Improving Dynamic Web Service Selection in WS-BPEL Using SPARQL

Neil Paiva Tizzo[1,3]
[1]PUC Minas
P. Caldas, MG, Brazil, 37701-355
e-mail: neil@pucpcaldas.br

Juan Manuel Adán Coello[2]
[2]PUC-Campinas, Cx.P. 317
Campinas, SP, Brazil, 13012-970
e-mail: juan@puc-campinas.edu.br

Eleri Cardozo[3]
[3]DCA-FEEC-UNICAMP
Campinas, SP, Brazil, 13083-970
e-mail: eleri@dca.fee.unicamp.br

Abstract— **The dynamic selection of Web services must be able to adapt to a changing environment as services are permanently inserted, removed and modified. However, most composition languages do not support such dynamic selection, or support it in a restrict sense, as in WS-BPEL. In this work we employ available technologies, especially the WS-BPEL and SPARQL languages, in order to develop a model for the dynamic selection of Web services. In this model the user expresses the service requirements in the form of a query that will be resolved at runtime. This guarantees the adaption of the system - in a changing environment - without requiring to refactor the entire composition. The restrictions of such a system are listed. An implementation of the model and a case study are reported as well.**

Keywords - Dynamic Web service selection, Service composition, Semantic Web services, SPARQL, WS-BPEL

I. INTRODUCTION

One of the benefits of Service Oriented Architectures (SOA) [1] is to promote the rapid development of software through the reuse of services. Services are software components that can be incorporated into distinct distributed applications. Currently, the most prominent technology to implement a software service is known as Web services.

A Web service is a software component or a logical unit of application that communicates through a set of Internet standards. As a component, it represents a functionality implemented in a "black-box" that can be reused without the concern of how the service was implemented.

Service compositions make use of one or more services in workflows in order to reach a specific objective. The final product of a composition is another service. In this context, the creation of a new service can be achieved simply by composing a set of existing services [2]. Service compositions can be static or dynamic [3]. The static composition consists of a workflow where the invoked services are chosen at design time, being its execution process invariant in terms of the primitive services employed in the composition. A dynamic composition can, at execution time, choose which services will be invoked. The static composition is adequate for well defined business processes acting on environments where the services do not change or change very infrequently. However, in highly dynamic environments, applications demand more flexibility in terms of service composition.

The composition process can range from manual to fully-automated (without human intervention). Between these two extremes, there is the semi-automatic composition in which software and humans cooperate in the composition process [4]. Automatic service composition can occur at design or execution time. In the latter case it is called dynamic composition because it must have the ability to adapt to a changing environment.

The dynamic service composition can build new services on-demand. Moreover, the dynamic composition is the only way to adapt the behavior of running components in highly available applications such as banking and telecommunication systems. Such applications cannot be brought offline in order to upgrade or replace the composite services [5]. Dynamic service composition deals with insertion, exclusion and modification of services without requiring to refactor the entire composition. It also favors fault tolerance as a faulty service can be replaced at runtime. For Li et al. [6], Web service composition is a very important way to offer novel value-added services on the fly, which entrusts with the existing services via reuse and integration. Despite all these, the Web service composition still is a highly complex task, and it is already beyond the human capability to deal with the whole process manually [7].

There are several languages for specifying service compositions, mainly static compositions. One of the most popular of such languages is Web Services Business Process Execution Language (WS-BPEL [8] or BPEL for short). Except for very restricted cases, this language does not allow dynamic compositions. This work proposes a semi-automatic service selection strategy that uses BPEL and SPARQL. The user writes the service requirements as SPARQL queries and sends it to a BPEL composition engine. Specific services are selected through the evaluation of the SPARQL query that returns the service endpoints. The composite service catches this endpoint, makes the necessary transformations, and invokes the selected service. In other words, we defined a model where the user (a person or another application) sends a SPARQL query to a BPEL process that dynamically selects a service at execution time.

SPARQL [9] is a Resource Description Framework

Partial support provided by CAPES, PUC Minas and FAPEMIG.

(RDF) [10] query language and it is considered a component of the Semantic Web. SPARQL allows queries consisting of triple patterns, conjunctions, disjunctions, and optional patterns. In this work, service profiles are specified in RDF language.

The article is organized as follows. Section II describes the background. Section III provides a general review of service composition, focusing on some related work. Section IV presents the proposed composition model. Section V describes an implementation (a use case) of the proposed model. Finally, section VI presents the conclusions and future work.

II. Background

In this section we will review briefly how BPEL supports the dynamism and also a short description of the RDF and SPARQL languages.

