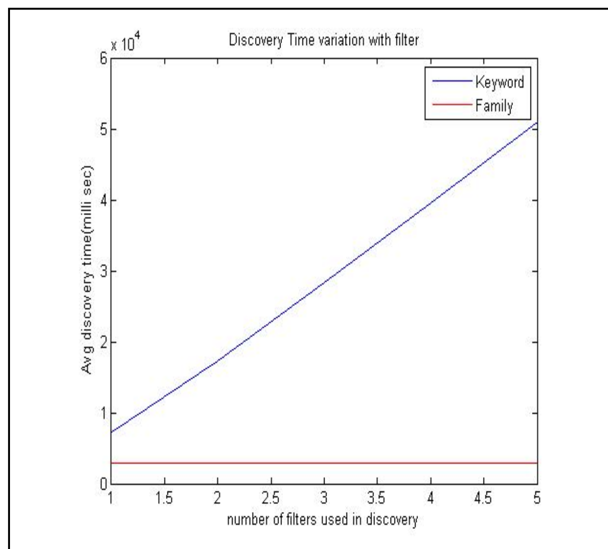

Fig. 8. Variation of discovery time with increasing filters

The family based discovery outperformed in comparison to keywords based discovery. The most important reason is that during the web service family creation all different required filters are applied one time in the background and a single web service family represents many semantically similar web services. Therefor number of web service families in comparison to number of service instance is very less.

10.0 CONCLUSION

This paper has novel approach toward rich analysis of dynamic web services, which helps not only in its managing but it also established an efficient discovery mechanism for searching similar services for composite web services. The concept of web service family is semantically richer concepts than web service community [1] as the service communities are not organised by input/output structures of its member so it needs more time in selecting suitable compatible services. In service composition the business context and input/output structures plays a very important role. The web service family organized the services not only on the basis of functionality but also on the basis of their business contexts and input/output structures. The paper presents the two layer discovery process for creating web service family, which are used efficiently in searching service instances as per the specific need of service composition. Abstract semantic groups of web services named as *business family* is introduced to manage web services at abstract label. These *business families* are further organized to create a dynamic data structure named as *DBWS* tree, which organizes *business families* in top down hierarchy on the basis of their business classification. Different algorithms and operations of the *DBWS* tree in presented. The coding schemes of *DBWS* tree resembles with *NAICS*. The paper focused on the searching of dynamic web services in real time. *Pull and push wave* operation is introduced for managing web service families in real time.

This paper focused on the discovery mechanism by filtering web services on the basis of many parameters, which are required in particular for selecting and executing web services belonging to composite services. The paper targets discovering web services for large scale business processes, which are built as composition (orchestrating or choreographing) of web services. Some of the key features of the proposed approach are as follow.

(1) *Existing systems are not affected*- All solutions and approaches are built using existing repositories and standards. Web service families, which are the basis of this discovery, are built using the information, which can be easily fetched from web services using existing repositories and standards.

(2) *New dynamic data structures are introduced*- New data structures like *business families* and *DBWS* tree are introduced to reorganized web services in more dynamic structure, which can help in managing services efficiently

(3) *Discovery time is improved*- Web service family based discovery has very less discovery time as compared with the existing keyword based discovery. Web service families are created by applying filters in back ground, which saves processing time. At any instance, number of web service families as compared to number of web services is always less.

(4) *Selection of web service used in composite service*- The proposed discovery process is very suitable for selecting and executing web services, which are used in large scale business processes as composition or orchestration. The web service family organizes the services not only on the basis of functionality but also on the basis of business context and input/output structures.

(5) *Real time updation of web service families*- *Pull wave* and *push wave* concept is used to keep web service families in sync with repositories.

The main focus of the future work will be towards building a working model of the proposed approach to test the above mentioned discovery approach and web service selection process for composite services in real time scenario. The future work shall also focus on the web service scheduling using the concept of web service family.

REFERENCES

[1] Sheng, Q.Z., Benatallah, B., Dumas, & M. Mak, E.O-Y. “SELF-SERV: a platform for rapid composition of web services in a peer-to-peer environment”, *Proceedings of the 28th VLDB Conference*, Hong Kong, China, 2002, pp 1051-1054,.

[2] Faisal Ahmad, Anirban Sarkar, Narayan C Debnath, "Analysis of Dynamic Web Services", *IEEE International Conference on Computing, Management and Telecommunications (ComManTel 2014)*, Vietnam, April 27 – 29,2014, PP 275 – 279.

[3] Sun, H., Wang, X., Zhou, B. and Zou, P., “Research and Implementation of Dynamic Web Services

Composition", *Advanced Parallel Processing Technologies (APPT)*, LNCS 2834, Springer-Verlag Berlin Heidelberg, 2003, pp.457–466.

