

A Web Service Registry for an Assisted-SOA

Zeina Azmeh

▶ **To cite this version:**

Zeina Azmeh. A Web Service Registry for an Assisted-SOA. RR-11003, 2010, pp.25. lirmm-00558917

HAL Id: lirmm-00558917

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00558917

Submitted on 24 Jan 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Web Service Registry for an Assisted-SOA

≪Research Report≫

Zeina Azmeh

October 15, 2010

Contents

1 Introduction **1**
1.1 Web Service Challenges 2
1.2 A Web Service Registry Proposition 2
1.3 Organization of this report 3

2 Automatic Web Service Tagging **4**

3 FCA-Based Web Service Classification **5**
3.1 Case study 5
3.2 The keywords lattice 7
3.3 The similarity lattice 8

4 QoS-Based Web Service Ordering **13**

5 Web Service Composition **15**
5.1 RCA-Based Web Service Classification 15
5.2 Web Service Triangles Approach 18

6 Perspectives **21**
6.1 Enhancing Coma++ with WordNet 21
6.2 WordNet for a WSDL's domain of functionality 21
6.3 Modelling abstract business processes 22

1 Introduction

A web service-based application in SOA vision can be built by discovering and composing sets of interoperable services, according to some specified functionality. A developer has to specify the pieces of functionality needed inside an aimed application, and each piece of functionality is defined as a set of parallel and sequential consecutive tasks. In other words, a piece of functionality inside an application is defined as a workflow of operations, offered by one or more services. This means that for each specified task the developer must discover a web service that offers an operation satisfying this task. This poses a number of challenges because a developer might not have enough knowledge about all of the existing service providers where he can find the required services. On the other hand, if many providers offering a needed service were found, a time-consuming comparison will be required in order to choose the provider offering the most suitable service. Many aspects must be taken into consideration while selecting a service to fulfill a task in the composition. This includes the functional aspect, the non-functional (QoS) and the needed adaptations to compose this service with other ones in the composition. Composing two consecutive services in a workflow implies the composition of two operations, an operation of each service. This necessitates the assignment of certain output parameters of the first operation with certain input parameters of the second one. This output-input assignment may not always be accomplished directly, because there may be some differences between the parameters and their data types that requires performing some adaptations. Thus, constructing a web service-based composite application can have variable levels of complexity, regarding the needed time and effort to build each part of functionality, consisting of finding and composing the required services, with performing the necessary adaptations. The situation may become more complicated if one or more of the composed services becomes no more usable. This will necessitate the replacement of these services with functionally similar services that can fulfill the left functional gap. Hence, finding services' substitutes, modifying services' proxies and performing new adaptations.

So briefly, the process of constructing a web service-based application consists of:

- Specifying the needed pieces of functionality inside the application, ex. billing, payment and shipping;
- Each piece of functionality is a workflow of tasks;
- Each task is an operation offered by a web service;
- To satisfy each needed piece of functionality, a search for services must be carried out for each specified task
- Service providers must be found and in case of multiple similar services, a compromise between functional and non-functional aspects of a service must be done, in order to choose the most suitable service, in addition to minimizing the required adaptations when composing the services;
- Services offering the needed tasks must be composed together, taking into consideration the needed adaptations;

- After the long process of discovering and composing the needed services, if a service becomes no more usable after a while, it must be replaced by another one that can fulfill the functional gap, after performing the needed modifications and adaptations.

1.1 Web Service Challenges

Web services face several challenging issues coming from several factors. Since, they are offered by various providers, remotely accessed, and sometimes provided for free, there is no guarantee of a continuous execution. An available functioning web service may crash and become unavailable at any time, which necessitates finding an equivalent one to replace it. Unfortunately, this can be hard to achieve since there is a lack of WSDL organizing facilities, especially after the deficiency of the UDDI[1] registries: "UDDI did not achieve its goal of becoming the registry for all Web Services metadata and did not become useful in a majority of Web Services interactions over the Web"[2].

1.2 A Web Service Registry Proposition

We propose a web service registry, a broker that mediates between service providers and service consumers, considering that a service consumer is a developer of web service-based composite applications. This registry enables providers of publishing their well-described documented services, and providing their estimated QoS information related to these services. It adapts two classifying techniques: Formal Concept Analysis (FCA) and Relational Concept Analysis (RCA).

The published services are organized into domains of functionality and in each domain similar services are classified in order to reveal replaceable services. We build our classification using FCA, which is a method of data analysis that takes as input a set of objects and a set of attributes with the relation between them indicating which objects have which attributes. After analyzing the input data, it clusters the objects into concepts depending on their attributes, and reveals the relationships between these concepts, thus, the relationships between the objects. The concepts can then be ordered in a lattice structure for a better browsing and navigation of objects and relationships among them. We adapt this method for classifying web services by considering that the set of objects is a set of web services, and the set of attributes is the set of the offered operations. The resulting classification is represented as a concept lattice for web services, from which we can easily select a needed service by browsing its offered operations, and we can also find the possible substitutes of a selected service. Using this method, services are also classified according to their QoS attributes specified by the providers. This enables services consumer of finding an aimed service depending on its functional and non-functional aspects, as well as finding its possible substitutes. A service consumer may either query the registry for services offering a specific operation, or he can design an abstract business process that describes his aimed composite application, in which he specify the set of needed tasks ordered in a workflow. The registry will analyze

[1] http://www.uddi.org/pubs/uddi_v3.htm

[2] E. Newcomer and G. Lomow, Understanding SOA with Web Services (Independent Technology Guides). Addison-Wesley Professional, 2004.

the specified abstract process and finds the set of web services that can fulfill each task, by querying the corresponding service lattice. After discovering the sets of services that can fulfill the tasks in order to satisfy the functionality required by the abstract process, another classification is built that reveals the composable services according to each pair of consecutive tasks. This second classification is based on an extension of FCA called RCA.

Inside the registry there are techniques to:

- Infer the domain of functionality for each published service, with the help of the WordNet ontology;
- Building FCA-based classifications for the similar services taking into consideration their QoS values, in order to have a better browsing of functionality and the substitutability relations;
- Designing and analyzing abstract business processes, then finding the candidate services that can fulfill the needed tasks, depending on the functional, non-functional aspects, and the preferences provided by the consumer;
- Building RCA-based classifications for the discovered services, in order to classify them by their composability levels according to each pair of consecutive tasks;
- Selecting the optimal set of composable services, by doing a compromise of QoS and adaptations levels, according to the consumer preferences;
- Keeping track of consumers and their service compositions, and notify them in case of a service failure with possible backups;
- By keeping track of service compositions, the registry acquires knowledge about the new resultant composite functionality;
- Consumers can combine several operations while designing their composition, under a new operation name, like this the registry learns the possible operation composition to realize a requested operation.

1.3 Organization of this report

The organization of this report represents the sequence of ideas of our work. Our first step is to extract the tags (keywords) in each WSDL file, which is explained in section 2. These keywords are used afterwards to classify the services into lattices using FCA, as we will see in section 3. We then explain in section 4 how to order a set of services according to their various QoS aspects. After classifying the services by their similar functionality, and their QoS aspects, we need to find a set of composable services, according to their QoS values and composability levels, illustrated in section 5. We present our future work in section 6.

2 Automatic Web Service Tagging

We propose an approach that automatically extracts a set of relevant tags from a WSDL [5]. We used a corpus of user-tagged services to learn how to extract relevant tags from untagged service descriptions. This approach adapts techniques from text mining in order to extract candidate tags out of a description, and techniques from machine learning to select relevant tags among these candidates.

