A Semantic Web Service Discovery and Composition Prototype Framework Using Production Rules

Georgios Meditskos and Nick Bassiliades

Department of Informatics
Aristotle University of Thessaloniki, Greece
{gmeditsk, nbassili}@csd.auth.gr

Abstract. The full realization of the semantic Web services demands efficient algorithms able to perform the procedures of service discovery, composition and invocation. In this paper, we present ProSeDisCo, our approach for developing a semantic Web service discovery and composition framework on top of the CLIPS rule-based system. More specifically, we describe our methodology of utilizing production rules over Web services semantic descriptions expressed in the OWL-S ontology. The purpose of these rules is to discover and create a Web service choreography that matches users' input and output requirements by utilizing a rule-based OWL reasoning engine in order to match semantically the requirements imposed by the users and the advertisements of the Web services.

Keywords: Semantic Web Services, OWL-S, Production Rules, OWL Reasoning.

1 Introduction

The advent of Web services is a proof that nowadays the need for communication among loosely coupled distributed systems is bigger than ever. Web services offer a well-defined interface through which other programs may interact by sending messages based on Internet protocols and Web standards. They may also be combined in order to achieve a complex service whose functionality cannot be achieved by a single one, a procedure that is called *service composition*. The description of a service interface is based on the Web Service Description Language (WSDL [23]) that describes the syntax of the input and output messages using XML, as well as other details needed for the invocation of the service. The communication is based on the Simple Object Access Protocol (SOAP [22]), an XML-based framework that provides a message construct that can be exchanged over a variety of underlying protocols.

Obviously, the procedures of discovery, composition and invocation of Web services are heavily based on human intervention. The XML representation of services' characteristics guarantees syntactic interoperability while it is unable to capture the semantics of the information. Programmers need to manually code any interaction procedure with a single or among many services and take care of potentially future modifications, resulting in a time consuming and costing process. In

B2B and e-commerce environments, quick and automated procedures among partners are of great importance and even if the technology of Web services is a step towards the automation of business interactions, it cannot serve the full potentials. Web services do not offer semantic description of their capabilities in order to enable the automated discovery and use by other programs, restricting their application only to static and predefined procedures defined by programmers.

Semantic Web initiative tries to solve such problems related to knowledge representation by suggesting standards, tools and languages for information annotation. Thus, the combination of Semantic Web techniques with Web services seems the best way to give the appropriate semantic notion to Web services in order to achieve the desirable level of automation. This combination has lead to the notion of *semantic Web services*. This term refers to existing Web services whose descriptions have been augmented with Semantic Web annotations techniques in order to allow automatic discovery, composition and invocation. Towards this need, many languages and frameworks have been proposed, such as OWL-S [14], WSMO [25] or WSDL-S [24], following different approaches [3] [16] [19].

In this paper, we present ProSeDisCo (**Pro**duction rules for **S**ervices **Dis**covery and **Co**mposition), a framework that supports Web service discovery and composition based on OWL-S descriptions. We have chosen OWL-S because we believe that it is a more mature approach (it has been under development since early 2001) and it has been used by a substantial number of research efforts. However, we do not state that it is the only standard that should be followed in the domain of semantic Web services. More efforts need to be done from any related initiative for the proposal of more complete frameworks, able to represent any real use case.

The development of a framework able to discover and compose Web services should have the ability to process and "understand" the semantic descriptions that follow these services. Since OWL-S is an ontology defined in the OWL [13] language, the underlying reasoning system should be able to draw the right conclusions that stem from the formal semantics of the OWL language in order to enable the semantic handling of Web services' characteristics. In our framework, we use a reasoning system that we have developed, namely O-DEVICE [11] [12], able to infer over OWL ontologies following an object-oriented approach.

ProSeDisCo is based on the utilization of appropriate rules over the object-oriented knowledge base that has been created by importing the semantic descriptions of services into the O-DEVICE system. These rules traverse the generated schema and select the appropriate services that meet users' needs in order to create a service composition plan (orchestration) that satisfies user's initial request.

The rest of the paper is organized as follows: in section 2 we give a short description about the underlying reasoning engine in order to make aware the reader about the general principles of the reasoning procedure and the resulting object-oriented representation of the information. In section 3 we present the main architecture of ProSeDisCo. In section 4 we describe the transformation procedure of users' requests into rules and objects of the knowledge base in order to facilitate the composition procedure. In section 5 we describe the principles of the Web service discovery procedure while in section 6 we present the composition algorithm. Section 7 gives a simple example of generating a composition plan. In section 8 we present related work and finally, in section 9, we conclude our work.

