Automated Composition of Semantic Web Services into Executable Processes

Paolo Traverso[1] and Marco Pistore[2]

[1] ITC-IRST, Trento, Italy
traverso@itc.it
[2] University of Trento, Italy
pistore@dit.unitn.it

Abstract. Different planning techniques have been applied to the problem of automated composition of web services. However, in realistic cases, this planning problem is far from trivial: the planner needs to deal with the nondeterministic behavior of web services, the partial observability of their internal status, and with complex goals expressing temporal conditions and preference requirements. We propose a planning technique for the automated composition of web services described in OWL-S process models, which can deal effectively with nondeterminism, partial observability, and complex goals. The technique allows for the synthesis of plans that encode compositions of web services with the usual programming constructs, like conditionals and iterations. The generated plans can thus be translated into executable processes, e.g., BPEL4WS programs. We implement our solution in a planner and do some preliminary experimental evaluations that show the potentialities of our approach, and the gain in performance of automating the composition at the semantic level w.r.t. the automated composition at the level of executable processes.

1 Introduction

One of the big challenges for the taking up of web services is the provision of automated support to the composition of service oriented distributed processes, in order to decrease efforts, time, and costs in their development, integration, and maintenance. Currently, the problem of the composition of web services is addressed by two orthogonal efforts. From the one side, most of the major industry players propose low level process modeling and execution languages, like BPEL4WS [1]. These languages allow programmers to implement complex web services as distributed processes and to compose them in a general way, e.g., by interleaving the partial execution of different services with programming control constructs, like if-then-else, while-loops, fork, choice, etc. However, the definition of new processes that interact with existing ones must be done manually by programmers, and this is a hard, time consuming, and error prone task. From the other side, research within the Semantic Web community proposes a top down unambiguous description of web services capabilities in standard languages like DAML-S [2] and OWL-S [10], thus enabling the possibility to reason about web services, and to automate tasks like discovery and composition. However, the real taking up of Semantic Web Services for practical applications needs the ability of generating automatically

S.A. McIlraith et al. (Eds.): ISWC 2004, LNCS 3298, pp. 380–394, 2004.

composed services that can be directly executed, in the style of BPEL4WS programs, thus reducing effort, time and errors due to manual composition at the programming level.

Several works have proposed different automated planning techniques to address the problem of automated composition (see [17,18,21,23]). However, the planning problem is far from trivial, and can be hardly addressed by "classical planning" techniques. In realistic cases, OWL-S process models describe nondeterministic processes, whose behaviors cannot be predicted prior to of execution (e.g., a flight reservation service cannot know in advance whether a reservation will be confirmed or canceled). Moreover, the internal status of a service (e.g., whether there are still seats available in a flight) is not available to external services, and the planner can only observe services invocations and responses. Finally, composition goals need to express complex requirements that are not limited to reachability conditions (like get to a state where both the flight and the hotel are reserved). Most often, goals need to express temporal conditions (e.g., do not reserve the hotel until you have reserved the flight), and preferences among different goals (try to reserve both the flight and the hotel, but if not possible, make sure you do not reserve any of the two). As a consequence, automated composition needs to interleave (the partial executions of) available services with the typical programming language constructs such as conditionals, loops, etc., similarly to composed services that are programmed by hand, e.g., in BPEL4WS.

In this paper, we propose a technique for the automated composition of web services described in OWL-S, which allows for the automated generation of executable processes written in BPEL4WS. Given a set of available services, we translate their OWL-S process models, i.e., declarative descriptions of web service processes, into nondeterministic and partially observable state transition systems that describe the dynamic interactions with external services. Goals for the service to be automatically generated are represented in the EAGLE language [11], a language with a clear semantics which can express complex requirements. We can thus exploit the "Planning as Model Checking" approach based on symbolic model checking techniques [13,9,6,11,5], which provides a practical solution to the problem of planning with nondeterministic actions, partial observability, and complex goals, and which has been shown experimentally to scale up to large state spaces. As a result, the planning algorithm generates plans that are automata and that can be translated to BPEL4WS code.

We implement the proposed techniques in MBP [4], a planner based on the planning as model checking approach, and perform an experimental evaluation. Though the results are still preliminary, and deserve further investigation and evaluation, they provide a witness of the potentialities of our approach. Moreover, we compare the experimental results with those obtained by applying the same technique to (state transition systems generated from) BPEL4WS processes. The comparison shows that automated composition performed at the high level of OWL-S process models is orders of magnitudes more efficient than the one applied at the low level of executable processes, thus demonstrating experimentally a practical advantage of the Semantic Web approach to web services.