We model the tag extraction problem as the following classification problem: classifying a word into one of the two *tag* and *no tag* classes. Our overall process is divided into two phases: a *training* phase and a *tag extraction* phase.

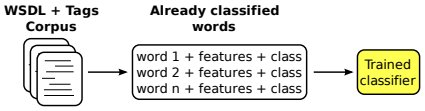


Figure 1: The training phase.

Figure 1 summarizes the behavior of the training phase. In this phase we have a corpus of WSDL files and associated tags, extracted from Seekda. From this training corpus, we first extract a list of candidate words by using text-mining techniques. Then several *features* (metrics) are computed on every candidate. A *feature* is a common term in the machine learning field. As an example, it may be the frequency of the words in their WSDL file. Finally, since manual tags are assigned to those WSDL files, we use them to classify the candidate words coming from our WSDL files. Using this set of candidate words, computed features and assigned classes, we train a classifier.

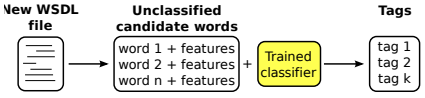


Figure 2: The tag extraction phase

Figure 2 describes the *tag extraction phase*. First, like in the *training phase*, a list of candidate words is extracted from an untagged WSDL file and the same features are computed. The only difference with the training phase is that we do not know in advance which of those candidates are true tags. Therefore we use the previously trained classifier to automatically perform this classification. Finally the tags extracted from the WSDL file are the words that have been classified in the *tag* class.

We have used a corpus of 146 user-tagged web services extracted from Seekda for experiment. The obtained results demonstrated the efficiency of the described approach.

The extracted keywords are used to classify web services, using Formal Concept Analysis, as explained in the next section.

3 FCA-Based Web Service Classification

We propose an approach based on Formal Concept Analysis (FCA) [1], [3] for classifying web services into a lattice structure, which facilitates their browsing and selection [6]. A service lattice reveals the relations between the services, easing the discovery of a needed service as well as the identification of its possible alternatives. This helps in the construction of service compositions and ensuring a continuous functionality by supporting them with backup services. In figure 3, we can see a general overview of the main steps of our approach, in which, two service lattices are built:

- a lattice for indexing the services by keywords extracted from their WSDLs, using the automatic tag extraction approach, described in Section 2,
- a lattice for similar service classification, which is build from a many-valued context that contains the similarity values that are calculated between operations offered by each pair of services.

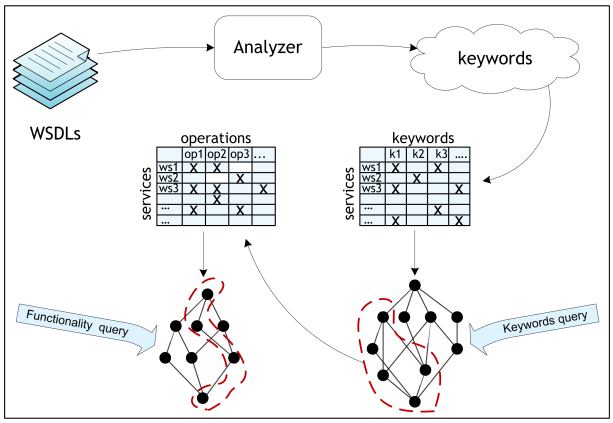


Figure 3: The main steps of our approach.

We explain our approach along with a case study described in the next section.

3.1 Case study

Let us consider the following travel scenario: a traveller needs to reserve a plane ticket to a desired city. Supposing that this traveller lives in a small city that has no airport, then, he should also travel to the city where the airport is located, in order to take the plane he reserved. Thus, he must also reserve a train ticket, from his home city to the airport city, taking into consideration the flight exact time with the time needed to travel between the two cities. This scenario can be achieved by a travel composite service (TCS), in which 3 functionalities must be satisfied:

- reserve a flight from home city, or airport city, towards a desired city,

- if a train reservation is needed (in case that the home city is not the airport city), then calculate the needed time (duration) to travel between the two cities by train,
- reserve a train ticket, regarding the exact flight time and the calculated duration.

The TCS can be realized by discovering services offering the described functionalities and composing them. It may look like the composition in Figure 4. In this orchestration, if the service $TrainWS$ crashes, for example, an equivalent service offering at least the two used operations $calcDuration()$ and $resTrain()$ must be searched and discovered, in order to recover the missing functionality, and ensure the continuity of the composition.

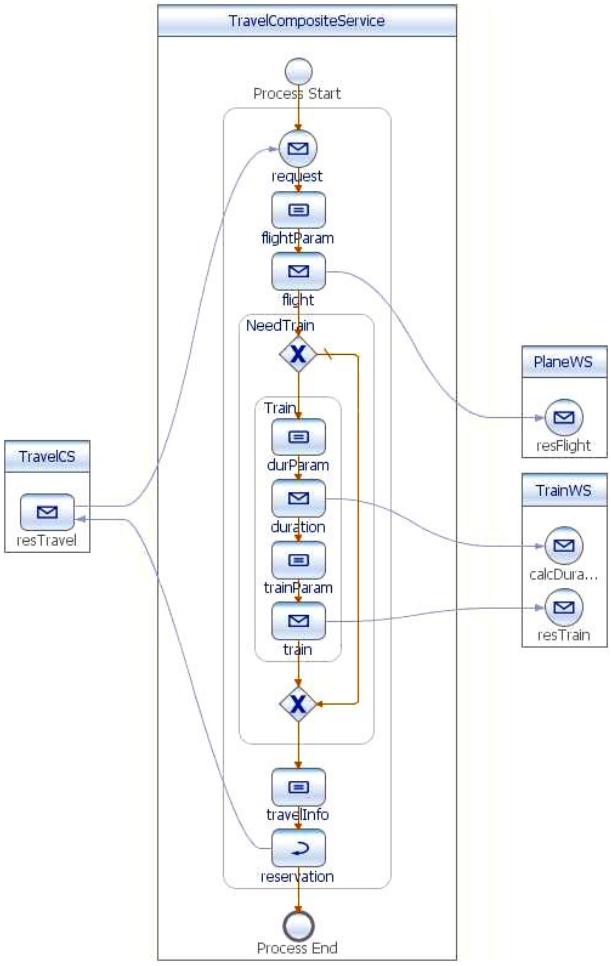


Figure 4: The travel composite service (TCS).

In the following we explain the construction of the keywords lattice, and the similarity lattice. We conclude by showing how to construct this composite travel service, and support it with backup services.

3.2 The keywords lattice

The extracted keywords in section 2 are used to generate the first lattice, which represents an indexation of services by keywords. Supposing a set of random web services for travel facilities, the corresponding lattice may resemble the lattice in Figure 5.

