Web Service Composition Plans in OWL-S

Eva Ziaka, Dimitris Vrakas, and Nick Bassiliades

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

Abstract. One of the main visions of Semantic Web has been the ability of software agents to compose atomic web services in order to facilitate the automation of complex tasks. One of the approaches used in the past in order to automatically construct composite web services has been AI planning. The most important advantage of this approach is its dynamic character that reduces the interference of the user. Although there have been various attempts to utilize planning algorithms and systems in the composition process, there has been little work in the field of converting web service composition plans in OWL-S. This paper studies the use of two well established standards in expressing plans and composite web services, namely the Planning Domain Definition Language (PDDL) and the Ontology Web Language for Services (OWL-S) and suggests a method for translating the produced PDDL plans of any planning system to OWL-S descriptions of the final composite web services. The result is a totally new web service that can later be discovered and invoked or even take part in a new composition.

Keywords: Web services composition, AI planning, Semantic web services, OWL-S, PDDL.

1 Introduction

Nowadays, many different systems all over the globe can communicate with each other through the Internet. The need for supporting interoperability of web applications so that they can be used by all platforms, no matter their implementation, has led to web services technology and a new, web-service-oriented way of programming. This new technology is based on open protocols, such as the XML and the well known HTTP transfer protocol.

There is often the need to execute more complex tasks that simple web services do not have the potential to complete on their own. In such cases, simple web services must cooperate so as to combine their functionalities to create a new complex web service that will hold the desirable functionality. Semantic information about all the available atomic web services is very important for their cooperation in web services composition field, so as to be able to understand the meaning of their inputs and outputs and to match them to achieve cooperation.

During the past decade a large number of approaches for composing web services have been proposed, some fully automated, other partially automated, whereas a lot of them are even completely manual. A promising way that aims at fully automated web services composition is the use of AI planning technology. Each web service is represented as a planning operator and the desired composite service's inputs and outputs form the initial state and the goals respectively. The plans that arise are encoded in languages such as PDDL [5] that describe the actions, that is the web services, that must be executed and the order of their execution.

The contribution of this paper focuses on the automatic translation of the plans, expressed in PDDL, to OWL-S descriptions [10] that take advantage of the OWL-S control constructs and facilitate the automatic invocation of the composite service. Specifically, information from the PDDL descriptions of the domain, the composition problem, and the plan is used to create a functional representation of the composition. This representation describes with a specific syntax the way each atomic web service is connected to each other in order to produce the final output. In a second phase, this functional representation is utilized to generate the OWL-S descriptions of the new composite web service.

In terms of functionality, the method described in this paper is merely based on the PDDL descriptions of the planning operators and does not explicitly deal with semantic information of the initial atomic services. Therefore, it can be applied to compositions arising from both syntactic and semantic matching of inputs and outputs of the atomic services. However, since the final expression will be encoded in OWL-S language, we will use the notion of semantic web service throughout the rest of the paper.

In the sections to follow, the relative research field is explored. The suggested technique is analyzed in detail and some conclusions along with future directions are given. Specifically, the rest of the paper is organized as follows:

In section 2, the field of automated web services composition using AI Planning techniques is presented and some studies on the field are exposed. In section 3, the developed method for translating the PDDL plans to OWL-S descriptions is analyzed. This section is divided into two sub-sections, reflecting the two phases of the method; in the first sub-section, the algorithm that creates the functional representation describing the composition is presented, whereas in the second sub-section, the method for converting this representation to OWL-S description is described. Finally, in the last section, conclusions of the research so far are given along with some ideas on how the developed algorithms and the web services composition procedure could be enhanced.

2 Related Work

The process of automated web services composition by the point of view of planning has been studied extensively. The most important advantage of this approach is the dynamic character that is offered to the composition process, which reduces a lot the interference of the user.

One of the most known systems in the field of web services composition via planning is SHOP2 (Simple Hierarchical Ordered Planner), [15]. It is based on HTN planning (Hierarchical Task Network) methods [14]. One basic difference between SHOP2 and the other HTN systems is that it locates all the actions of the plan in the same order that they will be later executed. In this way, the current state of the system in every step of the planning procedure is known and inference mechanisms or heuristic techniques can be used to augment the effectiveness and the efficiency of the whole process.

The functionality of SHOP2 consists of three basic steps. In the first one, the domain is constructed by the process OWL-S files of the available web services. The atomic services are represented by operators and methods for analyzing the complex services to simpler ones are constructed. In the second step, the composition problem is transformed to planning problem. This is realized by describing the problem as an abstract composite process that need decomposition with the use of methods so as to obtain simple processes that refer to web services. In the third step, the problem is solved by decomposing the tasks and creating the plan, i.e. is the description of the final composite service.

