Algorithm to Solve Web Service Complex Request Using Automatic Composition of Semantic Web Service

Brahim Batouche, Yannick Naudet,
Public Research Center Henri Tudor,
Luxembourg
{brahim.batouche, yannick.naudet}@tudor.lu

Frédéric Guinand
University of Le Havre,
France
frederic.guinand@univ-lehavre.fr

Abstract -- **Automatic composition of web services supports the solving of complex user request. The set of possible solutions can be represented by a graph, modeling the composition. Usually, this kind of approach is highly simplified by considering only sequences of services. This paper proposes an algorithm for automatic semantic web services composition, which generates a graph taking into account any composition structure. The request resolution process identifies possible composition structures and selects relevant services based on their semantic description. The resulted composition graph answers all requested functionality with coherent composition structures.**

Keywords - semantic web service; composition graph; automatic compositio; web service composition structure.

I. INTRODUCTION

Web services composition is a classical approach to answer complex queries that cannot be solved with one single service. Answering to such requests requires several steps: (1) finding suitable services; (2) finding how they can be composed together to answer the request; (3) create the corresponding composite service; (4) invocate it; (5) maintain it so that it can be reused later. The structure of the composite service depends obviously on the request, but also of the available services.

Composing services can be useful in many different domains, such as, e.g., tourism, transport, multimedia, etc. Some of them involve a dynamic environment where events at any time can affect previously computed compositions answering a request. A fundamental issue is then how to repair failures in a composite service execution, which can occur in dynamic environments. A typical example is when one of the services involved in the composition is faulty or can no more be executed. This fail is translated to a complex request and then use our algorithm to find another composition alternative.

In this paper we propose an algorithm for automatically finding all candidate compositions answering a complex request, without a priori knowledge of the composition structure. When the request does not formally specify any chaining between the requested elements, the algorithm must found suitable composition structures based on the available services. This problem is not trivial because there are many possible services combinations and composition structures. To determine the composition structure we base in the existing functionalities, which are automatically determined because the available services are supposed described semantically by OWL-S [1].

In section 2, we present related works. In Section 3, we first formalize the problem and detail it. Section 4 presents the composition structures and their semantics. In Section 5, we present our algorithm for automatic construction of a composition graph. In Section 6, we provide the experimentation results, and finally, we conclude in Section 7.

II. RELATED WORKS

Solving a complex request by services composition in dynamic environments, can be tackled by different approach.

The algorithm presented in [2] builds a composition graph answering a request. The algorithm identifies first the input and output of the request and search for a matching service. When none can be found a service having a matching output is selected and recursively, subsequent services having output matching with the input of the latter service and input matching with the request input are sought. The algorithm ends when a sequence of services starting with the request input and ending with its output is found, or when the set of available services has been searched. The provided composition graph does not allow the direct invocation of services. Also, it is still limited to sequences structure of composition.

In [3], the flooding algorithm is used. Once again, the proposed approach is limited to services sequences and does not allow composition execution.

In [3], an architecture for automatic web service composition is proposed. This architecture allows fast composition of OWL-S service. However, while authors

Copyright (c) IARIA, 2010 ISBN: 978-1-61208-108-3

provide interesting ideas for the design of the composite service and automating service invocation, they only consider sequences of services. Kazhamiakin and Pistore [5] proposed a model to answer a request using the composition of web services, the model supposes the available services are described by BPEL-WS. A request requires much functionality, which are identified and then used in a finite state-machine, implementing the composition structure. The state-machine provided does not allow composition execution.

In [5], a multi-agent system is used to automate the composition of services. The agents collaborate to provide the composition needed, an agent is presented by the OWL-S service and its functional parameters describe the agent role. With this system, we can consider the compositional structures: sequence, parallel and conditional. The conditional structure concerns only the functional parameter of the service.

According to the state of the art, many methods of automatic composition focus to find the needed functionalities to answer a request and there order but do not give the link to execute them. So, they usually consider only the sequence structure. To exceed these limits, we automate the detection of the composition structure needed and the selection of the services requested. To select automatically the services, we use the I/O dependence basing in the matching function. The matching function uses only the IO parameter [6] or uses the IOPE [8], which provides more accurate results.

III. Problem Formalization

A typical example of complex user request, which we will use as a basis to present our approach, is the following: “I want travel from City A to City B, reserve several hotel rooms in destination city where each book is billed separately, rent a car for six people, have the weather and plan for the destination city”. Such request needs first to be formalized in a machine processable way.