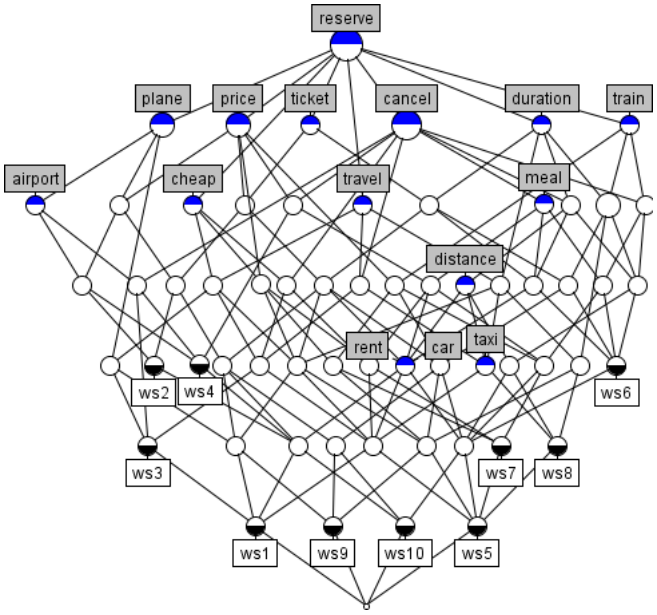


Figure 5: The concept lattice for a set of services and their keywords.

From this lattice, we can retrieve services in a certain domain by enquiring the lattice using keywords. Such a query returns a sub-lattice of services sharing the specified set of keywords, which are highly probable to provide similar functionalities. We perform a query on a lattice by expressing the set of keywords as a new line, added to the context $service \times keyword$. This results in a new concept in the lattice, labelled $query$. The services that answer the specified query are represented by the sub-concepts of the $query$ concept. In our scenario, we perform two keyword queries: $query1 = \{reserve, plane\}$ and $query2 = \{reserve, train, duration\}$. This results in the lattice shown in Figure 6, in which, we extract the services answering the two queries and they are:

- for $query1$: $ws_1, ws_2, ws_3, ws_4, ws_9$ and ws_{10},
- for $query2$: $ws_5, ws_6, ws_8, ws_9, ws_{10}$.

We further analyse the two sets of services corresponding to the two queries. We apply a similarity measure on the operations according to each set and we identify the groups of similar operations. For example, an operation labelled $reserveF$ in the lattice, represents a group of similar operations for performing a flight reservation. This way, we can build a new context of $service \times operation$ for each set of services.

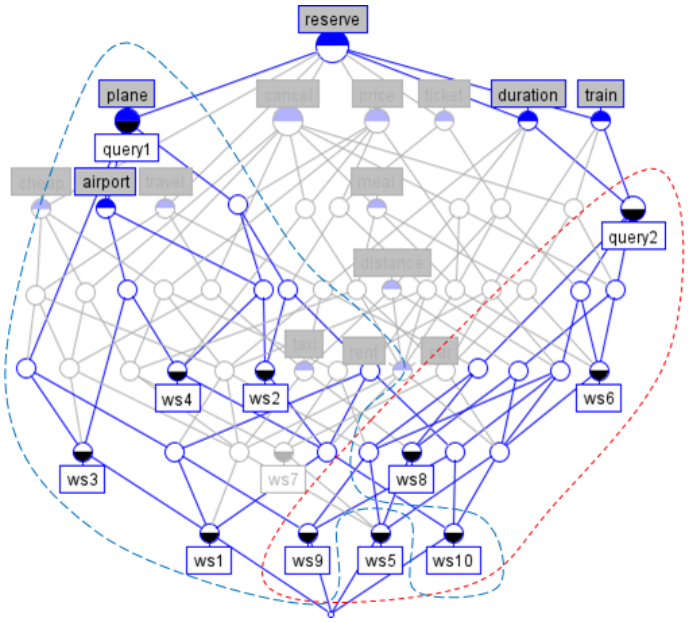


Figure 6: Queries as concepts in the *service* × *keyword* lattice.

We explain next how to build a many-valued context of similarity values, which are calculated between pairs of operations provided by different services.

3.3 The similarity lattice

For clarity sake, we illustrate our approach using an imaginary set of Web services for performing calculations. Each service from this set is parsed by a WSDL parser to extract its signatures. The set of services with their signatures are given unique identifiers, as listed in Table 1.

Table 1: A set of calculation services with their operations.

Services	**Id**	**Operations**	**Id**
Calculator1	ws_1	add(a,b)	op_{11}
		sub(a,b)	op_{12}
Calculator2	ws_2	add(a,b,c)	op_{21}
Calculator3	ws_3	add(a,b,c,d)	op_{31}
		sub(a,b,c)	op_{32}
		mult(a,b)	op_{33}
		add(a,b,c)	op_{34}

Next, a similarity measure must be chosen, and the operations signatures extracted from the WSDL files will be used by this similarity measure, according to its input format. Several similarity measures for web services exist in the literature. They evaluate the similarity according to the syntactic and semantic levels.The similarity is assessed in the form of values in the range [0,1]. If two

operations are sufficiently similar, the similarity value will approach 1, or else it will approach 0. The similarity measure is applied on pairs of operations provided by different services. We do not consider the similarity between operations provided by the same service (it is equal to 0), because when a service becomes dysfunctional, all of its operations become dysfunctional too.

A similarity measure (Sim) can be defined as follows:

$$Sim : \mathbb{O} \times \mathbb{O} \rightarrow [0, 1]$$
$$\forall\, op_{ij} \in \mathbb{O} \Longrightarrow Sim(op_{ij}, op_{ij}) = 1$$
(an operation with itself)
$$\forall\, op_{ij}, op_{ik} \in \mathbb{O} \Longrightarrow Sim(op_{ij}, op_{ik}) = 0$$
(operations in the same service)
$$\forall\, op_{ij}, op_{nm} \in \mathbb{O} \Longrightarrow Sim(op_{ij}, op_{nm}) \in [0, 1]$$
(operations in different services)

The calculated similarity values can be presented by a symmetric square matrix that we will call $SimMat$, as shown in Table 2. This matrix is of size $n = |\mathbb{O}|$, and its diagonal elements are all equal to 1, since we consider that the similarity of an operation with itself is equal to 1 as declared above.

Table 2: The similarity matrix ($SimMat$).

	op_{11}	op_{12}	op_{21}	op_{31}	op_{32}	op_{33}	op_{34}
op_{11}	1	0	0.75	0.5	0	0	1
op_{12}	0	1	0	0	0.75	0	0
op_{21}	0.75	0	1	0.75	0	0	0.75
op_{31}	0.5	0	0.75	1	0	0	0
op_{32}	0	0.75	0	0	1	0	0
op_{33}	0	0	0	0	0	1	0
op_{34}	1	0	0.75	0	0	0	1

From the similarity matrix $SimMat$, we can extract several binary contexts, by specifying threshold values $\theta \in]0, 1]$. Thus, the values of $SimMat$ that are greater or equal to the chosen threshold θ are scaled to 1, while other values are scaled to 0. The binary context that corresponds to $\theta = 0.75$ is shown in Table 3, we call it $SimCxt$.

Table 3: The binary context ($SimCxt$) for $\theta = 0.75$.

	op_{11}	op_{12}	op_{21}	op_{31}	op_{32}	op_{33}	op_{34}
op_{11}	x		x				x
op_{12}		x			x		
op_{21}	x		x	x			x
op_{31}			x	x			
op_{32}		x			x		
op_{33}						x	
op_{34}	x		x				x

The $SimCxt$ context is a triple $(\mathbb{O}, \mathbb{O}, RSim_\theta)$, where $RSim_\theta$ is a binary relation indicating whether an operation is similar to another operation or not.

$$(op_{ij}, op_{nm}) \in RSim_\theta \Longleftrightarrow Sim(op_{ij}, op_{nm}) \geq \theta$$

We use the SimCxt context to generate a lattice of operations, $\underline{\mathfrak{B}}(\mathbb{O}, \mathbb{O}, RSim_\theta)$. This lattice helps in discovering groups of similar operations, which are used later on to construct the services lattice.