Another technique, analyzed in [12], is based on situation calculus, where the states are not considered as instances of the environment but as sequences of actions that were executed in the past and resulted to this state. This technique uses also the language Golog (alGOL in LOGic), which is based on logic and the problems that are encoded in it can be solved by methods that use logic. For the appropriate representation of the planning problem in Golog, the language was extended so as to be able to contain constraints on the composition process defined by the user. These constraints in essence reflect the desired outputs. The OWL-S descriptions are used as requirements of the processes that must be executed and also as descriptions of the actions that are provided by the web services. The composition problem is transformed into the problem of finding the appropriate Golog program that when executed, all the defined constraints will be satisfied. In the solution process, intelligent agents are used whom knowledge base contains the preconditions and the results of the services, encoded in situation calculus terms. The available web services correspond to operators, primitive or composite. The role of the agents is the inference on the web services, in order to discover, execute and compose them.

A different and quite simple web services composition method is presented in [18]. It is based on regression in a state space. The algorithms belonging to this category start from exploring the goals that must be succeeded and seek for the actions that lead from the goals to the initial state. The method proposed introduces a new structure called SLM (Semantic Links Matrix) and is a table containing the values of semantic relevance between the parameters of the web services. For the construction of this table, the process models and the relative ontologies of the atomic services are used. Generally, the SLM structure groups the candidate web services based on their semantic relevance and in the same time provides information on their quality characteristics so as to ease the choice among them. The algorithm begins from the goals, but because of the SLM structure it does not need to calculate the previous states. In the step of locating the actions that satisfy the current goals, all the services that have a positive value in the relevance function are considered as candidates. The best service is chosen based on the QoS characteristics. The process continues until it reaches the initial state.

Another approach described in [17] uses model checking techniques for producing the plan. The algorithm consists of four steps. In the first step, the goal and the initial states are defined. In the second step, the model of the process on which the checks will be running is extracted. The web services that could be used for the domain are automatically detected and the state space where the solution is searched is constructed. Information on the services is retrieved by the ontologies and is inserted to the model. In the third step, the search algorithm in the plan space is executed and some plans that satisfy the goals are collected. In the fourth and last step, the best plan is chosen and is converted in a composite web service, encoded in BPEL.

A system which was developed recently and is analyzed in [6] is the system PORSCE. The approach is based on transforming the web services composition problem to a planning problem. The straight forward mapping of these two fields is exploited and the OWL-S descriptions of the available web services are used to construct PDDL plan files. The initial state is derived by the data given as input to the final web service by the user, whereas the goals are reflected by the desired outputs. The operators of the problem correspond to the available atomic web services that can be used. Their preconditions are mapped to the inputs of the services and theirs results to the outputs. Simultaneously, the ontologies that are connected to the types of the parameters of the available web services are used so as for the semantics of the concepts to be provided. The system starts by representing the composition problem with planning terms. Then, a solution to the problem is provided by an external planner, such as LPG-td [3], [4] or JPlan [8], according to the user's selection. Finally, the quality of the produced plan is measured based on some quality measures selected by the user at the beginning of the process and the results are provided to the user. There is also the possibility of replacing instantly some of the web services in the plan with other relevant, as they are discovered during the planning process.

Another approach that exploits the similarities between the AI planning and semantic web services composition research fields is the OWLS-Xplan [9]. This system uses the OWL-S descriptions of the available web services, the relevant OWL ontologies that define the types of the parameters in the descriptions and a planning query as input. After some preprocessing of the above data and the execution of the Xplan planning algorithm, the result is a plan describing the sequence of composed services that satisfies the goals.

The OWLS-Xplan approach consists of two basic parts. The first one is an OWLS2PDDL converter which converts the OWL-S descriptions along with the OWL ontologies to the equivalent PDDL domain and problem of the composition. Specifically, the conversion results to descriptions of the domain and problem in a XML dialect of PDDL (developed by the authors), referred to as PDDXML, that simplifies parsing, reading and communicating the descriptions using SOAP. An atomic operator is directly related to a service profile as they both provide a general description of their instances, actions and web services, respectively. A complex action can be linked to a service model that describes how simpler actions should cooperate to result to the composite one. Finally, the methods used in HTN planning are related to composite web services and may be used by the planner as a hierarchical task network during the planning process.