2 The O-DEVICE Reasoning System

O-DEVICE is a rule-based reasoning engine that handles OWL ontologies by performing a transformation of the information into an object-oriented environment. It is built over an existing rule system, namely CLIPS [4], and uses an embedded object-oriented language (COOL) that allows the manipulation of the object-oriented schema of the knowledge base by rules. O-DEVICE is an extension of a previously developed reasoning system for RDF [18] metadata, namely R-DEVICE [1].

O-DEVICE is a reasoning system disengaged from the triple-based functionality. Instead, we use triples in order to build an object-oriented schema of the ontologies by transforming them into classes and objects. TBOX and ABOX reasoning is performed via object-oriented production rules that implement a number of entailments. The TBOX inference rules alter the schema of the knowledge base in order to reflect the formal semantics of the OWL language, whereas the ABOX rules are used in order to perform classification (based on sufficient conditions) and Skolemization (based on existential quantifiers).

By importing the OWL-S ontology into O-DEVICE, the appropriate classes, slots and objects are created following the object-oriented principles. For every service profile that is submitted to the system, the corresponding object of the *Profile* class is generated, having encapsulated the input and output values into its *hasInput* and *hasOutput* slots. Thus, every service profile is treated as a single object and its input/output values are accessed directly by sending messages to the corresponding object. We do not elaborate further on our reasoning system since it is out of the scope of this paper. Further details concerning the reasoning method and technical characteristics can be found in [12].

3 Framework Architecture

The framework defines three types of users, i.e. administrators, providers and clients, each of which is eligible to perform a specific set of actions. Fig. 1 depicts the main architecture.

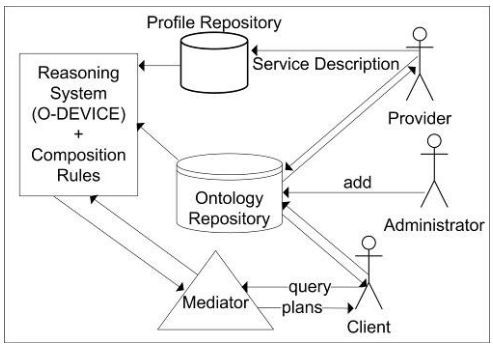

Fig. 1. Framework Architecture.

Providers: They are users that submit their service descriptions in the framework. More precisely, they are responsible for annotating their services based on the available ontology information of the *Ontology Repository* that the framework provides. They do not have the authority to add or alter the already existing ontology hierarchy, preventing in that way ontology inconsistencies. Each service description is submitted by uploading the corresponding service profile to the *Profile Repository*.

The *Profile Repository* stores service descriptions expressed in the OWL-S ontology. For the annotation of Web services, OWL-S defines four upper ontologies: *Service*, *Service Profile*, *Service Model* and *Service Grounding*. The *Service* ontology serves as an upper ontology of the other three and contains references to them. The *Service Profile* contains functional and non-functional properties for a service. The *Service Model* contains information about how the service works and the *Service Grounding* specifies details about how an agent can invoke the Web service, such as the protocol, the message formats and addresses. In our effort, we currently make use only of the input and output descriptions of service profiles during the discovery and composition procedures.

Administrators: They are responsible for the normal operation of the system. Their main responsibility is to enrich the framework with ontology information in order to enable providers to annotate semantically their services in a more accurate way. The providers may also contact the administrators in order to inform them about their needs when they are unable to describe correctly their services based only on the available ontology information. Notice that administrators are not responsible for a correct service description.

The administrators are also responsible for the removal of existing service descriptions due to unavailability or after a provider's request. Furthermore, statistical information about service requests are hold that can be used in order to detect services description with potential mistakes in their definitions. These services are unlike to be discovered due to description inconsistencies of their inputs or outputs.

Clients: Their role is limited only to the conduction of queries in order to retrieve services that meet their needs. Clients are responsible to define queries that refer to concepts and roles of the *Ontology Repository*, in order to be able for the system to process them appropriately.

4 Internal Representation of Users' Requirements

The ontologies of the *Ontology Repository* and the service advertisements are stored into the system in the object-oriented form that is supported by the underlying reasoning engine. For that reason, users' requirements should also be transformed into the object-oriented model of the framework.

4.1 Queries

Queries specify users' preferences about the type of input information they provide and the type of outputs they want to have as a result. They must refer to existing ontology concepts and roles in order to be understood by the system and to facilitate

the retrieval procedure. Queries are expressed using a variation of the deductive rule language that the system supports [1] which are then translated into CLIPS production rules in order to be applied to the knowledge base. The procedures of the query transformation and the collection of results are performed through a *Mediator* service which can be considered as the middleware between clients and the framework.