A. Request Formalization

A complex request is a combination of more focused or atomic sub-requests, which concerns each a service or functionality. We write: $R = F_R = \{F_{r_i}\}$, where F_{r_i} is a functionality requested. Our example requires four functionalities: transport, booking hotel, rent a car, city information. Each functionality has input/ output ($I_{F_{r_i}}/O_{F_{r_i}}$). Formally, we write a request as a triple: $R = <I_R, O_R, C>$, where $I_R = \bigcup_{i=1}^{card(F_R)} I_{F_{r_i}} = (i_1, \dots, i_k)^T$is the set of inputs, $O_R = \bigcup_{i=1}^{card(F_R)} O_{F_{r_i}} = (o_1, \dots, o_l)^T$ is the set of outputs, and $C = (c_1, \dots, c_m)^T$ is the set of conditions or constraints (related to data, service or composition). Conditions differ from constraints in that they must be verified to instance the input parameter of service, but the constraints to filter the set of available services, data provided by services or composition paths.

All the sets elements are URIs of concepts defined in ontologies. While I and O correspond to functional parameters which describe a domain ontology, *C* concerns both functional and non functional parameters. Quality of Service is an example of such parameter, as well as price, cardinality of some services output, etc. The non functional parameters values are found in the service description. The functional parameters values are identified after execution the informative service, which provide information without modify its source database.

- From the fail execution of service to request

The execution of composition can fail if one of its services fails. The fail can then be translated to a new request, which depends on the functionalities realized at the moment of fail. These functionalities correspond to a set of Terminated Input / Output (*TI/TO*). The new request formalized as $R = < \overline{I_R \cap TI}, \overline{O_R \cap TO}, C >$.

To configure the original composition graph (or find another alternative) we use reclusively our algorithm with the new request.

B. Composition Graph Formalization

The composition of services presents a set of functionalities and there structures, but usually does not give the execution possibility, e.g. [5] [9]. This brings us to define two types of composition graph.

Definition: ***The executable composition*** *graph allows the service execution, thus making the composition executable.* ***The abstract composition*** *graph represents only the structure of the composition and cannot be executed.*

This definition based in the existing (or not) the link to invocate the service. But [10] defines the abstract composition according to the existing (or not) the sub-service I/O of the composition. According to our definitions, an executable composition graph is an abstract composition graph whose nodes integrate services identifiers, (URIs), instead of input / output parameters only. Abstract graphs represent only composition of functionalities fitting a request, while

executable ones describe actual services chaining. We focus here on finding automatically suitable services composition structures that we model with an executable graph.

The executable composition graph corresponding to a complex request; represents a set of services paths which constitute possible answers. It is formulated as: $G = <N, V>$, where, V is a set of directed arcs and N is a set of nodes. We distinguish four types of nodes, *IS, AS, DA and SW,* where *IS* is an informative service, *AS* is an active service, which provides an action and modify its source database; *DA* is the data (information) provided by an *IS;* and *SW* is a switch node that represents a conditional structure, specifying a condition formula.

We define a node as $n = < NT, id, URI_S, I_s, O_s, URI_{DATA} >$, where NT is the node type, id is the identifier of starting parallel structure node ($id = \emptyset$ if n does not belong to a parallel structure), URI_S is the URI of OWL-S service ($URI_S = \emptyset$ if $NT = DA\ or\ SW$), I_s/O_s are respectively the Input and output of the service, they are defined from URI_S, and URI_{DATA} is the URI of data ($URI_{DATA} = \emptyset$ if $NT = IS, AS\ or\ SW$). The special node switch $n_{SW} =< NT = SW, c_i, LF >$, where c_i is the condition provided by the request and it is verified by the node, and LF is the linked node because the multiple paths in the graph can meet in one SW node, then a SW node embeds a hash function recording authorized successors of nodes. For this reason, the SW node is a kind of meta-node containing several nodes.

We add the node SN and EN which respectively starting and ending the composite graph, $SN = < O = I_R >$, which its output corresponds to the request input, and $EN = < \emptyset >$.

The functional parameters of the answer composite of services match with the functional parameters of the request. So, the non functional parameters values are calculated according to the parameter type and the composition structure used, for example, see [11] .

