A Multi-Layer and Multi-Perspective Approach to Compose Web Services

Djamal Benslimane[1], Zakaria Maamar[2], Yehia Taher[1],
Mohammed Lahkim[3], Marie Christine Fauvet[4], and Michael Mrissa[1]

[1]Claude Bernard Lyon 1 University, Lyon, France
[2]Zayed University, Dubai, United Arab Emirates
[3]King Saud University, Riyadh, Kingdom of Saudi Arabia
[4]Joseph Fourier University, Grenoble, France

Abstract

This paper presents a Web services composition approach that is built upon three layers known as component, community, and composite. The contribution of each layer towards this approach is assessed from two perspectives known as organization and management. Furthermore this paper discusses how Web services in a community are specialized into abstract and concrete. Interactions between abstract/-concrete Web services and composite Web services happen through a driver known as Open Service Connectivity. This driver permits first, binding any abstract Web service to any composite Web service and second, triggering any concrete Web service from any composite Web service.

Keywords. *Community, Composition, OSC, Web service.*

1. Introduction

Web services are now established as the technology of choice that makes applications interoperate. In addition businesses that deploy Web services can take advantage of their benefits in terms of *flexibility*, *scalability*, and *openness*. Flexibility means the ability of a business to adapt its core processes by selecting the appropriate operations that accommodate partners' requirements. Scalability means the capacity of a business to interact with a number of partners without having its core processes disrupted. Finally openness means the capacity of a business to privilege loosely-coupled solutions/open standards over tightly-coupled solutions/in-house standards.

Web services composition addresses the situation of users' requests that cannot be satisfied by any single, available Web service, whereas a composite Web service obtained by combining available Web services might be used. In an open environment like the Internet, a given functionality such as `BookOrdering` can be offered by several, independent Web services. Obviously, a client application should be able to change Web services with minor impact on the ongoing performance of the composite Web service that uses these Web services. Motives of changes include Web service's non-responsiveness to client requests or better arrangement with another, competitor Web service. However, changing Web services is tedious as the common practice is to make Web services bind to composite Web services in a hard-coded way. This is reported in various projects [2, 7, 10, 13]. In this paper we take Web services composition one step further by shedding the light on a new element that impacts the way Web services get bound during composition. We denote this new element by community and define it as a means to provide a common description of a desired functionality like `FlightBooking` without explicitly referring to any specific Web service like `EKFlightBooking` that implements this functionality.

In this paper we present a multi-layer and multi-perspective approach to compose Web services (Fig. 1). In this approach, composition spreads over three different layers, which we denote by *component*, *community*, and *composite*. In addition to these layers, composition is handled from two different perspectives, which we denote by *organization* and *management*. On one hand, organizing Web services means building a community that consists first, of an *abstract* Web service that identifies the common functionality of this community and second, of a dynamic set of *concrete* Web services that implement this functionality in different ways. On another hand, managing Web services means working out the computation modules that guide composition in terms of how Web services engage in community, how to monitor Web services' activities within a community, etc.

The research discussed in this paper strengthens the role of the community layer in developing adaptable composite Web services. This research also raises some questions that are tackled throughout this paper:

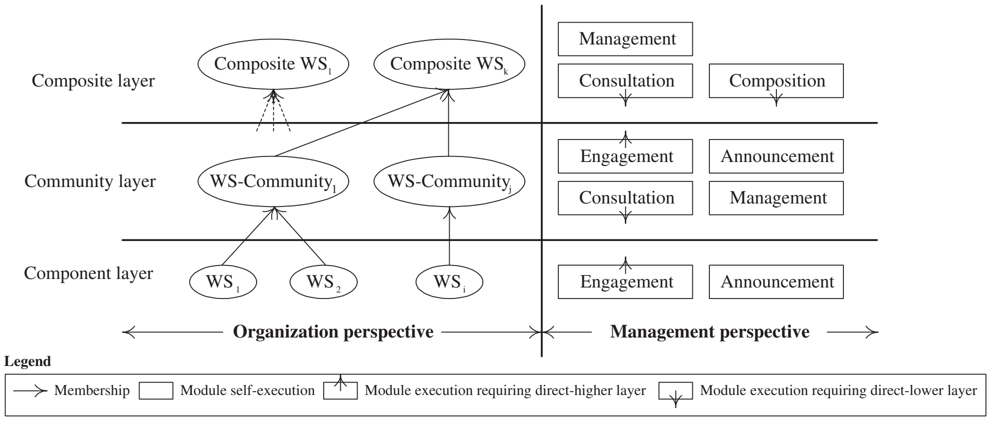

Figure 1. The proposed approach for Web services composition

1. How to characterize the component, community, and composite layers? And what types of interactions cross these layers according to the management and organization perspectives?
2. What are the factors that trigger the development/dismantlement of a community of Web services? And how much does this development/dismantlement affect the way composite Web services are structured and carried out?
3. How to classify Web services in a community into abstract and concrete types? What are the mechanisms that support mapping abstract Web services onto concrete Web services and *vice-versa*? How do abstract Web services bind to composite Web services? And how do composite Web services trigger concrete Web services through abstract Web services?
4. And how to specify the management operations per type of layer? How to keep track of these management operations? And how to embed these operations into modules of the management perspective?

The rest of this paper is organized as follows. Section 2 provides a macro-discussion on our Web services composition approach. A micro-discussion of this approach is given in Section 3 with emphasis on the role of communities of Web services. Section 4 discusses the way interactions happen between composite Web services and component Web services. Section 5 presents the implementation work. Section 6 concludes the paper.

2. The multi-layer/perspective approach

We discuss at the macro-level the component, community, and composite layers from organization and management perspectives. This discussion highlights also the cross-collaboration that occurs between the layers.

2.1. Layer description - organization perspective

The component layer contains Web services. Providers develop and describe Web services prior to informing them to potential users through UDDI registries. A Web service implements a functionality that permits satisfying users' needs. Some functionalities include `BookOrdering`, `WeatherForecast`, `AirfareQuote`, etc. A Web service's functionality can come along with non-functional details that usually revolve around the QoS notion.

The community layer is a means to gather Web services with the same functionality into groups known as communities. The component layer feeds the community layer with Web services. Although Web services belong to different providers, they can still offer the same functionality, regardless of how this functionality is defined, advertised, and performed by each Web service. We note that a community of Web services has a dynamic content: new Web services may join, other Web services may leave, some Web services may resume operation after suspension, etc. Conflicts in a community could arise, too. For example, a Web service does no longer reside in a community but its peers still believe it is in the community. Moreover, Web services do not always expose a cooperative behavior when they become members of a community. For example, they can announce misleading information such as lower execution-cost in order to enhance their participation opportunities in composite Web services. In [8], we discuss how a similar case affects the reputation level of Web services in a community.

The composite layer is motivated by the complex nature of users' needs, which sheds the light on the requirement of composing several Web services. The content of the community layer feeds the composite layer with the necessary component Web services. The feeding process consists of two steps. The first step identifies the communities of Web services that have the functionalities that permit satisfying a user's need. The identification is based on the unique functionality that labels each community. It is noted that labeling a community occurs through an abstract Web service that acts on behalf of this community. The second step identifies within the selected communities the Web services that implement the required functionalities. It is noted that implementing functionalities occurs through concrete Web services, which are identified upon recommendation of the abstract Web services of the selected communities.

2.2. Layer description - management perspective

The management perspective shows the different computation modules that intervene in each layer. Though some modules have similar labels in Fig. 1, their role differs according to the layer type. In Fig. 1, a specific representation formalism is used in the management perspective. A module is represented with a rectangle. A regular rectangle (i.e., no arrow) means that the execution of the module happens within the borders of a layer. A rectangle with an arrow pointing up means that the execution of the module requires elements from a direct-higher layer as shown by the organization perspective. Finally, a rectangle with an arrow pointing down means that the execution of the module requires elements from a direct-lower layer as shown by the organization perspective.

The component layer encompasses two modules: announcement and engagement. The announcement module is responsible for signing Web services up with UDDI registries. This permits informing providers of Web services about the existence of other Web services, so potential communities can be formed upon similarity assessment of Web services' functionalities [5]. The engagement module is responsible for framing the participation of Web services in communities. Some actions in the engagement module concern identifying of the appropriate community, adding/withdrawing requests of Web services to/from communities, and monitoring of Web services' activities within a community when they participate in compositions.