A query is based on a template rule that contains users' requirements, as Fig. 2 depicts.

```
1:(query <query-id>
2:  (owl-s:Profile
3:      (input <input1>)
4:      (input <input2>)
5:      ...
6:      (input <inputN>)
7:      (output <output1>)
8:      (output <output2>)
9:      ...
10:     (output <outputM>))
11:)
```

Fig. 2. Query template

The template rule consists of conditions that a service or a composition plan should satisfy. More specifically, lines 3 to 6 define users' preferences for the input information they willing to give and lines 7 to 10 define users' requirements about the outputs that the requested service should produce.

4.2 Mediator Service

The *Mediator* is responsible for translating the incoming template query into a production rule, to submit the resulting rule to the framework, to collect the results and finally to send the results back to the client. This procedure is depicted in Fig. 3.

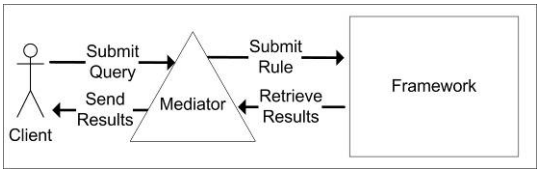

Fig. 3. The interaction between the client and the mediator

Each user's request for a service is represented by an object of the class `REQUEST`. This object stores a session id that the mediator assigns in order to determine uniquely each request, the request inputs and outputs. The definition of this class, following the native COOL syntax, is shown in Fig. 4.

```
(defclass REQUEST
  (is-a USER)
  (slot session_id (type INTEGER))
  (multislot inputs (type SYMBOL))
  (multislot outputs (type SYMBOL)))
```

Fig. 4. Class for representing a user request

The mediator is responsible for parsing the incoming query and generating the appropriate CLIPS production rule in order to submit it to the system. The submission of the rule has the effect of generating the internal representation of the query in the form of a request object. For the template query of Fig. 2, there is the corresponding template CLIPS rule of Fig. 5 that the mediator populates with values and then submits it to the system.

```
(defrule generate-requests
  (not (object
         (is-a REQUEST)
         (session_id <session-id>)))
=>
  (make-instance (gensym*) of REQUEST
    (session_id <session-id>)
    (inputs <input1> <input2> ... <inputN>)
    (outputs <output1> <output2> ... <outputM>)))
```

Fig. 5. Rule for generating objects of the REQUEST class

The system generates an object of the class `REQUEST` that stores the <session-id> in the slot `session_id`. This is a unique value and it is used in order to uniquely identify each client, since there is the possibility more that one clients to use the system simultaneously. The object also contains the user's request information, having the input requirements into the `inputs` multislot and the output requirements into the `outputs` multislot.

5 Semantic Web Service Discovery

The discovery of semantic Web services is the procedure of locating Web Services that satisfy specific requirements based on their semantic descriptions. In order to facilitate the semantic matchmaking procedure in our model, we use the rule-based OWL reasoner we have described in section 2, assuming that for each semantic Web service there is a corresponding service description in OWL-S that provide semantic information about Web services inputs and outputs.

Following the degree of matching between two concepts presented in [15], we define four matching functions in order to satisfy the matchmaking requirements between ontology concepts during the composition procedure. Recall that two concepts *i* and *j* are considered to match only if $i = j$ or *i* subsumes *j* or *j* subsumes *i*. These types of match are used later in order to define a matching score for a composition plan or a single Web service. The four matching functions are:

- ***sub(setA, setB)***: returns true if all the concepts of the *setA* set are semantically matched to the *setB* set, i.e. *setA* is a *semantic subset* of *setB* ($setA \subseteq setB$).
- ***exists(c, S)***: returns true if the *c* concept semantically matches at least one concept of the *S* set, i.e. *c semantically belongs* in the *S* set ($c \in S$).
- ***n-inter(setA, setB)***: returns true if the two sets have not any semantically similar concept, i.e. their *semantic intersection* is the empty set ($setA \cap setB = \varnothing$).
- ***n-same(setA, setB)***: returns true if the two sets have not *semantically the same* concepts ($setA \neq setB$).

6 Generating Composition Plans

In this section we analyze the algorithm for generating composition plans for a service request and we present the scoring function we use in order to rank the results.

6.1 Composition Algorithm

The composition algorithm follows a bottom-up approach: it starts from the request outputs and creates all the possible composition plans that achieve these goals based on the request inputs. The algorithm is implemented by a set of object-oriented CLIPS production rules that match registered Web services of the knowledge base based on their semantic descriptions of their inputs and outputs (*profiles*) and not just in their string representation. This service discovery procedure is based on the functions we have described in the previous section. The detailed algorithm is depicted in Fig. 6.

