2012 IEEE 19th International Conference on Web Services

From Abstract to Executable BPEL Processes with Continuity Support

Zeina Azmeh, Marianne Huchard
LIRMM, CNRS & UM2
34095, Montpellier cedex 5, France
{firstname.lastname}@lirmm.fr

Fady Hamoui
LIG & UPMF
38041, Grenoble cedex 9, France
fady.hamoui@imag.fr

Naouel Moha
UQAM - CP 8888,
Montréal, H3C 3P8, Canada
moha.naouel@uqam.ca

***Abstract*—The real value of Web services under the SOA paradigm lies in their ability to be assembled to obtain a new functionality. Assembling Web services can be achieved through a standard called BPEL, which creates executable processes by orchestrating Web service invocations. The problem with BPEL is the inability to separate the process description from its realization. In other words, it requires a prior retrieval of concrete Web services, which can be very challenging regarding the issues surrounding service discovery and selection.**

In this paper, we propose to separate a BPEL process description from its realization. We extend the notion of abstract BPEL processes, in order to enable developers to describe their desired orchestrations abstractly without identifying concrete services, according to three levels: the needed functionality, the expected QoS levels, and the composition flow. Then, the abstract BPEL process is realized by a selection framework that automatically discovers, classifies, and selects suitable services to render the process executable. Backup services are also discovered to assure the continuity of the realized process.

***Keywords*-SOA; Web service; abstract BPEL; QoS; Relational Concept Analysis (RCA).**

I. Introduction

Web services represent an important realization of Service-Oriented Architectures (SOA) [1]. Their real value lies in their ability to interoperate and be assembled together to obtain new composite services. Assembling Web services together can be achieved by orchestrating their invocations through a standard language, called BPEL [2]. BPEL is an XML-based language that defines a new Web service by orchestrating a set of existing services according to a desired functionality. The resulting service is called a business process, the involved services are called partners, and the message exchange is referred to as an activity.
Constructing an executable BPEL process consists of:

- first, determining the needed functionality that cannot be satisfied by a single Web service and dividing this functionality into smaller pieces;
- then, discovering a set of services for each needed piece of functionality;
- finally, selecting the needed services and orchestrating them according to some flow logic to achieve the process overall functionality.

The previous steps impose two principal problems: The first problem is the inability to separate the process description from its realization. Consequently, it requires a prior retrieval of concrete Web services, which can be very challenging. Current discovery mechanisms –embodied mainly in Web service search engines like Seekda [3] and Service-Finder [4]– limit the developer's expression capability to only keywords. They may return a large list of services that may not be all related to the used keywords, and thus might not match the searched functionality. We notice that depending only on the syntactic information inside the WSDL description may not be sufficient for guiding a developer to select a suitable service. The textual description, inside a WSDL interface, may not be enough to index a service sufficiently. Thus, several irrelevant services may be retrieved. This requires the developer to filter the services by hand to check their functionality. Moreover, an important factor for service selection is to take into consideration its quality of service (QoS) values [5]. The QoS plays a crucial role in determining a process quality.
According to the literature, the selection of a pertinent Web service can sometimes depend on a shared knowledge between the provider and the consumer (an ontology). This kind of solution may solve the problem of selection, but under the condition of having a unique ontology. If several ontologies were used, ontology mapping must be carried out, which is yet another challenge.
The second problem occurs after realizing the process. If one of the involved services disappears, an equivalent service must be identified to replace the missing piece of functionality and to maintain the process continuity. This may lead to the repetition of all the steps of constructing an executable BPEL process, which we described earlier.

In this paper, we present an extension of a previous work [6]. We propose a framework for achieving a separation of concerns: a process description from its realization.
Process Description: enables a developer to describe his desired business process abstractly, without identifying the needed concrete services in advance.
Process Realization: is an extension and improvement of what we presented in [6]. It enables the automatic realization of an executable process that satisfies the developer's provided process description. It works according to three main steps:

Discovery: including analyzing the process description, searching, retrieving and filtering Web services, in addition to parsing their WSDL descriptions.

Classification: according to functionality, composability, and QoS. In order to have a better view of the services, and so to make a better selection, especially when having comparable services.

Selection: of suitable services to realize the desired process. In addition to selecting backup services to support the continuity of a process, by recovering the missing piece of functionality of a broken service.

The paper is organized as follows: in the next section, we describe our proposed solution. In Section III, we present our conducted experiments to verify the validity of our solution. In Section IV, we list the related work and compare between them. Finally, in Section V, we conclude the paper with a summary of our contributions and perspectives.

II. Approach

Our framework works according to two stages: process description and process realization. It transforms automatically an abstract BPEL process into executable, by identifying suitable Web services. It works also on supporting the realized process with backup services, to ensure its continuity, in case of service failures.

During the process description stage, a developer can specify his process abstractly, without concrete services. It can be described using an abstract specification of the needed services. This specification indicates their functionality and their expected QoS. Then, the composition flow between these specified services is expressed inside the process description.

During the process realization stage, the specification of the needed services is analyzed. Then, concrete services are discovered and classified, in order to select the suitable ones. Finally, an executable BPEL process is generated, by filling the gaps inside the process description.

In the following, we describe in details each of the process description and realisation.

