Chapter 2
Semantic Support for Change Management

Semantic support is always considered as the key enabler for automation. With the help of machine-processible semantics, software agents can understand the functionality and performance of a Web service. They can then determine whether a service should be selected as a component of a LCS and how to integrate the service with other components.

Semantic Web service technologies have been proposed to automate the usage of Web services, such as service discovery and service composition. The existing approaches can be classified into two categories. The first type of approaches adds semantic markup information to a service's WSDL description [93, 23, 96]. The input, output, and operations of a Web service are annotated with machine-understandable information. Such semantic annotations can help software agent software agents discover and invoke a Web service. The limitation of this type of approaches is that they mainly focus on the semantics within a service description. They do not capture the inter-service relationship, which can be used to facilitate in locating Web services in a large scale and composing different services together. The second type of approaches builds up a meaningful organization of Web services. The services having the same functionality are grouped into the same categories, *a.k.a.*, *service communities* [62]. When a service provider registers the service to a service registry, such as UDDI, it will publish the service description, together with the information about its associated service community. The limitation of this type of approaches is that they lack sufficient semantic description for Web services. Moreover, service communities are *flat*. Querying a service community is always keyword-based and not optimized. In additional to their limitations, different approaches have different assumptions and focuses. This also makes them not suitable enough for supporting automatic change reaction.

In this chapter, we propose a semantic support that are expected to play a key role in automating the process of change management. The central of the semantic support is a Web service ontology that has a hierarchical structure,

X. Liu et al., *Change Management for Semantic Web Services,*
DOI 10.1007/978-1-4419-9329-8_2, © Springer Science+Business Media, LLC 2011

which leverages the advantages of current approaches and addresses their limitations.

2.1 A Tree-Structured Web Service Ontology

In this section, we propose a *Web service ontology* that provides enough semantics for the schema modifier and service selector. As depicted in Figure 2.1, the ontology has a tree-like structure. Each node in the tree represents a *service concepts*. It also captures the relationship between different nodes, which can be leveraged for efficiently locating and composing Web services.

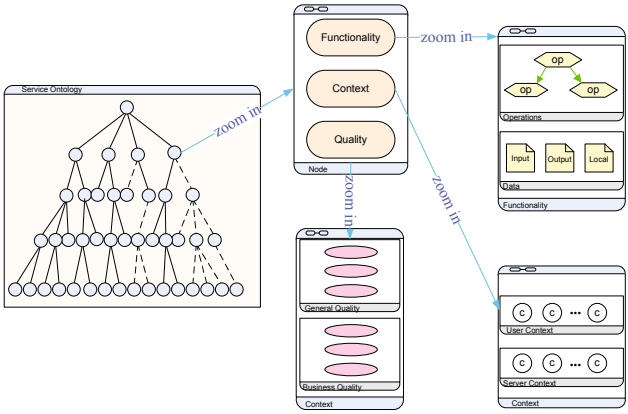


Fig. 2.1 A Web service ontology

2.1.1 Service Concepts

A service concept defines a type of Web services within a domain. It can be considered as a service category, which groups Web services based on their functionality. Using the service concepts of the ontology, Web services' functionalities are defined in a way that is clear and unambiguous to software agents. For example, classifying a Web service instance w to the "AirBooking" node of the ontology implies that it is an air reservation service. This ontology also facilitates a LCS's owner to establish its LCS. The owner can choose a node of this ontology as an abstract service to define the LCS schema.

A Web service has three major properties: *service functionality* ($\mathcal{F}^{\mathcal{S}}$), *service context* ($\mathcal{C}^{\mathcal{S}}$), and *service quality* ($\mathcal{Q}^{\mathcal{S}}$). $\mathcal{F}^{\mathcal{S}}$ specifies what a service offers to its users. $\mathcal{C}^{\mathcal{S}}$ constructs the environmental situation that a service operates with. $\mathcal{Q}^{\mathcal{S}}$ defines a set of parameters for service evaluation and selection. By

using the information delivered by these three features, a LCS can determine whether to choose a service as its member (i.e. whether the service provides the desirable functionality with satisfactory context and quality). These three features are elaborated on in the following sections.

2.1.1.1 Functionality

A service functionality ($\mathcal{F}^{\mathcal{S}}$) is collectively delivered by a set of *service operations*. The process of accessing a service is actually invoking one or more operations provided by the service. The operations consume the service input and generate the output of the service.