The second part of OWLS-Xplan is the developed heuristic hybrid Xplan AI planner that combines the benefits of the action-based FF-planner [7] with HTN planning. Xplan always finds a solution, if it exists in the state space, over the space of possible plans, in contrast to HTN approaches. It combines guided local search with graph planning and a simple form of hierarchical task networks to produce a plan. Also, a re-planning component is included to improve flexibility is cases changes happen in the world during planning, a property well needed in semantic web services composition field.

The solution analyzed in [16] also translates the composition problem to PDDL descriptions and suggests that in this way an appropriate planner could be found each time according to the problem so as to provide an improved solution. The paper presents a three step technique for the creation of a composite web service with the first step being the translation of the OWL-S descriptions and OWL ontologies to PDDL domain and problem descriptions; the second one is the creation of a plan that solves the problem with the execution of a planner; the third one is the translation of the plan to a new OWL-S description of the resulting composite web service. However, the paper focuses only on the first step of the procedure. Some assumptions are made to ease the translation function, such as considering that each atomic process has either effects or outputs but not both simultaneously. Also, the authors of the paper do not deal with OWL-S process models that have composite process using Repeat-While and Repeat-Until or Any-Order and Split-Join constructs. The algorithm proposed, deals separately with the OWL-S process model, the atomic and simple processes, the sequence, if-then-else, choice and split processes and with the OWL-S target service description to create the domain and problem descriptions. The process of choosing the appropriate planner for each problem and the translation of the plan to OWL-S description of the new service are not elaborated in the paper.

The aforementioned methods tackle the problem of web services composition using a variety of fully or partially automated techniques. However, they don't deal with the task of expressing the resulting composite service in OWL-S, taking advantage of the supported control constructs.

3 Translating PDDL to OWL-S

This section analyzes the method for translating a composite web service expressed in the PDDL language to the corresponding OWL-S description. The translation completes in two phases. The first one concerns the extraction of all the required information from the plan for the creation of a composite web service's functional representation. The second is about the conversion of this representation to an OWL-S description of the resulting composite web service.

3.1 Constructing the Composite WS

The first step in the creation of an OWL-S description based on data derived from a PDDL plan is the manipulation of these data and their conversion to a composite web service functional representation. This representation refers to the available simple or atomic web services and the order in which they should be executed and is structured using the OWL-S control constructs *sequence*, *split* and *split-join*.

In the following algorithm the functional representation of a composite web service C is represented as a predicate $f(a_0,a_1,...,a_n)$, where f is the control construct used to describe the composition structure and $a_0,a_1,\dots,a_n$ stand for the simple web services that participate in the composition. Each a_i could be another composite service or, in a simpler case, an atomic process, which is represented as $atomic(a_i)$.

The developed algorithm consists of three general steps, as shown in **Fig. 1**. The first step concerns the parsing of the files associated with the composition planning problem and the extraction of all the information needed in the next steps. In the second step, a web service composition graph is created. The nodes of the graph are the actions of the plan and the edges are the links that express the order constraints among the actions. The creation of the graph is based on the information collected from the previous step. Finally, in the last step, the composite web service functional representation is formed using the ordering constraints that are extracted from the composition graph. In the following paragraphs, these three steps are described in more detail.

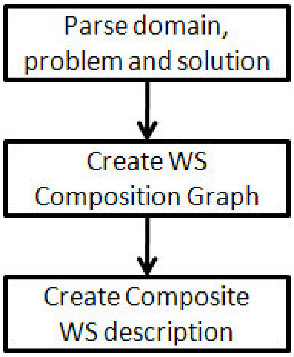
Parse domain, problem and solution
Create WS Composition Graph
Create Composite WS description

Fig. 1. Converting a PDDL plan to a composite web service functional representation

The initial available information is derived from the PDDL domain and problem files of the composition problem. For the parsing of these files, an external library, called PDDL4J, [13] is used. The types of information that are required by the translation process are the following: a) the name of the operator, b) the parameters list, c) the preconditions list, d) the effects list, e) the initial state and f) the goals of the problem. Finally, the resulting plan is parsed in order to extract information concerning the actions of the plan. Exploiting the syntax of this file, information on the actions used can easily be extracted. The data that are needed in the later steps of the algorithm involve the timestamp of each action, which is the time step when the action will be executed and the name, parameters and duration of it. The actions are read in the order that they are presented in the plan, so the procedure keeps track of this order.

When all these data are retrieved, the procedure continues combining them so as to create objects representing the steps of the plan. Every step contains the name of the action that will be executed, the parameters with which the action is called, the timestamp and duration of the action, the operator from which the action is derived, the substitution imposed on the operation, the list of preconditions that must hold for the action to be executed and the list of the effects, the facts that will change due to the execution of the action.