A. Process Description

A developer works on providing an abstract description of his desired BPEL process. We extend the notion of an abstract BPEL that is proposed by the BPEL standard. We define it as a process that is built on an abstract specification of the needed services. While, in BPEL standard, an abstract process specifies the external message exchange between parties only. It does not contain the internal details of the process flow, and it is not executable [2].
Thus, a developer starts by dividing the global needed functionality into smaller pieces. Each small piece of functionality requires a Web service to satisfy it. The set of required Web services must be specified by the developer inside an abstract WSDL interface (AWSDL). He uses this abstract interface afterwards to describe an abstract BPEL process (ABPEL).

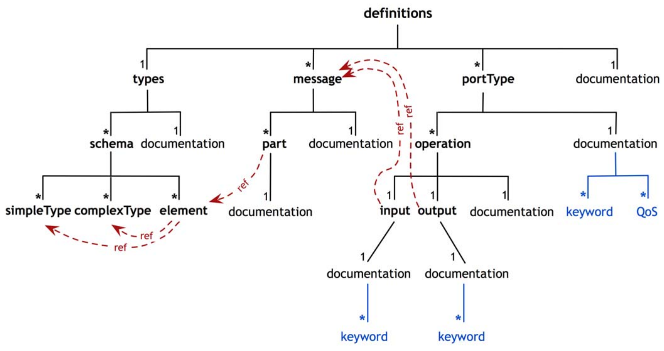


Figure 1. Abstract WSDL structure.

1) Abstract WSDL Interface (AWSDL): An AWSDL interface represents an utilization of the WSDL standard to specify one or more needed services abstractly. It may specify several abstract services ($PortTypes$) by their functional properties as well as their QoS properties, without specifying concrete parts (bindings and endpoints). We exploit the documentation tags, which can be defined for each element inside a WSDL interface, in order to describe these functional and QoS properties. AWSDL interface structure is illustrated in Figure 1.

Functional properties for each service are characterized by the operation it offers. We consider that a needed operation can be characterized by its input/output parameters and their types, but not by the operation name. We argue that an operation name can be as much a good indicator for the offered functionality as it can be bad. For example, if someone is searching for an operation that returns the capital city for a given country, an operation named $getCapital$ or $getCapitalForCountry$ would be exactly the required one. On the other hand, an operation named $getCountryInformation$ would be misleading. It may even get discarded if we only consider its name, although it returns the capital city among its outputs. Furthermore, we give the developer the option to say whether the parameters names are strict or flexible. If they are strict, this means that we should only search for an exact match of the provided parameters names. In the case of flexible names, we can search for synonyms or names that semantically match the requested ones (using WordNet for example). Hence, in case of a strict matching, a developer is allowed to provide several possible alternative names for each parameter name. For example, if the requested parameter was $city$ with strict matching, a developer may also provide words like $town$ and/or $metropolis$. The same thing is done for the parameter type. For example, if a developer is searching of a date parameter, he can specify its type to be $Date$ and/or $String$.

Expected QoS properties may also be expressed for each needed service inside an AWSDL interface, inside its documentation tag. We propose to represent the space of actual QoS numerical values by levels, ranging from $VeryBad$

QoS value to $VeryGood$ QoS value. These levels are calculated using a statistical technique called $BoxPlot$[1] [7]. The benefits of using such levels are two: first, it avoids developers from understanding the real meaning of a QoS value. For example: the performance of a service is better when it is lower, while the availability is better when it is higher. Second, a developer might not be aware of the actual values. He might request a service that is available 100%, while the best assured availability among the retrieved services might be 90%. Using the QoS levels, a developer can specify that he needs a certain service to have a $VeryGood$ level for a certain QoS property.
We also believe that specifying the expected QoS per service is better than specifying a global QoS for the whole process. This is because QoS properties vary in their importance according to the service functionality. Therefore, developers must be able to make a compromise between the QoS levels according to each service. For example: a printing service may have a bad availability and still be acceptable, unlike a bank account service, which should have a very good availability to be accepted.
Once all of the needed services are specified by their functionality and QoS levels, a developer can define his ABPEL process.

2) Abstract BPEL Process (ABPEL): An ABPEL process can be built exactly like a BPEL process and using any available development tool that supports Web service orchestration using BPEL (like $NetBeans$ 6.7.1, for example). The only difference is that it is built using abstract services defined in an AWSDL interface (Figure 2).

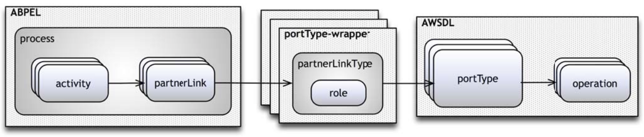


Figure 2. Abstract BPEL process.

Invoking an operation of a Web service begins by assigning values to its input parameters. This is expressed in BPEL language by the structures *<assign> <copy> <from .. /> <to .. />* and then *<invoke .. operation=".." .. />*. Parsing these structures enables us to identify the composition links between the services. Hence, we can identify two possible composition modes, saying that $service\text{-}A$ (source service) and $service\text{-}B$ (target service) are either:
Fully-Composable: when (some or all) of the output parameters of an operation (in $service\text{-}A$) are linked to all of the input parameters of an operation (in $service\text{-}B$); or,
Partially-Composable: when the input parameters of an operation (in $service\text{-}B$) are not only linked to output parameters of an operation (in $service\text{-}A$), but also to others.

