Towards Dynamic Orchestration of Semantic Web Services

Domenico Redavid[1], Stefano Ferilli[2], and Floriana Esposito[2]

[1] Artificial Brain Srl - Bari, Italy
redavid@abrain.it
http://www.abrain.it
[2] Dipartimento di Informatica, Università degli Studi di Bari, Italy
{ferilli,esposito}@di.uniba.it
http://www.di.uniba.it

Abstract. The Semantic Web together with Web services technologies enable new scenarios in which the machines use the Web to provide intelligent services in an autonomus way. The orchestration of Semantic Web Services now can be defined from an abstract perspective where their formal semantics can be exploited by software agents to replace human input. This paper tackles the more difficult use case, automatic composition, providing a complete solution to create and manage service processes in a semantically interoperable environment.

Keywords: Semantic Web Services, Orchestration, Process generation.

1 Introduction

From the Web services perspective, an orchestration is a declarative specification that describes a work-flow supporting the execution of a specific business process, operation, or service [12]. Currently, Web services technologies allow to describe orchestration, but only at design-time. Semantic Web Services (SWSs) provide an ontological framework for describing services in a machine-readable format. In [10] a software agent is proposed that applies logical reasoning on SWS descriptions in order to provide on-the-fly orchestration capabilities. With the adjective *dynamic* we denote the capability of an agent to design and manage in an automatic way an orchestration schema using the semantic descriptions of available services. The notion of *dynamic* orchestration becomes useful in scenarios where there is the need of run-time designing process integration using semantic descriptions of the involved entities. This work addresses the most difficult part for the realization of this scenario, i.e., the automatic composition mechanism. In fact, as will explained in the following, orchestration is mainly a representation problem that requires the semantic representation of constraints (preconditions, effects, etc.), currently not properly supported by the Semantic Web standard languages.

This paper is organized in four sections. Section 2 describes what orchestration means from the Semantic Web perspective, the requirements to realize it and the support offered by the main SWS languages. Section 3 provides details about the language

N.T. Nguyen (Ed.): Transactions on CCI X, LNCS 7776, pp. 16–30, 2013.

that better satisfies those requirements, to be used for the implementation of the composition use case. Section 4 describes the entire procedure to obtain a composed service written in the chosen language, while Section 5 proposes a complete example of the applied procedure. Finally, conclusions and future works complete the paper.

2 SWS Dynamic Orchestration

2.1 Orchestration and SWS Infrastructure

Orchestration for Web services must necessarily rely on XML-based descriptions. Their merely syntactical nature [12] represents an unsurmountable obstacle towards the automatic implementation of the operations that are necessary to accomplish dynamicity in orchestration. Let us, indeed, consider the following scenario:

> "Given a request (goal), an agent (dynamic orchestrator) *discovers* the possible SWSs able to accomplish the goal. At the same time, it *composes* the discovered services in order to have many possible ways to reach the goal. Having different alternatives available, it *selects* the best one exploring functional and non-functional properties of the different SWSs. The selection process can be used also during composition (sub-goal matching). Finally, the same agent manages the correct *invocation* of the selected services."

In this paper, dynamic orchestration of SWSs is inteded as carrying out one or more of the operations mentioned above. This is very different from the design of a work-flow for the execution of simple Web services [12]. The first step to realize this scenario is the automatization of the emphasized operations (discovery, composition, selection and invocation). These operations, amongst others (namely, publishing, deployment and ontology management) which are not directly involved in our scenario, were already identified as use cases for the SWS infrastructures into the *Usage Activities* dimension described in [3]. In the same work the architectural components (*Architecture* dimension) and semantic descriptions (*Service Ontologies* dimensions) needed to realize the

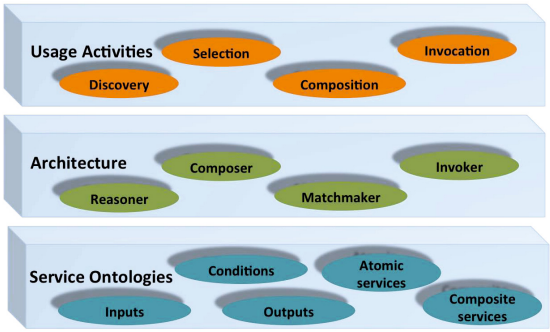


Fig. 1. Orchestration in the SWS Infrastructure

Usage Activities have been identified and used to define SWS infrastructures as formed by these three orthogonal dimensions. The *Architecture* includes a register, a reasoner, a matchmaker, a composer, and an invoker. The *Service Ontologies* dimension specifies the semantics of:

- Functional capabilities, such as inputs, output, and conditions (pre-conditions, effects, etc.);
- Non-Functional capabilities, such as category, cost and quality of service;
- Provider-related information, such as company name and address;
- Task or goal-related information and domain knowledge, defining, for instance, the type of the inputs of the service.