The second step creates the web service composition graph. The nodes list is identical to the list of actions of the plan. In essence, the contribution of this step is the computation of the edges, that is, the links between the actions. The general idea is to traverse all the actions and locate cases where one precondition of an action matches one effect of another. This ought to happen in theory because of the causal links that are present among the actions of the plan, which imply that the preconditions of the later actions will appear as effects of other previous actions. An order constraint link is then created between the two actions.

Algorithm 1 (Graph): Computes the web services composition graph
Inputs: P = {a_0,a_1,…,a_n}, the plan
Output: G = (P,E), web services composition graph

```
E = ∅
for i = n down to 1
   for each c ∈ prec(ai)
      for j = i-1 down to 0
         for each p∈ add(aj)
            if (c = p)
               E = E U {(aj,ai)}
return G = (P,E)
```

The algorithm that discovers such kinds of links is called *Graph* and starts from the last elements of the action list. Each one of its preconditions is then examined so as to discover a previous action in the plan that produces this fact. This means to discover an action that contains this fact in its effect list. So, another loop is needed to access all the previous possible producers of this imminent link. When such a previous action is found, a link is created among the two actions. This link illustrates an order constrain and ensures that the action that produces the fact will be executed before the one that consumes it in its preconditions list.

A simple example of the above procedure is depicted in **Fig. 2**. In this example there are two actions in the plan, the actions *Drop Ball B* with which a robot puts down the ball B and the action *Grab Ball A* that results in a state where the robot is holding the ball *A*. The algorithm examines first the action *Grab Ball A* and loops on its preconditions. In this case there is only one precondition, declaring that for executing this action, the robot's gripper must be free. So, somewhere in the plan there should be an action that realizes this fact. Exploring the previous actions of the plan, the algorithm confronts the action *Drop Ball B* and matches the fact under consideration with the second result of this action. Automatically, an order constraint link is created between the two actions meaning that the robot should definitely perform first the action *Drop Ball B* so as to be able then to perform the action *Grab Ball A*.

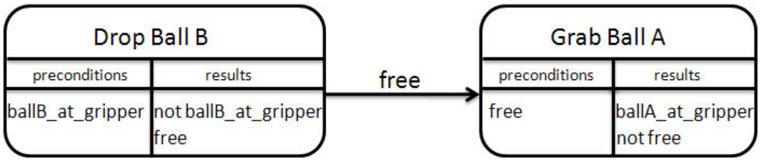

Fig. 2. Example on discovering links

When all the edges and the corresponding order constraints are discovered in the plan, the procedure can continue and exploit these relationships in order to construct a composite web service functional representation that illustrates in a more formal way how the actions of the composite service relate to each other. This representation is built upon the control constructs that OWL-S uses to describe the different possible connections between web services. In the algorithm we use three basic control constructs: *sequence*, *split* and *split-join*. The control *sequence* declares that all its members should be executed in the exact order they appear. The control *split* is used to describe cases of parallel execution of web services. The last control, *split-join*, describes the case where a split occurs in the plan and the parallel executions connect again in a next step in one web service. It is important that the web services that happen to be last in the parallel executions, have to synchronize their outputs to supply the web service following the connecting point with the sufficient inputs.

The general algorithm that constructs the composite web service's functional representation consists of 2 basic steps, presented in Algorithm 2 (*Basic*) and Algorithm 3 (*Join*). Before the execution of these algorithms, a manipulation of the data gathered so far is needed. First, the order constraints list is reduced by removing all the constraints not needed. Then the algorithm *Basic* is called, locates the web services that will be invoked first and creates functional representations of the sub-compositions that start from these services. All these representations are then added to an empty split control. Up to this point, the first version of the requested functional representation is ready. But some refinement steps should be performed in order to provide a more concise representation. So, next in the developed algorithm, a process named *Join* takes place and simplifies the functional representation by replacing *split* controls with *split-join* where possible. The generated functional representation of algorithm *Basic* contains null expressions and unnecessary controls, such as a split control with only one parameter. In the following paragraphs a more detailed description of the translation procedure is provided.

The output of *Graph* algorithm may contain some unnecessary ordering constraints, so the first step is about locating such constraints and removing them from the set. Unnecessary constraints are the ones that can be implied by others, so there is no need for their existence in the set. One order constraint A can be inferred by others if there exists another constraint B with the same left part as A and a constraint C whose left part is identical to the right part of constraint B and its right part is identical to the right part of constraint A. An example will clarify more the above situation. Let the set $\{A<C, A<B, B<C\}$ be the set of constraints of the composition problem. Examining the need of existence of the first order constraint, which is interpreted as 'the web service A must be executed before the execution of the service C', the constraint $A<B$ has the same web service at the left part. The process continues by exploring the set for constraints that have service B in the left part, because this is the right part of the constraint $A<B$. Such a constraint exists and is the third of the set. Also, this constraint has identical right part with the first constraint that is examined in the process. This means that the constraint $A<C$ is unnecessary because it can be inferred by the constraints $A<B$ and $B<C$, so it is removed from the set.