[1]Available online: http://www.lirmm.fr/∼azmeh/tools/boxplot.html

As soon as the ABPEL process is defined, the framework starts searching and retrieving concrete services to instantiate the process, in order to render it executable.

B. Process Realisation

Services will be retrieved, filtered, classified, and selected (details in [6]), in order to instantiate the specified ABPEL process and make it executable. Moreover, services that are equivalent to the selected ones (backups) will also be retrieved and kept, in order to support the continuity of the process.
This stage starts functioning once it receives the AWSDL interface and according to the three steps that we have identified in Section I: Discovery, Classification and Selection.

1) Discovery: the AWSDL interface is analyzed and the set of requirements is extracted, including the needed functionality (parameters names and types) and the expected QoS for each specified service. The extracted words (parameters names) for each needed service are used to query a Web service search engine. Then, the returned set of services is retrieved together with the QoS values for each service. Each set is filtered, in order to identify the services that are compatible with the needed functionality and discard the rest. Therefore, each service WSDL description is parsed in order to extract the operation signatures (input and output parameters with their types). A service is functionally compatible if it offers an operation having the exact requested signature. It is adaptable compatible if it has a matching signature, but the types are not exactly the same. We use rules for deciding adaptability, for example: an int value can be converted to String but not the inverse.

2) Classification: the sets of filtered services are classified using a classifying method called Relational Concept Analysis (RCA) [8], according to their QoS properties and values, as well as the composability between them. In the simplest case, the input of RCA can be one table of services described by the QoS they provide. We remind here that we calculate QoS levels from actual numerical values, using the BoxPlot technique, as we explained earlier in Section II-A1. For example, in Table I, four services of a set $WS1$ (possible candidates for an abstract service) are described by their availability (A) and performance (P) levels. We can notice that when a service has a good level of QoS, it also has medium and bad. The explanation is very simple if we take back the numerical values that these levels represent. For example: if a $BadA$ represents an availability that is higher than 10%, $MedA$ higher than 40%, and $GoodA$ higher than 90%, then a service's availability that is higher than 90% is also higher than 40% and higher than 10%. An RCA-based classification is represented by a concept lattice that organizes services by their QoS, as illustrated in Figure 3. In this lattice (generated using ConExp [9]), we can have a general view of the services and the relations between them. In order to interpret it, we have to follow some simple rules:

Table I
A TABLE OF SERVICES $WS1$ DESCRIBED BY THEIR QOS LEVELS.

$WS1$	$BadA$	$MedA$	$GoodA$	$BadP$	$MedP$	$GoodP$
$ws1.1$	×	×		×	×	×
$ws1.2$	×	×	×	×		
$ws1.3$	×	×	×	×	×	×
$ws1.4$	×			×	×	

The nodes inside a lattice are called concepts. Each concept represents a group of services sharing some QoS levels. In our figures, the services appear in the white labels, while QoS levels appear in the gray ones. Concepts have generalization relations between them, represented by the edges between them. We read the lattice from top to bottom. Higher concepts are more general than lower ones. An edge between two concepts means that the lower concept is a sub-concept of the higher one. For example, in Figure 3, the concept $c5$ is a super-concept for all the other concepts in the lattice. They all inherit $BadP$ and $BadA$. If we consider $c0$, we notice that it is a sub-concept of $c4$ and a super-concept of $c1$. So, the service $ws1.1$ is better than $ws1.4$, because it inherits its properties and has better ones. Furthermore, $ws1.3$ is the best service of all, because it is located at the bottom concept (inheriting all the properties including $GoodA$ and $GoodP$). Choosing between $ws1.2$ and $ws1.1$ requires doing a compromise between $GoodA$ and $GoodP$.

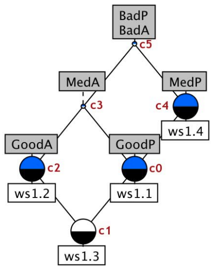


Figure 3. The concept lattice for the services in Table I.

In our case, since we have several sets of services, we have to link the tables by the composability relations existing between the services of different sets. Evaluating the composability is done by checking how the services can be linked, compared to the composition modes extracted from the ABPEL file, as we described in II-A2. So, if in the ABPEL file, a $service\text{-}A$ is supposed to be orchestrated with $service\text{-}B$, then the set of services retrieved for $service\text{-}A$ will be checked for its composability with the set of services retrieved for the $service\text{-}B$ (Fully-Composable or Partially-Composable). A table of services versus services will be constructed for each composition mode. These tables are then used by the RCA method to generate the classification of services by both their QoS levels and composability modes. For example, if we consider another set of services, let us say $WS2$. In Table II, we find the services in $WS2$, described by their QoS levels.

Table II
A TABLE OF SERVICES WS2 DESCRIBED BY THEIR QOS LEVELS.