The community layer encompasses four modules: announcement, engagement, consultation, and management. The announcement module is responsible as described above. This permits to inform composite Web services about the existence of communities and their respective functionalities. The management module is responsible for updating the content of a community when it comes to adding/withdrawing new/existing Web services and checking Web services' credentials for security reasons. The consultation module supports browsing the UDDI registries where Web services are announced, so that these latter are invited to join in communities. Finally, the engagement module is responsible for framing the participation of communities (through Web services) in composite Web services. Some actions in the engagement module concern identifying Web services, adding/withdrawing requests of Web services to/from composite Web services, and monitoring the interactions with composite Web services during Web services selection, binding, and triggering.

The composite layer encompasses three modules: consultation, composition, and management. The management module is responsible for updating the content of a composite Web service when it comes to adding/withdrawing new/existing component Web services. We recall that a component Web service is selected out of a community. The consultation module supports browsing the UDDI registries where communities of Web services are announced, so that these latter are invited to be part of composite Web services. Finally, the composition module implements the specification of a composite Web service. This specification dictates at design-time multiple elements like execution order of component Web services, data dependencies between component Web services, and corrective strategies in case component Web services raise exceptions. At runtime, the composition specification is executed to identify and trigger component Web services, to oversee their execution, to coordinate their actions to avoid conflicts, just to cite a few. Different specification languages for Web services composition exist, e.g., the Web Services Choreography Description Language [6] and the Web Services Business Process Execution Language [1].

3. Community of Web services

The following is a micro-description of the community layer. The other two layers are to a certain extent commonly used in Web services projects.

3.1. Definitions

In Longman Dictionary, community is "*a group of people living together and/or united by shared interests, religion, nationality, etc*". In the field of Web services, Benatallah et al. define community as a collection of Web services with a common functionality, although these Web services have different non-functional properties like different providers and different QoS parameters [2]. Medjahed and Bouguettaya use community to cater for an ontological organization of Web services sharing the same domain of interest [9].

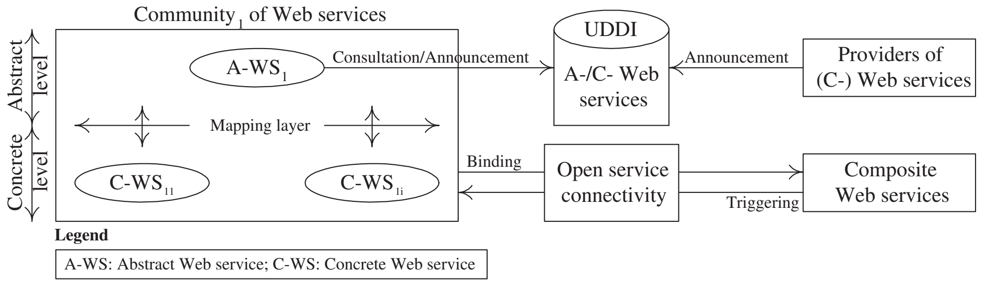
Community$_1$ of Web services
Abstract level
Concrete level
A-WS$_1$
Consultation/Announcement
Mapping layer
C-WS$_{11}$
C-WS$_{1i}$
UDDI
A-/C- Web services
Announcement
Providers of (C-) Web services
Binding
Open service connectivity
Triggering
Composite Web services
Legend
A-WS: Abstract Web service; C-WS: Concrete Web service

Figure 2. Operations in a Web services community

3.2. Role and management of a community

The development of a community is a designer initiative. The designer develops a Web service by defining its functionality, describing it with WSDL, and posting it on an UDDI registry. We qualify this Web service as abstract for three reasons: it leads a community, it does not implement the functionality it offers, and it does not participate in any composition. The engagement of an abstract Web service in composition occurs through other Web services, which we qualify as concrete. Providers of concrete Web services join the community that hosts the abstract Web service subject in order to offer the functionality of this abstract Web service. Assessing functionality similarity is discussed in our previous research [12]. Contrarily to an abstract Web service, a concrete Web service implements its functionality by defining and deploying the necessary code operations.