IV. STRUCTURES OF WEB SERVICE COMPOSITIONS

Existing web services languages supporting composition model different structures in different ways. Taking the most commonly known, we observed the following. The structures modeled by OWL-S are: "sequence", "any-order", "if-then-else", "choice", "while", "until", "split" and "split-joint". Differently, BPEL4WS [12] uses: "sequence", "switch" "while" "Pick" and "flow". A mapping between the two representations involves three operators: equivalence (e.g., if-then-else is equivalent to switch; choice is equivalent to pick); composition (e.g., the flow structure in BPEL4WS can be decomposed into two structures of OWL-S: split and split-joint); and identity (for constructs that cannot be realized with structures of the other representation, e.g., any-order is not identity (see Section 4.2)). In order to insure interoperability with the different representations and keeping a generic approach, we focus on elementary structures (sequence, if-then-else, split, split-joint), from which many others can be modeled.

A. *Composition Structures Illustration*

A composition may comprise several different structures, which can themselves contain combinations of structures. A tree representation helps understanding and visualizing the composition: the leaves are services; the nodes and the root are the compositions structures. The path corresponds to read of composition tree which follow a prefixed depth approach. The Figure 1 shows for our example the composition tree and the corresponding composition flow. The used services are: available train (AT), available flight (AF), book train (BT), book flight (BF), available hotel (AH), book hotel (BH), available rentals car (ARC), rent car (RC), plan touristic map (PT), city weather (CW).

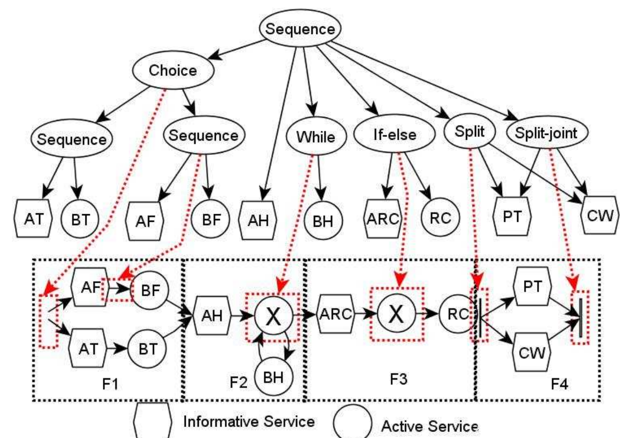

Figure 1: Service composition illustrated by tree and flow

B. *Characteristic of Composition Structures*

In the following, we detail the characteristics of structures we retained and explain how they identified from the request.

Copyright (c) IARIA, 2010 ISBN: 978-1-61208-108-3

Sequence "$\longrightarrow$": This structure defines an order between services. The order can be detected directly or indirectly. There are two ways to detect the order directly: (1) - Checking the match between services IOPE; (2) – Checking the priority between the services answering the question: *which service cancels the other when it is cancelled?*
Since the order operation is transitive: $(A \rightarrow B) \wedge (B \rightarrow C) \Rightarrow (A \rightarrow C)$. To detect indirectly the order between A and C. We base on the order of services (e.g., B) which have the order with A, C.
Choice "+": (or or-split): This structure represents a choice between several services that have a same functionality. $Choice(A, B_1, B_2, \ldots, B_k) \equiv (A \rightarrow B_1) \vee (A \rightarrow B_2) \vee \ldots \vee (A \rightarrow B_k)$, knowing that service "A" precedes services B_i and the services B_i have not the different functionality.
Any-Order "$\odot$": This structure is not elementary and represents a random invocation of services. This structure can be expressed using choice and sequence structures: $A \odot B \equiv (A \longrightarrow B) + (B \longrightarrow A)$. Therefore this structure is replicable.
If-then-else "$\otimes_c$": This structure checks a condition of request to instance the functional parameter of the service. The structure follows a service if ones of its parameter (functional/ non functional) correspond to a condition.
Split "$\vdash$": This structure indicates a simultaneous start of multiple services (or services chains). Services that can be parallelized have the same predecessor and provide different types of outputs. Each service starts a new sub-path in the composition. All services chains starting at a split will be executed in parallel and ended with a split-joint. $Split(A, B_1, B_2, \ldots, B_k) \equiv (A \rightarrow B_1) \wedge (A \rightarrow B_2) \wedge \ldots \wedge (A \rightarrow B_k)$.
Split-joint "$\dashv$": This structure ends a parallel structure, where the sub-composition paths belong to a same "split". The last services B_i in parallel chains have the same successor A. $split - Joint\ (B_1, B_2, \ldots, B_k, A) \equiv (B_1 \rightarrow A) \wedge (B_2 \rightarrow A) \wedge \ldots \wedge (B_k \rightarrow A)$, where services "$B_i$" end the parallel sub-composition paths. It is possible that all services chains in a same "split" do not end in the same "split-joint".
While "$\circledast_c$" and *until* "$\circledcirc_c$": These structures are not elementary and used for iterative service invocation. They can be constructed with if-then-else and sequence structures: $\circledast_c\,(A) = \otimes_c \longrightarrow A \wedge A \longrightarrow \otimes_c$ and $\circledcirc_c\,(A) = A \longrightarrow \otimes_c \wedge \otimes_c \longrightarrow A$.