$WS2$	$BadA$	$MedA$	$GoodA$	$BadP$	$MedP$	$GoodP$	$'ws2.1'$	$'ws2.2'$	$'ws2.3'$	$'ws2.4'$	$'ws2.5'$	$'ws2.6'$
$ws2.1$	×			×	×	×	×					
$ws2.2$	×	×	×	×	×	×		×				
$ws2.3$	×	×	×	×	×	×			×			
$ws2.4$	×	×		×	×					×		
$ws2.5$	×			×							×	
$ws2.6$	×	×		×								×
$query2$	×	×		×	×							×

We notice that we added an attribute per service, for example: $'ws2.1'$. This attribute is considered as an identifier that helps in having a concept per service, for the simplifying lattice interpretation. We also added these attributes for the services in the set $WS1$. We also notice an object called $query2$, in the last line of this table. It is used to express the expected QoS levels for the needed service (specified in the AWSDL interface), which in our case, is $MedA$, $MedP$ for $WS2$ (we add also $query1$: $MedA$, $GoodP$ into Table I for $WS1$). This is further detailed in the selection step.

Moreover, considering that the needed composition mode is fully-composable, having $WS1$ as the source and $WS2$ as the target, we express this relation between the two sets of services in Table III. For example, $ws1.1$ is fully-composable with $ws2.2$ and $ws2.3$.

Table III
THE FULLY-COMPOSABLE (FC) RELATION BETWEEN THE SERVICES OF $WS1$ AND $WS2$.

FC	$ws2.1$	$ws2.2$	$ws2.3$	$ws2.4$	$ws2.5$	$ws2.6$
$ws1.1$		×	×			
$ws1.2$				×		
$ws1.3$		×			×	
$ws1.4$			×			×

Consequently, RCA takes these three tables: $WS1$, $WS2$, and FC, then generates the two lattices in Figure 4.

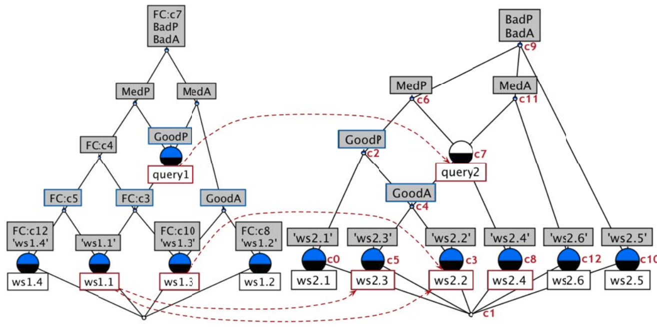


Figure 4. The concept lattices of $WS1$ and $WS2$ classifying services by QoS levels and FC composability (calculated by Galicia [10] & ConExp [9]).

3) Selection: is performed on the generated lattices, by identifying the services that have the expected QoS levels. These expected QoS levels are expressed as queries that are classified into the lattices (by adding them into the corresponding tables as new services), as we can notice in the two previous lattices, $query1$ and $query2$ (Figure 4). These queries help in navigating into the lattices, according to an algorithm introduced in [11]. It extracts the most suitable services that are composable and have the expected QoS levels. This is especially important when considering real business processes scenarios, where several lattices of larger sizes are generated. In short, this algorithm works on identifying a set of services for each lattice, in respect to the requested query and the composition relations. These services are located at the sub-concepts of the concept where a query appears. For example: for $query1$ in the left lattice, we can identify $ws1.1$ and $ws1.3$. Correspondingly, for $query2$ in the right lattice, we can identify $ws2.3$, $ws2.2$, and $ws2.4$, in the concepts $c5$, $c3$, and $c8$ respectively. These services have the expected QoS levels or better. In order to verify their composability, we have to follow their relational attributes, appearing in the gray labels as $FC : c_i$. In our example: $ws1.1$ can be fully-composed with $ws2.2$ ($FC : c3$) and $ws2.3$ ($FC : c5$). Similarly, $ws1.3$ is fully-composable with $ws2.2$ ($FC : c3$). In contrast, $ws1.3$ can be fully-composed with $ws2.5$ ($c10$) but $ws2.5$ does not satisfy $query2$. Moreover, $ws2.4$ ($c8$) satisfies $query2$, but can not be fully-composed neither with $ws1.1$ not $ws1.3$.

Thus, the set of possible service selections can be for example: either $ws1.1$ with $ws2.3$, having $ws2.2$ as a backup for $ws2.3$. Or else, $ws1.3$ with $ws2.2$, having $ws1.1$ as a backup for $ws1.3$.

III. Validation

We carried out our experiments according to two parts: describing an abstract BPEL process called $WeatherProcess$, which works on finding the weather information of a given ip address; then realizing an executable copy of this process using concrete Web services.