A. *The Dynamism in BPEL*

BPEL is an XML-based programming language to describe high level business processes based on Web services. The BPEL process is basically a flowchart as the expression of an algorithm. Each stage in the process is called an activity. It has a collection of primitive activities, such as invoke, receive and reply. These primitive activities can be combined in more complex activities using some of the structures supplied by the language: sequence of stages (sequence); branch of execution flow (switch); loop (while); alternative branch (pick); and parallel execution of stages (flow). Currently, the BPEL language supports two forms of dynamism [11]:

- The amount of Web services to be consumed is unknown at design time: an invoke activity inside a loop can invoke countless services;
- The service address (endpoint) to be consumed is unknown at design time.

The services that a business process interact with are designed as Partner Links (Fig. 1). Each Partner Link is characterized by a partnerLinkType. The dynamism is carried through by the concept of “dynamic binding” of Partner Links. Dynamic binding allows the addition of new services through the configuration of input in execution time [12].

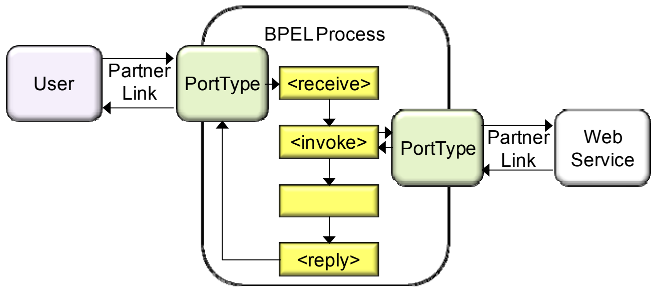

Fig. 1. The role of the partner link and portType in a BPEL process.

The BPEL process is dependent on the information of the Web Services Description Language (WSDL) interface defined in the portType [13], while the endpoint reference (that maps the binding of a service) allows the dynamic redefinition of the service localization. In essence, the endpoint reference is a dynamic alternative to the element of the static service defined in the WSDL interface.

B. *RDF - A Language to Describe Resources*

RDF [10] is recommended by the World Wide Web Consortium (W3C) to model the meta-data assigned to the Web resources. The fundamental concept of RDF is the triple: a resource (the subject) is linked to another resource (the object) through an arc labeled with a third resource (the predicate). This means that the subject has a property (predicate) valued by object.

The XML language does not allow to express the data semantics, i.e., there is no meaning associated to the nested markers, which compels the applications to make their own interpretation.

C. *SPARQL - An RDF Query Language*

SPARQL is a query language and data access protocol for the Semantic Web. It is defined in terms of the W3C's RDF data model and works for any data source that can be mapped into RDF. It is defined by three documents: SPARQL query language [9], SPARQL protocol [14], and SPARQL XML results format [15].

The SPARQL protocol is a way of transmitting the query from a SPARQL customer to a SPARQL processor. The protocol is described in two forms: as an independent abstract interface of any concrete accomplishment, implementation or binding to another protocol; and as HTTP and SOAP bindings of this interface [14].

A simple query example is shown in Code 1. In this query, we wish to return the Unified Resource Identifier (URI) from the Web blog of “Neil Paiva Tizzo” in a local data repository, called blog.rdf.