The next procedure that takes place is the *Basic* procedure, shown in Algorithm 2. The first step of this algorithm is the location of the so called 'clear' services, the web services that are executed first in the plan. The main characteristic of these services is that they are not consumers in any causal link, which means that there is no need for another web service to be executed before them. Such services can be located by searching for the existence of each web service in the plan as a right part of an order constraint. If this search returns no results, then the service can be marked as “clear”. For example, having the set of web services {*A, B, C*} and the order constraints {*A*<*B, B*<*C*} it can be easily inferred that only the service *A* is clear, because it does not appear as a right member of any order constraint. For each clear web service, the construction of sub-representations of the desired composition takes place. In essence, the relationship among a clear web service and all its children, all the services that can be executed after the completion of the clear service, is revealed.

Algorithm 2 (Basic): Computes an initial composite service with Sequence and Split constructs
Inputs: G = (V,E), the web service graph
Output: C, a composite service

```
// R is the set of root nodes in G
set R ← {r∈V: ∀x ∈ V, (x→r)∉E }
if |R| = 0 then return NULL
if |R| = 1 then
   set G'← the tree in G with r∈R
            as the root
  return sequence(r, Basic(G'-{r}))
set c ← {}
for each r in R
   set G' ← the tree in G with r∈R
             as the root
   set c ← c ∪ Basic(G'-{r})
return Join(split(c))
```

In the next steps of the algorithm *Basic*, the number of clear services is examined. In the trivial case, where there are no such services, a null value is returned. If there is only one clear service, then the only representation that can be constructed is a simple sequence of the clear service and the composition of the child. So in this point, the algorithm calls recursively itself with the rest of the graph as a parameter. This is because the expression beginning from the clear service must contain all the information about the expressions that can be built from the children of this service.

If there are more than one clear services, then an empty composite web service is created and for every clear service the *Basic* procedure is invoked having as parameter the *Graph* without the service in question. All the returned functional representations are then added to a *split* control. The resulting *split* expression is simplified by an algorithm that will be analyzed later in the paper. A short example is given to clarify the procedure. Suppose there are a clear service *A* and two children *B* and *C*. The functional representation returned from the algorithm, in terms of control constructs, will be *seq*(*A*,*split*(*Basic*(*B*),*Basic*(*C*))). Supposing that there are no other web services in the plan, the final result will be *seq*(*A*,*split*(*B*,*C*)).

Next, the composition representation that resulted from the clear services (algorithm *Basic*) is simplified by the algorithm *Join* (Algorithm 3). The main function of this algorithm is to replace the *split* controls with *split-join*, wherever this is possible. In every step, two parameters of the functional representation are examined for the existence of a common part. If one such part is found, it is removed from both the parameters and the results are added to a new *split-join* relationship. Finally, a new *sequence* control is created, the *split-join* is added as the first parameter and the common part is added as a second parameter.

Algorithm 3 (Join): Replaces split with split-join where possible in a composite service
Inputs: $C=f(a_0,a_1,\ldots,a_n)$, a composite service with sequence and split constructs
Output: C, a composite service with sequence, split and split-join constructs

```
do
   for each ($a_i$,$a_j$): i,j in [0,n]
      Set L($a_i$,$a_j$) = 0
      if $a_i = a_i' \cup k$, $a_j = a_j' \cup k$ then
         L($a_i$,$a_j$) = |k|
   $(a_x, a_y) = \arg\max_{(a_i,a_j)} (L(a_i,a_j))$
   $L_{xy}$ = max($L_{ij}$)
   if $L_{xy}$ > 0 then
      Let $f_{ax}(a_{x0},a_{x1},...,a_{xn})$ the
         construct containing k in $a_x$
      Let $f_{ay}(a_{y0},a_{y1},...,a_{yn})$ the
         construct containing k in $a_y$
      $k_1=k_2=k$
      if $f_{ax}$ = split then
         $k_1 = f_{ax}(a_{x0},a_{x1},...,a_{xn})$
      if $f_{ay}$ = split then
         $k_2 = f_{ay}(a_{y0},a_{y1},...,a_{yn})$
         C= C-{$a_x$,$a_y$}
         $C=C\cup seq(s+j(a_x',a_y'),s(k_1,k_2))$
while $L_{xy}$ > 0
return C
```

For each pair (a_i,a_j) of parameters, the size of their common part is stored in the structure $L(a_i,a_j)$. The size of x is expressed as $|x|$ and refers to the number of simple web services that take part in the functional representation of x. When all the pairs are traversed, the one with the largest common part is selected, that is the pair (a_x,a_y). If the size is a positive number, then the next step checks whether the common part is in a *split* control in the two parameters of the selected pair. If so, the *split* expression must not be divided instead it should be completely removed.