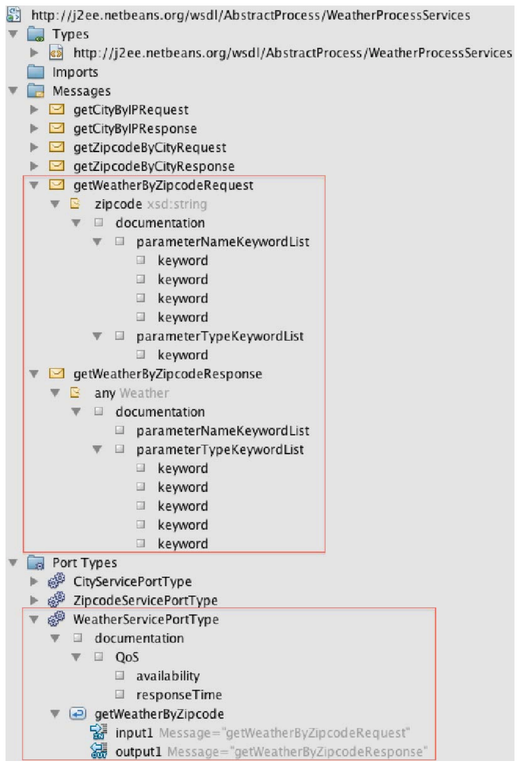
A. Describing the WeatherProcess

We divided the process functionality into three pieces: getting the city corresponding to the given ip, getting the zip code of this city, and finally, getting the weather information for the obtained zip code.
We constructed an AWSDL interface and described three abstract services (PortTypes): $CityService$, $ZipcodeService$, and $WeatherService$. Each service of them provides an operation: $getCityByIP$, $getZipcodeByCity$, and $getWeatherByZipcode$, respectively. We specified for each operation its input and output parameters together with their types, in addition to possible alternative names. We also specified the expected QoS levels[2] for each service, as we can see in Table IV[3].

[2]We set our Web service search engine to be Service-Finder [4]. It provides two QoS properties for each service, availability (A) & response time (RT).

[3]More details can be found on: http://www.lirmm.fr/∼azmeh/icws/

In Figure 5, we can see the corresponding AWSDL interface. We expand the messages description for the operation $getWeatherByZipcode$. We can see that for the input parameter ($zipcode$), the user provided a list of four equivalent keywords, which are listed in the WSDL source code. They are as follows: $zipcode$, zip, $postal$, and $postalcode$. The parameter type is specified to be $String$. For the output parameter, we can notice that its name is specified as any. This means that the user is not asking for a specific parameter name, but he is interested by the complex type, which is in our case $Weather$. In this case, the user provided a list of 5 equivalent keywords for the $Weather$ type. They are as follows: $Weather$, $WeatherInfo$, $Forecast$, $WeatherForecast$, and $WeatherReport$.



```
http://j2ee.netbeans.org/wsdl/AbstractProcess/WeatherProcessServices
▼ Types
    ▶ http://j2ee.netbeans.org/wsdl/AbstractProcess/WeatherProcessServices
  Imports
▼ Messages
    ▶ getCityByIPRequest
    ▶ getCityByIPResponse
    ▶ getZipcodeByCityRequest
    ▶ getZipcodeByCityResponse
    ▼ getWeatherByZipcodeRequest
        ▼ zipcode xsd:string
            ▼ documentation
                ▼ parameterNameKeywordList
                      keyword
                      keyword
                      keyword
                      keyword
                ▼ parameterTypeKeywordList
                      keyword
    ▼ getWeatherByZipcodeResponse
        ▼ any Weather
            ▼ documentation
                  parameterNameKeywordList
                ▼ parameterTypeKeywordList
                      keyword
                      keyword
                      keyword
                      keyword
                      keyword
▼ Port Types
    ▶ CityServicePortType
    ▶ ZipcodeServicePortType
    ▼ WeatherServicePortType
        ▼ documentation
            ▼ QoS
                  availability
                  responseTime
        ▼ getWeatherByZipcode
              input1 Message="getWeatherByZipcodeRequest"
              output1 Message="getWeatherByZipcodeResponse"
```

Figure 5. The abstract WSDL describing the needed services for the scenario $WeatherProcess$.

Afterwards, we defined our abstract process, the $WeatherProcess$, by orchestrating the three abstract services with fully-composable composition links.

B. Realizing the WeatherProcess

Using each set of keywords, we retrieved a corresponding set of services (WSDL interfaces), after omitting repeated and invalid endpoints. Then, we parse each WSDL interface and filter by the requested functionality, to determine whether it is compatible, adaptable, or has to be discarded. In Table V, we can see the number of services for each set and at each step.

Table IV
THE SPECIFICATION OF EACH ABSTRACT SERVICE NEEDED FOR THE WEATHERPROCESS DESCRIPTION.

Abstract WS	Operation	Input			Output			QoS	
		parameter	alternatives	type(s)	parameter	alternatives	type(s)	A	RT
CityService (WS1.i)	getCityByIP	ip	ipAddress	String	city	cityName	String	Good	Good
ZipcodeService (WS2.j)	getZipcodeByCity	city	cityName	String	zipcode	zip, postal, postalcode	String	Good	Good
WeatherService (WS3.k)	getWeatherByZipcode	zipcode	zip,postal, postalcode	String	any	–	Weather, WeatherInfo, Forecast, WeatherForecast, WeatherReport	Good	Good

Table V
THE NUMBER OF FILTERED SERVICES FOR EACH SET.