The paper is structured as follows. In Section 2, we give an overview of the approach and introduce an explanatory example that will be used all along the paper. In Section 3, we explain how OWL-S process models can be translated into state transition systems,

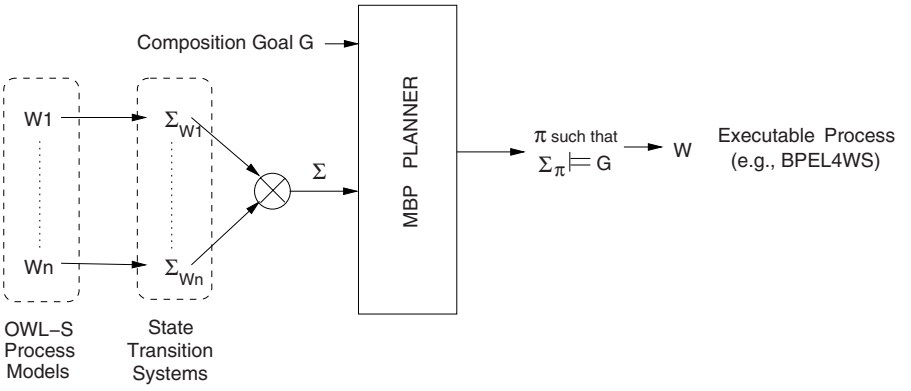

Fig. 1. OWL-S based Automated Composition.

while in Section 4 we describe the goal language. We explain how we do planning for web service composition in Section 5. We provide a preliminary experimental evaluation in Section 6, and a comparison with related work in Section 7.

2 Overview of the Approach

By automated composition we mean the task of generating automatically, given a set of available web services, a new web service that achieves a given goal by interacting with (some of) the available services. More specifically, we take as our starting point the OWL-S Process Model ontology [10], i.e., a declarative description of the program that realizes the service. Given the OWL-S process model description of n available services $(W_1, \ldots, W_n)$, we encode each of them in a state transition system $(\Sigma_{W_1}, \ldots, \Sigma_{W_n})$, see Fig. 1. State transition systems provide a sort of operational semantics to OWL-S process models. Each of them describes the corresponding web service as a state-based dynamic system, that can evolve, i.e., change state, and that can be partially controlled and observed by external agents. This way, it describes a protocol that defines how external agents can interact with the service.

From the point of view of the new composed service that has to be generated, say W, the state transition systems $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ constitute the environment in which W has to operate, by receiving and sending service requests. They constitute what, in planning literature, is called a planning domain, i.e., the domain where the planner has to plan for a goal. In our case, the planning domain is a state transition system Σ that combines $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. Formally, this combination is a parallel composition, which allows the n services to evolve independently. Σ represents therefore all the possible behaviors, evolutions of the planning domain, without any control performed by the service that will be generated, i.e., W.

The Composition Goal G (see Fig. 1) imposes some requirements on the desired behavior of the planning domain. Given Σ and G, the planner generates a plan π that controls the planning domain, i.e., interacts with the external services $W_1, \ldots, W_n$ in

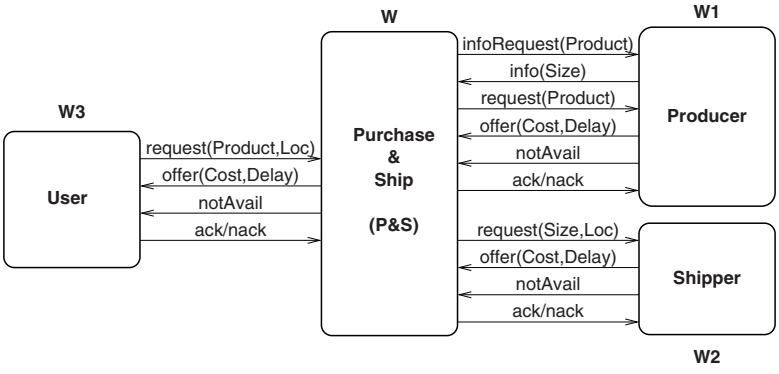

Fig. 2. A Simple Example.

a specific way such that the evolutions satisfy the goal G. The plan π encodes the new service W that has to be generated, which dynamically receives and sends invocations from/to the external services $W_1, \ldots, W_n$, observes their behaviors, and behaves depending on responses received from the external services. The plan π must therefore have the ability of encoding normal programming constructs, like tests over observations, conditionals, loops, etc. As we will see, π is encoded as an automaton that, depending on the observations and on its internal state, executes different actions. We can translate π into process executable languages, like BPEL4WS.

In the rest of the paper, we will describe step by step the automated composition task introduced above through the following example.

Example 1. Our reference example consists in providing a purchase & delivery service, say the P&S service, which satisfies some user request. We do so by combining two separate, independent, and existing services: a producer Producer, and a delivery service Shipper. The idea is that of combining these two services so that the user may directly ask the composed service P&S to purchase and deliver a given product at a given place. To do so, we exploit a description of the expected interaction between the P&S service and the other actors. In the case of the Producer and of the Shipper the interactions are defined in terms of the service requests that are accepted by the two actors. In the case of the User, we describe the interactions in terms of the requests that the user can send to the P&S. As a consequence, the P&S service should interact with three available services: Producer, Shipper, and User (see Fig. 2). These are the three available services W_1, W_2, and W_3, which are described as OWL-S process models and translated to state transition systems. The problem is to automatically generate the (plan corresponding to the) P&S service, i.e., W in Fig. 1.