The algorithm starts by selecting all the Web services that achieve some (or all) request outputs and stores them in the *Children* set (line 03). Based on this set and on the user's outputs (*F* set), the algorithm utilizes the *findClusteredChildren* function that creates the *ClusteredChildren* set. This set contains elements of the form *f({Web services},{Outputs})* that denote all the possible Web service sets that achieve all request outputs. In other words, each *f* element contains the set of single Web services that produce the required outputs.

For each *f* element, the algorithm creates a *COMP* construct (line 08). A *COMP* construct contains compositions of Web services and it has an *in*, *out* and *component* field that stores the inputs, outputs and the Web services of the composition plan respectively. Each Web service in the *component* field is represented as a *CWS* construct (line 10) with *in* and *out* fields and a *composition* field that holds the *COMP* construct where the *CWS* Web service belongs. If an input of a COMP construct is matched with a request input, then it has the form *f(Input,usr)*, meaning that this input can be directly satisfied by the user request. Otherwise, it has the form *f(Input,nil)* and denote that the source of this input is still undetermined. Each *COMP* construct is stored into the *CS* set (line 17).

For each $comp_i$ of the *CS* set (line 19), the undetermined inputs are collected in the *F* set and the algorithm follows a similar recursive procedure in order to determine compositions of Web services that satisfy them (line 21). If the $comp_i$ does not have any undetermined inputs, then it is a complete composition plan and it is stored into the *FCS* set (line 22). The algorithm terminates when the *CS* list is empty, i.e. when all the $comp_i$ constructs have been examined and the *FCS* set is returned as the result.

The *FCS* set contains composition plans that satisfy *exactly* request outputs, i.e. the plans do not produce outputs that are not requested by the user. Although this is a desirable feature, we plan to modify the algorithm in order to propose more relaxed results especially in cases where the model is unable to create a composition plan that satisfies the requested outputs only.

01: BEGIN
02: FCS = Ø, CS = Ø, F = O_{usr}, Children = Ø
03: $\forall$ $ws_i \in$ WS, **sub(out(ws_i), F)** $\rightarrow$ Children = Children $\cup$ {ws_i}
04: IF Children = Ø **THEN RETURN** Ø

```
05: ClusteredChildren = findClusteredChildren(Children, F)
06: IF ClusteredChildren = Ø THEN RETURN Ø
07: ∀ f(wsc_i,out_i) ∈ ClusteredChildren
08:    CREATE comp_i ∈ COMP, out(comp_i) = out_i, in(comp_i) = Ø, components(comp_i) = Ø
09:    ∀ ws_i ∈ wsc_i,
10:       CREATE cws_i ∈ CWS, service-id(cws_i) = ws_i, out(cws_i) = Ø, in(cws_i) = Ø, composition(cws_i) = comp_i
11:          ∀ out ∈ out(ws_i), out(cws_i) = out(cws_i) ∪ { f(out,usr) }
12:          ∀ in ∈ in(ws_i),
13:             IF exists(in, I_usr) THEN in(cws_i) = in(cws_i) ∪ { f(in,usr) }
14:             ELSE in(cws_i) = in(cws_i) ∪ { f(in,nil) }
15:          in(comp_i) = in(comp_i) ∪ in(ws_i),
16:          components(comp_i) = components(comp_i) ∪ { cws_i }
17:    CS = CS ∪ { comp_i }
18: REPEAT
19: ∀ comp_i ∈ CS,
20:    F = Ø
21:    ∀ in ∈ in(comp_i) ∧ NOT exists(in, I_usr), F = F ∪ { in }
22:    IF F = Ø THEN FCS = FCS ∪ { comp_i }
23:    ELSE
24:       Children = Ø
25:       ∀ ws_i ∈ WS, sub(out(ws_i), F) → Children = Children ∪ {ws_i}
26:       IF Children ≠ Ø THEN
27:          ClusteredChildren = findClusteredChildren(Children, F)
28:          IF ClusteredChildren ≠ Ø THEN
29:             ∀ f(wsc_i,out_i) ∈ ClusteredChildren
30:                CREATE comp_k ∈ COMP, out(comp_k) = out(comp_i), in(comp_k) = in(comp_i) - F,
31:                                   components(comp_k) = components(comp_i)
32:                ∀ ws_i ∈ wsc_i,
33:                   CREATE cws_i ∈ CWS, service-id(cws_i) = ws_i, out(cws_i) = Ø, in(cws_i) = Ø,
34:                                   composition(cws_i) = comp_k
35:                   ∀ out ∈ out(ws_i), ∃ cws_k ∈ components(comp_i) ∧ exists(f(out,nil), in(cws_k)) →
36:                      out(cws_i) = out(cws_i) ∪ { f(out,cws_k) },
37:                      in(cws_k) = in(cws_k) - { f(out,nil) }
38:                      in(cws_k) = in(cws_k) ∪ { f(out,cws_i) }
39:                   ∀ in ∈ in(ws_i),
40:                      IF exists(in, I_usr) THEN in(cws_i) = in(cws_i) ∪ { f(in,usr) }
41:                      ELSE in(cws_i) = in(cws_i) ∪ { f(in,nil) }
42:                   in(comp_k) = in(comp_k) ∪ in(ws_i),
43:                   components(comp_k) = components(comp_k) ∪ { ws_i }
44:                CS = CS ∪ { comp_k }
45:       CS = CS - { comp_i }
46: UNTIL CS = Ø
47: RETURN FC
48: END
49:
50: PROCEDURE findClusteredChildren(Children, F) {
51:    ClusteredChildren = Ø
52:    ∀ ws_i ∈ Children, ClusteredChildren = ClusteredChildren ∪ { f({ws_i},out(ws_i)) }
53:       REPEAT
54:       Flag = 0
55:       ∀ f(wsc_i,out_i) ∈ ClusteredChildren,
56:          IF ∃ ws_i ∈ Children ∧ n-inter(out(ws_i), out_i) THEN out_i = out_i ∪ out(ws_i), wsc_i = wsc_i ∪ {ws_i}
57:          Flag = 1
58:          ClusteredChildren = ClusteredChildren - { f({ws_i},out(ws_i)) }
59:       UNTIL Flag = 0
60:       ∀ f(wsc_i,out_i) ∈ ClusteredChildren,
61:          IF n-same(out_i, F) THEN ClusteredChildren = ClusteredChildren - { f(wsc_i,out_i) }
62: RETURN ClusteredChildren
63: }
```

