A Minimalist Approach to Semantic Annotations for Web Processes Compositions*

Marco Pistore[1], Luca Spalazzi[2], and Paolo Traverso[3]

[1] Università di Trento - Via Sommarive 14 - 38050 Povo - Trento - Italy
pistore@dit.unitn.it
[2] Università Politecnica delle Marche - Via Brecce Bianche - 60131 Ancona - Italy
spalazzi@diiga.univpm.it
[3] ITC-irst - Via Sommarive 18 - 38050 Povo - Trento - Italy
traverso@irst.itc.it

Abstract. In this paper we propose a new approach to the automated composition of distributed processes described as semantic web services. Current approaches, such as those based on OWL-S and WSMO, in spite of their expressive power, are hard to use in practice. Indeed, they require comprehensive and usually large ontological descriptions of the processes, and rather complex (and often inefficient) reasoning mechanisms. In our approach, we reduce to the minimum the usage of ontological descriptions of processes, so that we can perform a limited, but efficient and useful, semantic reasoning for composing web services. The key idea is to keep separate the procedural and the ontological descriptions, and to link them through semantic annotations. We define the formal framework, and propose a technique that can exploit simple reasoning mechanisms at the ontological level, integrated with effective reasoning mechanisms devised for procedural descriptions of web services.

1 Introduction

The importance of describing web services at the *process-level* is widely recognized, a witness being the standard languages for describing business processes, like BPEL [1], and the most popular standards for semantic web services, like OWL-S [4] and WSMO [22]. In a process-level description, a web service is not simply represented as an "atomic" component that can be executed in a single step. Instead, the interface of the service describes its behavior, i.e., a flow of interactions with other services structured according to different control constructs, e.g., sequentially, conditionally, and iteratively. Behavioral descriptions of web services can be published in standard languages, e.g., as *abstract* BPEL *specifications*, OWL-S *process models*, and WSMO *interfaces*. They constitute a key element for several application domains where web services are proposed as the basis for interoperability and integration of (business) processes that are distributed over the network. This is the case, for instance, of several e-Government, e-Banking, and e-Commerce applications.

* This work is partially funded by the European project FP6-507482 "Knowledge Web" , by the MIUR-FIRB project RBNE0195K5, "Knowledge Level Automated Software Engineering", and by the MIUR-PRIN 2004 project "Advanced Artificial Intelligence Systems for Web Services".

Y. Sure and J. Domingue (Eds.): ESWC 2006, LNCS 4011, pp. 620–634, 2006.

Recent research is focusing on the key problem of the automated composition of web services described at the process level [10, 7, 3, 20, 15, 13, 14]. However, the research is still at an early stage. From one side, some approaches do not deal with semantic web services, and cannot thus exploit the ability to do reasoning about what services do. This is the case of techniques for composing BPEL processes [15, 13] and of theoretical frameworks for the composition of services represented as finite state automata [7, 3]. From the other side, the approaches that have been proposed so far to exploit semantic descriptions (see, e.g., [10, 20, 14, 22]) are based on the idea that processes should be described by means of comprehensive ontologies. They have the practical disadvantage to require long descriptions that are time- and effort- consuming, and that are very hard to propose in practice for industrial applications. Such semantic descriptions of web services are based on expressive languages such as OWL [9] or WSMO [22], and require complex reasoning mechanisms. Indeed, for instance, the OWL family of languages are based on the description logics $\mathcal{SHIQ}$ and $\mathcal{SHIOQ}$, that have reasoning services that are EXPTime and NEXPTime, respectively [19].

In this paper, we propose a practical approach to the composition of semantic web services. We aim at automated composition techniques that exploit a limited, but still useful, amount of semantic reasoning. The key idea is to keep separate the procedural and the ontological descriptions, and to use semantic annotations to link them. First, the behavior of a web service is defined in languages that have been designed to describe processes. Then, the semantics of data exchanged and of the operations performed by the processes is described in a separate ontological language. Finally, the two descriptions are linked by semantic annotations of the behavioral descriptions that map to the ontological concepts. Annotations are necessary to give semantics to the exchanged data (e.g., which relations exist between the data given in input to the service and the data received as answers from the service), as well as to define the effects and outcomes of the service executions (e.g., to identify the successful executions of the service and distinguish them from the failures, and to describe the effects associated to the successful executions).