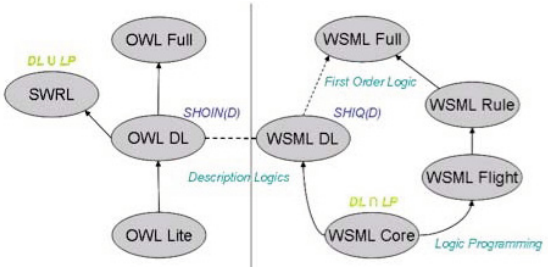
The dynamic SWS orchestration, as defined above, needs to be matched with these dimensions in order to identify its requirements. The results of this investigation are summarized in Figure 1. In the architecture dimension, the *reasoner* is the most important component to implement these use cases because it allows the semantic *matchmaking* of the service properties enabling the automatic *composition* of the services, and the automatic choice of the suitable parameters for the *invocation* of the services. The service ontologies dimension is necessary because it allows to define the semantics of functional and non-functional properties owned by atomic and composite services that participate to the orchestration, as well as the semantics of their control constructs and data-flow. Ideally, the semantic descriptions of inputs and outputs should exist independently from the SWS specification. They should be chosen within existing ontologies describing a particular domain of knowledge and, at most, they might be refined. Hence, in order to define atomic SWSs just the semantic representation of conditions over inputs and outputs is needed, whereas to define composite SWSs the semantic representation of control constructs and data-flow is needed as well. Thus, Dynamic orchestration becomes mainly a representation problem. The SWS frameworks proposed in the literature provide different support to dynamic orchestration. In the next section we analyze the formal support offered by the two leading efforts for SWS representation (OWL-S and WSMO)[1].

2.2 WSMO and OWLS Formal Support

The Web Service Modeling Ontology (WSMO) provides a framework for semantic descriptions of Web services. It consists of four core elements that, properly linked, make up the semantic description of the service: **Ontologies**, **Goals**, **Web Services**, and **Mediators**. The semantics of these entities can be specified using one of the formal languages defined by the Web Service Modeling Language[2] (WSML). WSML includes several language variants, based on three different logical formalisms: Description Logics [1] (WSML-DL), First-Order Logic [4] (WSML-Full) and Logic Programming [9]

[1] Both OWL-S (`http://www.w3.org/Submission/OWL-S`) and WSMO (`http://www.w3.org/Submission/WSMO`) are currently submissions in the Semantic Web Services Standardization process.

[2] Web Service Modeling Language (WSML), W3C Member Submission, 3 June 2005. `http://www.w3.org/Submission/WSML/`



	Max usable expressive power combining the different types of SWS representation	
	WSMO	**OWL-S**
DL	WSML-DL (SHIQ(D))	OWL-DL (SHOIN(D))
LP	WSML-RULE (undecidable)	SWRL (undecidable)
DL & LP	WSML-Core (DL ∩ LP)	SWRL DL-Safe Rule (DL ∪ LP)

Fig. 2. OWL+SWRL vs. WSML formal support

(WSML-Rule and WSML-Flight). Furthermore, it defines the **WSML-Core** variant that can be identified as the intersection between a particular Description Logic (a subset of $\mathcal{SHIQ}$) and Horn Logic (without function symbols and equality) [6]. WSML-DL is, in general, incompatible with both WSML-Flight and WSML-Rule. Therefore, complete reasoning about service descriptions specified using WSML-DL, Rule and Flight is unfeasible because WSML-Full, the common super-language, is undecidable.

The Web Ontology Language for Services (OWL-S) is an OWL[3] ontology describing three essential aspects of a service (i.e., its advertisement, process model, and protocol details) using three different *modules*: **Service Profile**, **Service Model**, and **Service Grounding**. OWL-S itself is an OWL ontology. OWL, having formal foundations in Description Logics [1], provides three increasingly expressive sub-languages:

- **OWL-Lite**, that can be used to model classification hierarchies and simple constraints. OWL-Lite has the lowest computational complexity among OWL sub-languages.
- **OWL-DL**, used when the maximum decidable expressivity is required. It corresponds to $\mathcal{SHOIN}(\mathcal{D})$ DL [8].
- **OWL-Full**, although allowing for as much expressivity as RDF, is undecidable, and therefore it could be hardly be used for reasoning.

However, since (as will be pointed out in the following) the decidable fragment of SWRL [7] represents a reasonable advancement in reconciling Logic Programming and Description Logics, OWL-S is in a better position than WSMO as it allows to seamlessly integrate SWRL with OWL.

[3] OWL-S is based on OWL 1.0 recently extended with the new recommendation OWL 2, as reported in `http://www.w3.org/TR/owl-overview/`

2.3 OWL-S and WSMO in SWS Infrastructures

In this section we focus on the usage activities and service ontologies dimensions to compare and contrast the two proposed main framework: WSMO and OWL-S[4]. Indeed, the architecture dimension concerns the components needed to implement the use cases that do not affect the purposes of this comparison.