Code 1. An example of SPARQL query.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?uri
FROM
<http://www.inf.pucpcaldas.br/~neil/dynamic-
composition/blog.rdf>
WHERE  {
  ?x foaf:name "Neil Paiva Tizzo" .
  ?x foaf:weblog ?uri .
}
```

The first line of the query simply defines a PREFIX for the foaf namespace, so it is not necessary to type it in full each time it is referenced. The SELECT clause specifies what the query should return, in this case, a variable named uri. SPARQL variables are prefixed with either ? or $. FROM is an optional clause that provides the URI of the dataset to use (here, it indicates the URI of the blog´s graph on the Web). Finally, the WHERE clause consists of a series of triple patterns, expressed using Turtle-based syntax. These triples together comprise what is known as a graph pattern. The query attempts to match the triples of the graph pattern to the model. Each graph pattern's variable that matches the model's nodes becomes a query solution, and the values of the variables named in the SELECT clause become part of the query results.

III. Related Work

The service composition has two types of participants: the service provider and the service requester. The service provider publishes services that offer information and functionalities for use, and the requester consumes them.

Currently, two techniques of automatic composition divide the attention of the specialists: those based on workflow and those based on AI planning [7]. Techniques based on AI planning aim to automatically generate the workflow. In this sense, Thakkar et al. [16] propose an extension to the "inverse rules" query reformulation algorithm to generate a universal integration plan to answer user queries. Ambite and Kappor [17] have presented an approach to automatically generate data processing workflows in response to a user data request, including the necessary data integration and translation operations. Sirin et al. [18] give a list of several arguments explaining the ways in which the HTN approach is promising for service composition. They explore how to use SHOP2 HTN planning system to do automatic composition of OWL-S Web service. Tran et al. [19] propose a composition of semantic Web services based on planning algorithms that realizes depth search. The algorithms include name matching, computation of the similarity semantics, cycle detention and optimization.

In the real world, there usually are several possible plans which can solve one specific high-level business process. Chen et al. [20] address the aspect of finding multiple composition plans and then selecting the most appropriate for a user.

The dynamism in BPEL must be understood in the context of the techniques based on workflow. In the remainder of this section we will review briefly the composition based on workflow.

A. *Service Composition Based on Workflow*

This technique is used, mostly, in situations where the requester has already defined the process model, that is, the service is specified as a workflow of linked activities stating the flow of data and control.

The automatic composition based on workflow will find and select, for each activity defined in the workflow, just one service that is capable of successfully realizing it. This process is also known as service matching [21].

An activity can be described as a set of functional requirements, such as input, output, prerequisite and effect (IOPE), and, in some cases, non-functional requirements such as price, locality, Quality of Service (QoS), events, objective, and resources. The published service that better satisfies such requirements is linked to the service requester and will be invoked during the execution of the workflow. Service matching is done by a composition engine.

Fig. 2 illustrates a service requester before (a) and after (b) the composition. The service requester is defined as a workflow with activities At1, At2 and At3. The composition engine searches, based on the matching rules, the available repositories looking for published services that better satisfy the requirements of the activities. In this example, it selects the services S1, S2 and S3 to match the activities At1, At2 and At3, respectively. In the right part of the Fig. 2, the matching is represented by the dotted arrows and the published service S4 was not used.

EFlow [22] is an example of this kind of composition where the data and control flow are defined manually, however, the service can be bound at design time or at run-time.

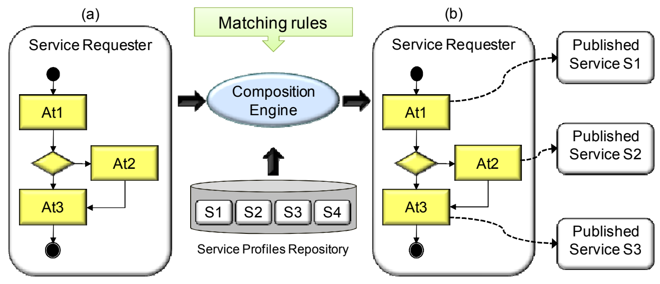

Fig. 2. The service composition regards the area of Workflow.

Tang and Ai [23] use genetic algorithms to solve the problem of Web service selection (called as composition) with QoS restrictions. A user describes a workflow of abstract services and the algorithm selects, for each abstract service presented in the workflow, an implementation (a concrete Web service) in such a way that the overall QoS is maximal. QoS selected criteria are: response time, price, reputation, reliability and availability.

Klusch et al. [24] propose a technique for service selection where the provider does not exactly match the input and output parameters of the service requester. Five rules based on formal (logical) descriptions and Information Retrieval (IR) techniques, presented in decrease priority order, establishes the similarity degree between both.

As one of the objectives of this work is to evaluate the ability of some available technologies (in this case BPEL and SPARQL) to realize the dynamic service selection, the input and output parameters of the requester and the provider must exactly match each other. The selection is, so, based on no functional parameters like price, location etc.

Next section presents the composition model that is able to select a service that will be invoked at runtime through a SPARQL query defined by the user.

IV. The Dynamic Composition Model

Fig. 3 shows the schematic diagram of the service composition model that dynamically selects a service using a SPARQL query. Here the actors are: the published services, the composite service (a BPEL process), the service selector and the user. Each one of these actors can be stored in different Web servers. The communication between actors, and between the actors and the repository, is illustrated by the dotted arrows; and the control flow between the BPEL activities is denoted by solid arrows. The numbers show the execution order.

The user (a person or other application) specifies the selection requirements and sends them to the composite service (step 1). The service selector receives the selection requirements from the composite service, makes an internal transformation to obtain an SPARQL query, queries the available RDF files (that represent the service profiles), and returns to the composite service the results that match the requirements (step 2).

The composite service (see section IV.C) is a BPEL process that has at least three activities that must be considered: the first one calls the service selector; the second one modifies the partner link to make the dynamic binding (step 3); and the last one invokes the selected service (step 4).

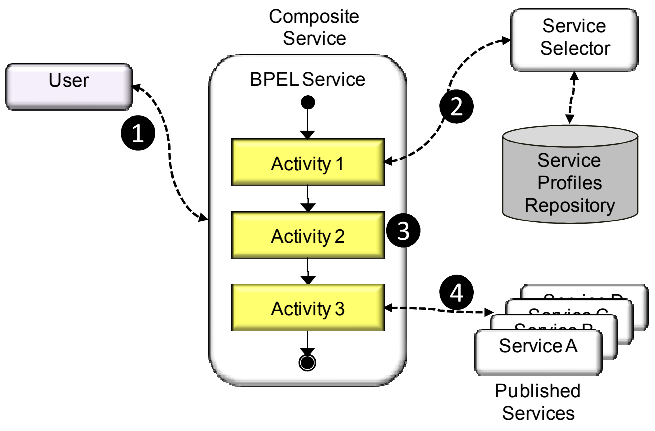

Fig. 3. The service composition model.

The service profile is an RDF file that is published in the Web. This is a more practical alternative to the use of formal and inflexible service description repositories, such as Universal Description, Discovery, and Integration (UDDI) [25]. The service profiles are stored in Web servers that can be indexed by search engines, accessed from Web browsers or from programming languages.

To make an SPARQL query it is necessary to specify the URI of the RDF file. As, in our model, the service profiles are spread over the Internet, the service selector can obtain its location in many different ways, defined as search domains:

1. The composite service or the user can send a list of URI (URI of RDF files) to the service selector;
2. The composite service (or the user) can send a list of the URIs of the servers where the services profiles are stored;
3. The composite service (or the user) can trust third parties to search the Web for service profiles.

As SPARQL allows only one URI to be specified, and it needs to be an URI of an RDF file, our query is in fact a “pseudo SPARQL query”. The service selector reads this pseudo SPARQL query and creates real SPARQL queries. For example, if two URI are specified in the FROM clause, the service selector is designed to create one SPARQL query for each one; Code 2 is an example of a pseudo SPARQL query. It needs to be transformed in two SPARQL queries; one of them is showed in Code 1.

The first search domain is the most restricted. The services that can be selected at runtime were chosen a priori by the user. This solution is very trustworthy but less adaptive.

In the second search domain, it is necessary to search the server to find the service profiles. It means that the service profile (or profiles) will only be known at execution time. A service profile can be inserted or deleted in the server, but the composite service and the service selector remain unchanged. This search domain increases the adaptability of the composition.

Code 2. An example of pseudo SPARQL query.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?uri
FROM {
  <http://www.inf.pucpcaldas.br/~neil/dynamic-
  composition/blog.rdf> &&
  <http://www.inf.pucpcaldas.br/~paiva/dynamic-
  composition/blog.rdf>
}
WHERE {
  ?x foaf:name "Neil Paiva Tizzo" .
  ?x foaf:weblog ?uri .
}
```