In the resulting operation lattice, groups of mutually similar operations can be identified by the concepts having equal extent and intent sets. We call such concepts as square concepts, because they form square gatherings on the binary context matrix. We define a group G_{op} of mutually similar operations Op_{Sim} as:

$$G_{op} = \{Op_{Sim} \mid (Op_{Sim}, Op_{Sim}) \in \mathfrak{B}(\mathbb{O}, \mathbb{O}, RSim_{\theta})\}$$

The notion of square concepts can be better recognized by performing a mutual column-line interchange in the $SimCxt$, the resulting interchanged context is shown in Table 4.

Table 4: The interchanged ($SimCxt$) context.

	op_{11}	op_{34}	op_{21}	op_{31}	op_{12}	op_{32}	op_{33}
op_{11}	x	x	x				
op_{34}	x	x	x				
op_{21}	x	x	x	x			
op_{31}			x	x			
op_{12}					x	x	
op_{32}					x	x	
op_{33}							x

From the lattice in Figure 7 as from the interchanged context in Table 4, we can identify the groups of similar operations, and they are the following:

- $\{op_{11}, op_{34}, op_{21}\}$
- $\{op_{21}, op_{31}\}$
- $\{op_{12}, op_{32}\}$
- $\{op_{33}\}$

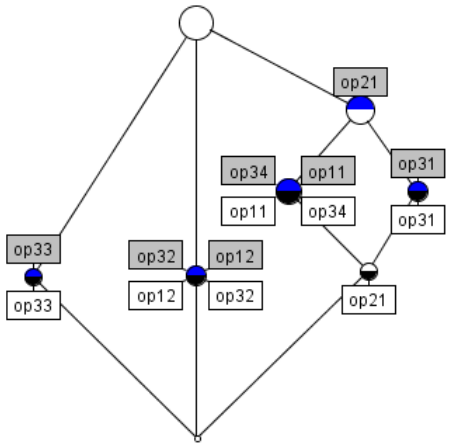


Figure 7: The generated lattice for ($SimCxt$) shown in Table 3.

The groups of similar operations, denoted as $\mathbb{G}$, are used to define the final binary context. This context is a triple $(\mathbb{W}, \mathbb{G}, R)$, in which the relation R indicates whether or not a service offers the functionality represented by the corresponding group of similar operations.

We will represent each group of operations by an identifier that corresponds to the indices of the operations. The new context is shown in Table 5: From the

Table 5: The final services × groups context.

	(11,34,21)	(21,31)	(12,32)	(33)
ws_1	x		x	
ws_2	x	x		
ws_3	x	x	x	x

final binary context, we can generate the corresponding service lattice, which is shown in Fig 8. From the final generated lattice, shown in Figure 8, we can

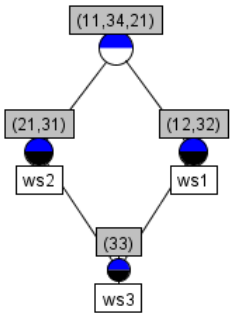


Figure 8: The final service lattice with possible backups.

notice the following:

- ws_1, ws_2, and ws_3 offer the functionality denoted by $(11, 34, 21)$, so they can replace each other for this specific functionality
- ws_3 can replace ws_1 and ws_2, and it offers an additional functionality (33)

We can also infer immediately which services offer a specific functionality (denoted by a specific label), by regarding the indices in the label.

We return to our scenario after having generated two similarity lattices, corresponding to $query1 = \{reserve, plane\}$ and $query2 = \{reserve, train, duration\}$.

The lattice that corresponds to $query1$ is shown in Figure 9 (top), and the one corresponding to $query2$ in Figure 9 (bottom).

Using these two lattices, the selection of services offering required operations is straightforward. In our scenario, we need three tasks: flight reservation, train reservation (if needed), and calculating the duration needed to travel by train to the aimed destination. By regarding the lattice in Figure 9 (top), we notice that all of the services offer an operation for plane reservation. In this case, a service selection might be done regarding the extra operations that the services provide, like for example the operation $rentCar$. When we select a certain service, we can immediately extract the set of backup services that are able to replace it if it fails. In the same way, we can select a service for train reservation with obtaining the duration information. Thus, the composition in Figure 4 can be easily achieved and supported with backups, as in Figure 9. Supposing that we selected the service ws_4 named $PlaneWS$, and used its operation $resFlight$ (which is grouped with other similar operations under the name $reserveF$ in the lattice). We can notice that for the operation $resFlight$, any other service

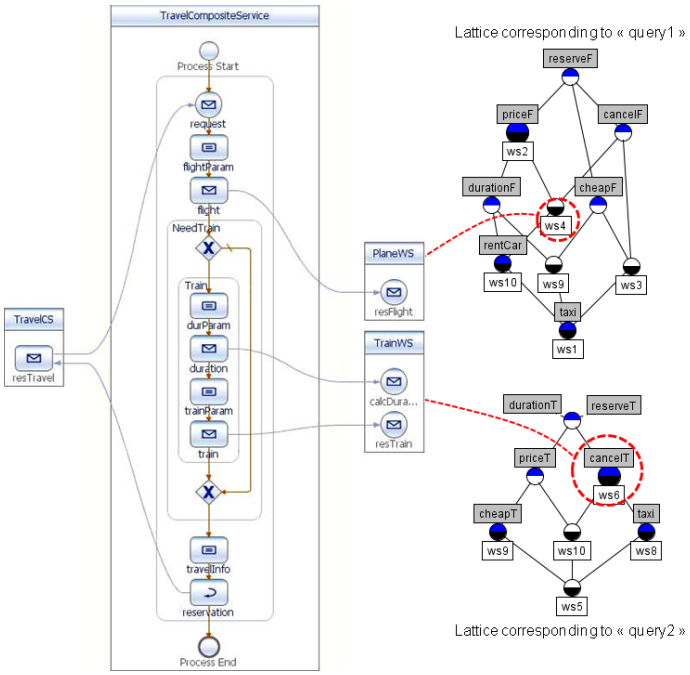


Figure 9: The TCS composition with its corresponding backups.

in the lattice can be a backup for ws_4. In case where all of the operations of ws_4 were used, we notice that only ws_{10} and ws_1 can be backups for ws_4. Similarly, we selected the service ws_6, named $TrainWS$, and used two of its provided operations: *calcDuration* and *resTrain*. We notice that ws_6 can be replaced by the services ws_{10}, ws_8 and ws_5. Thus, we have discovered immediate backups for the service $TrainWS$ as we did for the service $PlaneWS$. We can notice that the service ws_{10} exists in the two lattices, as a backup for both $TrainWS$ and $PlaneWS$. In this case, if both of these services crash, we can replace them both by a single service, which is ws_{10}. This service provides the same functionalities of the two services, with three extra operations that are $priceT$, $rentCar$ and $durationF$ as can be seen in the lattices in Figure 9.

Our approach has enabled us of an easy service discovery and selection, in order to build our aimed scenario. It has also facilitated the discovery of backup services to support service composition and ensure its continuous functionality.

In the following section, we will include QoS aspect to our web service classification, in order to select a service with the best available QoS values.