	WS1.i	WS2.j	WS3.k
Retrieved from Service-Finder	94	768	39
Filter1 (Valid)	94	748	37
Filter2 (Functionally-Compatible)	16	96	21

We calculated afterwards the QoS levels for the remaining services of each set. We organized the services into tables with their QoS levels, including the three QoS queries, resulting into three tables (a table per set of services). Then, we extracted the composition links from the ABPEL file, and evaluated the composition modes between each pair of services. This resulted into tables, showing the composability between the services. The collection of tables are finally used by the RCA method, and three concept lattices are generated, as we can see in Figure 6.
Using the lattices navigation algorithm in [11], we extracted the (highlighted) services in Table VI to be the best choices.

Table VI
INFORMATION ABOUT THE SERVICES SATISFYING THE QUERIES AND THE SELECTED ONES (HIGHLIGHTED).

WS	Name	Operation	A(%)	RT(ms)
1.59	Ip2LocationWebService	IP2Location (in) IP:string (out) CITY:string,..	100	257
1.5	GeoCoder	IPAddressLookup (in) ipAddress:string (out) City:string,..	100	328
1.3	IP2Geo	ResolveIP (in) ipAddress:string (out) City:string,..	100	798
2.198	MediCareSupplier	GetSupplierByCity (in) City:string (out) Zip:string,..	85	304
2.8	ZipcodeLookupService	CityToLatLong (in) city:string (out) Zip:string,..	100	439
3.1	USWeather	GetWeatherReport (in) ZipCode:string (out) WeatherReport:string	85	384
3.23	Weather	GetCityForecastByZIP (in) ZIP:string (out) ForecastReturn:complex	100	237

When selecting a service from an extracted set of services, the other services in the set can play the role of backups. For example, if we select the service $WS1.59$ for the first abstract service, $WS1.5$ and $WS1.3$ will be backups for it. In case it disappeared, one of them can replace it to fill the missing functionality and assure the process continuity.

Finally, the selected services are used to generate the desired executable BPEL process, and the backup services are kept to be used when needed.

IV. RELATED WORK

In this section, we study the state of the art of Web service composition, according to two categories: manual and automatic composition. We conclude the section by a comparison of the studied works.

A. Manual Composition

We may cite several graphical tools (Triana [12], CAT [13], SWORD [14], WSTK [15], ZenFlow [16], BPEL2B [17], which help users visually in building their desired compositions. We describe only three of them.
In Triana [12], a composite service is created by dragging the services and connecting them. Composed applications may be written as BPEL4WS graphs. They may be executed from within Triana or any Web services choreography engine or published to the network.
CAT [13], which takes existing WSDL descriptions and extends them with off-the-shelf domain ontologies. It uses these ontologies in examining a user's solution and generating suggestions about how to proceed.
SWORD [14] defines its own simple world model based on defining for each Web service logic rules. A composite service can be realized according to an execution plan.

B. Automatic Composition

In [18], the authors present an approach that extends the heuristic-based approach proposed by [19] by adding QoS constraints. The algorithm receives as input a request, which consists of the provided input concepts, required output concepts and QoS constraints. Each concept is defined in a domain ontology. It produces as output a set of services.
In [20], the authors propose a semantic-based framework for the automatic composition of Web services. They generate composite services from high-level declarative descriptions.
The Web Services Composition Platform, StarWSCoP [21], is introduced with several modules, in particular: a service registry and a discovery engine, a composition engine, a wrapper to achieve interoperability of heterogeneous services and a QoS estimation. It focuses on QoS-based dynamic Web services composition by extending WSDL descriptions with QoS attributes.
In [22], Agflow is presented as a QoS-aware middleware supporting quality-driven Web service compositions. A composite service is specified as a collection of generic service tasks described in terms of service ontologies and combined according to a set of control and data flow dependencies.