In the following, we describe informally the three available services. Producer accepts requests for information on a given product and, if the product is available, it provides this information (e.g., the size). The Producer also accepts requests for buying a given product, in which case it returns an offer with a cost and production time. This offer can be accepted or refused by the external service that has invoked the Producer. The Shipper service receives requests for transporting a product of a given size to a

given location. If delivery is possible, Shipper provides a shipping offer with a cost and delivery time, which can be accepted or refused by the external service that has invoked Shipper. The User sends requests to get a given product at a given location, and expects either a negative answer if this is not possible, or an offer indicating the cost and duration of the service. The user may either accept or refuse the offer. Thus, a typical (nominal) interaction between the user, the combined purchase & delivery service P&S, the producer, and the shipper would go as follows:

1. the user asks P&S for product p, that he wants to be transported at location l;
2. P&S asks the producer for some data about product p, namely its size, the cost, and how much time does it take to produce it;
3. P&S asks the delivery service the price and time needed to transport an object of such a size to l;
4. P&S provides the user an offer which takes into account the overall cost (plus an added cost for P&S) and time to achieve its goal;
5. the user sends a confirmation of the order, which is dispatched by P&S to the delivery and producer.

Of course this is only the nominal case, and other interactions should be considered, e.g., for the cases the producer and/or delivery services are not able to satisfy the request, or the user refuses the final offer. At a high level, Fig. 2 describes the data flow amongst our integrated web service, the two services composing it, and the user. This can be perceived as (an abstraction of) the WSDL description of the dataflow. □

3 From OWL-S Process Models to State Transition Systems

OWL-S process models [10] are declarative descriptions of the properties of web service programs. Process models distinguish between *atomic processes*, i.e., non-decomposable processes that are executed by a single call and return a response, and *composite processes*, i.e., processes that are composed of other atomic or composite processes through the use of control constructs such as sequence, if-then-else, while loops, choice, fork, etc.

Example 2. The OWL-S process model of the Shipper service (see Example 1) is shown in Fig. 3 (left). The OWL-S model has been slightly simplified for readability purposes, by removing some (redundant) tags in the description of the processes. The Shipper service is a composite service consisting of the atomic processes DoShippingRequest, AcceptShippingOffer, and RefuseShippingOffer. DoShippingRequest receives in input a description of the Size and of the destination Location of the product to be delivered. The conditional output models the fact that the service returns as output an offer only if the shipping is possible, and returns a NotAvailable message otherwise. The offer includes the price (Cost) and the duration (Delay) of the transportation. If the transportation is possible (control construct IfThenElse), the shipper waits for a nondeterministic external decision (control construct Choice) that either accepts (AcceptShippingOffer) or refuses (RefuseShippingOffer) the offer.

Similarly, we can model the interactions with the producer with a composition of atomic processes AskProductInfo, DoProductRequest, AcceptProductOffer, and

```
<process:CompositeProcess rdf:ID="Shipper">
 <process:Sequence>
  <process:AtomicProcess rdf:about="#DoShippingRequest"/>
  <process:CompositeProcess>
   <process:IfThenElse>
    <process:ifCondition rdf:resource="#ShippingPossible"/>
    <process:then>
     <process:CompositeProcess>
      <process:Choice>
       <process:AtomicProcess rdf:about="#AcceptShippingOffer"/>
       <process:AtomicProcess rdf:about="#RefuseShippingOffer"/>
      </process:Choice>
     </process:CompositeProcess>
    </process:then>
   </process:IfThenElse>
  </process:CompositeProcess>
 </process:Sequence>
</process:CompositeProcess>

<process:AtomicProcess rdf:ID="DoShippingRequest">
 <process:Input rdf:ID="size">
  <process:parameterType rdf:resource="#Size">
 </process:Input>
 <process:Input rdf:ID="destination">
  <process:parameterType rdf:resource="#Location">
 </process:Input>
 <process:ConditionalOutput rdf:ID="price">
  <process:coCondition rdf:resource="#ShippingPossible"/>
  <process:parameterType rdf:resource="#Cost">
 </process:ConditionalOutput>
 <process:ConditionalOutput rdf:ID="duration">
  <process:coCondition rdf:resource="#ShippingPossible"/>
  <process:parameterType rdf:resource="#Delay">
 </process:ConditionalOutput>
 <process:ConditionalOutput rdf:ID="na">
  <process:coCondition rdf:resource="#NoShippingPossible"/>
  <process:parameterType rdf:resource="#NotAvailable">
 </process:ConditionalOutput>
</process:AtomicProcess>

<process:AtomicProcess rdf:ID="AcceptShippingOffer> ...

<process:AtomicProcess rdf:ID="RefuseShippingOffer> ...
```

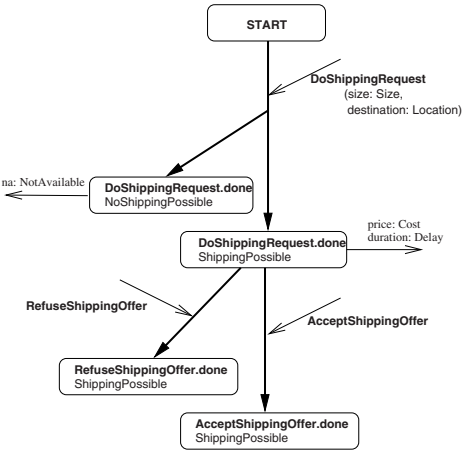

Fig. 3. The OWL-S Process Model and the State Transition System of the Shipper Service.