The compositional structures (St) are illustrated in the graph by arc or node, $St = < N, V >$ (see Figure 1). The structures while and until $= \{SW, V\}$, the structure if-then-else$= \{SW, \emptyset\}$ and the structure sequence, choice, split and split-joint$= \{\emptyset, V\}$, where V is respectively a sequence-arc, set of sequence-arc, split-arc and split-joint-arc. This gives to distinct three types of arc $\{\longrightarrow, \vdash, \dashv\}$. An arc is defined by its type, departure node and destination node.

V. ALGORITHM GENERATING THE COMPOSITION GRAPH

Our algorithm processes progressively the request to build the executable composition graph. In the following, we define our terminology.

We name in the graph *current layer* l_k, the set of nodes in the graph having a same depth level, currently being processed: $l_k=\{n_i\}$. Initially $l_k=\{SN\}$. One step of the algorithm corresponds to full covers of l_k. The node of l_k being processed named *current node.*
The temporary buffer is used to store the set of nodes following the current node, and not preceding *EN*. When precede *EN* are placed directly in the *end layer* of the graph.

The algorithm is illustrated in Figure 2. From a request, it fills the current layer and processes it. For each node in the current layer, selects the next services according to their matching with functionalities F_R. The set of nodes created from next services is first put in the temporary buffer, which is later put in the next layer. When the current node output matches with one of the F_{r_i}outputs, the algorithm carries on with next node in current layer. That has the input of a F_{r_i}not yet covered. Otherwise, services having inputs matching the current node output are selected. Corresponding nodes are created after checking does not already exist in the set of nodes *N* in *G*.

An arc-sequence is created between the current node and the next nodes. When a selected service is an IS, it is invoked to obtain the information it provides before creating the arc. When the data are obtained, the algorithm creates an arc sequence between the node of the service and data nodes created, then it replaces the service node by the set of data nodes.
When a next node has been newly created, the algorithm checks the existence of a condition. The next node contains a condition if the output of the service it represents corresponds to one of request conditions. In

Copyright (c) IARIA, 2010 ISBN: 978-1-61208-108-3

this case we create a SW-node and linked to the node by an arc-sequence. The node following the SW-node is then selected according to the first node output.
In case all F_R have been covered, the next node is affected to the *end layer.* Otherwise, it is put into the temporary buffer, and later to the next layer.

The checking of split-structures is performed when the temporary buffer is full, containing all the nodes matching the current node. The checking of split-joint-structure is performed when the next node is selected. Therefore, the algorithm checks split-joint structure before the split structure.

The process checks the existence of a split-joint structure starting from next node. If it is selected from *N,* then it is possible to find a node which can precede the next node. In this case a complete check is performed, otherwise only a partial check is necessary. The complete check considers all nodes of the current layer. The partial check considers a current node and current layer nodes which have not been yet processed.

The algorithm creates a split-joint-arc when the split-joint is verified, i.e., the follow conditions are verified:
-The starting nodes of the split-joint-arc have a same split-structure, i.e., they contain the same identifier of split structure $id(n')$, where n' is the node starting the parallel structure.
-The nodes have the same succeeding node n^+, where n^+ ends the parallel structure.

$\forall\, n_i, n_j\ prcced\ n^+, \boldsymbol{if}\ (n_i, n_j)\text{contains}\left(id(n')\right)\ \boldsymbol{then}$
$CreatArcSplitJoint\,(n_i, n_j, n^+), n_i.delet\left(id(n')\right), n_j.delet\left(id(n^*)\right)$

Concatenation of parallel structures is possible. When nodes of same split-structure don't regroup in a same split-joint structure, the node $n^+$is included in the structure split, so it can be grouped with the remaining nodes. $\exists\, n_i.\text{contains}\left(id(n')\right), n_i \in l_k : n^+.add\left(id(n')\right)$.