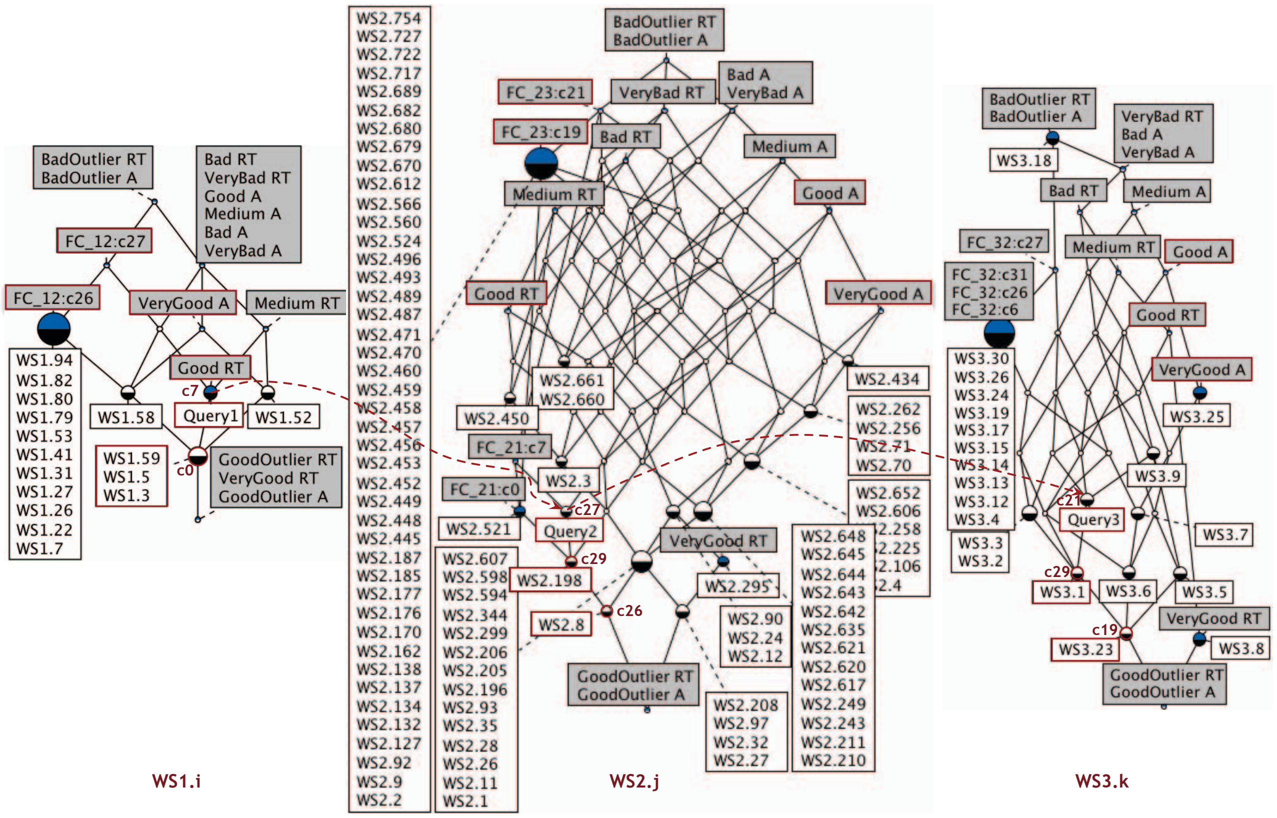


Figure 6. The concept lattices for the compatible sets of services with Good A, Good RT queries, and FC mode.

Very similar to the previous, [23] presents a broker-based architecture to facilitate the selection of QoS-based services. The objective of service selection is to maximize an application-specific utility function under QoS constraints. The problem is modeled in two ways: the combinatorial model and the graph model.
Another similar work is presented in [24]. It proposes an approach for achieving dynamic semantic Web service composition. It is based on the METEOR-S WS composition framework with an added-on constraint analyzer module.

C. *Discussion*

In the manual composition building, users can search by keywords and retrieve services. They still have to check for the composability manually. Such works do not provide QoS information, nor enable searching by a required QoS level.

In the automatic composition building, the works we presented are all based on semantics (which imposes other issues, as we mentioned earlier). Furthermore, they assume that Web services are annotated with semantic information (beyond WSDL). Such semantic information might not be available for current Web services, although it might become available in the future [25]. Some of these works calculate the global QoS value for the whole composition. Users (meaning developers) are not allowed to specify a needed QoS for a certain service. Moreover, QoS is specified by only numeric values.

The manual and automatic composition approaches do not support service backups. However, some of the automatic composition works support dynamic reconfiguration for the business process, when a service changes (disappear or have different QoS values). This reconfiguration can cause an overhead if a backup is needed, because of the several remote interactions with the service registry.

We summarize the presented works in Table VII, according to the following criteria: manual or automatic composition, using semantics, achieving service discovery, checking the functional compatibility, identifying composable services, considering QoS, discovering backup services.

V. Conclusion

In this paper, we discussed the problem of building a business process using BPEL and the challenges surrounding it. We proposed to separate a process description from its realization. Process description is defined abstractly by developers via an AWSDL interface and an ABPEL process. An AWSDL specifies all the needed services by their functionality and QoS, while an ABPEL defines their orchestration. Process realization is achieved automatically through a framework for Web service discovery, classification (based on RCA), and selection. We validated our proposition using real Web services for a WeatherProcess scenario.

Table VII
WORKS COMPARISON ACCORDING TO THE SPECIFIED CRITERIA.

Work	Manual	Automatic	Semantic WS	Discovery	Functionality	Composability	QoS (per WS)	QoS (globally)	Backups
Triana [12]	✓	×	×	✓	×	×	×	×	×
CAT [13]	✓	×	✓	✓	×	×	×	×	×
SWORD [14]	✓	×	✓	×	×	×	×	×	×
Oliveria et al. [18]	×	✓	✓	×	×	✓	×	✓	×
Medjahed et al. [20]	×	✓	✓	✓	✓	✓	×	✓	×
StarWSCoP [21]	×	✓	✓	×	×	✓	×	✓	×
AgFlow [22]	×	✓	✓	×	×	✓	✓	✓	×
QoS-Broker [23]	×	✓	✓	×	×	✓	×	✓	×
METEOR-S [24]	×	✓	✓	×	×	✓	×	✓	×
Our Framework	×	✓	×	✓	✓	✓	✓	×	✓

We studied the related work, by listing two kinds of works, manual and automatic. When dealing with the manual technique, the user must apply a functional filtration and must check for the composability between two services. Considering the automatic composition techniques, the resulting compositions suffer from inflexibility because the QoS is calculated globally for the whole composition and they do not support backups identification.

In a future work, we plan to enrich the specification of the needed functionality with semantic descriptions. We also intend to realize dynamic Web service substitution and to conduct further experiments on more complex scenarios.