Fig. 6. Composition algorithm.

6.2 Assigning Scores

Since the resulting composition plans or the single Web Services for a request may be more than one, we use the three matching types of concepts of [15] in order to assign scores. These scores denote the relevance of the results to a specific request and affect the way the results are presented to the client. For each matching type between concepts, we define a weight *w* that denotes the degree of similarity of the concepts.

- ***exact* match**: This type of match exists when the two matched concepts are the same or equivalent (in the terms of OWL class equivalence). In our model, we consider direct subclass relationships as subsumption relationships and they are handled by the other two types of matches. The exact matching weight is $w_e = 3$.
- ***plug in* and *subsume* matches**: These types of matches denote that there is a subsumption relationship between the matched concepts (including direct subclass relationships). The weight that is assigned to these matches depends on the distance between the two concepts in the ontology. The distance *d* between two concepts C_1 and C_2 is defined as the number of concepts that exist in the (shortest) path from C_1 to C_2 in the ontology (including also in the sum the C_1 and C_2). For example, the distance between two concepts with a direct subclass relationship is 2. In that way, for plug in matches we define $w_p = 1 + 2/d$ and for subsume $w_s = 2/d$, denoting that plug in matches are always preferable than subsume.
- ***fail* match**: In this case, the matching fails (disjoint concepts).

The score *s* for a single Web service is computed between request inputs and Web service's inputs and between request outputs and Web service's outputs. In a similar manner, the score for a composition plan is computed by regarding it as a single Web service with inputs the initial plan inputs and outputs the final plan outputs. The scoring function is defined as the sum of the minimum *w* after the matching procedure of the inputs of the request and the service and the minimum *w* after the matching procedure of the outputs of the request and the service.

$$s = \min\{weights_{in}\} + \min\{weights_{out}\}$$

In the case of composition plans with similar scores, we rank them according to the number of the Web services that are involved in the composition plan, i.e. for two composition plans C_a and C_b that contain *n* and *k* Web services respectively, we consider that $rank(C_a(n)) > rank(C_b(k))$, if $n < k$.

7 An Example

In this section we present an example of generating a composition plan for a service request. Suppose the simple ontology of Fig. 7 that describes the domain of an online bookstore. We assume that this ontology is already processed by the underlying OWL rule reasoner that has generated the corresponding class hierarchy into the object-oriented knowledge base.