RefuseProductOffer; and the interaction with the user with the processes DoP&SRequest and EvaluateOffer (with the latter process the user specifies whether an P&S offer is accepted or not). □

We encode OWL-S process models as state transition systems. These describe dynamic systems that can be in one of their possible *states* (some of which are marked as *initial states*) and can evolve to new states as a result of performing some *actions*. A *transition function* describes how (the execution of) an action leads from one state to possibly many different states. System's evolutions can be monitored through *observations* describing the visible part of the system state. An *observation function* defines the observation associated to each state of the domain.

Definition 1 (state transition system). *A* (nondeterministic, partially observable) state transition system *is a tuple* $\Sigma = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$*, where:*

- $\mathcal{S}$ *is the set of* states*,* $\mathcal{A}$ *is the set of* actions*, and* $\mathcal{O}$ *is the set of* observations*.*
- $\mathcal{I} \subseteq \mathcal{S}$ *is the set of* initial states*.*
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to 2^{\mathcal{S}}$ *is the* transition function*; it associates to each current state* $s \in \mathcal{S}$ *and to each action* $a \in \mathcal{A}$ *the set* $\mathcal{T}(s, a) \subseteq \mathcal{S}$ *of next states.*
- $\mathcal{X} : \mathcal{S} \to \mathcal{O}$ *is the* observation function*.*

State transition systems are *nondeterministic*, i.e., an action may result in several different outcomes. This is modeled by the fact that the transition function returns sets of states.

Nondeterminism is needed since we do not know *a priori* which outcome will actually take place, e.g., whether a given product is available or not when we submit a request. Moreover, our state transition systems are *partially observable*, i.e., external agents can only observe part of the system state. In our case the external communications can be observed, while the internal status and variables are private to a given service. Partial observability is modeled by the fact that different states may result in the same observation (i.e., they are indistinguishable).

We associate a state transition system to each available OWL-S process model describing a web service. Intuitively, this is done as follows. The states $\mathcal{S}$ are used to codify the different steps of evolution of the service (e.g., what position has been reached inside the composite process of the Shipper) and the values of the predicates defined internally to the service (e.g., predicate ShippingPossible of Fig. 3). The actions $\mathcal{A}$ are used to model the invocation of the external atomic processes (e.g., DoShippingRequest). The observations $\mathcal{O}$ are used to model the outputs of the invoked external processes.

Example 3. In the case of the Shipper service process model (see Fig. 3 (right)), the states model the possible steps of the service: START, which holds initially, and DoShippingRequest.done, AcceptShippingOffer.done, RefuseShippingOffer.done, associated to the intermediate phases of the composite process. The internal variables ShippingPossible and NoShippingPossible describe the values of the corresponding conditions in the OWL-S model. In Fig. 3 (right), we represent explicitly only the information on the step in the composite process and on the predicates that are true in a given state. For simplicity, we do not represent the values for the the input and output parameters of the service invocation. Therefore, a single rounded box in Fig. 3 (right) corresponds to several states of the actual state transition structure.

The actions correspond to the invocation of the atomic processes DoShippingRequest, AcceptShippingOffer, and RefuseShippingOffer. The most complicated action is the first one: it has two parameters specifying the size and the destination location. Moreover, it has two possible non-deterministic outcomes. The first one, corresponding to the case the shipper is able to do the delivery, leads to the state where condition ShippingPossible is true. The second outcome, corresponding to the case the shipper is not able to do the delivery, leads to the state where condition NoShippingPossible is true.

The observations associated to each state define an assignment to the OWL-S conditional outputs. For instance, in the state corresponding to DoShippingRequest.done and ShippingPossible, values are defined for the output parameters price and duration. □

The formal definition of the translation of an OWL-S process model into a state transition system is conceptually simple, but it is complicated by several technical details. For this reason, and for lack of space, we do not present this definition here. The interested reader may refer to [19] for a detailed discussion of a translation similar to ours. In that case, Petri nets are used as target models. The states of our state transition systems can be seen as the markings in the Petri nets of [19].

4 Composition Goals

Composition goals express requirements for the service to be automatically generated. They represent conditions on the evolution of services, and as shown by the next example, they express requirements of different strengths and at different levels of preference.

Example 4. In our example (see Fig. 2), a reasonable composition goal for the P&S service is the following:

Goal 1: The service should try to reach the ideal situation where the user has confirmed his order, and the service has confirmed the associated (sub-)orders to the producer and shipper services. In this situation, the data associated to the orders have to be mutually consistent, e.g., the time for building and delivering a product shall be the sum of the time for building it and that for delivering it.

However, this is an ideal situation that cannot be enforced by the P&S service: the product may not be available, the shipping may not be possible, the user may not accept the total cost or the total time needed for production and delivery ... We would like the P&S service to behave properly also in these cases, otherwise it is likely to loose money. More precisely, it has to reach a consistent situation, where the P&S confirms none of the two services for production and delivering:

Goal 2: The P&S service should absolutely reach a fall-back situation where every (sub-)order has been canceled. That is, there should be no chance that the service has committed to some (sub-)order if the user can cancel his order.