4 QoS-Based Web Service Ordering

Web services have many QoS aspects such as availability, response time, reliability, etc. Selecting a service from a set of services offering different values of QoS may not be a straightforward task.

Supposing a set of services {a,b,c,d,e,f,..} with different values of availability and performance time as in figure 10. In such case, we need to select the service that makes the best compromise between the aspects of QoS supposing than the value 1 is the best and 10 is the worst. In the literature, we can find a similar problem about finding skylines objects[3]. Using the skyline operator, an object is determined to be interesting if it is not dominated by any other object. In this view, we can observe that the interesting service are a, i, and k.

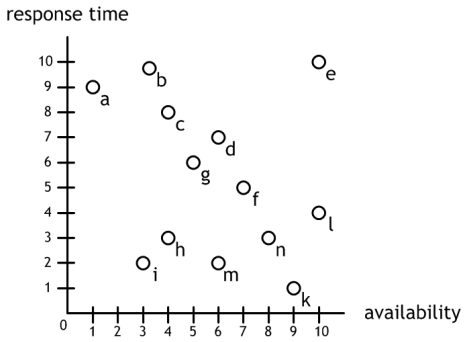


Figure 10: Services with availability and performance time values.

We applied the FCA classification on services with their QoS values, in order to have this kind of service ordering. We considered that a service that has a certain QoS value, has also all the values that are lower. In figure 11, we can see the binary context corresponding to services and QoS values shown in figure 10. The resulting lattice of the context in figure 11 is illustrated in the figure 12.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
	a=1	a=2	a=3	a=4	a=5	a=6	a=7	a=8	a=9	a=10	rt=1	rt=2	rt=3	rt=4	rt=5	rt=6	rt=7	rt=8	rt=9	rt=10
a	X	X	X	X	X	X	X	X	X	X									X	X
b			X	X	X	X	X	X	X	X										X
c				X	X	X	X	X	X	X								X	X	X
d						X	X	X	X	X							X	X	X	X
e										X										X
f							X	X	X	X					X	X	X	X	X	X
g					X	X	X	X	X	X						X	X	X	X	X
h				X	X	X	X	X	X	X			X	X	X	X	X	X	X	X
i			X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X
k									X	X	X	X	X	X	X	X	X	X	X	X
l										X				X	X	X	X	X	X	X
n								X	X	X			X	X	X	X	X	X	X	X
m						X	X	X	X	X		X	X	X	X	X	X	X	X	X

Figure 11: Binary context of services in figure 10 with their QoS values.

In this lattice, we can notice that the skylines services appear at the bottom. They are also the services a, i, and k as given by the skyline operator.

[3]S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In Proceedings of the 17th International Conference on Data Engineering (ICDE), pages 421-431, 2001.

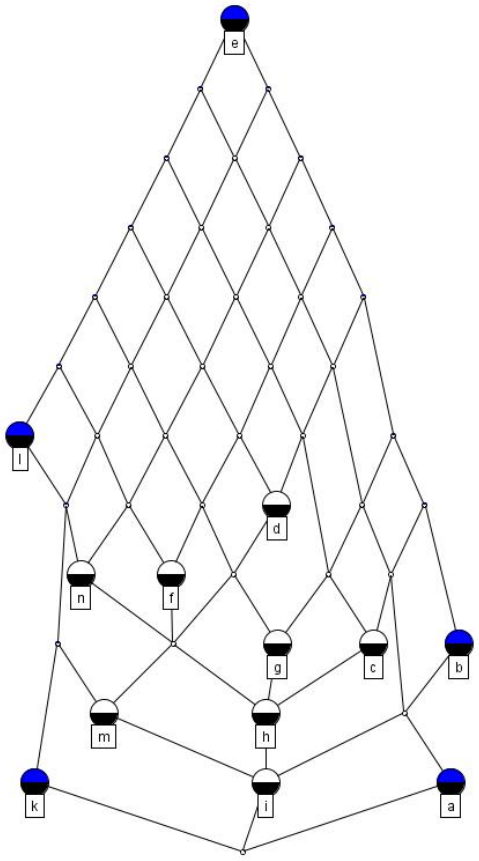


Figure 12: The service lattice for the context in figure 11.

We can also observe that to choose a service that makes the optimal compromise between the QoS values (availability and response time), we have to select a service that dominates a greater number of services. For example, the service i would be the best choice because it dominates more services than any other service in the lattice. On the other hand, the service e, which is located at the top of the lattice can be considered as the worst choice because it does not dominate any service. We can check the correction of these conclusions by returning to the actual values of each service. Service i has the values (a=3, rt=2), which is probably the best compromise among all the other values, while service e has the values (a=10, rt=10), which is definitely the worst compromise ever.

The generated lattice has so many empty concepts which can be eliminated using the Galois Sub-Hierarchy (GSH) algorithms[4].

[4]Arevalo G., Berry A., Huchard M., Perrot G., Sigayret A.: Performances of Galois Sub-hierarchy-building algorithms. ICFCA 2007, LNCS/LNAI 4390, pp166-180, Springer Verlag.

5 Web Service Composition

We propose an approach based on a variant of FCA called Relational Concept Analysis (RCA) [2] for the identification of composable services. It allows the classification of web services according to their QoS and composability levels and proposes a query mechanism that allows users to specify their required QoS and composability levels. In the case of multiple choices that satisfy users' requirements, we enhance our approach with a mechanism based on vectors for identifying the optimized choice.

5.1 RCA-Based Web Service Classification

We explain our approach along with an abstract composition of three sequential tasks, as in figure 13. We try to identify the concrete services offering the needed functionalities, in order to instantiate the defined orchestration. Our objective is to select composable services with the highest affordable QoS values, and the least required adaptations.



Figure 13: An abstract composition of three sequential tasks.

We have 3 sets of services: S_{1i}, S_{2j} and S_{3k} corresponding to the three tasks. For each service of the three sets, we assume certain values of QoS (for availability (A) and response time (RT)). We categorize the QoS values into 5 levels: very low, low, medium, high and very high, for each of the A and RT values. A service that offers a specific QoS level, does also offer QoS levels that are lower. For each pair of consecutive services (according to our defined orchestration), we assume three composability values: exact, partial and disjoint. We base our classification on Formal Concept Analysis (FCA), which is a data analysis method that aims at extracting concepts from entities described by attributes. The description of entities is encoded in a formal context. For example, in Fig. 14, the third table S_{3k} is a formal context where entities are services for the Task 3, and are described by the values of availability and response time (attributes). Concepts are maximal sets of entities (extent) sharing a maximal set of attributes (intent), and they are organized in a classification provided with a lattice structure. The lattice associated with the context S_{3k} is shown in Fig. 18. Each concept in this lattice inherits the attributes of its ascendants (super-concepts), and has lesser attributes than its descendants (sub-concepts). Reversely, each concept inherits the services of its descendants (sub-concepts). For example, the concept c_0 represents the services s_{31} and s_{33}, which covers the QoS values of the services in the upper concepts, and offers better ones (like: low A and low RT). The concept c_5 indicates that s_{33} has very low RT and high A. The concept c_5 is a sub-concept of the concept c_0, which shows that s_{33} has better QoS values than s_{31}.