Fig. 2 illustrates the operations in a community of Web services. Our definition of community goes beyond gathering similar Web services in a community and considers a community as a means for providing a common description of a desired functionality (e.g., `FlightBooking`) without explicitly referring to any concrete Web service (e.g., `EKFlightBooking`) that implements this functionality. In a community, a mapping layer exists between the abstract and concrete levels. The role of this layer is to establish correspondences, according to specific ontologies [4], between (i) input and output arguments of and (ii) operations of the abstract Web service and all the concrete Web services that reside in the community. In addition, the mapping layer establishes how a concrete Web service implements the functionality of the abstract Web service. This implementation varies from one concrete Web service to another according to the identified argument and operation correspondences (Section 4.2). Indeed, a concrete Web service could rely on other concrete Web services that are located in other communities to complete the functionality. This might not be the case with another concrete Web service that implements the functionality itself (i.e., self-sufficient). Formally, we define a community C_i with the triple $C_i =< AWS_i, CWS_{(i,j)}, Mapping(AWS_i, CWS_{(i,j)}) >$ where $A.WS_i$ is the abstract Web service, $CWS_{(i,j)}$ is the set of concrete Web services, and $Mapping(AWS_i, CWS_{(i,j)})$ is the set of correspondences between the abstract Web service and all the concrete Web services in the community C_i.

When the functionality of an abstract Web service satisfies a user's needs as prescribed in the composite Web service specification, this abstract Web service agrees first, to bind to the composite Web service and second, to interact with all the concrete Web services that reside in its community. The objective of this interaction is to identify a concrete Web service that will handle the triggering requests that will originate from the composite Web service. In [8], we used the contract-net protocol [11] to identify concrete Web services. The binding and triggering between composite Web service and abstract/concrete Web service occur through a specific driver that we denote by *Open Service Connectivity* (OSC). For simplicity purposes we do not show in Fig. 2 how the designer of a composite Web service consults the UDDI registry so that the abstract Web services are identified. More details on the role of the OSC are given in Section 4.

Similar to our Web services classification into abstract and concrete, Bianchini et al. suggest a service ontology that helps organize Web services in three abstraction layers [3]: concrete Web services, abstract Web services, and subject categories. Concrete Web services are directly invocable. Each cluster of similar concrete Web services is associated with an abstract Web service that is not invocable. Finally, subject categories organize Web services into standard taxonomies and provide a topic-driven access to the underlying abstract Web services.

4. The open service connectivity driver

4.1. Role and offered functionalities

In Fig. 2, the role of the OSC is to achieve the binding of any abstract Web service to any composite Web ser-

Table 1. Open service connectivity's proposed functions

Function	Description
`Find`(*functionality*)	Search an UDDI registry for an abstract Web service on the basis of the required *functionality*.
`GetDetail`(*AWS*)	Request binding information on an abstract Web service using the output of `Find`(*functionality*).
`Bind`(*AWS*)	Establish a transparent binding with a concrete Web service using the details returned by `GetDetail`(*AWS*).
`GetListOfCWS`(*AWS*)	Request the set of concrete Web services in the community that an abstract Web service leads using the output of `Find`(*functionality*).
`GetDetail`(*CWS*)	Request binding information on a concrete Web service from the list returned by `GetListOfCWS`(*AWS*).
`Trigger`(*CWS*)	Establish a triggering request with a concrete Web service using the details returned by `GetDetail`(*CWS*).

vice and the triggering of any concrete Web service from any composite Web service. This happens regardless of the concrete Web service in a community that will perform the functionality that the composite Web service requires. The OSC translates composite Web services' and abstract/concrete Web services' queries into commands that both understand. Indeed both composite and abstract/concrete Web services need to be OSC compliant, i.e., they must be capable of issuing and responding to OSC commands. These commands are related to function calls, error codes, and data types. Monitoring these commands turns also out to be useful for auditing and billing purposes, as this permits knowing which function was executed, how much time the execution lasted, which exception was raised, etc. Table 1 lists some functions the OSC provides.

Prior to binding a composite Web service, an abstract Web service associates a logical name with the concrete Web service that is selected for participation in this composite Web service. Participation mechanisms are outside this paper's scope [7]. During binding, the abstract Web service submits two details to the composite Web service through the OSC. The first detail is the logical name of the concrete Web service. The use of logical names avoids referring to concrete Web service with hard-coded references and guarantees concrete Web services' anonymity if they wish so. The second detail concerns the operations of the abstract Web service, which correspond to the operations to identify at the level of concrete Web services. This permits including the invocation actions to these operations in the specification of the composite Web service. At run time, the concrete Web service receives commands from the composite Web service through its logical name and the OSC. These commands result in executing the operations of the concrete Web service. However, only the operations of the abstract Web service are reported in the specification of the composite Web service. As a result, functions for correspondences between concrete and abstract Web services are deemed appropriate. This is shown in the next section.