[4] Raj, R.G., Abdul-Kareem, S., "Information Dissemination And Storage For Tele-Text Based Conversational Systems' Learning", *Malaysian Journal of Computer Science*, Vol. 22(2):2009. Pp. 138-159.

[5] Preuner, G. and Schrefl, M., 'Integration of web services into workflows through a multilevel schema architecture', Proceedings of the 4th IEEE Int'l *Workshop on Advanced Issues of E-commerce and Web-based Information Systems (WECWIS*), 2002.

[6] Hansen, M., Madnick, S. and Siegel, M., Bussler, C. et al. (Eds.): "Process Aggregation Using Web Services*", Web Services, E-Business, and the Semantic Web WES, LNCS* 2512, Springer-Verlag Berlin Heidelberg, 2002,pp.12–27.

[7] Moohebat, M., Raj, R.G. , Kareem, S.B.A., Thorleuchter, D., "Identifying ISI-indexed articles by their lexical usage: A text analysis approach", *Journal of the Association for Information Science and Technology*, Vol. 66, No. 3, pp. 501–511. doi: 10.1002/asi.23194.

[8] X. Dong, A. Halevy, J. Madhavan, E. Nemes and J. Zhang. "Similarity Search for Web services". *In Proceedings of the 30th VLDB Conference*, Toronto, Canada, 2004.

[9] W. Abramowicz, K. Haniewicz, M. Kaczmarek and D. Zyskowski. "Architecture for Web services filtering and clustering". In *Internet and Web Applications and Services*, (ICIW '07), 2007.

[10] R. Nayak and B. Lee. "Web service Discovery with Additional Semantics and Clustering". In *Web Intelligence, IEEE/WIC/ACM International Conference*, 2007.

[11] I. Constantinescu, W. Binder and B. Faltings. "Flexible and efficient matchmaking and ranking in service directories". In Proceedings of the *IEEE International Conference on Web Services* (ICWS'05), 2005.

[12] L. B. Huang, V. Balakrishnan, R.G. Raj, "Improving the relevancy of document search using the multi-term adjacency keyword-order model." *Malaysian Journal of Computer Science*, Vol. 25, No. 1, 2012, pp. 1-10.

[13] Marco Aiello, Christian Platzer, Florian Rosenberg, Huy Tran, Martin Vasko, Schahram Dustdar, "*Web Service Indexing for Efficient Retrieval and Composition*", Proceedings of the 8th IEEE International Conference on E-Commerce Technology and the 3rd IEEE, 26-29 June 2006, pp 63.

[14] Fatih Emekci, Ozgur D. Sahin, Divyakant Agrawal, Amr El Abbadi, "A Peer-to-Peer Framework for Web Service Discovery with Ranking", In proceedings of the *IEEE International Conference on Web Services* (ICWS'04), 2004.

[15] Benatallah,B Dumas,M , & Sheng, Q Z. "Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services". *Distributed and Parallel Databases*,Vol 17, 2009, pp. 5-37.,.

[16] Faisal Ahmad, Suvamoy Changder, Anirban Sarkar, Title – "Architectural framework for web service dynamics: a layered approach", *22nd International Conference on Software Engineering and Data Engineering,* September, 2013.

[17] Faisal Ahmad, Suvamoy Changder, Anirban Sarkar, "Web service execution model for cloud environment", *ACM SIGSOFT Softw*. Eng. Notes Vol. 38, No. 6 (November 2013), PP 1-13.

[18] NAICS: North American Industry Classification System, http://www.census.gov/eos/www/naics/ (accessed 10 Jan. 2014). http://www.naics.com/history-naics-code/

[19] N. A. Saadon ,R. Mohamad , "Cloud-based Mobile Web Service Discovery framework with semantic matchmaking approach"*, Software Engineering Conference (MySEC*), 2014 8th Malaysian ,pp 113 – 118.

[20] S. Hamza; Mohamed khider, B.Aïcha-Nabila ; K. Okba ; A. Youssef, "A Cloud computing approach based on mobile agents for Web services discovery"*, Second International Conference on Innovative Computing*

Technology (INTECH), pp 297 – 304, 2012.

[21] G. Khan; S. Sengupta; A. Sarkar, "*WSRM:* A Relational Model for Web Service Discovery in Enterprise Cloud Bus (ECB)"*, 3rd International Conference on Eco-friendly Computing and Communication Systems (ICECCS*), pp 217 – 222, 2014.

[22] Xizhe Zhang ;Shenyang, China ; Shuai Feng ;Ying Yin ;Bin Zhang, "Community discovery of public cloud web services based on structural networks"*, Ninth International Conference on Natural Computation (ICNC)*, 2013,pp 1129 – 1133.