The checking for the existence of a split structure is performed between the current node and the nodes in the temporary buffer. If these nodes have different functionalities i.e., different output, then we create a split-arc and add the identifier of the split structure $id(n')$ to these nodes. $\forall\, n_i, n_j follow\ n', \boldsymbol{if}\ M\left(out_{n_i}, out_{n_j}\right) > \varepsilon\ \boldsymbol{then}$
$n_j.\text{add}\left(id(n')\right), n_i.\text{add}\left(id(n')\right), CreatArcSplit(n', n_i, n_j)$
When a node n_k follows the node n_j which appears in parallel structure, $n_j\text{contains}\left(id(n')\right)$, then we affect n_k to this structure, $n_k.add\left(id(n')\right), n_j.delet\left(id(n')\right)$.

When all nodes of the current layer are processed, the next layer becomes the current layer and so on until the next layer is empty. The algorithm terminates when this state is reached.

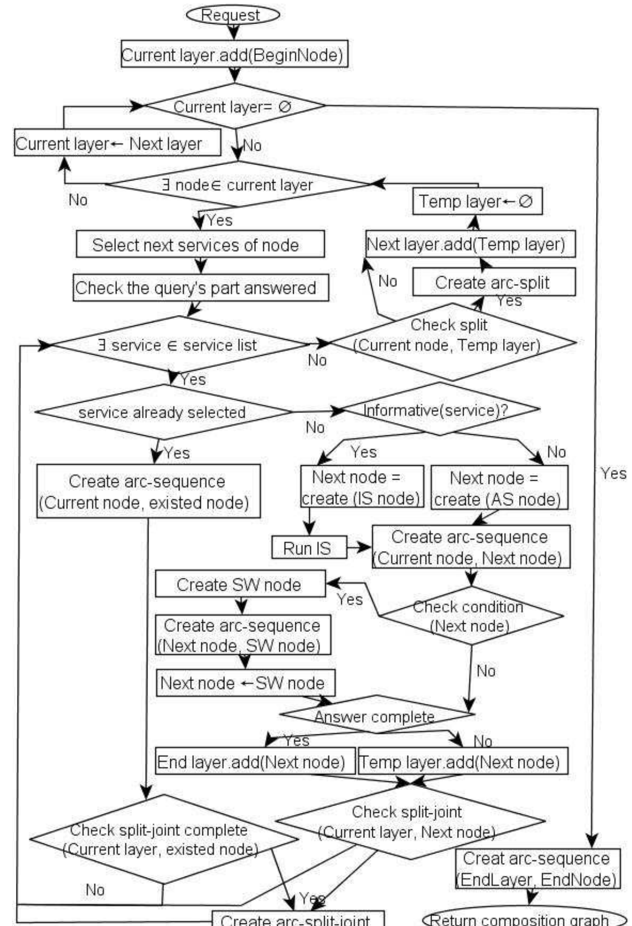

Figure 2: Algorithm of solving complex request

Finally, the complexity of each step of the algorithm graph construction composition is about $O\ (|l_k|.|S|\ )$, where S is the set of selected service. Since, the algorithm is based in the flooding algorithm. To assure the process logic, we check the composition structures according to the flooding algorithm step.

VI. EXPERIMENTATION AND RESULTS

We have tested our example request on a base, containing the services OWL-S descriptions and varying all the functionality needed in the request.

After running the algorithm, we verify: - the service composite answering a request has all requested functionalities, - its internal composition structure is coherent, i.e., there is no false detection of structures.
Different APIs were used Jena [13], SPARQL [14]; to check the data constraint and the conditions, OWL-S API [15], to check the service constraints, and Pellet [16], to

Copyright (c) IARIA, 2010 ISBN: 978-1-61208-108-3

check the matching level between services I/O and request I/O.