Different parties have registered their Web services by submitting the descriptions in the framework. We assume that there are 4 Web services with the following functionalities:

- WS_{tpub}**(in:{*Title*}, out:{*Publisher*})**: the Web service takes as input a book title and returns its publisher.
- WS_{tper}**(in:{*Title*}, out:{*Person*})**: the web service takes as input the title of the book and returns instances of any person of the domain.
- WS_{ti}**(in:{*Title*}, out:{*ISBN*})**: the web service takes as input the title of the book and returns the ISBN number.
- WS_{ip}**(in:{*ISBN*}, out:{*Price*})**: the web service takes as input an ISBN number and returns the price of the book.

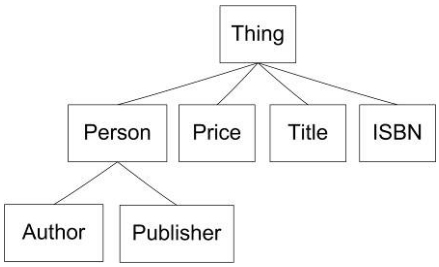

Fig. 7. A simple taxonomy for an online bookstore.

The profiles of the above Web services have been submitted in the framework by the providers who want to advertise their services. These advertisements contain semantic annotations for the inputs and outputs of the service they describe, using concepts of the ontology in Fig. 7.

Suppose, a client wants to retrieve the *price* and the *publisher* of a particular book based on the *title* and submits the query depicted in Fig. 8 to the Mediator. Queries should utilize existing ontology concepts for the input and output parameters.

```
(query book-price
  (owl-s:Profile
    (input bookstore:Title)
    (output bookstore:Publisher)
    (output bookstore:Price)))
```

Fig. 8. The input query of retrieving the publisher of a book.

The Mediator is responsible for assigning a session id to the query, e.g 345 and translating the incoming query into a production rule that submits to the system (Fig. 9). This rule generates an object of the `REQUEST` class, which is the internal representation of user's request.

```
(defrule generate-requests
  (not (object
         (is-a REQUEST)
         (session_id 345)))
=>
  (make-instance (gensym*) of REQUEST
    (session_id 345)
    (inputs bookstore:Title)
    (outputs bookstore:Price)
    (outputs bookstore:Publisher)))
```

Fig. 9. The production rule that transforms a query into a REQUEST object.

The generation of the new object in the knowledge base, triggers the pattern matching procedure of the defined production rules that facilitate the composition procedure. The algorithm starts by selecting all Web services that match all or some of the request outputs. In the example, we have two request outputs and the WS_{ip}, WS_{tpub}, WS_{tper} Web services are selected that satisfy them and are added in the *Children* list. Since this list is not empty, the algorithm continues by generating the *ClusteredChildren* list using the function *findClusteredChildren*. The resulting list is:

$$\{f(\{W_{tpub}, W_{ip}\}, \{Publisher, Price\}), f(\{W_{tper}, W_{ip}\}, \{Person, Price\})\}$$

For each element *f* of the *ClusteredChildren* list, the algorithm creates a *COMP* construct. The *COMP* construct is an abstraction of a composition plan that stores the Web services, the inputs and the outputs of the so far generated plan. Each Web service that participates in the composition plan is represented as a *WSC* construct that holds the inputs, the outputs (along with the corresponding web services that provide and consume those inputs and outputs) and the id of the original Web service. The *COMP* construct (*comp1*) for the first element of the *ClusteredChildren* list is depicted in Fig. 10.

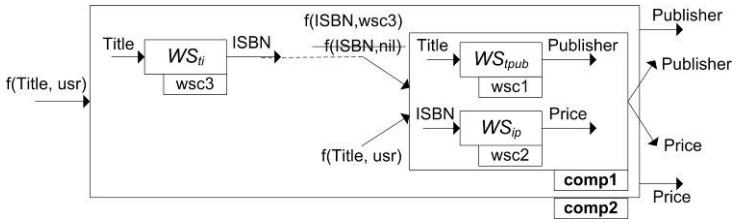

Fig. 10. A composition plan for the request.

The *f(Title,usr)* input denotes that this is a request input whereas the *f(ISBN,nil)* denotes that currently this is an undetermined input. The next step of the algorithm is to determine a composition plan that satisfies the undetermined input of the *wsc1* component that belongs to the *comp1* composition plan. Following the same procedure, a second *COMP* construct (*comp2*) is created for the Web services whose outputs satisfy the *ISBN* input. There is only one Web service that produces the required input, i.e. the WS_{ti} Web service. The *comp2* construct is created (and replaces *comp1*) and the algorithm changes the *f(ISBN,nil)* input of *wsc1* to *f(ISBN,* WS_{ti}*)* in order to denote the Web service from which will take the output. Since all the inputs of *comp2* are request inputs (i.e. there are no other *f(Input,nil)* to satisfy) the algorithm terminates the procedure for creating *COMP* constructs and the composition plan *comp2* is stored into the *FCS* list.