It is worthy to note that there may be dependency relationships between different service operations. For example, an Airline service may provide several operations, such as `user_login` and `flight_reservation`. Typically, an user needs to login before (s)he can reserve an air ticket. Therefore, there is a dependency between `user_login` and `flight_reservation`. The dependencies between service operations (referred to as *operation-level dependencies*) need to be strictly enforced when accessing a service.

A service data is also an essential aspect of a service functionality. A service is actually affected by the outside with a set of input and responses with a set of output [2].

Therefore, we can define a service functionality as a binary: service data (D) and service operations (OP). D is an essential aspect of the behavior of a service. A service can be viewed as a transducer that translates a set of input items into a set of output items [2]. Besides of these input (I), output (O) data, D also contains internal data (N), which the service generates and consumes them by itself. OP consists of a set of operations and their dependency relationships.

There are several ontology languages such as OWL-S and WSMO [23, 96] that we can use to describe the proposed ontology. Take OWL-S as an example. There is a mapping between the proposed service ontology and the OWL-S definition. A service data can be mapped to the property of hasInput, hasOuput, hasLocal. A service operation can be mapped to the definition of an atomic process. The relationship between two operations can be defined by a composite process in an OWL-S ontology.

Suppose that an airline service provides a flight ticket reservation functionality. It has the service data as follows:

(I)={user_information, departure_date, arrival_date, departure_city, arrival_city}.
(O)={flight_information}
(N)={login_confirmation, payment_information}
The airline service also has the service operations as follows:

user_Login: It takes user_information as input and generates the output of login_information and payment_information.

flight_Reservation: It takes the input of login_confirmation, payment_information, departure_date, arrival_date, departure_city, arrival_city and generates the output of flight_inforamation.

It can be specified in OWL-S as depicted in Figure 2.2.



Fig. 2.2 An example of an airline service ontology in OWL-S

2.1.1.2 Quality

The service quality ($\mathcal{Q}^S$) is the main criterion to evaluate a service [26, 65, 69, 47]. A Web service is always associated with a service in the real world (e.g. an airline company or a hotel). Therefore, we define the service quality in twofold: *general quality* (Q_G) and *business quality* (Q_B).

- General quality consists of a set of domain-free parameters that measure the quality of the service that is consumed through the Web. It mainly presents the measurement of properties that are related to the access of an operation *op* of a Web service. Examples of these parameters include *availability*, *reliability*, and *latency*. The general quality also includes the security requirements that are compliant with an operation *op* of a Web service. Examples of the parameters include *encryption*, *authentication*, and *reputation*.
- Business quality consists of a set of domain-specific parameters that measure the quality of the service that is consumed in the real world. For

example, in a travel domain, the parameters can be the *fee* that is charged to access a service, *comfort_Lodge* for how comfort is a lodge service, and *convenience_Internet* for how convenient the access to the Internet.

Table 2.1 Measurement of QoWS parameters

Category	Parameter	Definition	Domain
General	Latency	$\text{Time}_{process}(op)$ + $\text{Time}_{results}(op)$ where $\text{Time}_{process}$ is the time to process *op* and $\text{Time}_{results}$ is the time to transmit/receive the results	number
	Reliability	$\text{N}_{success}(op)/\text{N}_{invoked}(op)$ where $\text{N}_{success}$ is the number of times that *op* has been successfully executed and $\text{N}_{invoked}$ is the total number of invocations	number
	Availability	UpTime(*op*)/TotalTime(*op*) where UpTime is the time *op* was accessible during the total measurement time TotalTime	number
	Encryption	Equal to *1* iff messages are encrypted	number
	Authentication	Equal to *1* iff consumers are authenticated	number
	Reputation	The degree of confidence that a service will deliver the quality it promises, ranging from 0 to 1.	number
Business (travel domain)	Fee	Dollar amount to execute the operation	number
	comfort_Lodge	The degree of comfort provided by a lodge service, ranging from 0 to 1	number
	convenience_Internet	The degree of convenience of accessing Internet, ranging from 0 to 1	number

We propose a QoWS model as depicted in Table 2.1. It is not meant to be exhaustive and can be extended. It gives the formal definitions of a set of representative quality parameters. Other quality parameters can be added to extend the current model.

2.1.1.3 Context

The context of a Web service ($\mathcal{C}^{\mathcal{S}}$) models the environmental situation that a service resides in. It can be any metadata that explicitly describes the meaning of the data items exchanged between Web services. For example, *context=(currency, dollars)* defines a context of currency in dollars. Therefore, $\mathcal{C}^{\mathcal{S}}$ contains the important information for correctly interacting with a service and needs to be modeled in a formal way.