Some remarks are in order. First of all, there is a difference in the "strength" in which we require Goal 1 and Goal 2 to be satisfied. We know that it may be impossible to satisfy Goal 1: we would like the P&S service to *try* (do whatever is possible) to satisfy the goal, but we do not require that the service guarantees to achieve it in all situations. The case is different for Goal 2: there is always a possibility for the P&S service to cancel the orders to the producer and shipper, and to inform the user. We can require a guarantee of satisfaction of this goal, in spite of any behavior of the other services. Moreover, Goal 1 and Goal 2 are not at the same level of desire. Of course we would not like a P&S service that satisfies always Goal 2 (e.g., by refusing all requests from the user) even when it would be possible to satisfy Goal 1. We need then to express a strong preference for Goal 1 w.r.t. Goal 2. Informally, we can therefore describe the composition goal as follows:

Composition Goal: *Try* to satisfy Goal 1, *upon failure*, *do* satisfy Goal 2. □

As the previous example shows, composition goals need the ability to express conditions on the whole behavior of a service, conditions of different strengths, and preferences among different subgoals. The EAGLE language [11] has been designed with the purpose to satisfy such expressiveness requirements. Let propositional formulas $p \in \mathcal{P}rop$ define conditions on the states of a state transition system. *Composition goals* $g \in \mathcal{G}$ over $\mathcal{P}rop$ are defined as follows:

$$g := p \mid g \textbf{ And } g \mid g \textbf{ Then } g \mid g \textbf{ Fail } g \mid \textbf{Repeat } g \mid$$
$$\textbf{DoReach } p \mid \textbf{TryReach } p \mid \textbf{DoMaint } p \mid \textbf{TryMaint } p$$

Goal **DoReach** p specifies that condition p has to be eventually reached in a strong way, for all possible non-deterministic evolutions of the state transition system. Similarly, goal **DoMaint** q specifies that property q should be maintained true despite non-determinism. Goals **TryReach** p and **TryMaint** q are weaker versions of these goals, where the plan is required to do "everything that is possible" to achieve condition p or maintain condition q, but failure is accepted if unavoidable. Construct g_1 **Fail** g_2 is used to model preferences among goals and recovery from failure. More precisely, goal g_1 is considered first. Only if the achievement or maintenance of this goal fails, then goal g_2 is used as a recovery or second-choice goal. Consider for instance goal **TryReach** c **Fail DoReach** d. The sub-goal **TryReach** c requires to find a plan that tries to reach condition c. During the execution of the plan, a state may be reached from which it is not possible to reach c. When such a state is reached, goal **TryReach** c fails and the recovery goal **DoReach** d is considered. Goal g_1 **Then** g_2 requires to achieve goal g_1 first, and then to move to goal g_2. Goal **Repeat** g specifies that sub-goal g should be achieved cyclically, until it fails. Finally, goal g_1 **And** g_2 requires the achievement of both subgoals g_1 and g_2. A formal semantics and a planning algorithm for EAGLE goals in fully observable nondeterministic domains can be found in [11].

Example 5. The EAGLE formalization of the goal in Example 4 is the following.

TryReach */* Goal 1 */*
(AcceptProductOffer.done & AcceptShippingOffer.done &
EvaluateOffer.done & EvaluateOffer.accepted &
DoP&SRequest.price = DoShippingRequest.price + DoProductRequest.price &
DoP&SRequest.duration = DoShippingRequest.duration + DoProductRequest.duration)
Fail
DoReach */* Goal 2 */*
(RefuseProductOffer.done & RefuseShippingOffer.done &
EvaluateOffer.done & ¬ EvaluateOffer.accepted)

Propositions like AcceptProductOffer.done are used to describe the states of the planning domain Σ corresponding to specific states of state transition systems obtained from the OWL-S processes (process Producer in our case). Propositions like DoShippingRequest.price or EvaluateOffer.accepted refer to the values of the input/output messages in service invocation. □

5 Automated Composition

The planner has two inputs (see Fig. 1): the composition goal and the planning domain Σ which represents all the ways in which the existing services can evolve. Formally, this combination is defined as the parallel composition $\Sigma = \Sigma_{W_1} \times \ldots \times \Sigma_{W_n}$ of the state transition systems of the existing services. The automated composition task consists in finding a plan that satisfies the composition goal G over a domain Σ. We are interested in complex plans, that may encode sequential, conditional and iterative behaviors, and are thus expressive enough for representing the flow of interactions of the synthesized composed service with the other services as well as the required observations over the other services. We therefore model a plan as an automaton.

Definition 2 (plan). *A* plan *for planning domain* $\Sigma = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$ *is a tuple* $\pi = \langle \mathcal{C}, c_0, \alpha, \epsilon \rangle$*, where:*

- $\mathcal{C}$ *is the set of* plan contexts.
- $c_0 \in \mathcal{C}$ *is the* initial context.
- $\alpha : \mathcal{C} \times \mathcal{O} \rightharpoonup \mathcal{A}$ *is the* action function*; it associates to a plan context* c *and an observation* o *an action* $a = \alpha(c, o)$ *to be executed.*
- $\epsilon : \mathcal{C} \times \mathcal{O} \rightharpoonup \mathcal{C}$ *is the* context evolutions function*; it associates to a plan context* c *and an observation* o *a new plan context* $c' = \epsilon(c, o)$.