In the last search domain, the composite server (or the user) will trust a search engine to search the Web for service profiles. It means that neither the server nor the service profiles are known before execution time. This solution is the most adaptive, but, at the same time, the least reliable. Swoogle [26], for example, could be defined as the search engine for this kind of domain.

A. The Service Profile

We use a very simple service profile because we want to focus our attention on the ability to dynamically select a service based on its profile, but this is not a restriction.

The service profiles are described in RDF files. Each service has its own file and they are distributed over the Internet in different servers. The service selector needs to read each service profile specified in the search domain and to format an appropriate answer to the composite service. Code 3 contains an example of a service profile in RDF format. The declarations of the namespaces were omitted.

Code 3. The profile of the service A.

```
<rdf:RDF>
  <rdf:description rdf:ID="id111">
    <ws:location rdf:datatype="xsd:string">Pocos
      de Caldas</ws:location>
    <ws:price rdf:datatype="xsd:integer">
      100
    </ws:price>
    <ws:service>
      <ws:uri
        rdf:about="http://192.168.16.23:8888/
        BPELDinamicoJoseki-
        ServicoA-context-root/ServicoA">
      </ws:uri>
    </ws:service>
  </rdf:description>
</rdf:RDF>
```

The RDF file makes the description about a resource, service A, identified by the tag `<rdf:description`

`rdf:ID="id111">`. The location, price and URI of the service are defined in this file.

B. The Service Selector

The service selector functionally works as a service broker. Our service composition model is able to select Web services at runtime through SPARQL queries. After the execution of a query, the path (an URI) of the selected service is returned to the composite service to make the dynamic binding. Physically, the task of selecting a service is done by the query service and by the SPARQL server (Fig. 4). The query service receives the requirements to select a service from the user, assembles a SPARQL query and sends it to the SPARQL server.