REFERENCES

[1] E. Newcomer and G. Lomow, *Understanding SOA with Web Services*. Addison-Wesley Professional, 2004.

[2] “Web Services Business Process Execution Language Version 2.0”, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[3] “Seekda, WS search engine”, http://webservices.seekda.com

[4] D. Cerizza, A. Funk, A. Turati, A. Beffani, E. D. Valle, H. Lausen, I. Celino, N. Steinmetz, S. Brockmans, W. Schoch: “D1.4: Service-finder refined design of service-finder as a whole”, Tech. Rep., April 2009. [Online]. Available: http://www.service-finder.eu/attachments/D1.4.pdf

[5] D. A. Menascé, “Qos issues in web services.” *IEEE Internet Computing*, vol. 6, no. 6, pp. 72–75, 2002.

[6] Z. Azmeh, M. Driss, F. Hamoui, M. Huchard, N. Moha, and C. Tibermacine, “Selection of composable web services driven by user requirements.” In Proceedings of *ICWS'11*. IEEE Computer Society, 2011, pp. 395–402.

[7] W. A. Larsen and J. Tukey, “Variations of box plots,” vol. 32, 1978, pp. 12–16.

[8] M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev, “Relational concept discovery in structured datasets.” *Ann. Math. Artif. Intell.*, vol. 49, no. 1-4, pp. 39–76, 2007.

[9] Conexp. [Online]: http://conexp.sourceforge.net/

[10] GaLicia, “Galois lattice interactive constructor,” 2002, http://www.iro.umontreal.ca/ galicia.

[11] Z. Azmeh, M. Huchard, A. Napoli, M. Rouane Hacene, and P. Valtchev, “Querying Relational Concept Lattices,” in *CLA'11: The 8th International Conference on Concept Lattices and their Applications*, France, 2011, pp. 377–392.

[12] S. Majithia, M. S. Shields, I. J. Taylor, and I. Wang, “Triana: A graphical web service composition and execution toolkit.” In Proceedings of *ICWS'04*. IEEE, 2004, pp. 514–.

[13] J. Kim and Y. Gil, “Towards interactive composition of semantic web services.” *National Conference on Artificial Intelligence*, 2004.

[14] S. R. Ponnekanti and A. Fox, “Sword: A developer toolkit for web service composition',' in *Proceedings of the 11th International WWW Conference (WWW2002)*, Honolulu, HI, USA, 2002.

[15] “The web services toolkit (wstk)”, http://www.alphaworks.ibm.com/tech/webservicestoolkit.

[16] A. Martínez, M. Patiño-Martínez, R. Jiménez-Peris, and F. Pérez-Sorrosal, “Zenflow: A visual web service composition tool for bpel4ws,” *Visual Languages and Human-Centric Computing, IEEE Symposium on*, vol. 0, pp. 181–188, 2005.

[17] I. Aït-Sadoune and Y. A. Ameur, “Stepwise design of bpel web services compositions: An event-b refinement based approach.” in *SERA (selected papers)*, ser. Studies in Computational Intelligence, R. Y. Lee, O. Ormandjieva, A. Abran, and C. Constantinides, Eds., vol. 296. Springer, 2010, pp. 51–68.

[18] F. G. A. de Oliveira Jr. and J. M. P. de Oliveira, “Qos-based approach for dynamic web service composition.” *J. UCS*, vol. 17, no. 5, pp. 712–741, 2011.

[19] T. Weise, S. Bleul, D. E. Comes, and K. Geihs, “Different Approaches to Semantic Web Service Composition.” in *Proceedings of The Third International Conference on Internet and Web Applications and Services (ICIW'08)*, A. Mellouk, J. Bi, G. Ortiz, K. W. D. Chiu, and M. Popescu, Eds. IEEE Computer Society Press: Los Alamitos, CA, USA, 2008, pp. 90–96.

[20] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid, “Composing Web services on the Semantic Web.” *The VLDB Journal*, vol. 12, no. 4, Springer-Verlag, pp. 333–351, 2003.

[21] H. Sun, X. Wang, B. Zhou, and P. Zou, “Research and implementation of dynamic web services composition.” in *APPT*, ser. Lecture Notes in Computer Science, X. Zhou, S. Jähnichen, M. Xu, and J. Cao, Eds., vol. 2834. Springer, 2003, pp. 457–466.

[22] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, “Qos-aware middleware for web services composition.” *IEEE Trans. Software Eng.*, vol. 30, no. 5, pp. 311–327, 2004.

[23] T. Yu, Y. Zhang, and K.-J. Lin, “Efficient algorithms for web services selection with end-to-end qos constraints.” *TWEB*, vol. 1, no. 1, 2007.

[24] R. Aggarwal, K. Verma, J. A. Miller, and W. Milnor, “Constraint driven web service composition in meteor-s.” in *IEEE SCC*. IEEE Computer Society, 2004, pp. 23–30.

[25] M. D. Ernst, R. Lencevicius, and J. H. Perkins, “Detection of web service substitutability and composability.” in *WS-MaTe 2006: International Workshop on Web Services — Modeling and Testing*, Palermo, Italy, June 9, 2006, pp. 123–135.