The contexts are the internal states of the plan; they permit to take into account, e.g., the knowledge gathered during the previous execution steps. Actions to be executed, defined by function α, depend on the observation and on the context. Once an action is executed, function ϵ updates the plan context. Functions α and ϵ are deterministic (we do not consider nondeterministic plans), and can be partial, since a plan may be undefined on the context-observation pairs that are never reached during plan execution.

The execution of a plan over a domain can be described in terms of transitions between configurations, where each configuration (s, c) describes the current state s of the domain and the current context c of the plan (see [7] for a formal definition). Due to the nondeterminism in the domain, we may have an infinite number of different executions of a plan. However, they can be presented in a compact way as an *execution structure*, i.e, a state transition system defined by the configurations of the plan and by the transition among these configurations. In the following, we denote by Σ_π the execution structure that represents the evolutions of the domain Σ controlled by the plan π. In a planning problem, the execution structure Σ_π that must satisfy the composition goal G (see Fig. 1). If $\Sigma_\pi \models G$, we say that π *is a valid plan for* G *on* Σ. A formal definition of $\Sigma_\pi \models G$ can be found in [11].

However, notice that when executing a plan, the plan executor cannot in general get to know exactly what is the current state of the domain: the limited available access to the internal state of each external service inhibits removing the uncertainty present at the initial execution step, or introduced by executing nondeterministic actions. For instance, in the case of the Shipper service describe in Fig. 3, the executor has no access to the values of predicates ShippingPossible and NoShippingPossible, even if it can infer these values from the observable outcomes of action DoShippingRequest (namely, an offer or a NoAvailable message).

In presence of partial observability, at each plan execution step, the plan executor has to consider a set of domain states, each equally plausible given the initial knowledge and the observed behavior of the domain so far. Such a set of states is called a *belief state* (or simply *belief*) [8,6]. Executing an action a evolves a belief B into another belief B' which contains all of the possible states that can be reached through a from some state of B. The available sensing is exploited initially, and after each action execution: if observation o holds after executing action a, the resulting belief shall rule out states not compatible with o. Thus in general, given a belief B, performing an action a (executable in all the states of B) and taking into account the obtained observation o gets to a new belief $Evolve(B, a, o)$:

$$Evolve(B, a, o) = \{s' : \exists s \in B . s' \in \mathcal{T}(s, a) \wedge \mathcal{X}(s') = o\}.$$

Planning in this framework consists in searching through the possible evolutions of initial beliefs, to retrieve a conditional course of actions that leads to beliefs that satisfy the goal. The search space for a partially observable domain is what is called a *belief space*; its nodes are beliefs, connected by edges that describe the above $Evolve$ function. The search in a partially observable domain can be described as search inside a fully observable "belief-level" domain Σ_K whose states are the beliefs of Σ, and whose nondeterministic transition function mimics $Evolve$.

Algorithms for planning under partial observability can be obtained by suitably recasting the algorithms for full observability on the associated knowledge-level domain. Actually, the following result holds [7]:

Fact 3. *Let Σ be a ground-level domain and g be a knowledge-level goal for Σ (i.e., a goal expressing conditions on the beliefs reached during plan execution). Let also Σ_K be the knowledge level domain for Σ. Then π is a valid plan for g on Σ if, and only if, π is a valid plan for g on Σ_K.*

Thus, given a composition goal and a planning domain, solving the problem implies using dedicated algorithms for planning under partial observability with EaGLe goals, or, alternatively, planning for the fully observable associated knowledge level domain. We pursue this latter approach, so that we can reuse existing EaGLe planning algorithms under full observability [11]. We generate the knowledge level domain by combining the state transition systems defined previously. Similarly to what happens for the ground level domains, this computation consists of a parallel composition. Finally, we plan on this domain with respect to an EaGLe goal. Fact 3 guarantees that the approach outlined above for planning under partial observability with EaGLe goals is correct and complete. A potential problem of this approach is that, in most of the cases, knowledge-level domains are exponentially larger than ground domains. In [6], efficient heuristic techniques are defined to avoid generating the whole (knowledge-level) planning domain. These techniques can be extended to planning with EaGLe goals.

We have therefore the algorithms for generating a valid plan π that satisfies the composition goal. Since π is an automaton, it can be easily translated to executable process languages, like BPEL4WS. The generated code is not human-readable, however, it reflects the contexts defined in the plan (see Definition 2), which in turn reflect the structure of the goal. This makes it possible to monitor the execution of the BPEL4WS code and detect, for instance, when the primary goal of a composition (e.g., Goal 1 in Example 4) fails and a subsidiary goal (e.g., Goal 2) becomes active.

6 Experimental Evaluation

In order to test the effectiveness and the performance of the approach proposed in this paper, we have conducted some experiments using the MBP planner.