The SPARQL server has a SPARQL endpoint, i.e., an URI that receives the solicitations from SPARQL clients through the Internet and process the queries, being able to use a local RDF file or a remote one.

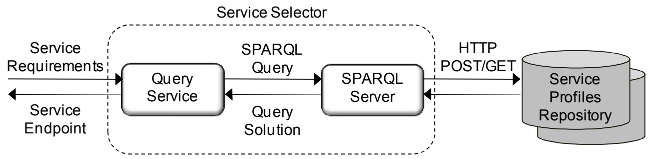

Fig. 4. The service selector is composed by the query service and by the SPARQL server.

In this work, the second form of the SPARQL protocol implementation was used: all communication is made over the HTTP protocol. In such a way, the query service which wishes to execute a SPARQL query sends its request in the form of a HTTP GET or POST method. In turn, the SPARQL server executes the query by accessing the service profile stored in a Web server using the same GET or POST method. The reply is coded in XML and returned to the solicitant.

The use of SPARQL in our model showed some limitations of the language to be used in a dynamic service composition environment. For dynamic service composition it would be very useful to specify a server or even do not specify any server. So, beyond assembling the query, another important task that needs to be done by the service selector is to deal with this limitation. Depending on the search domain, the service selector needs to:

- Assemble all the service files in just one file;
- Query file by file and assemble the answer.

C. The Composite Service

The composite service, that is a BPEL process, is composed of, at least, three activities. The first activity invokes the service selector as described previously. The second activity is a BPEL assign activity that dynamically defines the partner link. The last activity invokes the selected service.

In BPEL, a dynamic binding is done by defining a dynamic endpoint. This process is detailed below (step 3 in Fig. 3):

a) Define a variable of type `endpointReference` to store the address of the service that will be invoked. This variable can later be assigned to a partner link in order to modify the address and the name in execution time. We called this variable `partnerReference`:

```
<variable name="partnerReference"
element="wsa:EndpointReference"/>
```

b) This variable must follow the standard for Web services addressing, this means that it must have the namespace `wsa`: defined in the top of the BPEL file as:

```
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03
/addressing"
```

c) By default, a BPEL process uses strings as input and output. To modify this, it is necessary to define or import a new XML Schema type and use it as part of the message types. So, we will import the XML Schema that defines the Web service addressing. This must be done in the WSDL file of the BPEL process. The `192.168.16.26` and the `8888` are, respectively, the IP address and the port number of the server that hosts the XML Schema:

```
<schema attributeFormDefault="qualified"
    elementFormDefault="qualified"
    targetNamespace="http://schemas.xmlsoap.org/
    wsdl/"
    xmlns="http://www.w3.org/2001/XMLSchema">
  <import
    namespace="http://schemas.xmlsoap.org/ws/
    2003/03/addressing"
    schemaLocation="http://192.168.16.26:8888/
    orabpel/xmllib/ws-addressing.xsd"/>
</schema>
```

d) The variable `partnerReference` has two parts: the address and the name of the service. For now, its content will be a null reference. That is made through the following XML code fragment:

```
<assign name="nullPartnerReference">
  <copy>
    <from>
      <EndpointReference
        xmlns="http://schemas.xmlsoap.org/
        ws/2003/03/addressing">
      <Address/>
      </EndpointReference>
    </from>
    <to variable="partnerReference"/>
  </copy>
</assign>
```

e) Assign the service address to the variable `partnerReference`. In this case, the address of the service to be invoked is stored in the variable returned by the service selector, called `serviceSelectorOutputVariable`:

```
<assign name="setEndpointReference">
  <copy>
    <from
      variable="selectorServiceOutputVariable"
      part="result"/>
    <to variable="partnerReference"
    query="/wsa:EndpointReference/wsa:Address"/>
  </copy>
</assign>
```

f) Finally, the last step is to copy the value of the modified variable `partnerReference` to the attribute `partnerLink` of the service to be invoked, called `selectedService`:

```
<assign name="setPartnerlink">
  <copy>
    <from variable="partnerReference"/>
    <to partnerLink="selectedService"/>
  </copy>
</assign>
```

The dynamic binding in the BPEL process decouples the business logic from partner addressing, making processes more adaptive and portable. In essence, it allows the system to adapt to conditions that would not otherwise be known at design time [11]. Here, we improved the BPEL dynamic capability by selecting a service with a SPARQL query that is defined by the user at execution time.