Since we also want to capture the composition relations, existing between pairs of services in an orchestration, we use a variant of FCA called Relational Concept Analysis (RCA), which includes relations in the attributes used for the concept mining. RCA data are given in the form of a Relational Context Family

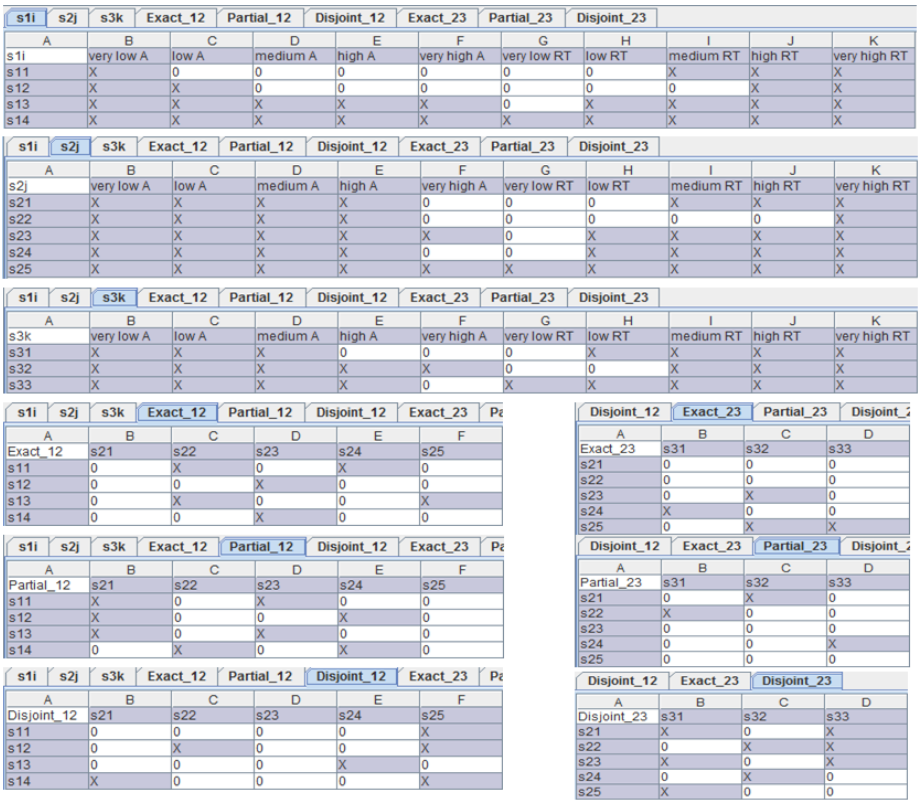


Figure 14: The relational contexts family (RCF) corresponding to the three sets of services.

(RCF). An RCF is a set of binary relationships (tables): some (non-relational) tables describe entities by attributes, while some (relational) tables describe relations between entities. The non-relational part of the RCF contains one table for each task: entities are services that can make the task, and attributes are QoS values. Thus, our RCF is composed of three non-relational contexts: $S_{1i} * QoS$, $S_{2j} * QoS$ and $S_{3k} * QoS$, and two sets of three relational contexts: composition of $S_{1i} * S_{2j}$ and composition of $S_{2j} * S_{3k}$, according to the three considered levels. As we can see in Fig. 14, the three relational contexts for composition of $S_{xi} * S_{yj}$ are called $Exact_{xy}$, $Partial_{xy}$ and $Disjoint_{xy}$. We run the RCA algorithm on the RCF, illustrated in Fig. 14. The RCA takes as input all the tables and iterates on two steps: (1) building a concept lattice on each non-relational context of services concatenated with the corresponding relational contexts that express composition, that is on $S_{1i} * QoS$ concatenated with $Exact_{12}$, $Partial_{12}$ and $Disjoint_{12}$, on $S_{2j} * QoS$ concatenated with $Exact_{23}$, $Partial_{23}$ and $Disjoint_{23}$ and on $S_{3k} * QoS$; (2) transform the six relational contexts to integrate the concepts found at that current step (to use this knowledge in the next iteration). During this transformation, we use the existential scaling operator which is one of the scaling operators provided by RCA. Services that form the columns of a composition context are replaced by concepts that group services. If (S_{xy}, S_{zt}) initially holds in a composition context (with composition level cl) and S_{zt} is in

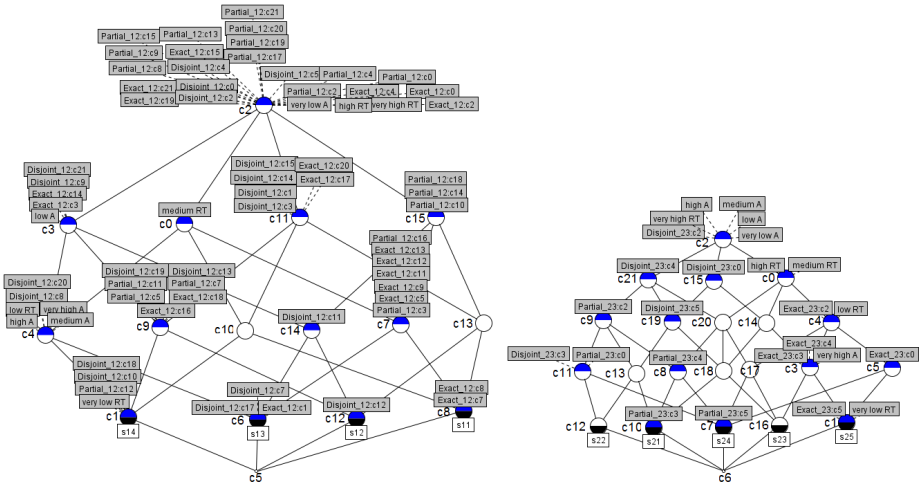
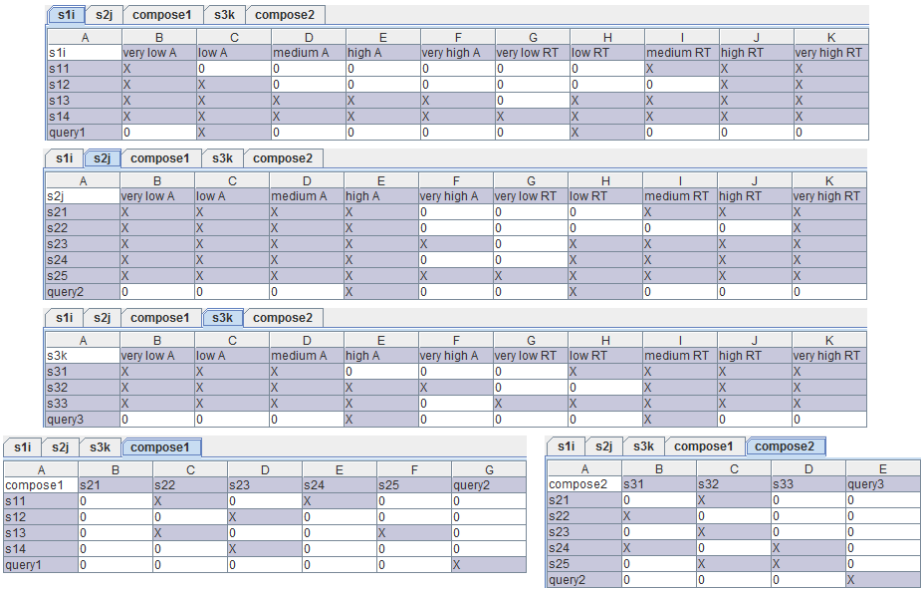


Figure 15: The services lattices for Task 1 and Task 2.