We have run MBP on six variants of the purchase and ship case study, of different degrees of complexity. In the easiest case, CASE 1, we considered a reduced domain with only the user and the shipper, and with only one possible value for each type of objects in the domain (product, location, delay, cost, size). In CASE 2 we have considered all three protocols, but again only one possible value for each type of object. In CASE 3 we

		With refuse		Without refuse	
	Building K	Planning	Result	Planning	Result
CASE 1	0.2 sec.	0.1 sec.	YES	0.1 sec.	NO
CASE 2	0.3 sec.	0.3 sec.	YES	0.3 sec.	NO
CASE 3	1.5 sec.	5.1 sec.	YES	3.4 sec.	NO
CASE 4	3.8 sec.	19.5 sec.	YES	17.9 sec.	NO
CASE 5	4.1 sec.	65.9 sec.	YES	71.5 sec.	NO
CASE 6	12.3 sec.	2899 sec.	YES	3885 sec.	NO

Fig. 4. Results of the Experiments.

have considered the three protocols, with two objects for each type, but removing the parts of the shipper and producer protocols concerning the size of the product. CASE 4 is the complete example discussed in Section 2. In CASE 5, one more web service is added to the domain, which provides support for the installation the product, once it has been delivered. CASE 6, finally, extends CASE 5 by allowing three values for each type of object. We remark that CASE 1 and CASE 2 are used to test our algorithms, even if they are admittedly unrealistic, since the process knows, already before the interaction starts, the product that the user will ask and the cost that will be charged to the user. In the other cases, a real composition of services is necessary to satisfy the goal. In all six cases we have experimented also with a variant of the shipper service, which does not allow for refusing an offer. This variant makes the composition goal unsatisfiable, since we cannot unroll the contract with the shipper and satisfy the recovery goal (see Example 5) in case of failure of the primary goal.

The experiments have been executed on an Intel Pentium 4, 1.8 GHz, 512 MB memory, running Linux 2.4.18. The results, in Fig. 4, report the following information:

- Building K: the time necessary to build the three knowledge-level domains.
- Planning: the time required to find a plan (or to check that no plan exists) starting from the knowledge-level domains.
- Result: whether a plan is found or not.

The last two results are reported both in the original domains and in the domains without the possibility of refusing a shipping offer. The experiments show that the planning algorithm returns the expected results. The performance is satisfactory: also in CASE 5, where the composition involves four external services, the composition is built in about one minute. The time required to obtain the composition grows considerably if we increase the number of available values for each object (CASE 6). Indeed, the different values are encoded into the domain states and have a strong impact on the size of the search space. A different, more advanced management of data types and variables, not requiring a direct encoding into the states, would mitigate this effect.

We perform a further set of experiments where we apply the same approach to the composition of web services at the level of BPEL4WS code. We translate the BPEL4WS code implementing the three existing services into state transition systems. The translation is technically different from (but conceptually similar to) the one described for OWL-S models, and is described in [20]. We perform this translation for the same cases

		With refuse		Without refuse	
	Building K	Planning	Result	Planning	Result
CASE 1	2 sec.	6 sec.	YES	5 sec.	NO
CASE 2	5 sec.	30 sec.	YES	13 sec.	NO
CASE 3	289 sec.	2008 sec.	YES	1642 sec.	NO
CASE 4	1058 sec.	16536 sec.	YES	13327 sec.	NO

Fig. 5. Results of the Experiments with BPEL4WS Domains.

considered in the previous experiment. Then we run the same MBP planning algorithm on the resulting planning domains. The results are reported in Fig. 5 (in CASE 5 and 6 we have stopped the planner after more than 5 hours of execution time). The comparison shows that automated composition performed at the level of OWL-S process models is much more efficient than composition applied at the level of executable processes. For instance, in CASE 4, both the time needed to generate the planning domain and the time for planning are three orders of magnitude higher for BPEL4WS domains. The reason is that OWL-S process models are at a higher level of abstraction w.r.t. BPEL4WS process models. Planning with BPEL4WS domains is done at the level of single messages between processes, while planning with OWL-S models is at the level of atomic web services. These results are a demonstration of the fact that OWL-S process models shift the representation at the right level of abstraction for composing web services, and show a practical advantage of the Semantic Web approach to web services composition.

7 Conclusions, Related and Future Work

In this paper, we have shown how OWL-S process models can be used to generate automatically new composed services that can be deployed and executed on engines for modern process modeling and execution languages, like BPEL4WS. This is achieved by translating OWL-S process models to nondeterministic and partially observable state transition systems and by generating automatically a plan that can express conditional and iterative behaviors of the composition. Our preliminary experimental evaluation shows the potentialities of the approach, and the practical advantage of automated composition at the semantic level w.r.t. the one at the level of executable processes.

Different planning techniques have been applied to the composition of web services, however, to the best of our knowledge, only the approach proposed in this paper is able to generate BPEL4WS processes that are directly executable. Existing approaches apply different planning approaches, ranging from HTNs [23] to regression planning based on extensions of PDDL [12], to STRIPS-like planning for composing services described in DAML-S [21], but how to deal with nondeterminism, partial observability, and how to generate conditional and iterative behaviors (in the style of BPEL4WS) in these frameworks is still an open issue. In [17], web service composition is achieved with user defined re-usable, customizable, high level procedures expressed in Golog. The approach is orthogonal to ours: Golog programs can express control constructs for the generic composition of web service, while we generate automatically plans that encode web service composition through control constructs. In [16], Golog programs are

used to encode complex actions that can represent DAML-S process models. However, the planning problem is reduced to classical planning and sequential plans are generated for reachability goals. In [19], the authors propose an approach to the simulation, verification, and automated composition of web services based on a translation of DAML-S to situation calculus and Petri Nets, so that it is possible to reason about, analyze, prove properties of, and automatically compose web services. However, the automated composition is again limited to sequential composition of atomic services for reachability goals, and does not consider the general case of possible interleavings among processes and of extended business goals.