With respect to the *Usage activities* dimension, both WSMO and OWL-S take into account all the use cases that we have identified for dynamic orchestration. Their only difference is that WSMO requires to model the service requester features. This implies the need for tackling the interoperability problems between different WSMO elements within the framework itself. In order to overcome this issue, different types of mediators have been defined and considered in every usage activity, while OWL-S relies on ontology matching methods developed within the Semantic Web. This is reflected also in the *Service ontologies* dimension, where WSMO proposes a richer service description model than OWL-S. Consequently, some extra features related to Web service execution (like goals, mediator and communication protocols) need to be represented by WSMO model. The shortcoming of this approach is that any WSMO service usage is limited to its declared goals at design time.

The formal semantics offered by WSMO and OWL-S is fundamental to achieve *dynamic* orchestration. Figure 2 compares the formalisms on which the two frameworks are based. WSMO seems to be unrelated to the Semantic Web stack of languages and formalisms, except for its basic level (URI and XML), that guarantees syntactic interoperability on the Web. It offers mappings with the RDF syntax and the $\mathcal{SHIQ}$ Description Logic which, however, does not cover the whole OWL-DL. WSMO's global approach to service ontologies definition can be summarized as follows: it starts with the definition of a FOL-like language able to represent all the possible aspects of a Web service, then it defines several sub-languages restricting the original expressive power to well-known fragments (DL or LP), and (partially) maps such fragments onto current Semantic Web standard languages. Given the known incompatibility between DL and LP [2], currently the only implemented reasoning available for WSMO, when dealing with both rules and ontologies, is restricted to their common subfragment, i.e. DLP. Hence, as regards the implementation goal, to the best of our knowledge WSMO's inference capabilities are significantly reduced with respect to a pure Semantic Web based counterpart. OWL-S, instead, is based natively in OWL, whose natural extension towards rules is SWRL. SWRL's decidable fragment (DL-safe rules [11]) is the largest possible union of primitives from both DL and LP formalisms rather than their intersection. Therefore, from the perspective of two out of three main dimensions of SWS infrastructures, OWL-S turns out to be better equipped to encompass the requirements of *dynamic* orchestration.

[4] Semantic Annotations for WSDL and XML Schema (SAWDL) — W3C Recommendation, `http://www.w3.org/TR/sawsdl/` — has not been considered because it does not allows the process model representation.

3 OWL-S Support for Orchestration Dimensions

3.1 OWL-S Framework Details

OWL-S provides a Semantic Web Services framework to formalize an abstract description of a service. Since it is an upper ontology described with OWL, every described service maps onto an instance of this concept. The upper level **Service** class is associated with the following three other classes:

Service Profile. It specifies the functionality of a service. This concept is the top-level starting point for the customizations of the OWL-S model that supports the retrieval of suitable services based on their semantic description. It describes the service by providing several types of information: Human-readable information, Functionalities, Service parameters, Service categories.

Service Model. It exposes to clients how to use the service by detailing the semantic content of requests, the conditions under which particular outcomes will occur, and, where necessary, the step-by-step processes leading to those outcomes. It defines the concept *Process*, that describes the composition of one or more services in terms of their constituent processes. A *Process* can be *Atomic* (a description of a non-decomposable service that expects one message and returns one message in response), *Composite* (consisting of a set of processes within some control structure that defines a workflow) or *Simple* (used as an element of abstraction, i.e., may be used either to provide a view of a specialized way of using an atomic process, or a simplified representation of a composite process for reasoning purposes).

Service Grounding. A grounding is a mapping from an abstract to a concrete specification of those service description elements that are required for interacting with the service. In general, a grounding indicates a communication protocol, a message format and other service-specific details.

Since this work aims at the automatic generation of services workflow and at its representation as an OWL-S composite process, more details about OWL-S process model are needed.

3.2 OWL-S Process Model

In this section we report the basic notions about the OWL-S process model with some considerations on the guidelines that should be followed in order to have useful metadata for the Web services to be described. Each OWL-S process is based on an IOPR (Inputs Outputs Preconditions Result) model. The *Inputs* represent the information required for executing the process. The *Outputs* represent the information the process returns to the requester. *Preconditions* are conditions that are imposed over the *Inputs* of the process and that must hold for the process to be successfully invoked. Since an OWL-S process may have several results with corresponding outputs, the *Result* entity of the IOPR model provides a means to specify this situation. Each result can be associated to a result condition, called *inCondition*, that specifies when that particular result can occur. Therefore, an *inCondition* binds inputs to the corresponding outputs. Such conditions are assumed to be mutually exclusive, so that only one result can be obtained