The same procedure holds for the second *f* element of the *ClusteredChildren* list and a similar composition plan is created and added to the *FCS* list with a *Person* output instead of *Publisher* due to the WS_{tper} Web service. Appropriate production rules traverse the *FCS* list and assign scores to each composition plan. Based on the scoring function we have described in section 6.2, the composition plan of Fig. 10 is assigned with a score value equal to 6. More specifically, the minimum weight *w* for the inputs is 3 since there is an exact match between request input and plan input. The minimum weight *w* of outputs is again 3 since there is an exact match between request outputs and plan outputs. Thus:

$$s_1 = \min\{weights_{in}\} + \min\{weights_{out}\} = w_e + w_e = 3 + 3 = 6$$

For the second composition plan, the minimum weight *w* for the inputs is 3 since there is an exact match between request input and plan input. The minimum weight *w* of outputs is now 2 since there is a plug in match between request outputs and plan outputs. Thus:

$$s_2 = \min\{weights_{in}\} + \min\{weights_{out}\} = w_e + w_p = 3 + 2 = 5$$

Since $s_1 > s_2$, the composition plan of Fig. 10 is preferable.

8 Related Work

Many research efforts have been focused on the field of Web service discovery and/or composition. In this section we briefly present some of these approaches.

In [17] a toolset for Web service composition is presented. In SWORD each service is represented by a rule that expresses the inputs that should hold in order for the rule to be activated and to produce particular outputs. These rules are used in an expert system (JESS [7]) in order to determine if a composite service that produces a desirable output can be realized using existing services (that are represented as rules). The difference of our framework is that each service is described using service description standards, such as OWL-S and we utilize an OWL reasoner in order to match services based on semantic descriptions and not just on simple string matches, as a native rule engine does.

In [20] the authors propose a prototype for semi-automating Web service composition. Users create a workflow of services by presenting the available choices at each step. Web services descriptions are defined in DAML-S and through an OWL Prolog reasoner, the system inferences and selects the matching services based on subsumption relationships. Services are also filtered based on constraints which the user may specify on their attributes. In contrast to this approach, our framework targets at the automatic generation of a composition plan based on the initial user's input and output requirements only.

IRS-III [2] is a framework which takes a semantic broker based approach to creating applications from semantic Web services by mediating between a service requester and one or more service providers. The aim of the framework is to enable the automated or semi-automated construction of semantically enhanced systems over the internet. In contrast to our model, this framework implements and extends the WSMO conceptual model for Web services description, publication and execution.

METEOR-S [21] is a project that deals with the problems of semantic Web service description, discovery and composition. It associates semantics to Web services, covering input/output, functional/operational descriptions, execution and quality and exploits them in the entire Web process lifecycle encompassing semantic description, discovery and composition of Web services. In this project, Web services are described using WSDL-S descriptions instead of the WSMO or the OWL-S ontology.

In [10], the authors describe their approach for semantic Web services matchmaking. They argue that hybrid approaches to semantic matching that exploit both formal and implicit semantics may improve the retrieval performance of semantic service matching over purely logic-based ones. In their approach, they

utilize both logic based reasoning and content based information retrieval techniques for services specified in OWL-S. Our system currently supports matchmaking based only on simple subsumption relationships but we plan to extend the procedure with more techniques.

In [8] the authors present a logical framework for automated Web service discovery which is based on the WSMO conceptual model. They have implemented their approach in the F-Logic [9] reasoning engine Flora2 [5]. However, they deal only with the discovery aspect of Web services in contrast to our work where we define a Web service composition algorithm.

9 Conclusions

In this paper we describe ProSeDisCo, a production rule-based framework for Web services discovery and composition. Based on the implementation of a rule-based OWL reasoner (O-DEVICE) on top of the CLIPS production rule engine, we define a model for the facilitation of Web services discovery and composition. Taking advantage of the efficiency of the well-known underlying rule engine, we use production rules in order to a) semantically match user requests with Web services OWL-S descriptions and b) implement a Web service composition algorithm.

Rules are considered to play a key role for the full realization of the semantic Web. Since semantic Web services are heavily based on this initiative, we believe that rules will also affect every aspect of them. As far as the OWL-S ontology is concerned, rules can be used in order to define pre and post conditions that should be satisfied before and after the execution of a Web service, e.g. using the SWRL [6]. Since our model is based on rules, we can easily extend it in order to represent such conditions using the supported deductive rule language [1].