Since this procedure is performed twice, once for every parameter of the couple, the results are two new common parts that should be removed respectively from the parameters. This is realized in parameters a_x' and a_y'. The resulting expressions are added as members of the *split-join* control, symbolized as '*s+j*', which in turn is added as a parameter of the sequence control. Then, the common parts are combined in a *split* control, symbolized as '*s*' and the result becomes the second parameter of the *sequence* control. Finally, this new *sequence* representation replaces the two parameters in the initial composite web service, a_x,a_y. All the previous steps are repeated for the altered composite web service *C* until no common part exists between its' parameters. Then, *C* is returned, as was formed from the procedure and represents a composition having *sequence*, *split* and *split-join* control constructs that functionally represents the data flow among the participating simple web services.

After the completion of *Join*, the null parameters of the functional representation created so far are cleared and the pointless control constructs are removed, e.g. the expression *split*(*A*) becomes *A*. Finally, the duplicate references to control constructs are eliminated This means, that the expression *seq*(*seq*(*A*,*B*),*C*) is transformed to the equivalent one *seq*(*A*,*B*,*C*).

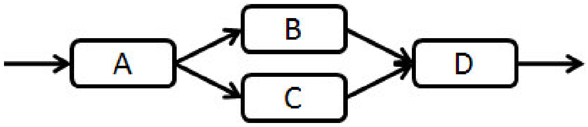

Fig. 3. Composition example

A short example of the whole procedure is given to clarify its workings. In **Fig. 3** a web services composition plan is depicted in a graphical way. The clear service is only the service *A*, so the result of the *Basic* algorithm, before calling the algorithm *Join*, will be *seq*(*A*,*split*(*seq*(*B*,*D*),*seq*(*C*,*D*))). The *Join* algorithm will notice that the parameters *seq*(*B*,*D*) and *seq*(*C*,*D*) have the service *D* as a common part, so the *split* control construct can be replaced by a *split-join* one. By removing the common part from each parameter, the results are the representations *seq*(*B*,*null*) and *seq*(*C*,*null*) and they are added as parameters in a new *split-join* control. Since the common part is not in a *split* expression in none of the two parameters, the resulting common part is just the service *D* and the new *sequence* representation is constructed as follows: *seq*(*split-join*(*seq*(*B*,*null*),*seq*(*C*,*null*)),*D*). This representation replaces the *split* of the initial expression and the result is the representation *seq*(*A*,*seq*(*split-join*(*seq*(*B*,*null*),*seq*(*C*,*null*)),*D*)).

After the completion of the clearing algorithm the functional representation is transformed to *seq*(*A*,*seq*(*split-join*(*B*,*C*),*D*)) which finally becomes *seq*(*A*,*split-join*(*B*,*C*),*D*) at the last step, which is an accurate functional representation of the composition.

3.2 Creating OWL-S Descriptions

Up to this point, a functional representation has been constructed that supplies sufficient information on the data flow of the composition. But, for the procedure to be complete so as to provide the user with a new semantic web service ready for execution, the OWL-S description has to be constructed. This is done based on this representation. The descriptions that are constructed by the algorithm are the process and the profile descriptions. The OWL-S API, which can be found at [11], was used for their creation. This OWL-S API is a JAVA library providing functions that facilitate the creation of OWL-S descriptions.

First, the process file is created by the algorithm 4, *OWLSProcess*. The algorithm takes as input parameter the composite web service representation *C*, as formed by the previous algorithms and discerns two cases. If *C* is an atomic service, then the appropriate parts of the OWL-S process description are created that describe the service along with its inputs and outputs. Specifically, for every input of the atomic service, an input element is created by calling the *InputElement* function of the OWL-S API. All the input elements are gathered in a list which is then set as the value of the *hasInput* field of the OWL-S process description. The same steps are followed for the creation of the output list which is the value of the *hasOutput* field in the description.