We apply this idea to the case of processes described in BPEL. More precisely, we give semantics to abstract BPEL processes in terms of state transition systems, in such a way that variables that are used in messages exchanged among BPEL processes constitute the state variables of the associated state transition systems. The meaning of these variables is defined by an annotation function that maps them to an ontological language, which, in this paper, is based on the $\mathcal{ALN}$ description logic and a generalized acyclic TBox [2]. Given this formal framework, we can express composition requirements as *semantic goals*, i.e., expressions in a language whose terms refer to ontological descriptions. We define formally the automated composition problem with semantic goals and propose an automated composition technique that translates semantic goals into *ground goals*, i.e., goals that refer to the state variables of the process. We can thus exploit efficient automated composition algorithms that have been devised for non-annotated BPEL compositions [15, 13].

The paper is structured as follows. In Section 2, we describe a reference example that is used all along the paper. In Section 3 we formally define semantic annotations for state transition systems that describe BPEL processes, and the language for

describing semantic goals. In Section 4, we formally define the composition problem, while in Section 5 we describe the automated composition technique. We conclude with a description of some related work.

2 Overview of the Approach: An Example

We aim at the automated synthesis of a new composite service that interacts with a set of existing component web services in order to satisfy a given composition requirement. More precisely, we assume that we have already identified the services that will be combined into the composite service[1], and that we are now facing the problem of defining the executable process that can interact with these existing services in order to achieve the composition requirement.

Example 1. Our running example consists in the composition of existing transport and accommodation services in order to provide a Virtual Travel Agency (VTA) service. The VTA is responsible for defining a suitable vacation package, according to the requests of the user. The selection of the service providers may depend on the constraints given by the end user and by domain knowledge: for instance, if the destination of the trip is Paris and the duration is one week, we know that the trip can be done by flight or by train (but not, e.g., by ship), and that suitable accommodations are in hotel and in guest houses (but not in apartments). We can hence assume that we have selected four suitable services (FlightReservation, TrainReservation, HotelReservation, Guesthouse-Reservation).

According to our approach each web service exploited in the composition defines:

- an ontology defining the relevant terminology,
- an interface process defining the interactions necessary to execute the service, and
- an annotation of the choreography that defines (partial) correspondences between the ontology and the process.

In the following, we assume that the ontologies provided by the different services have been mapped into a global common ontology that defines all the relevant concepts of the composition scenario. We will also assume that the interface processes are annotated according to this global ontology.

Example 2. The common ontology for the VTA composition scenario discussed in Example 1 is depicted in Figure 1 using the standard description logic notation. This ontology contains a part that is general for the VTA domain (Date, Client, Location, Trip, Accommodation), and that can be seen as part of the domain knowledge. It also contains other concepts that are specific of the actual web services that we are going to exploit in the composition (Flight, Train, Hotel, GuestHouse), and that can be obtained by mapping the local ontology of each web service into the common ontology.

[1] This step is of course very complex. We will not further discuss these steps in the paper, since our focus is different. We assume that any of the techniques for service discovery and selection discussed in the literature are applied to these steps.

$$
\begin{aligned}
\mathsf{Date} &\doteq \forall \mathsf{year}.\mathsf{Number} \sqcap \forall \mathsf{month}.\mathsf{Number} \sqcap \forall \mathsf{day}.\mathsf{Number} \\
\mathsf{Client} &\doteq \forall \mathsf{name}.\mathsf{String} \sqcap \forall \mathsf{gender}.\mathsf{Gender} \\
\mathsf{Gender} &\sqsubseteq \top \\
\mathsf{Status} &\sqsubseteq \top \\
\mathsf{Location} &\doteq \forall \mathsf{name}.\mathsf{String} \\
\mathsf{Trip} &\doteq \forall \mathsf{id}.\mathsf{String} \sqcap (\leq 1\mathsf{id}) \sqcap (\geq 1\mathsf{id}) \sqcap \forall \mathsf{date}.\mathsf{Date} \sqcap \forall \mathsf{start}.\mathsf{Location} \sqcap \\
&\quad \forall \mathsf{destination}.\mathsf{Location} \sqcap \forall \mathsf{pax}.\mathsf{Client} \sqcap \forall \mathsf{status}.\mathsf{Status} \\
\mathsf{Accomodation} &\doteq \forall \mathsf{id}.\mathsf{String} \sqcap (\leq 1\mathsf{id}) \sqcap (\geq 1\mathsf{id}) \sqcap \forall \mathsf{date}.\mathsf{Date} \sqcap \\
&\quad \forall \mathsf{location}.\mathsf{Location} \sqcap \forall \mathsf{pax}.\mathsf{Client} \sqcap \forall \mathsf{status}.\mathsf{Status} \\
\mathsf{Flight} &\sqsubseteq \mathsf{Trip} \sqcap \forall \mathsf{seatNumber}.\mathsf{String} \\
\mathsf{Train} &\sqsubseteq \mathsf{Trip} \sqcap \forall \mathsf{seatNumber}.\mathsf{String} \\
\mathsf{Hotel} &\sqsubseteq \mathsf{Accommodation} \sqcap \forall \mathsf{roomNumber}.\mathsf{String} \\
\mathsf{GuestHouse} &\sqsubseteq \mathsf{Accommodation} \sqcap \forall \mathsf{roomNumber}.\mathsf{String}
\end{aligned}
$$