4.2. Abstract-Concrete correspondences

The abstract-concrete correspondence process supports the execution of commands that a composite Web service submits, through the OSC, to a specific concrete Web service. However all these commands are expressed according to the definition of an abstract Web service that represents the community. Thus, correspondence functions are required between the abstract and concrete Web services. These functions initially focus on the operations that implement the functionality of a Web service whether abstract or concrete. Let us assume two abstract and concrete Web services with different `operation` names. Once the operation's name difference is resolved, additional differences could arise and concern operations' input and output arguments in terms of number, name, type, structure, data unit, and constraints. Correspondences between abstract and concrete Web services are handled with *mapping rules*.

To illustrate the use of mapping rules, we assume the following fictive example. `FirstUrgentHelp` community gathers Web services dealing with emergency transportation functionality. This community has `FirstUrgentHelp` abstract Web service that offers `PickUpPatient()` abstract operation. This operation requires patient's first and last names, gender, and local address as input parameters. As output message, it returns a cost in Euros for transporting a patient. Signing `FirstCare` concrete Web service up in `FirstUrgentHelp` community requires defining mapping rules between its `SendUrgentVehicle()` operation and `PickUpPatient()` operation of `FirstUrgentHelp`. Some details on both operations are given in Fig. 3-(A/B).

(A)

Abstract Web service Interface
Operation:PickUpPatient
Input Message:
Fname: String
Lname: String
Sex: String
Address: String
Ouput Message:
EuroCost: float

(B)

Concrete Web service Interface
Operation:SendUrgentVehicle
Input Message:
Name: String
Gender: Integer
Street: String
City: String
Ouput Message
DollarCost: float

(C)

Mapping rules
Abstract to Concrete
Rule $_1$: Name =AdaptName(Fname,LName)
Rule $_2$: Gender =AdaptGender(Sex)
Rule $_3$: Street =AdaptStreet(Address)
Rule $_4$: City =AdaptCity(Address)

Concrete To Abstract
Rule $_5$: EuroCost =AdaptCost(DollarCost)

Figure 3. Abstract/Concrete Web services mappings

5. Implementation

In the following we discuss the implementation of each layer of the proposed approach for Web services composition. We selected bicycle competition as a running case. Each participant has a watch bracelet that submits various details to the control center including location, heartbeat pulse, blood pressure, etc. The bracelet has mainly two roles: warn the control center in case of health problems, and submit the geographic coordinates of the participant's location. Upon receiving a warning signal, the control center dispatches an ambulance to the scene after mapping the coordinates to a specific address that is known to drivers. To achieve its mission with success, the control center runs `Competition` composite Web service that uses two Web services along with their respective operations: `FirstCare` with `SendUrgentVehicle()` operation, and `FirstAmbulance` with `SendAmbulance()` operation. We recall that `FirstCare` Web service runs in `FirstUrgentHelp` community. Same comment is made on `FirstAmbulance` Web service, which is part of another community.

For the component layer, we implemented the different Web services using Java and Axis Java2WSDL utility to automatically generate WSDL descriptions from Java files. These descriptions are afterwards posted on an UDDI registry, which is deployed using jUDDI Registry Framework. Requests to Web services are directed using Apache SOAP (2.3). Here are details on each Web service:

- FirstCare Web service:
  - Operation: SendUrgentVehicle()
  - Inputs: Name, Gender, Street, City
  - Output: DollarCost
- FirstAmbulance Web service
  - Operation: SendAmbulance()
  - Inputs: PatientName, Gender, Location
  - Output: EuroCost