Algorithm 4 (OWLSProcess): Creates the OWL-S process description
Inputs: $C = f(a_0,a_1,..a_n)$
Output: The OWL-S process description of C

```
if f = atomic then
   A = OWLSAPI.AtomicProcessElement
   LI = LO = {}
   for each $p_i \in$ prec($a_0$)
      $k_i$ = OWLSAPI.InputElement($p_i$)
      LI = LI + {$k_i$}
   OWLSAPI.hasInput(LI)
   for each $o_i \in$ add($a_0$)
      $m_i$ = OWLSAPI.OutputElement($o_i$)
      LO = LO + {$m_i$}
   OWLSAPI.hasOutput(LO)
   PE = OWLSAPI.PerformElement
   return PE.add(A)
else
   CC = OWLSAPI.ControlContruct(f)
   CC.add(CLO(C))
   return CC
```

If *C* is not just an atomic service, but instead a composite one, then the appropriate control construct element is created (*seq*, *split*, *split-join*) according to *f* and the algorithm *CLO* is called to create the list of the services that takes part in this element. Then, this list is added to the control construct element and this is the object that the *OWLSProcess* algorithm returns. In fact, this object contains all the information about the OWL-S process description of *C*.

Algorithm 5 (CLO): Creates the List Object containing the atomic services of the composite one
Inputs: $C = f(a_0,a_1,..a_n)$
Output: LO: the List Object

```
if n = 0 then
   return null
LO = OWLSAPI.ListObjectElement
LO.First = OWLSProcess($a_0$)
LO.Rest = CLO(f($a_1$,$a_2$,…,$a_n$))
return LO
```

The algorithm *CLO* has as input a composite web service functional representation, which is in essence a functional representation with OWL-S control construct connecting the participants services, and creates using the OWL-S API a *List Object* element with the atomic services as parameters. The *list object* is a structure with *First* and *Rest* parts and could be described by an expression like: *First*(a_0,*Rest*(*First*(a_1),*Rest*(…))).

In *CLO* algorithm, the first parameter of the expression is examined and the *OWLSProcess* algorithm is called for this. The result becomes the head of the constructing list, because it is the service or the composition of services that will be executed first. Then, the *CLO* algorithm is called recursively for *C'*, the composite web service *C* with a_0 omitted. The result of this call is set as the *Rest* part. Finally, the constructed list object is returned.

The last step in converting the composite web service functional representation to OWL-S description is the creation of the profile description. Here, the composite web service is treated as an atomic service with specific inputs and outputs. The construction of this description is merely based on the methods provided by the OWL-S API's functions.

4 Conclusions and Future Work

Web services are playing an important role in the web applications development field, with which many different systems through the globe can communicate and exchange data using the World Wide Web. Users that need a specific functionality can retrieve the desired web services from the UDDI registries and use them to create the output they are looking for.

SOA architecture has contributed to the rapid and easy web applications development, using as units the web services and combining them to create new, complex services of advanced functionality that can serve even as complete business models. The composition methods studied in this paper differ on user's involvement level. Some initial solutions, of limited autonomy, use workflows and leave the details regarding the location the appropriate services execution and their order to the user. In some more creative solutions, the user doesn't have to find the exact services that will be used, but just provides a description of them. The discovering of services that match with the descriptions and the execution of the resulting workflow are automatically performed without the intervention of the user.

In later studies, the autonomy of the composition procedure is increased. Semantic information concerning the web services is used to describe in a semantic level their functionality. Languages such as OWL-S are used for this purpose. In this way, concept matching becomes possible and so is the check whether two or more services can cooperate. The semantic information is used also by automatic web services composition via planning methods, which are examined in this paper. The composition problem is treated as a planning problem and solved by algorithms of the field.

The result is a plan encoded in planning languages, such as PDDL+ that describes the services that will be used for the composition and the way in which they will be combined to create the desired composite web service. But, for this final service to be available to other users too and to be published in a UDDI registry as an atomic web service and take part to possible future compositions, semantic description of the service have to be created.

The contribution of this paper focuses on converting the PDDL+ plans that constitute the composite web service to OWL-S descriptions of the new web service. Information extracted from the domain of the composition problem is used to construct a composite web service functional representation that describes sufficiently the composition. Then, this representation is used to create the OWL-S profile description of the composite web service, containing information on its inputs and outputs. Also, the OWL-S process description is constructed, that analyzes the way the atomic services are used for the production of the final composite web service.

As for future plans, a complete system could be developed as an extension to the already existing automatic web services composition systems, taking advantage of the algorithms proposed by this paper to construct new semantic web services and publish them in UDDI registries so as to be available to everyone who could be seeking such functionality. In this way, an integrated solution to the composition problem would be provided. Already developed solutions could be used to this direction, such as the system SiTra described in [2], which transforms the OWL-S description of a web service to BPEL, the execution language for web services.