for each possible situation. When an *inCondition* is satisfied, there are properties associated to this event that specify the corresponding output (*withOutput* property) and, possibly, the *Effects* (*hasEffect* properties) produced by the execution of the process. *Effects* are changes in the state of the world. The OWL-S conditions (*Preconditions*, *inConditions* and *Effects*) can be represented as SWRL logic formulas. Formally, *Input* and *Output* are subclasses of the more general class *Parameter*, declared in turn as a subclass of *Variable* in the SWRL ontology. Every parameter has a type, specified using a URI. Such a type is needed to refer it to an entity within the domain knowledge of the service. The type can be either a *Class* or a *Datatype* (i.e., a concrete domain object such as a string, a number, a date and so on) in the domain knowledge. Nevertheless, we argue that providing descriptions of Web service parameters using concrete datatypes adds very little semantics. For example, consider a service S whose input was declared as *Datatype* within a knowledge domain, e.g., a string. This means that the reference knowledge model of this input parameter is a concrete XML Schema datatype rather than an entity within a domain ontology. This mismatch becomes critical in automatic composition of services. Indeed, suppose that, during a hypothetical composition process, one needs to find another service whose output will be fed into S. Then, the composer must necessarily consider those services that have as output a resource of the same type as our input parameter, i.e. a string. Thus, any service that returns a string as an output can be composed with S, which would result in meaningless compositions of completely unrelated services due to the fact that the parameters have been poorly described form a semantic viewpoint. In the rest of this paper we will consider only those services having parameters declared as entities in a domain ontology (i.e., not as datatype). Furthermore, OWL-S Composite processes (decomposable into other Atomic or Composite processes) can be specified by means of the following *control constructs* offered by the language: **Sequence**, **Split**, **Split-Join**, **Any-Order**, **Choice**, **If-Then-Else**, **Iterate**, **Repeat-While** and **Repeat-Until**, and **AsProcess**. One crucial feature of a composite process is the specification of how its inputs are accepted by particular sub-processes, and how its various outputs are produced by particular sub-processes. Structures to specify the *Data Flow* and the *Variable Bindings* are needed. When defining processes using OWL-S, there are many places where the input to one process component is obtained as one of the outputs of a previous step, short-circuiting the normal transmission of data from service to client and back. For every different type of *Data Flow* a particular *Variable Binding* is given. Formally, two complementary conventions to specify *Data Flow* have been identified: **consumer-pull** (the source of a datum is specified at the point where it is used) and **producer-push** (the source of a datum is managed by a pseudo-step called *Produce*). Finally, we remark that a composite process can be considered as an atomic one using the OWL-S Simple process declaration. This allows to treat a Composite service as an Atomic one during the application of a Composer tool.

4 Automatic Composition for OWL-S

In this section we explain how to obtain OWL-S composite services using Semantic Web languages and tools. The proposed approach combines them with works presented

in [14] and [13] endowing semantic interoperability. The procedure can be summarized as follows: 1) a SWRL representation is extracted from the available set of OWL-S atomic and simple services using a given encoding; 2) a SWRL composer generates a plan of rules that encodes the services; 3) the SWRL plan is interpreted to produce the OWL-S composite service.

4.1 Encoding OWL-S Atomic Processes with SWRL Rules

In this section we explain our approach for transforming process descriptions into sets of SWRL rules. SWRL [7] extends the set of OWL axioms to include Horn-like rules [9]. The proposed rules are in the form of an implication between an antecedent (*body*) and consequent (*head*), both consisting of a conjunction of zero or more atoms. The intended meaning can be read as: "whenever the conditions specified in the antecedent hold, the conditions specified in the consequent must also hold". For our purposes, it is important to highlight two SWRL characteristics: every rule must fulfil a *safety* condition (only variables that occur in the antecedent of a rule may occur in its consequent) and every rule with a conjunctive consequent can be transformed into multiple rules, each having an atomic consequent [9]. Furthermore, we work exclusively with SWRL DL-safe rules [11] fragment. Within OWL-S, conditions (logical formulas) can be declared using languages whose standard encoding is in XML, such as SWRL. Body and head are logical formulas, whereby OWL-S conditions can be identified with the body or with the head of a SWRL rule. Such conditions are expressed over *Input* and *Output*. Therefore, if the above requirement is met, conditions will be also expressed in terms of a domain ontology and thus will have the correct level of abstraction. After these considerations, we can describe the guidelines we follow for encoding an OWL-S process into SWRL.

- For every result of the process there exists an *inCondition* that expresses the binding between input variables and the particular result (output or effect) variables.
- Every *inCondition* related to a particular result will appear in the antecedent of each resulting rule, whilst the *Result* will appear in the consequent. An *inCondition* is valid if it contains all the variables appearing in the *Result*.
- If the *Result* contains an *Effect* made up of many atoms, the rule will be split into as many rules as the atoms. Each resulting rule will have the same inCondition as the antecedent and a single atom as the consequent.
- The *Preconditions* are conditions that must be true in order to execute the service. Since these conditions involve only the process *Inputs*, they will appear in the antecedent of each resulting rule together with *inConditions*. In this work we consider all the *Preconditions* as being always true.