Other techniques have been applied to related but somehow orthogonal problems in the field of web services. The interactive composition of information gathering services has been tackled in [22] by using CSP techniques. In [14], given a specific query of the user, an interleaving of planning and execution is used to search for a solution and to re-plan when the plan turns out to violate some user constraints at run time. In [3] automated reasoning techniques, based on Description Logic, are used to generate a composite web service. In that paper, however, the composition is seen as the problem of coordinating the executions of a given set of available services, and not as the problem of generating a new web service interacting with them.

The work in [15] is close in spirit to our general objective to bridge the gap between the semantic web framework and the process modeling and execution languages proposed by industrial coalitions. However, [15] focuses on a different problem, i.e., that of extending BPEL4WS with semantic web technology to facilitate web service interoperation, while the problem of automated composition is not addressed.

In the future, we aim at a solution that avoids the computationally complex power-set construction of the knowledge level domain, by providing algorithms for natively planning with extended goals under partial observability. Some preliminary results in this directions for a different goal language are presented in [7]. Moreover, we plan to integrate the automated composition task with reasoning techniques for discovery and selection at the level of OWL-S service profiles. Finally, we intend to test our approach over realistic case studies in projects for private companies and for the public administration we are currently involved in.

Acknowledgements. Special thanks to Piergiorgio Bertoli, Loris Penserini and Mark Carman for the interesting discussions on automated composition of web services. This work has been partially funded by the FIRB-MIUR project RBNE0195K5, “Knowledge Level Automated Software Engineering”.

References

1. T. Andrews, F. Curbera, H. Dolakia, J. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weeravarana. Business Process Execution Language for Web Services Version 1.1, 2003.
2. A. Ankolekar. DAML-S: Web Service Description for the Semantic Web. In *Proc. 1st Int. Semantic Web Conference (ISWC’02)*, 2002.

3. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of E-services That Export Their Behavior. In *Proc. 1st Int. Conf. on Service-Oriented Computing (ICSOC'03)*, 2003.
4. P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a Model Based Planner. In *Proc. IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.
5. P. Bertoli, A. Cimatti, M. Pistore, and P. Traverso. A Framework for Planning with Extended Goals under Partial Observability. In *Proc. 13th Int. Conf. on Automated Planning and Scheduling (ICAPS'03)*, 2003.
6. P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc. 17th Int. Joint Conference on Artificial Intelligence (IJCAI'01)*, 2001.
7. P. Bertoli and M. Pistore. Planning with Extended Goals and Partial Observability. In *Proc. 14th Int. Conf. on Automated Planning and Scheduling (ICAPS'04)*, 2004.
8. B. Bonet and H. Geffner. Planning with Incomplete Information as Heuristic Search in Belief Space. In *Proc. 5th Int. Conf. on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, 2000.
9. A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.
10. The OWL Services Coalition. OWL-S: Semantic Markup for Web Services — Technical White paper (OWL-S version 1.0), 2003.
11. U. Dal Lago, M. Pistore, and P. Traverso. Planning with a Language for Extended Goals. In *Proc. 18tn National Conference on Artificial Intelligence (AAAI'02)*, 2002.
12. D. Mc Dermott. The Planning Domain Definition Language Manual. Technical Report 1165, Yale Computer Science University, 1998. CVC Report 98-003.
13. F. Giunchiglia and P. Traverso. Planning as Model Checking. In *Proc. 5th European Conf. on Planning (ECP'99)*, 1999.
14. A. Lazovik, M. Aiello, and Papazoglou M. Planning and Monitoring the Execution of Web Service Requests. In *Proc. 1st Int. Conf. on Service-Oriented Computing (ICSOC'03)*, 2003.
15. D. Mandell and S. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In *Proc. 2nd Int. Semantic Web Conference (ISWC'03)*, 2003.
16. S. McIlraith and R. Fadel. Planning with Complex Actions. In *Proc. 9th Int. Workshop on Non-Monotonic Reasoning (NMR'02)*, 2002.
17. S. McIlraith and S. Son. Adapting Golog for composition of semantic web Services. In *Proc. 8th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'02)*, 2002.
18. S. McIlraith, S. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.
19. S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. 11th Int. World Wide Web Conference (WWW'02)*, 2002.
20. M. Pistore, P. Bertoli, F. Barbon, D. Shaparau, and P. Traverso. Planning and Monitoring Web Service Composition. In *Proc. 11th Int. Conf. on Artificial Intelligence: Methodology, Systems, Applications — Semantic Web Challenges (AIMSA'04)*, 2004.
21. M. Sheshagiri, M. desJardins, and T. Finin. A Planner for Composing Services Described in DAML-S. In *Proc. AAMAS'03 Workshop on Web Services and Agent-based Engineering*, 2003.
22. S. Thakkar, C. Knoblock, and J.L. Ambite. A View Integration Approach to Dynamic Composition of Web Services. In *Proc. ICAPS'03 Workshop on Planning for Web Services*, 2003.
23. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S Web Services Composition using SHOP2. In *Proc. 2nd Int. Semantic Web Conference (ISWC'03)*, 2003.