Since a service context can be of any type of metadata, there are many types of contexts, such as *location*, *time*, *currency*, *temperature*, *weight*, *length*, and *users*. Therefore, we define C as a set of context types that are related to the service invocation. $C = \{c_1, ..., c_n\}$, where c_i is a context type.

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
    <env:Header>
        <Context xmlns="http://sg.fmi.uni-passau.de/context">
            <Location>
                <address useType="Office">
                    <addressLine keyName="Street" keyValue="60">Fairfax Rd</addressLine>
                    <addressLine keyName="City" keyValue="40">Falls church</addressLine>
                </address>
            </Location>
            <Client>
                <payment useType="Online Payment">
                    <paymentLine keyName="Type" keyValue="20">Visa</paymentLine>
                    <paymentLine keyName="IssueBy" keyValue="30">the United States</paymentLine>
                </payment>
            </Client>
        </Context>
    </env:Header>
        <env:Body>
            <!-- serialized object data -->
        </env:Body>
</env:Envelope>
```

Fig. 2.3 A context transported in a SOAP header block

WS-context is an OASIS standards to manage the lifecycle of a Web service context [67]. In WS-Context, the context lifecycle is as follows. First, to initiate an activity, a service requests a new context from the WS-Context service via a begin message and specifies a timeline for sharing the context. The begin action will then return a begin message plus a context. In this case, the interaction with the service will be associated with the context delivered in a SOAP message [90]. The context-aware service interaction will be terminated once by timing out or by explicitly instructing the context service to end. Figure 2.3 shows an example of a SOAP message that delivers a context information.

2.1.2 Service Relationships

We define two types of relationships between two nodes in the ontology tree: *inheritance* and *dependency*. The inheritance is between two service nodes if one inherits its features (e.g., service data and operations) from another one. This relationship can be used to facilitate in service discovery. Dependency is between two service nodes if one's invocation relies on another's invocation. This relationship can be used to automate service composition. We elaborate on these two types of relationships as follows.

2.1.2.1 Inheritance Relationship

In the traditional ontology system, an ontology only contains the relationship of "consumption" (i.e., is-a) between a node and its children. That is, the children inherit all the properties from the node. This feature is not perfectly in line with the ontology designed for a Web service domain, where partial consumption relationship (i.e., has-of) may exist between two nodes [31]. These two types of inheritance are described as follows.

- An *is-a* relationship is between a node and its parent if the node inherits all the properties (i.e., operations and service data) of the parent. The node also has the properties that the parent node does not have. For example, an Airline service is a child of a Transportation service. It provides the transportation service with a special feature, i.e., through a flight. Therefore, there is an *is-a* relationship between the Airline service and the Transportation service.
- A *has-of* relationship is between a node and its parent if the node partially inherits the properties from the parent. For example, a Flight Quote service is a child of an Airline service. It only provides the service of getting the quote of a flight, but not other airline-related services, such as checking flight status, reserving a flight, or electronic check in. Therefore, there is a *has-of* relationship between the Flight Quote service and the Airline service.

One of the important evaluation criteria of defining an ontology is to check whether the definition can achieve the common understanding among different parties. To achieve this goal, we allows these two types of inheritance in our proposed ontology to make it more intuitive. The inheritances are represented in different ways. We use a line to denote an is-a relationship and a dashed line to denote a has-of relationship.

2.1.2.2 Dependency Relationship

It is worth to note that although Web services are autonomous and they can be invoked independently, some dependency constraint may need to be enforced when they are combined together. For example, a user can invoke an airline service and a hotel service individually. However, if a user wants an airline+hotel package, the check-in and check-out information of the hotel service depends on the flight information, which is the output of the airline service. The dependency relationships between different types of services within a domain are usually generated by domain experts.

A dependency constraint can be specified as a triplet $\{S_1, S_2, D\}$. It means that the invocation of S_1 depends on the invocation of S_2 when they cooperate together since S_2's output will be used as the input of S_1 with respect to the data items included in D. The dependency constraint is not necessarily transitive. That is, if S_1 depends on S_2 and S_2 depends on S_3, S_1 does not

necessarily depend on S_3. Suppose only S_1 and S_3 attend the cooperation, the dependency constraints $\{S_1, S_2, D_1\}$ and $\{S_2, S_3, D_2\}$ do not need to be enforced since S_2 is not involved in the cooperation. In this case, there may be no dependency relationship between S_1 and S_3.

2.2 Heuristic Query Process

Reacting to a change requires to refer back to the hierarchical structure of the corresponding service ontology, seeking for desired semantics. In this section, we propose an approach to efficiently query the service ontology.

2.2.1 Query Categories

Querying a Web service ontology is required to retrieve the desired service nodes and locate Web services that subscribe to them. Meanwhile, the dependency relationship between different types of services also needs to be retrieved for composing these services together. Therefore, there are three types of ontology queries required as below.

- *Functionality-based service query:* It is to traverse a Web service ontology to find a service node which offers a specified functionality. Since a service functionality has two facets: service operations and service data, there are two types of queries here: operation-based and data-based. This query will be used to update the list of a LCS schema's abstract services.
- *Dependency relationship query:* : It is to check the dependency relationship between two given service nodes. This query will be used to automatically modify a LCS schema.
- *Web service query:* It is to find a list of Web services that offer the functionality defined by the given service node. This query will be used to automatically locate a Web service.

Retrieving the dependency relationship is always performed after identifying the service node in a Web service ontology. Since the information is included in the definition of the service concept, the query can be easily done by looking into the definition. Retrieving the list of Web services given a service node can rely on the association between service nodes and Web services. The association is built when creating the service node and maintained after that. It is always stored in a service registry, such as UDDI registry. By looking up the association, the desired Web service list can be obtained.

2.2.2 Heuristic Query Process

The structure of ontology is hierarchical and extensible by nature. Once the ontology structure becomes large with the increment of the available service functionalities, the process of identifying a proper piece of functionality would turn out to be time consuming. It is important and beneficial for change reaction to make this process efficient and accurate.

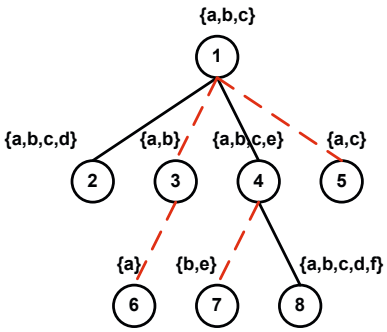


Fig. 2.4 An ontology tree

An intuitive heuristic query can be achieved by leveraging the difference between is-a and has-of relationships. That is, if a service node does not contain the required operations, its has-of children should not contain either. Therefore, only its is-a children will be further explored. For example, in the tree depicted in Figure 2.4, suppose we want to find a node providing $\{a, b, c, d, f\}$ (here, a and f refer to service operations or service data). The query starts at the root node. Since the root node does not provide d and f, the query will be further performed in the is-a children. Therefore, node 2 and node 4 will be checked. Continue the process until the desired node (i.e., node 8) is found. The process is described in Algorithm 1.

Algorithm 1 Heuristic Query on a Web Service Ontology Tree