Also, the possibility of creating the grounding OWL-S descriptions of the composite web service could be explored. In this description, the exact data flow among the atomic services will be described and the result will be an even more automated solution. So far, our approach provides the order and the way of the execution of the services taking part in the composition. However, the information of which output is offered as input to the next service is not provided from the OWL-S descriptions of the composite service. This procedure is left to the system that tries to execute the resulting service. It is obvious that by providing this kind of information through the grounding description, the development of systems that execute complex services is greatly simplified.

Moreover, characteristics concerning the quality could be considered for the composite web service. In case there is such data in the semantic descriptions of the atomic web services, procedures that take advantage of them could be developed to construct the quality characteristics of the resulting composite service.

Finally, we aim at integrating web service composition via planning into a decision support system for industrial risk reduction, which represents risk case studies via domain dependent ontologies including the mechanism for building up the risk as a composition of simple physical processes [1].

Acknowledgements. This work has been supported by the Project "Integrated European Industrial Risk Reduction System (IRIS)" (7th Framework Programme, Theme: 4 – NMP, FP7-NMP-2007-LARGE-1, CP-IP 213968-2).

References

1. Angelides, D., Xenidis, Y.: Fuzzy vs. Probabilistic Methods for Risk Assessment of Coastal Areas. In: Linkov, I., Kiker, G.A., Wenning, R.J. (eds.) Environmental Security in Harbors and Coastal Areas: Management using Comparative Risk Assessment and Multi-Criteria Decision Analysis. NATO Security through Science Series (Series C: Environmental Security), pp. 251–266. Springer, Heidelberg (2007) ISBN: 978-1-4020-5801-1

2. Bordbar, B., Howells, G., Evans, M., Staikopoulos, A.: Model Transformation from OWL-S to BPEL Via SiTra. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 43–58. Springer, Heidelberg (2007)
3. Gerevini, A., Saetti, A., Serina, I.: LPG-TD: a Fully Automated Planner for PDDL2.2 Domains (short paper). In: 14th Int. Conference on Automated Planning and Scheduling (ICAPS 2004), booklet of the system demo section, Whistler, Canada (2004)
4. Gerevini, A., Saetti, A., Serina, I.: LPG-td a planning system (2005), `http://zeus.ing.unibs.it/lpg/`
5. Ghalab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL – the Planning Domain Definition Language. Technical report. Yale University, New Haven, CT (1998)
6. Hatzi, O., Meditskos, G., Vrakas, D., Bassiliades, N., Anagnostopoulos, D., Vlahavas, I.: Semantic Web Service Composition using Planning and Ontology Concept Relevance with PORSCE II. In: Proceeding of the 2009 Web Intelligence and Intelligent Agent Technology, Milan, Italy, pp. 418–421 (2009)
7. Hoffman, J., Nebel, B.: The FF Planning System: Fast Plan Generation Through Heuristic Search. Journal of Artificial Intelligence Research 14, 253–301 (2001)
8. JPlan: Java Graphplan Implementation, `http://sourceforge.net/projects/jplan`
9. Klusch, M., Gerber, A., Schmidt, M.: Semantic Web Service Composition Planning with OWLS-XPlan. In: Proceedings of the AAAI Fall Symposium on Semantic Web and Agents. AAAI Press, Arlington (2005)
10. Martin, D., Burstein, M., Lassila, O., McIlraith, S., Narayanan, S., Paolucci M., Parsia, B., Payne, T., Sirin, E., Srinivasan,N., Sycara, K.: OWL-S: Semantic Markup for Web Services (2004), `http://www.daml.org/services/owl-s/1.1/`
11. OWL-S API, `http://www.daml.ri.cmu.edu/owlsapi/`
12. Peer, J.: Web Service Composition as AI Planning – a Survey. Technical report. University of St. Gallen (2005)
13. Pellier, D.: PDDL4J (2008), `http://sourceforge.net/projects/pddl4j`
14. Sacerdoti, E.: The nonlinear nature of plans. In: Proc. of the International Joint Conference on AI, pp. 206–214 (1975)
15. Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D.: HTN planning for web service composition using SHOP. Journal of Web Semantics 1(4), 377–396 (2004)
16. Yang, B., Qin, Z.: Composing semantic web services with PDDL. Inform. Technol. J. 9, 48–54 (2009)
17. Yu, H.Q., Reiff-Marganiec, S.: Semantic Web Services Composition via Planning as Model Checking. Technical Report. CS-06-003, University of Leicester (2006)
18. Zhang, P., Huang, B., Sun, Y.: Automatic Web services composition based on SLM. In: Workshop on Semantic Web and Ontology, SWON 2008 (2008)