The first guideline is needed because there may be processes in which the binding is implicit in their OWL-S descriptions. Let us consider, for example, an atomic process having a single output. In this case there might be no *inCondition* binding input and output variables since, being the output the only outcome, such a binding is obvious. This would prevent our encoding with SWRL rules because the second guideline would not be applicable. However, we can add a new *inCondition* that makes explicit such implicit binding. For example, suppose we have a service that returns book

information, whose process is declared having one input (*?process:BookName*), one output (*?process:BookInfo*), and no condition. We should write the corresponding rule as "*kb:BookTitle(?process:BookName)* $\rightarrow$ *bibtex:Book(?process:BookInfo)*", but since variable *process:BookInfo* does not appear in the antecedent of the rule, this is not a valid SWRL rule. Since every service produces its output by manipulating the inputs, we may suppose that a *hasTransf* predicate exists, always true, that binds every input to the output. Adding this predicate to the rule antecedent we obtain the implicit *inCondition* and hence a valid rule. The entities that make up the SWRL rule (OWL Classes, properties and individuals) can be defined in different ontologies. For this reason we apply the matching of ontologies referred by the rules in order to enable semantic interoperability during composition. In particular, only the OWL classes need to be aligned, because properties in the rule are relations between considered classes and individuals are instances of considered classes. The matching procedure, which can be made by applying one of several learning techniques in the literature [5], produces equivalence assertions between classes. For example, consider the classes *books:Book* and *univ:Book*, where *books* and *univ* are the namespaces of two different ontologies *books.owl* and *univ-bench.owl*, respectively, describing the same domain. The result of the alignment will be an OWL axiom asserting that *books:Book* and *univ:Book* are equivalent classes. This axiom will be added to the knowledge base containing the SWRL rules.

4.2 The Composition Algorithm

Our SWRL composer prototype implements a backward search algorithm for the composition task and enhances the algorithm proposed in [14]. It works as follows: it takes as input a knowledge base containing SWRL rules (with the descriptions of the equivalent OWL classes) and a goal specified as a SWRL atom, and returns every possible path built by combining the available SWRL rules in order to achieve such a goal. These rules fulfil the SWRL safety condition. Specifically, the algorithm performs backward chaining starting from the goal in the same way as Prolog-like reasoners work for query answering. The difference is that this algorithm works on SWRL DL-safe rules instead of Horn clauses. This means that, besides the rule base, it takes into account also the Description Logic ontology the rules refer to. The SWRL rule path found, and consequently the resulting OWL-S service composition, will be valid (in the sense that it will produce results for the selected goal) only if the SWRL rules in the path are DL-safe. In other words, DL-safety means that rules are true for individuals that are *known*, i.e. that appear in the knowledge base[5]. The implemented prototype performs DL-safety check. This guarantees that the application of rules is grounded in the ABox and, consequently, that the services embodying those rules can be executed.

4.3 SWRL Plan Analysis

The set of paths obtained as a result of the composer can be considered as a SWRL rules plan (referred to as *plan* in the following) representing all possible combinable OWL-S

[5] It might not be the case in general, given the Open World Assumption holding in Description Logics, see [11] and Chapter 2 in [1].

Atomic processes that lead to the intended result (the goal). According to the OWL-S specification for a composed process and its syntax, the composition of atomic services obtained through the SWRL rule composer can be represented by means of an OWL-S composite service. In this section we will analyze a possible encoding. An OWL-S composed process can be considered as a tree whose nonterminal nodes are labeled with control constructs, each of which has children that are specified through the OWL property *components*. The leaves of the tree are invocations of the processes mentioned as instances of the OWL class Perform, a class that refers to the process to be performed. Bearing in mind the characteristics of the plan built by means of the method specified in Section 4.1, we identify the OWL-S control constructs to be used to implement the plan by applying the guidelines reported in Table 1. Currently, the OWL-S specification does not completely specify Iteration and its dependent constructs (Repeat-While and Repeat- Until), nor how the *asProcess* construct could be used. For this reason they are not discussed in this paper, but they will be considered in future work.

4.4 Encoding the SWRL Plan with OWL-S Constructs

For our purposes, each rule is represented in the composer as an object called *Rulebean*, that has various features and information that could be helpful to the composition, and specifically:

- The atoms in the declared precondition of the rule;
- The URI of the atomic process the rule refers to;
- The number of atoms in which the grafted rules have correspondence;
- A list containing pointers to the other Rulebeans with which it is linked.

The information about the atoms of the preconditions allows to check the presence of IF conditions that could lead to identify a situation that needs an If-Then-Else construct. The URI of the atomic process referred by the rule is needed because the leaves of the constructs tree must instantiate the processes to be performed. Finally, since each rule can be linked to other rules, it is necessary to store both their quantity and a pointer to the concatenated rules. In this way each Rulebean carries inside the entire underlying structure. This structure is implemented as a tree of lists, where each list contains the Rulebeans that are grafted on the same atom. Now let's show in detail the steps needed to encode a plan with the OWL-S control constructs. Referring to Figure 4, the procedure involves the application of three subsequent steps depending on the number n of grafted rules:

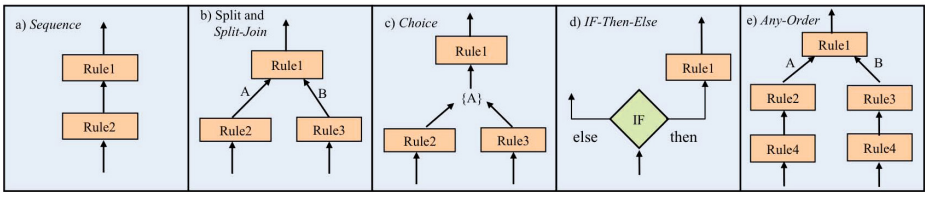


Fig. 3. The OWL-S control constructs in the plan

Table 1. OWL-S *Control Constructs* identified in the plan

Sequence. Represents the situation in which the rules are geared to each other sequentially, i.e. the head of a rule corresponds to an atom in the body of another one. Since this indicates a sequential execution, the Sequence construct will be used. According to the specification, *Sequence* is a construct whose components are executed in a given order and the result of the last element is used as the result of the whole sequence (Fig. 3 a)).
Split and Split-Join. Represents the situation where two or more rules with different head atoms are grafted directly into two or more atoms in the body of a particular rule. In this circumstance there is a branch that is evaluated and encoded with a Slit or Split-Join construct. According to the specifications, Split-Join is a construct whose components are executed simultaneously, i.e. they run concurrently and with a certain level of synchronization (Split is the particular case where synchronization is unnecessary). The condition for using this construct requires that its components can overlap in the execution, i.e. they are all different (Fig. 3 b)).
Choice. Represents the situation where two or more rules, with the same head atoms, are grafted directly into one of the atoms in the body of a particular rule. In this circumstance there is a branch that is evaluated and encoded with the Choice construct. According to the specifications, Choice is a construct whose components are part of a set from which any element can be called for execution. This construct is used because the results from the rules set can easily overlap, no matter which component is going to be run because the results are always of the same type (Fig. 3 c)).
If-Then-Else. It could represent the situation where the body of a rule are the atoms identifying a precondition. In this case, the service that identifies the rule to be properly executed needs that its precondition be true. In this circumstance, therefore, the precondition was extracted and used as a condition in the If-Then-Else construct. According to the specifications, If-Then-Else construct is divided into three parts: the 'then' part, the 'else' part and the 'condition' part. The semantics behind this construct is to be understood as: "if the 'condition' is satisfied, then run the 'then' part, otherwise run the 'else' part." (Fig. 3 d)).
Any-Order. Represents a situation similar to the Split-Join, but this particular case covers those circumstances where control constructs or processes are present multiple times in the structure of the plan, and it is important that their execution does not overlap in order to prevent a process break. This type of situation can be resolved through the use of the Any-Order construct because its components are all performed in a certain order but never concurrently (Fig. 3 e)).

1. Search all Rulebeans grafted with a number of rules equal to zero ($n = 0$) (Figure 4 a)).
   a. Store therein an object that represents the "leaf", i.e. an executable process.
2. Search all Rulebeans grafted with a number of rules equal to one ($n = 1$) (Figure 4 b).
   a. Check the engaged list:
      i. If there is only one Rulebean, the node will be of type "Sequence";
      ii. If there are multiple Rulebeans, the node will be of type "Choice";
   b. Store the object representing the created structure in the Rulebean.
3. Search all Rulebeans grafted with a number of rules greater than one ($n \geq 2$) (Figure 4 c).
   a. For each grafted list follow the steps in 2.a;
   b. Make the following checks on the structure:
      i. If there are repeated Rulebeans add a node of type "Any-Order";
      ii. If there are no repeated Rulebeans add a node of type "Split-Join";
   c. Store the object representing the created structure in the Rulebean.

Since the If-Then-Else construct overlaps with the constructs assigned during this procedure, it is identified in another way. During the creation of a Rulebean, a check is performed to verify if there are atoms in the body of the rule labeled as belonging to a precondition. In such a case, the Rulebean will be identified as the 'Then' part of the construct, and the atoms of the precondition will form the 'If' condition. The 'Else' part will be identified as the complementary path, if any (for the "Sequence" construct it does not exist, of course). Finally, the data flow is implemented in accordance with the consumer-pull method, i.e. the binding of variables is held exactly at the point in which it occurs.

Table 2. OWL-S Atomic services test set

Ws NAME	TEXTUAL DESCRIPTION	INPUTS	OUTPUTS
Service-10	This service returns the information of a book whose title best matches the given string	books:Title	books:Book
Service-12	A book search engine	books:Title	books:Book
Service-15	This service informs about a person who works as co-publisher of a certain book	books:Book	books:Person
Service-28	This service returns the author of the given novel	books:Novel	books:Person
Service-9	This service returns the price of a book; person is an optional input	books:Person; books:Book	concept:Price
Service-37	This service returns the name of books given the publication number	univ:Publication	univ:Book

5 Experimental Analysis

In this section we present an example that shows the applicability of our method. The considered services were chosen from the OWLS-TC dataset [6].