V. A USE CASE

In this section we present some details about the implementation of the model described above, how the code was distributed, and the tools used to develop, deploy and execute it. The use case is simplified in a way that it represents only the relevant characteristics of the service selection in our model. But it can be easily extended to incorporate many services and complex service descriptions.

The services were implemented in Java using the Oracle JDeveloper, an integrated environment to development Java-based SOA applications [27].

Seven hosts were used to execute and distribute the system as shown in Fig. 5. Three different kinds of servers were used: Oracle Application Server, Apache Tomcat and Joseki.

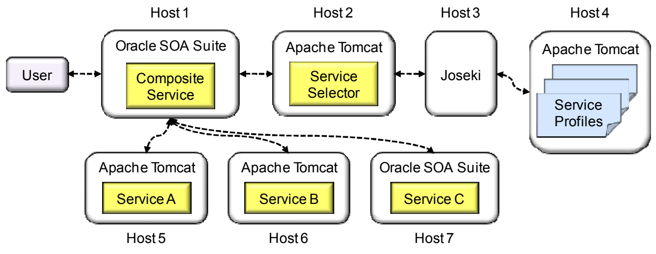

Fig. 5. The distributed architecture of the system.

Three Web services were implemented: A, B and C and deployed in three different servers. As a severe restriction imposed by the BPEL language, the dynamic binding can only be done between the services that have the same interface.

The Oracle Application Server is distributed as part of the Oracle SOA Suite [28]. The composite service and service C were deployed in this server. Apache Tomcat is an implementation of the Java Servlet and Java Server Pages technologies [29]. We used the Apache HTTP Server Version 2.2 to host the service profiles and also the services A and B. Joseki is an HTTP engine that supports the SPARQL Protocol and the SPARQL RDF query language [30]. We used Joseki RDF Server 3.2 as part of the functionalities of the service selector.

As Fig. 5 shows, the user sends a request to the composite service deployed in a BPEL server (part of the Oracle SOA Suite). An invoke BPEL activity calls the service selector, deployed in an Apache Tomcat. This service sends the SPARQL query to the Joseki server. The Joseki server accesses the service profiles and the answer of the query is sent back to the composite service. At the composite service, the answer (a service URI) is used to create the dynamic partner link and, finally, an invoke activity calls the select service.

To demonstrate the use of our model and implementation, it is necessary to create the profiles of the service A, B and C. Except for the value of the price and its localization, the profiles of services B and C are identical to the profile of the service A (shown in Code 3).

Code 4 shows the query sent by the user to the composite service. It queries the URI of a service that has a price less than 140. The clause FROM defines the location of the service profiles.

The FILTER keyword in SPARQL restricts the results of a query by imposing constraints on values of bound variables. These value constraints are logical expressions that evaluate to boolean values, and may be combined with logical operators. For instance, the filter in Code 4 modifies the query to return only a list of service URI that has a price lower than 140. In this case, only the URI of service A matches this condition.

Code 4. The query sent by the user to the composite service.

```
PREFIX ws:
<http://www.inf.pucpcaldas.br/neil/webservice#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?uri
FROM {
  <http://192.168.16.29:89/wsdl/servicoA.rdf> &&
  <http://192.168.16.29:89/wsdl/servicoB.rdf> &&
  <http://192.168.16.30:89/wsdl/servicoC.rdf>
}
WHERE {
  ?contributor ws:price ?price ;
  ws:service ?uri .
  FILTER (xsd:integer(?price)< xsd:integer(140))
}
```

It must be noted that Code 4 is not a valid SPARQL query because it specifies that the query must be performed in three different RDF files (serviceA.rdf, serviceB.rdf and serviceC.rdf), but we introduced this modification to define the search domain, in this case, a list of RDF files that can be stored in different servers (see section IV.B).

The service selector transforms this query into a standard SPARQL query, one query for each RDF file, to use the Joseki server to query file by file. The standard SPARQL query for the RDF file of service A is shown in Code 5. The query for services B and C are analogous. Afterwards, it assembles the answer to the composite service.

Service A is the only service that satisfies the requirements of the user. The XML document returned from the query is shown in Code 6. The service selector retrieves the URI and sends it back to the composite service.

Code 5. The standard SPARQL query for the service.

```
PREFIX ws:
<http://www.inf.pucpcaldas.br/neil/webservice#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?uri
FROM {
  <http://192.168.16.29:89/wsdl/servicoA.rdf>
}
WHERE {
  ?contributor ws:price ?price ;
  ws:service ?uri .
  FILTER (xsd:integer(?price < xsd:integer(140))
}
```

The BPEL process consumes the URI returned to instantiate the endpoint reference and invokes the service A.