```
1: Function HQ(r, Q) {r is a service node, Q is a set of required operations or data}
2: if (r.P == Q) return r; {find the match}
3: if ((Q-r.P) <> ∅){the current node does not cover the required set}
4:    N = get_Is_a_Children(r);
5:    for all (n ∈ N)
6:      S=HQ(n, Q); {the search will be performed on the is-a children}
7:      if (S <>NULL) return S;
8:    return NULL;
9: if (r.P-Q) <> ∅) {the current node provides more than required}
10:    N = get_Has_of_Children(r);
11:    for all (n ∈ N)
12:     S=HQ(n, Q; {the search will be performed on the Has-Of children}
13:      if (S <>NULL) return S;
14:    return NULL;
```

There is an assumption of this approach. That is, there is no *confusing nodes* in an ontology tree. A confusing node is the one that is a has-of child of another node and it also has is-a children. The confusing node may cause the potential desired node to be ignored due to the heuristic pruning. For example, in the tree depicted in Figure 2.5, node 7 is a has-of child of node 3 and it has an is-a child node 13. Suppose that we want to find a node that provides {a, k} (i.e., node 13). We cannot find the node following the process in Algorithm 1 since node 13 is unreachable.

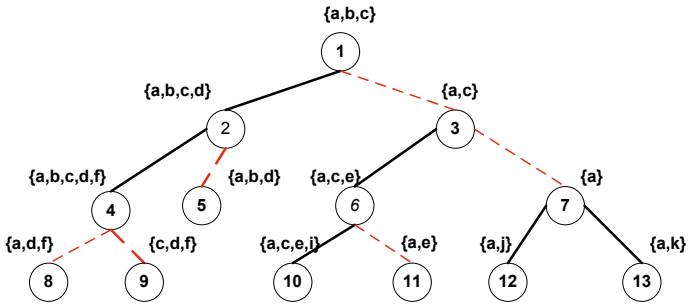


Fig. 2.5 Another ontology tree

Allowing confusing nodes in a Web service ontology tree will improve the flexibility of designing the service ontology. It can also make the ontology structure more intuitive for understanding. Therefore, the proposed query scheme is quite limited. To address the limitation, we first identify the confusing nodes in an ontology tree. We then modify the heuristic query process to ensure that the potential desired node won't be missed.

Algorithm 2 shows the steps of identifying confusing node by coloring the ontology tree. The tree nodes are initially *white*. If a node is identified as a

confusing node, it will be colored red. The coloring process starts at the root node. All the has-of children of the current node will be colored *pink*, showing that they are the potential confusing node (Lines 5-6). If a *pink* node also has is-a children, it is then identified as a confusing node and colored *red* (Lines 7-8). Once a node is colored red, its parents will also be colored *red* to ensure that the confusing node won't be ignored during the query process (Lines 11-13).

Algorithm 2 Tree Coloring