Web services nonfunctional parameters can also be represented using rules, e.g. in the case of the selection of a Web service based on quality measures. We argue that the declaretiveness of rules is a powerful tool and it can be used during the discovery as well as the composition procedure. For the latter, we plan to enhance the model with the ability of defining composition plans via rules, e.g. in the case of a commonly used service that has a static composition plan and to facilitate the plan execution procedure.

Acknowledgments. This work was partially supported by a PENED program (EPAN M.8.3.1, No. 03EΔ73), jointly funded by the European Union and the Greek government (General Secretariat of Research and Technology) and by the Greek R&D General Secretariat through a bilateral Greek-Ukrainian project (EPAN-M.4.3-A.4.3.6.1, No. 148-γ)

References

1. Bassiliades, N., Vlahavas, I.: R-DEVICE: An Object-Oriented Knowledge Base System for RDF Metadata, International Journal on Semantic Web and Information Systems, Amit Sheth , Miltiadis D. Lytras (Ed.), Idea Group, Vol. 2, No. 2, pp. 24-90, 2006

2. Cabral, L., Domingue, J., Galizia, S., Gugliotta, A., Norton, B., Tanasescu, V., Pedrinaci, C. (2006) IRS-III: A Broker for Semantic Web Services based Applications, The 5th International Semantic Web Conference (ISWC 2006), Athens, GA, USA
3. Cabral, L., Domingue, J., Motta, E., Payne, T. and Hakimpour, F. (2004). Approaches to Semantic Web Services: An Overview and Comparisons. In proceedings of the First European Semantic Web Symposium, ESWS 2004, Heraklion, Crete, Greece. LNCS 3053
4. CLIPS: A Tool for Building Expert Systems, http://www.ghg.net/clips/CLIPS.html
5. FLORA-2, The FLORA-2 web site. http://flora.sourceforge.net
6. Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission, 21 May 2004. http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/
7. JESS: Rule Engine for Java, http://herzberg.ca.sandia.gov/jess/
8. Kifer, M., Lara, R., Polleres, A., Zhao, C., Keller, U., Lausen, H., Fensel, D.: A Logical Framework for Web Service Discovery. In Semantic Web Services Workshop at ISWC, Hiroshima, Japan, November 2004
9. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM, 42(4):741–843, July 1995
10. Klusch, M., Fries, B., Sycara, K.: Automated Semantic Web Service Discovery with OWLS-MX. Proceedings of 5th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), Hakodate, Japan, 2006
11. Meditskos, G., Bassiliades, N.: Towards an Object-Oriented Reasoning System for OWL, Int. Workshop on OWL Experiences and Directions, B. Cuenca Grau, I. Horrocks, B. Parsia, P. Patel-Schneider (Ed.), 11-12 Nov. 2005, Galway, Ireland, 2005
12. O-DEVICE: http://lpis.csd.auth.gr/systems/o-device/o-device.html
13. OWL Web Ontology Language Overview, http://www.w3.org/TR/owl-features/
14. OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/
15. Paolucci, M., Kawamura, T., Payne, T., Sycara, K.: Semantic matching of web services capabilities. In: Proceedings of the First International Semantic Web Conference, LNCS 2342, Springer-Verlag (2002) 333–347
16. Paolucci, M., Srinivasan, N., Sycara, K.: Expressing WSMO Mediators in OWLS. In Proceedings of the workshop on Semantic Web Services: Preparing to Meet the World of Business Applications held at the 3rd International Semantic Web Conference (ISWC 2004), Hiroshima, Japan, 2004
17. Ponnekanti, S. R., Fox, A.: SWORD: A Developer Toolkit for Web Service Composition. In Proceedings International WWW Conference(11), Honolulu, Hawaii, USA, 2002.
18. Resource Description Framework (RDF), http://www.w3.org/RDF/
19. Ruben, L., Roman, D., Polleres, A., Fensel, D.: A Conceptual Comparison of WSMO and OWL-S, Proceedings of the European Conference on Web Services (ECOWS 2004)
20. Sirin, E., Hendler, J., Parsia, B.: Semi-automatic composition of Web services using semantic descriptions. In Proceedings of Web Services: Modeling, Architecture and Infrastructure workshop in conjunction with ICEIS2003, 2002
21. Sivashanmugam, K., Miller, J., Sheth, A., Verma, K.: Framework for Semantic Web Process Composition, International Journal of Electronic Commerce, Winter 2004-5, Vol. 9(2) pp. 71-106
22. SOAP Version 1.2 Part 1: Messaging Framework, http://www.w3.org/TR/soap12-part1/
23. Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl
24. WSDL-S, http://www.w3.org/Submission/WSDL-S/
25. WSMO: Web Service Modeling Ontology D2v1.3, http://www.wsmo.org/TR/d2/v1.3/