For the community layer, we developed `FirstUrgentHelp` community as reported earlier. It has an abstract WSDL-file and an XML description-file. The abstract WSDL-file describes a community's operations and input/output parameters and is either posted on an UDDI registry or manually downloaded. This file contains a binding section that points towards the XML description-file. This latter contains identifiers, access endpoint URLs, and non-functional characteristics of the provided Web services. This is illustrated in Listing 1. We also illustrate in Fig. 3-C some of the mapping rules we developed due to the heterogeneities between abstract and concrete Web services' operation and inputs/ouptputs.

```
<CommunityXMLFile>
  <ConcreteWS>
      <ID>SC1</ID>
          <QoSModel>
              <Price>P1</Price>
              <Reputation>R1</Reputation>
              <availability>A1</availability>
          </QoSModel>
      <URLadapter>URLAdapter1</URLAdapter>
  </ConcreteWS>
   ...
</CommunityXMLFile>
```

Listing 1. Community XML description file

For the composition layer, we developed the OSC driver as a Java^{TM} library that includes functions for selecting abstract and concrete Web services. A client for communities was also developed, which triggers the various functions of the OSC driver. In the current implementation, the client can perform the following operations: select an abstract Web service, list currently available concrete Web services, manually or automatically select a concrete Web service using some criteria, and invoke the selected Web service. The matchmaking of operations from abstract to concrete format is performed via auxiliary adapters, implemented as Web services, thus enabling the OSC driver to seamlessly communicate with concrete Web serviced via adapters' endpoints.

6. Conclusion

In this paper we presented a multi-layer and multi perspective approach for Web Services composition. This approach is built upon component, community, and composite layers. Each layer has a role in framing the progress of a composition scenario. This role is assessed from organization and management perspectives. Besides the approach, the paper highlighted the value-added of building Web services communities to composition. A community consists of an abstract Web service and a set of concrete Web services. To support composite Web services and component Web services interactions, we discussed the open service connectivity driver, which aims at making the process of binding Web services flexible at runtime.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, IBM Corporation and Microsoft Corporation, 2003.

[2] B. Benatallah, Q. Z. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1), January/February 2003.

[3] D. Bianchini, V. De Antonellis, and M. Melchiori. Capability Matching and Similarity Reasoning in Service Discovery. In *Proceedings of The Open Interop Workshop on Enterprise Modelling and Ontologies for Interoperability (EMOI-INTEROP'2005) held in conjunction with CAiSE'2005 conference*, Porto, Portugal, 2005.

[4] C. Bussler, D. Fensel, and A. Maedche. A Conceptual Architecture for Semantic Web Enabled Web Services. *SIGMOD Record*, 31(4), 2002.

[5] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity Search for Web Services. In *Proceedings of The 30th International Conference on Very Large Data Bases (VLDB'2004)*, Toronto, Canada, 2004.

[6] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web Services Choreography Description Language, Version 1.0. W3 Candidate Recommendation, 2005.

[7] Z. Maamar, S. Kouadri Mostéfaoui, and H. Yahyaoui. Towards an Agent-based and Context-oriented Approach for Web Services Composition. *IEEE Transactions on Knowledge and Data Engineering*, 17(5), May 2005.

[8] Z. Maamar, M. Lahkim, D. Benslimane, and Ph. Thiran. Towards An Approach for Specifying and Managing Communities of Web Services. Technical report, Zayed University, King Saud University, Claude Bernard Lyon 1 University, and Namur University, January 2006.

[9] B. Medjahed and A. Bouguettaya. A Dynamic Foundational Architecture for Semantic Web Services. *Distributed and Parallel Databases, Kluwer Academic Publishers*, 17(2), March 2005.

[10] Q. Z. Sheng, B. Benatallah, Z. Maamar, M. Dumas, and A. H. H. Ngu. Enabling Personalized Composition and Adaptive Provisioning of Web Services. In *Proceedings of The 16th International Conference on Advanced Information Systems (CAiSE'2004)*, Riga, Latvia, 2004.

[11] R. Smith. The contract Net Protocol: High Level Communication and Control in Distributed Problem Solver. *IEEE Transactions on Computers*, 29, 1980.

[12] Y. Taher, D. Benslimane, M.-C. Fauvet, and Z. Maamar. Towards an Approach for Web Services Substitution. In *Proceedings of The 10th International Database Engineering & Applications Symposium (IDEAS'2006)*, Delhi, India, 2006.

[13] Y. Wang and E. Stroulia. Semantic Structure Matching for Assessing Web-Service Similarity. In *Proceedings of The 1st International Conference on Service-Oriented Computing (ICSOC'2003)*, Trento, Italy, 2003.