```
1: Initially, all the nodes in the tree are colored white
2: Procedure Color_Tree(r)
3: N_i=get_Is_a_Children(r);
4: N_h=get_Has_of_Children(r);
5:  for all (n ∈ N_h)
6:    n.color=pink; Color_Tree(n);
7: if (r.color==pink) and (N_i <> ∅)
8:    Red_Color(r)
9:    for all (n ∈ N_i)
10:     Color_Tree(n);
11: Procedure Red_Color(r)
12: r.color=red; n=get_Parent(r);
13: if (n.color <> red) Red_Color(n);
```

Figure 2.6 shows the result of applying Algorithm 2 to the ontology tree.

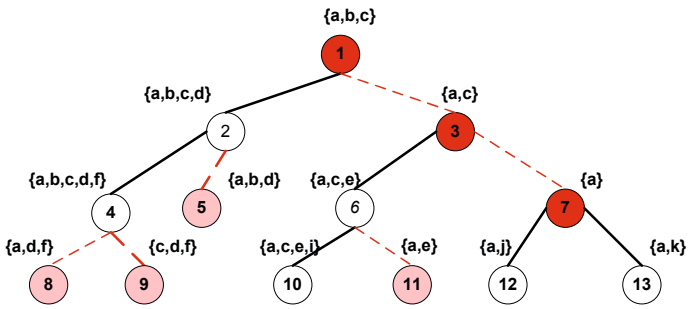


Fig. 2.6 The colored ontology tree

Algorithm 3 shows the steps of the querying the colored tree. The heuristic query process is modified to ensure that the desirable node will be located with the existence of confusing nodes. For the *white*-colored or *pink*-colored nodes, the query process is performed in the similar way as in Algorithm 1. The difference is, for the red-colored node, the query will be performed on both is-a and has-of Children of the node (Lines 3-8). The advanced query

process keeps the heuristic by pruning the search space for non red-colored nodes, as well as ensures the reachability of any targeted nodes.

Algorithm 3 Advanced Heuristic Query

```
1: Function IHQ(r, 𝒬) {r is a service node, 𝒬 is a set of required operations or data}
2: if (r.𝒫 == 𝒬) return r; {find the match}
3: if (r.color==red) {the current node is a confusing node}
4:    N =getr_all_Children(r);
5:    for all (n ∈ N)
6:      S=IHQ(n, 𝒬); {the search will be performed on all children}
7:      if S <> NULL return S;
8:    return NULL;
9: if ((𝒬-r.𝒫) <> ∅){the current node does not cover the required set}
10:   N = get_Is_a_Children(r);
11:   for all (n ∈ N)
12:     S=HQ(n, 𝒬); {the search will be performed on the is-a children}
13:     if (S <>NULL) return S;
14:   return NULL;
15: if (r.𝒫-𝒬) <> ∅) {the current node provides more than required}
16:   N = get_Has_of_Children(r);
17:   for all (n ∈ N)
18:    S=HQ(n, 𝒬; {the search will be performed on the Has-Of children}
19:     if (S <>NULL) return S;
20:   return NULL;
```

The proposed query process can be used to perform an exact-match query, i.e., looking for the node provides exactly what it is required. It can also be extended to perform a partly-match query. For example, finding a node provides more than or less than what it is required. For the sake of space, the extension of the algorithm will be included in the future work.

http://www.springer.com/978-1-4419-9328-1

Change Management for Semantic Web Services
Liu, X.; Akram, S.; Bouguettaya, A.
2011, XVII, 164 p., Hardcover
ISBN: 978-1-4419-9328-1