the extent of a concept C at the current step, thus in the current version of the composition context we add (S_{xy}, C). This can be interpreted as S_{xy} can be composed with at least one service from the set of services grouped in C with the composition level cl. For example, since we have $(s_{13}, s_{25}) \in Exact_{12}$ and since s_{25} is in the extent of Concept c_1 (in lattice S_{2j}), the scaled $Exact_{12}$ table will contain (s_{13}, c_1). By regarding the lattices in Fig. 15, we can find for example, that the service s_{13} (in the leftmost lattice) has in its intent $Exact_{12} : c1$. This means that it can be composed with services contained in the concept $c1$ (in the middle lattice). Thus, s_{13} can be composed at least with s_{25}. At a next step this will allow to group a set of services (implementing a first task) because they can be composed with services of another group of services (implementing another task). The process stops when no new concept emerge during the FCA analysis. At the end of the process, a concept lattice is generated for each set of services. In each lattice, services are classified into concepts, showing their QoS levels as well as their composability levels with other services, according to the orchestration. Concept c_7 in the S_{1i} lattice groups services s_{13} and s_{11}, and indicates that they can be partially composed with services of concept c_3 of the S_{2j} lattice, which groups s_{23} and s_{25}.

Navigation by Query Integration. We define a query mechanism that enables us to navigate into a lattice, in order to choose a combination of services that satisfies the aimed orchestration, with our aimed levels of QoS and composability. A query determines the required QoS for each requested task, as well as, the needed composability level (required adaptations) between the services. We define a QoS query as a new line (a new entity), integrated into the corresponding service*QoS binary context, specifying the least expected QoS level. In our example, we perform three QoS queries: $query_1$, $query_2$ and $query_3$ on $S_{1i} * QoS$, $S_{2j} * QoS$ and $S_{3k} * QoS$ respectively. They are as follows:

- $query_1$ looks for a service from S_{1i}, with a low A and a low RT;
- $query_2$ tries to retrieve a service from S_{2j}, with a high A and a low RT;



s1i | s2j | compose1 | s3k | compose2

A	B	C	D	E	F	G	H	I	J	K
s1i	very low A	low A	medium A	high A	very high A	very low RT	low RT	medium RT	high RT	very high RT
s11	X	0	0	0	0	0	0	X	X	X
s12	X	X	0	0	0	0	0	0	X	X
s13	X	X	X	X	X	0	X	X	X	X
s14	X	X	X	X	X	X	X	X	X	X
query1	0	X	0	0	0	0	X	0	0	0

s1i | s2j | compose1 | s3k | compose2

A	B	C	D	E	F	G	H	I	J	K
s2j	very low A	low A	medium A	high A	very high A	very low RT	low RT	medium RT	high RT	very high RT
s21	X	X	X	X	0	0	0	X	X	X
s22	X	X	X	X	0	0	0	0	0	X
s23	X	X	X	X	X	0	X	X	X	X
s24	X	X	X	X	0	0	X	X	X	X
s25	X	X	X	X	X	X	X	X	X	X
query2	0	0	0	X	0	0	X	0	0	0

s1i | s2j | compose1 | s3k | compose2

A	B	C	D	E	F	G	H	I	J	K
s3k	very low A	low A	medium A	high A	very high A	very low RT	low RT	medium RT	high RT	very high RT
s31	X	X	X	0	0	0	X	X	X	X
s32	X	X	X	X	X	0	0	X	X	X
s33	X	X	X	X	0	X	X	X	X	X
query3	0	0	0	X	0	0	0	X	0	0

s1i | s2j | s3k | compose1

A	B	C	D	E	F	G
compose1	s21	s22	s23	s24	s25	query2
s11	0	X	0	X	0	0
s12	0	0	X	0	0	0
s13	0	X	0	0	X	0
s14	0	0	X	0	0	0
query1	0	0	0	0	0	X

s1i | s2j | s3k | compose1 | compose2

A	B	C	D	E
compose2	s31	s32	s33	query3
s21	0	X	0	0
s22	X	0	0	0
s23	0	X	0	0
s24	X	0	X	0
s25	0	X	X	0
query2	0	0	0	X

Figure 16: The RCF after integrating the QoS queries and specifying the composability level.

- $query_3$ demands a service from S_{3k}, with a high A and a medium RT.

In order to select services that satisfy a specific composability level, we merge the relational contexts that express levels equal or greater than the requested level. In other words, if a *partial* composability level can be accepted, this means that if we find an *exact* level, it is even better. In this example, we demand an "exact" composability level between S_{1i} and S_{2j}, and a "partial" level between S_{2j} and S_{3k}. This gives us a variation of our RCF that includes queries, where the relational context $compose1$ is the context $exact12$ and $compose2$ is the fusion of $exact23$ and $partial23$, as shown in Fig. 16.

This latter RCF generates three lattices, from which we can extract the services combinations that meet our requested QoS and composability. These lattices are illustrated in Fig. 17, from which we can identify the services that satisfy the QoS queries, which appear in the sub-concepts of the concept where a $query_n$ appears, and are as follows:

- services s_{13} and s_{14} for $query_1$;
- services s_{23}, s_{24} and s_{25} for $query_2$;
- services s_{32} and s_{33} for $query_3$.

5.2 Web Service Triangles Approach

We can have several service combinations that satisfy our desired orchestration. If we are searching for an optimized selection, this means that we have to find the set of services that meet the optimal compromise of QoS levels and composability. Thus, services that can be coupled according to our acceptable levels of

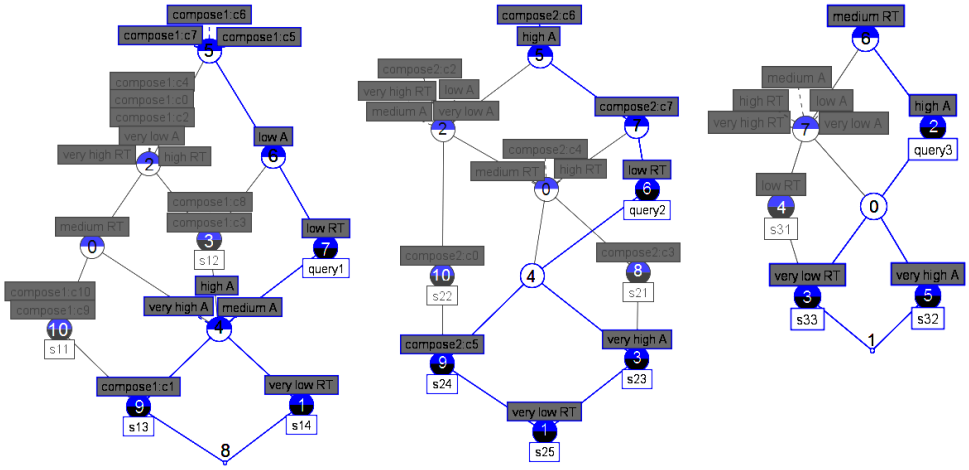


Figure 17: The service lattices with the integrated queries.

required adaptations and QoS. In order to meet this issue, we propose to represent the services by vectors. A vector per set of services, on which, services are ordered according to their QoS. We also define a vector for composability levels. Each composition can be regarded as a triangle, having a head corresponding to the composability level, and the two other heads corresponding to the pair of services to be composed, as shown in Fig. 19.

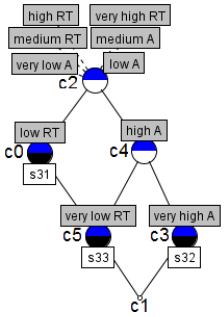


Figure 18: The lattice associated with the context S_{3k} of Task 3.