Code 6. The SPARQL XML resulted from the query of Code 4.

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-
results#">
  <head>
    <variable name="uri"/>
  </head>
  <results>
    <result>
      <binding name="uri">
        <uri>
          http://192.168.16.23:8888/
          BPELDinamicoJoseki-ServicoA-context-
          root/ServicoA
        </uri>
      </binding>
      </result>
  </results>
</sparql>
```

In this example, only service A satisfies the user requirements, but in a more general case, more than one service can satisfy the user requirements, so the SPARQL query will return many service URLs. In this case, many solutions can be adopted; the easier one is just to select the first service returned.

VI. Conclusions and Future Work

This paper presented a service composition model able to select the service that will be invoked at runtime through a SPARQL query defined by the user. The query returns the URI of the service that will be consumed by the dynamic composition. The proposed model is able to create a dynamic and flexible service selector that adapts to the insertion, exclusion and modification of services from a list of possibilities without requiring to refactor the entire composition. It is also possible that this dynamic composition uses third-party endpoints, creating new opportunities for dynamic service compositions. Dynamic composition also favors fault tolerance as a faulty service can be replaced at runtime.

By dynamically binding composite services, the BPEL process becomes more agile by adapting quickly to the changes in business conditions. By decoupling business logic from partner addressing, it is possible to make processes much more adaptable and portable. The dynamic binding activities in the composite service should be generated as much as possible, assuming that it is up to the author of the composite service to write these activities for every service that wishes to be selected dynamically.

One of the main objectives of this work was to perform dynamic service selecting using just available technologies. With this requirement, the following problems were identified:

- Only services that have the same interface can be taken into account in the dynamic binding of the BPEL language. In a real world, this will be a very strong restriction;
- It is necessary to use a SPARQL server to execute the query. It will be very useful to incorporate such functionality to the engine of the composite process description language;
- The FROM clause of the SPARQL language is used to specify a RDF file, but for service selector it would be very useful to specify a repository or a server (as in the concept of search domain) where the service profiles are stored. To workaround this problem it is necessary to implement new functionalities in the service selector;
- The query is dependent on service profiles, written in RDF; this may be too high an entry bar for many users; also, it would be good to be able to have a looser coupling between a query and service profiles, e.g., if RDF terms change in a profile, a query may miss it even if it would match. These problems can be solved by adding a translator that receives the user query, described in a high abstraction level, and translates it to a SPARQL query;
- The Web service profiles are described in RDF format. This restriction is important, since Web services descriptions are usually not in RDF format, but currently there is nothing similar as SPARQL for OWL-S [31], for example.

We would like to circumvent some of these limitations in future work. Our first idea is to specify a process language with a native service query and dynamic binding independent of service interfaces.

References

[1] Hao He. (2003, September) What Is Service-Oriented Architecture. [Online]. http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html

[2] Mark Turner, David Budgen, and Pearl Brereton, "Turning Software into a Service," *Computer*, vol. 36, no. 10, pp. 38-44, October 2003.

[3] Schahram Dustdar and Wolfgang Schreiner, "A survey on web services composition," *Int. J. Web and Grid Services*, vol. 1, no. 1, pp. 1-30, 2005, http://www.infosys.tuwien.ac.at/Staff/sd/papers/A%20survey%20on%20web%20services%20composition_Dustdar_Schreiner_inPress.pdf.

[4] Neil Paiva Tizzo, Matthias Fluegge, Ivo J. G. Santos, and Edmundo R. M. Madeira, "Challenges and techniques on the road to dynamically compose web services," *ACM International Conference Proceeding Series*, vol. 263, pp.

40 - 47, 2006.

[5] Atif Alamri, Mohamad Eid, and Abdulmotaleb El Saddik, "Classification of the state-of-the-art dynamic web services composition techniques," *International journal of web and grid services*, vol. 2, no. 2, pp. 148-166, 2006.