5.1 SWRL Plan Generation

Let us consider the subset of atomic services in Table 2. To obtain SWRL rules that satisfy the requirements described in Section 4.1, we have modified the atomic services as follows:

- For every parameter having a datatype as type, we create a class in the domain ontology having a datatype property with the corresponding datatype as range. The considered dataset did not require the application of this step.
- For each service, we create two logical formulas. The former consists of unary atoms having the *parameterType* URI as a predicate and the input as an argument, for each input. The latter consists of a unary atom having the *parameterType* URI as a predicate and the output as an argument. We set these two formulas as the antecedent and the consequent of a new SWRL rule, respectively.

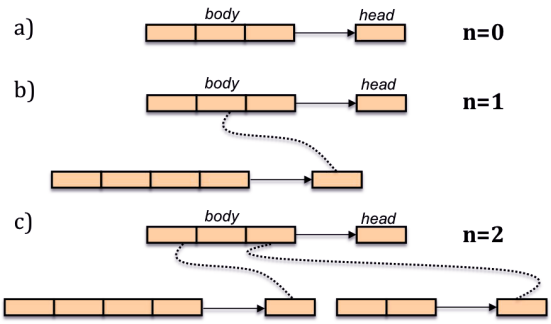


Fig. 4. The different types of grafted SWRL rules in the plan

[6] OWL-S service retrieval test collection,
`http://projects.semwebcentral.org/projects/owls-tc/`

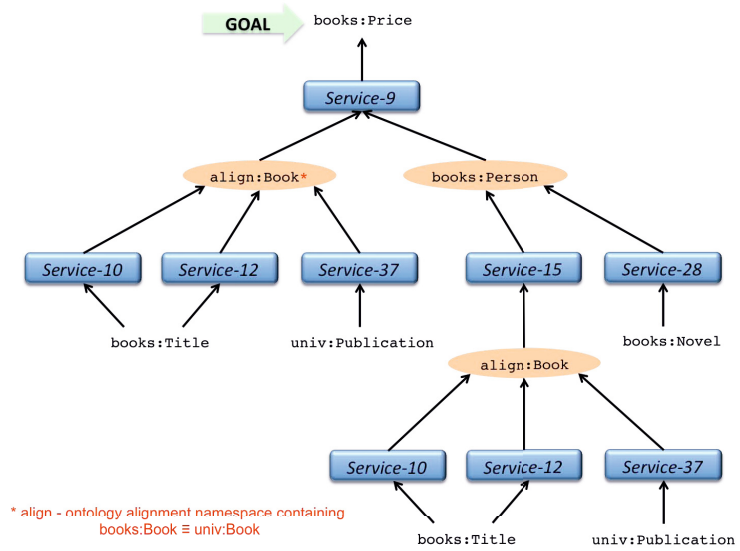


Fig. 5. The SWRL plan obtained by applying the composer to the services of Table 2

– Since every service produces its output by manipulating the inputs, we can suppose the existence of a *hasTransf* predicate, always true, that binds every input to the output, which guarantees the SWRL safety condition. Then we add *hasTransf* predicates to the antecedent of the rule built in the previous step. With this modification the antecedent can be identified with a new *inCondition*.

The SWRL rule set obtained in this way is given as input to the composer described in Section 4.2 and the resulting composition is shown in Figure 5.

5.2 OWL-S Composite Service Generation

After obtaining the composition plan, we apply the procedure described in Section 4.4. As a result we obtain the OWL-S constructs tree of Figure 6. In practice, we start searching Rulebeans grafted with a number of rules equal to zero, finding Service-10, Service-12, Service-37 (*books:Book* and *univ:Book* are equivalent classes) and Service-28. These will be tree leaves. We go on searching Rulebeans grafted with a number of rules equal to one, finding Service-15. It has two rulebeans grafted on the same Atom *books:Book*, thus we use a "Choice" construct. To link the obtained structure with Service-15 (another tree leaf) we use the "Sequence" construct, naming this structure $C1$. We continue serching Rulebeans grafted with a number of rules greater than one, finding Service-9. It has Service-10, Service-12 and Service-37 grafted on Atom *books:Book*, and Service-28 (another tree leaf) and $C1$ grafted on Atom *books:Person*. Both pairs are linked with a "Choice" construct, and we call them $C2$ and $C3$, respectively. Since $C2$ and $C3$ contain repeated Rulebeans (the "Choice" over Service-10,

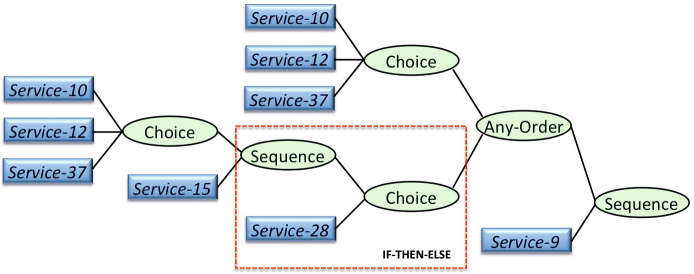


Fig. 6. The tree construct of the obtained composition

Service-12 and Service-37), we model this situation with the "Any-Order" construct rather than with the "Split-Join". The depicted "If-Then-Else" construct is obtained by applying the following consideration. Suppose that the precondition of Service-28 states that, in order to execute the service, the input must be necessarily a *books:Novel* (an OWL subclass of *books:Book*). Then, we can use this assertion as the 'If' condition, the execution of the service as the 'Then' part, and the non-execution of the service as the 'Else' part.