The optimized choice would be the triangle that has the minimal area, in case of a two services composition. Otherwise, it will be the minimal sum of the services triangles according to an orchestration. Thus in our example, by regarding the triangles in Fig. 19, we notice that the triangle (E, s_{25}, s_{33}) represents an optimized composition between S_{2j} and S_{3k} (corresponding to Task2 and Task3, respectively). Accordingly, if we consider the triangle (E, s_{13}, s_{25}) that shares an edge with the previous triangle, it represents a composition between S_{1i} and S_{2j} (corresponding to Task1 and Task2, respectively). These two triangles together may be an optimized combination for the required orchestration, after comparing them with all the existing triangles[5].

[5]We haven't shown all the triangles in Fig. 19, for the sake of simplicity.

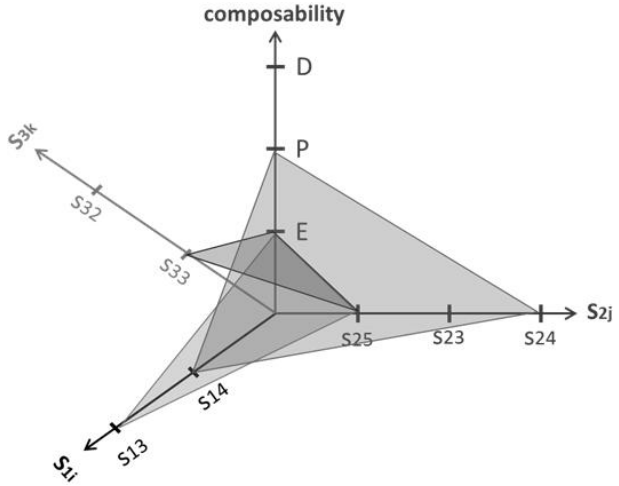


Figure 19: Optimizing the service selection.

6 Perspectives

We have many ideas to improve and complete our works. In the following, we present some of them.

6.1 Enhancing Coma++ with WordNet

Coma++ [6] gives us a primary matching of WSDL which can be improved using WordNet [7] together with the service functionality domain, in order to match the fields semantically. After doing so, we can annotate the fields with a WordNet concept to have a semantic service.

By matching the fields using Coma++ with the semantic enrichment, we can find the similar fields and thus, the similar functionalities (operations). This will also help us in the identification of the adaptations, i.e. the extra fields (inputs) in an operation that has a similar functionality of another one that doesn't contain these fields.

How can we judge if two operations have a similar functionality or not?

The two services must have the same domain and the two operations must have similar names and common fields (inputs and outputs).

Steps:

1. Extract the domain of functionality for each WSDL interface;
2. Using the Coma++ API to automate the matching process between all the pairs of a WSDL interfaces collection having the same domain;
3. Two operations have a similar functionality, if they have similar names, a similar input, and a similar output;
4. For each domain, there must exist a domain ontology, which can be used to annotate the corresponding services;
5. According to the similarity of inputs-inputs and for inputs-outputs, we can continue our classification based on FCA for finding Web services' substitutes, and RCA for finding composable Web services.

6.2 WordNet for a WSDL's domain of functionality

We want to divide the WSDL interfaces into their functionality domains. This categorization can be done with the help of WordNet. In figure 20, we can see the steps that are needed to annotate the WSDL file with the WordNet ontology. Every WSDL will be parsed by a TreeTagger, which is a tool for annotating text with part-of-speech and lemma information. Terms of this annotated text will be mapped onto the WordNet ontology. This gives us a set of WordNet concepts for each WSDL file. In order to find out the domain, we search for the branch in WordNet that contains all the concepts for a certain WSDL file.

Similarity may be calculated afterwards between the set of WordNet concepts retrieved for each WSDL file, in order to find similar operations.

[6] http://dbs.uni-leipzig.de/Research/coma.html
[7] http://wordnet.princeton.edu/

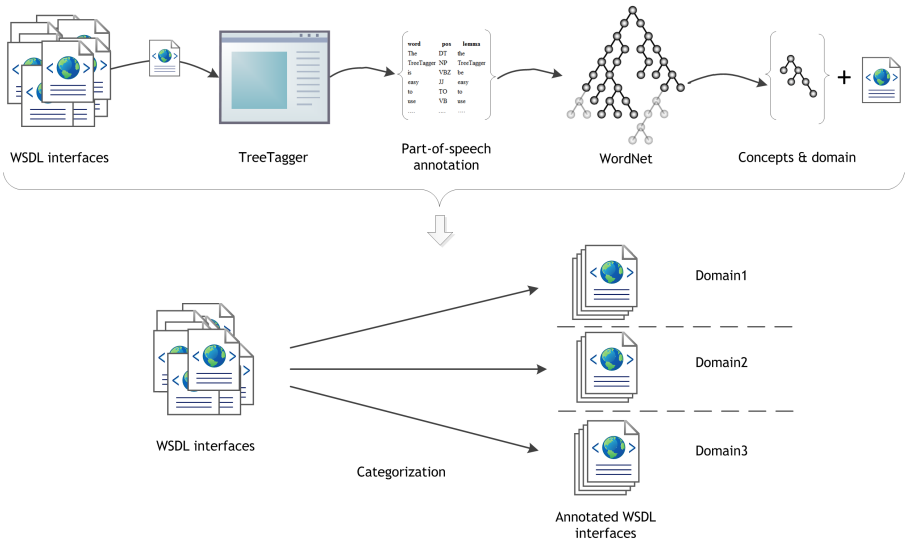


Figure 20: WSDL categorization into functionality domains using WordNet.

6.3 Modelling abstract business processes

An abstract business process [8] is a partially specified process that is not intended to be executed. It may hide some of the required concrete operational details, and may have a descriptive role of the observable process behavior and template.

- A user defines his orchestrations as abstract business processes;
- for each orchestration he defines a list of user-input-variables;
- for each task inside the orchestration, a domain is specified, in order to identify the set of services offering this needed task;
- the composability level can be determined regarding the user indicated output-input that must be coupled together.

References

[1] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., Heidelberg, 1999.

[2] M. Huchard, M.R. Hacene, C. Roume, and P. Valtchev. Relational concept discovery in structured datasets. *Annals of Mathematics and Artificial Intelligence*, 49(1-4):39–76, 2007.

[3] R. Wille. Restructuring lattice theory: an approach based on hierarchies of concepts. *Ordered Sets*, 83:445–470, September 1982.

[8] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf

[4] Z. Azmeh, M. Huchard, C. Tibermacine, C. Urtado, and S. Vauttier. WS-PAB: A tool for automatic classification & selection of web services using formal concept analysis. In *Proc. of (ECOWS 2008)*, pages 31–40, Dublin, Ireland. IEEE Computer Society.

[5] J.R. Falleri, Z. Azmeh, M. Huchard, and C. Tibermacine. Automatic Tag Identification in Web Service Descriptions. In *Proc. of (WEBIST 2010)*, pages 40–47, Valencia, Spain.

[6] Z. Azmeh, M. Huchard, C. Tibermacine, C. Urtado, and S. Vauttier. Using Concept Lattices to Support Web Service Compositions with Backup Services. In *Proc. of (ICIW 2010)*, pages 363–368, Barcelona, Spain. IEEE Computer Society.