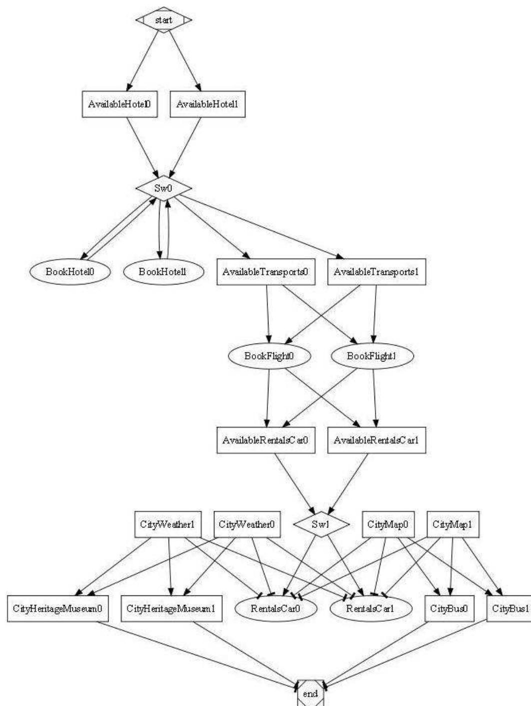

Figure 3: Resulted executable composition graph

The Figure 3 illustrates the composition graph given by the algorithm. The composition path is semantically correct because it contains all requested functionalities: transport, booking hotel, etc. And the composition structures used are coherent with the used services. E.g., choice: between the service “Available Hotel 0” and “Available Hotel 1”. Sequence: between “Book Flight” and “Available Rentals Car”. While: the node *sw0* checks the number of booking hotel. If the condition is true then another booking is made; else the loop is left. If-then else: the node *sw1* checks the number of car rented according to the type of car provided, if the available rental car does not take six people then rent two cars. Split/Split-joint: the service “City Heritage Museum” and “City Bus” will be executed in parallel.

VII. CONCLUSION AND PERSPECTIVES

In this paper, we have proposed an algorithm for multi-structure web services composition. It allows answering a user request by composing available matching services using all possible composition structures.

The composition graph provided by the algorithm will mainly be used as input for giving a search space authorized to optimize the composition of services. Additionally, we have also shown how to deal with composition execution failures (in this case, the composition graph can be adapted).

Finally, the solutions to a request contained in the composition graph can be formalized using classical languages like, e.g., BPEL-WS, OWL-S, etc., and stored in the services base for re-use.

In future works, we consider all ways to detect a sequence between services and we integrate the precondition/effects to calculate a level of matching between the services and request.

REFERENCES

[1] D Martin, et al.: OWL-S: Semantic Markup for Web Services. *W3C Member Submission*, 22, 2004.

[2] G. Silva, F. Pires, and V. Sinderen. An Algorithm for Automatic Service Composition, *1 st International Workshop on Architectures, Concepts and Technologies for service Oriented Computing*. pp. 65-74, Barcelona Spain. July 2007.

[3] S. Oh, B. On, E.J. Larson, and D. Lee. BF*: Web Services Discovery and Composition as Graph Search Problem, 6-8, *e-Technology, e-Commerce, and e-Services, IEEE International Conference on*, 784-786, 2005.

[4] K. Matthias and G. Andreas, Semantic web service composition planning with OWLS-XPlan, *In Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web, pp. 55-62, 2005.*

[5] *R. Kazhamiakin* and M. Pistore, A Parametric Communication Model for the Verification of B*PEL4WS* Compositions, *Formal Techniques for Computer Systems and Business Processes,* 318-332, Trento, Italy. 2005.

[6] D. Pellier and H. Fiorino. Un modèle de composition automatique et distribuée de services web par planification, *Revue d'Intelligence Artificielle ,v23,13-46, 2009.*

[7] *M. Klusch, B. Fries, M. Khalid, and K. Sycara, OWLS-MX: Hybrid OWL-S Service Matchmaking, In Proceedings of 1st Intl. AAAI Fall Symposium on Agents and the Semantic Web*. 2005.

[8] A.B. Bener, V. Ozadali, and E.S.Ilhan. Semantic matchmaker with precondition and effect matching using SWRL. *Expert Systems with Applications*, 36, 9371-9377, 2009.

[9] S.V. Hashemian, and F. Mavaddat. A Graph-Based Approach to Web Services Composition. *Proceedings of Symposium on Applications and the Internet*, 183-189, 2005.

[10] M. Mihhail, M. Riina, and T. Enn. Compositional Logical Semantics for Business Process Languages. Pro of *ICIW*. 2007.

[11] C. Wan, C. Ullrich, L. Chen, R. Huang, J. Luo, and Z. Shi. On Solving QoS-Aware Service Selection Problem with Service Composition, *Grid and Cooperative Computing*, 2008.

[12] T. Andrews, et al: ‘Business Process Execution Language for Web Services Version 1.1’, IBM, May, 2003.

[13] http://jena.sourceforge.net/ (11/03/2010)

[14] http://www.w3.org/TR/rdf-sparql-query/ (11/03/2010)

[15] http://www.mindswap.org/2004/owl-s/api/ (11/03/2010)

[16] http://www.mindswap.org/2003/pellet/ (11/03/2010)

Copyright (c) IARIA, 2010
ISBN: 978-1-61208-108-3