6 Conclusions and Future Work

The SWS frameworks proposed in the literature provide different support to dynamic orchestration. In the first part of this work we conducted a comparative study to elicit differences and analogies among them, and remarked the capabilities necessary to enable dynamic orchestration: the needed requirements and the suitability of OWL-S and WSMO to support such requirements. As shown in Section 2, the set of requirements is implied by means of the use cases, namely automatic discovery, selection, composition and invocation, required to make the SWS orchestration dynamic. The formal language underlying the SWS frameworks is the key for an effective realization of these use cases. For this reason, we described the formal support enabling reasoning on the semantic descriptions of the services offered by WSMO and OWL-S (based on OWL+SWRL and WSML, respectively). Then, we compared the formalisms underlying OWL+SWRL and WSML from the point of view of the expressive power actually exploitable for reasoning on service descriptions. As a result of this comparison, OWL-S turns out to be a more suitable candidate for *dynamic* orchestration. Automatic composition of SWSs is a more complex process to achieve using only tools built on Description Logics [1]. In Section 3 we presented a complete composition method for OWL-S services using Semantic Web languages and tools and working in semantically interoperable environments. The applied procedure can be summarized as follows: 1) The set of OWL-S atomic and simple (i.e., composite) services are represented by means of SWRL rules; 2) The SWRL composer is applied on them obtaining a plan of SWRL rules; 3) the SWRL plan is interpreted to produce the OWL-S composite service using OWL-S control constructs. The constructs identified in this way are used to build the OWL-S control

construct tree that is directly serializable using the syntax of the language. As a future work, it is important to find ways to manage the remaining constructs (Iteration) and improve the composer in order to reuse internal parts of composite processes during composition. Another aspect that deserves attention concerns a Semantic Web intrinsic issue, i.e. the absence of primitives for retracting knowledge due to the monotonic nature of DL knowledge bases. Finally, the implementation of software agents able to manage dynamic SWS orchestration by considering low-level details as, for instance, Quality of Service, Service Level Agreement and the coordination for the concrete Web services Invocation would allows a large-scale use. As in previous work, the emphasis will be placed on the exclusive use of technologies developed for the Semantic Web.

Acknowledgments. Special thanks to Luigi Iannone, Terry R. Payne and Ignazio Palmisano for working with us in the definition and refinement of the paper themes.

References

1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook. Cambridge University Press (2003)
2. Borgida, A.: On the relative expressiveness of description logics and predicate logics. Artificial Intelligence 82(1-2), 353–367 (1996)
3. Cabral, L., Domingue, J., Motta, E., Payne, T.R., Hakimpour, F.: Approaches to Semantic Web Services: an Overview and Comparisons. In: Bussler, C., Davies, J., Fensel, D., Studer, R. (eds.) ESWS 2004. LNCS, vol. 3053, pp. 225–239. Springer, Heidelberg (2004)
4. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press, New York (1972)
5. Euzenat, J., Shvaiko, P.: Ontology matching. Springer (2007)
6. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: combining logic programs with description logic. In: Proceedings of the 12th International Conference on World Wide Web, pp. 48–57. ACM Press, New York (2003)
7. Horrocks, I., Patel-Schneider, P.F., Bechhofer, S., Tsarkov, D.: OWL rules: A proposal and prototype implementation. J. of Web Semantics 3(1), 23–40 (2005)
8. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From SHIQ and RDF to OWL: the making of a Web Ontology Language. J. Web Sem. 1(1), 7–26 (2003)
9. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer (1987)
10. McIlraith, S.A., Son, T.C., Zeng, H.: Semantic Web Services. IEEE Intelligent Systems 16(2), 46–53 (2001)
11. Motik, B., Sattler, U., Studer, R.: Query Answering for OWL-DL with rules. Journal of Web Semantics: Science, Services and Agents on the World Wide Web 3(1), 41–60 (2005)
12. Peltz, C.: Web Services Orchestration and Choreography. Computer 36(10), 46–52 (2003)
13. Redavid, D., Ferilli, S., Esposito, F.: SWRL Rules Plan Encoding with OWL-S Composite Services. In: Kryszkiewicz, M., Rybinski, H., Skowron, A., Raś, Z.W. (eds.) ISMIS 2011. LNCS, vol. 6804, pp. 476–482. Springer, Heidelberg (2011)
14. Redavid, D., Iannone, L., Payne, T.R., Semeraro, G.: OWL-S Atomic Services Composition with SWRL Rules. In: An, A., Matwin, S., Raś, Z.W., Ślęzak, D. (eds.) ISMIS 2008. LNCS (LNAI), vol. 4994, pp. 605–611. Springer, Heidelberg (2008)