[6] Gexin Li, Shuiguang Deng, Haijiang Xia, and Chuan Lin, "Automatic Service Composition Based on Process Ontology," *Third International Conference on Next Generation Web Services Practices*, pp. 3-6, October 2007.

[7] Jinghai Rao and Xiaomeng Su, "A Survey of Automated Web Service Composition Methods," in *Lecture Notes in Computer Science*. Berlin / Heidelberg: Springer, 2005, pp. 43-54.

[8] BPEL XML.org. (1993-2009) Online community for the Web Service Business Process Execution Language OASIS Standard. [Online]. http://bpel.xml.org/

[9] Eric Prud'hommeaux and Andy Seaborne. (2008, January) SPARQL Query Language for RDF. [Online]. http://www.w3.org/TR/rdf-sparql-query/

[10] Graham Klyne and Jeremy J. Carroll. (2004, February) Resource Description Framework (RDF): Concepts and Abstract Syntax. [Online]. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[11] Jeremy Bolie et al., *BPEL Cookbook: Best Practices for SOA-based integration and composite applications development*, Markus Zirn and Harish Gaur, Eds.: Packt Publishing, 2006, http://www.oracle.com/technology/pub/articles/bpel_cookbook/carey.html.

[12] Matjaz Juric. (2005, April) theserverside.com. [Online]. http://www.theserverside.com/tt/articles/article.tss?l=BPELJava

[13] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. (2001, March) World Wide Web Consortium. [Online]. http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[14] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. (2008, January) SPARQL Protocol for RDF. [Online]. http://www.w3.org/TR/rdf-sparql-protocol/

[15] Dave Beckett and Jeen Broekstra. (2008, January) SPARQL Query Results XML Format. [Online]. http://www.w3.org/TR/2008/REC-rdf-sparql-XMLres-20080115/

[16] Snehal Thakkar, Craig A. Knoblock, and José Luis Ambite, "A View Integration Approach to Dynamic Composition of Web Services," *Proceedings of 2003 ICAPS Workshop on Planning for Web Services*, 2003, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.7322&rep=rep1&type=pdf.

[17] José Luis Ambite and Dipsy Kapoor, "Automatic Generation of Data Processing Workflows for Transportation Modeling," *Proceedings of the 8th annual international conference on Digital government research: bridging disciplines & domains*, vol. 228, no. Digital Government Society of North America, pp. 82 - 91, May 20-23 2007.

[18] Evren Sirin, Bijan Parsia, Dan Wu, James Hendler, and Dana Nau, "HTN planning for Web Service composition using SHOP2," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 4, pp. 377-396, October 2004, http://dx.doi.org/10.1016/j.websem.2004.06.005.

[19] B. D. Tran, P. S. Tan, and A. Goh, "Composing OWL-S Web Services," in *21st International Conference on Distributed Computing Systems*, September 2007, pp. 322-329.

[20] Kun Chen, Jiuyun Xu, and Stephan Reiff-Marganiec, "Markov-HTN Planning Approach to Enhance Flexibility of Automatic Web Services Composition," *IEEE International Conference on Web Services (ICWS)*, pp. 9-16, July 2009, http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?isnumber=5175786&arnumber=5175801&count=143&index=1.

[21] Vassileiros Tsetsos, Christos Anagnostopoulos, and Stathes Hadjiefthymiades, "Semantic Web Service Discovery: Methods, Algorithms and Tools," in *Semantic Web Services: Theory, Tools and Applications*, Jorge Cardoso, Ed.: IDEA Group Publishing, 2007.

[22] Fabio Casati, Ski Ilnicki, Li-jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan, "eFlow: a Platform for Developing and Managing Composite e-Services," Software Technology Laboratory, HP Laboratories, Palo Alto, CA, Technical Report HPL-2000-36, 2000.

[23] Maolin Tang and Lifeng Ai, "A hybrid genetic algorithm for the optimal constrained web service selection problem in web service composition," *Proceeding of the 2010 World Congress on Computational Intelligence*, July 2010, http://eprints.qut.edu.au/33293/1/c33293.pdf.

[24] Matthias Klusch, Benedikt Fries, and Katia Sycara, "Automated semantic web service discovery with OWLS-MX," *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pp. 915-922, May 2006.

[25] UDDI XML.org. (1993-2009) Online community for the Universal Description, Discovery and Integration OASIS Standard. [Online]. http://uddi.xml.org/

[26] Tim Finin et al. (2004) Swoogle Semantic Web Search. [Online]. http://swoogle.umbc.edu/

[27] Oracle. (2009) Oracle JDeveloper. [Online]. http://www.oracle.com/technology/products/jdev/index.html

[28] Oracle. (2009) Oracle SOA Suite. [Online]. http://www.oracle.com/technologies/soa/soa-suite.html

[29] Apache Software Foundation. (1999-2009) Apache Tomcat. [Online]. http://tomcat.apache.org/index.html

[30] Hewlett-Packard Development Company. (2009) Joseki - A SPARQL Server for Jena. [Online]. http://www.joseki.org/

[31] David Martin et al. (2004, November) OWL-S: Semantic Markup for Web Services. [Online]. http://www.w3.org/Submission/OWL-S/