Fig. 1. The terminology of the running example

male : Gender, female : Gender,
available : Status, notAvailable : Status, booked : Status, cancelled : Status

Fig. 2. The common part of each ABox in the running example

Notice that, in the definition of Trip *and* Accommodation*, we have the role* status *whose values are restricted to be of the concept* Status*. This role captures what is the current status of the client request. Indeed, when a trip (accommodation) is available, the status assumes the value* available*; when a trip (accommodation) is not available, the status assumes the value* notAvailable*; and finally, when a trip (accommodation) has been booked (cancelled), the status assumes the value* booked *(*cancelled*). The possible values (instances) for the concept* Status *are listed in the ABox in Figure 2.*

In our approach, the interface processes defining the interaction behaviors of the component services are defined in abstract BPEL. BPEL [1] provides an operational description of the (stateful) behavior of web services on top of the service interfaces defined in their WSDL specifications. An abstract BPEL description identifies the partners of a service, its internal variables, and the operations that are triggered upon the invocation of the service by some of the partners. Operations include assigning variables, invoking other services and receiving responses, forking parallel threads of execution, and nondeterministically picking one amongst different courses of actions. Standard imperative constructs such as if-then-else, case choices, and loops, are also supported.

Example 3. In Figure 3 we report (the relevant parts of) the abstract BPEL *specification of the FlightReservation service in the scenario discussed in Example 1.*[2]

[2] The specification contains some annotations in boldface: they are not part of the BPEL language, and we will explain their meaning later on in this section.

```
<process name="FlightReservation">
  <variables>
    <variable name="req" messageType="flightRequest"/>
      <!-- "req" contains parts "/req/start", "/req/des", and "/req/date" -->
    <variable name="pax" messageType="paxInformation"/>
      <!-- "pax" contains part "/offer/client" -->
    <variable name="offer" messageType="flightOffer"/>
      <!-- "offer" part "/offer/fl" -->
  </variables>
  <sequence name="main">
    <receive operation="request" variable="req"
      semann="/req/start : Location, /req/dest : Location, /req/date : Date"/>
    <switch name="checkAvailability">
      <case name="isNotAvailable">
        <invoke operation="not_avail" semann="/offer/fl : Flight, /offer/fl.status = notAvailable"/>
      </case>
      <otherwise name="isAvailable">
        <assign name="prepareOffer">
          <copy><from opaque="yes" semann="/offer/fl : Flight, /offer/fl.start = /req/start,
                /offer/fl.destination = /req/dest, /offer/fl.date = /req/date"/>
            <to variable="offer" part="fl"/></copy>
        </assign>
        <invoke operation="offer" inputVariable="offer" />
        <pick name="waitAcknowledge">
          <onMessage operation="ack" variable="pax"
            semann="/pax/client : Client, /offer/fl.pax = /client/pax, /offer/fl.status = booked"/>
          <onMessage operation="nack" semann="/offer/flight.status = cancelled"/>
        </pick>
      </otherwise>
    </switch>
  </sequence>
</process>
```

Fig. 3. The annotated BPEL process of the *FlightReservation* service

The process starts with a declaration of the variables that are used in input/output messages: req *is the input variable that specifies the start and destination locations and the date of the flight;* pax *specifies the details on the client booking the flight;* offer *is the flight offered to the client, including flight identifier and seat number. The* messageType *declaration specifies the structure of a variable used for sending/receiving messages. Such structure is detailed in the* WSDL *specification associated to the* BPEL *code, which we omit for lack of space.*

The rest of the abstract BPEL *specification describes the interaction flow. The FlightReservation service is activated by a request from a client (*receive *instruction corresponding to operation* request*). The information on desired flight submitted by the client is stored in variable* req*. Depending on its internal availability (*switch *instruction named* checkAvailability*), the flight provider can either send an answer refusing the request (*invoke *instruction corresponding to operation* not_avail*), or prepare and send the information regarding a specific flight. In the latter case, the flight and seat number are determined and assigned to variable* offer *within the* assign *statement named* prepareOffer*. The way in which the information is obtained is not disclosed and published by the abstract* BPEL*: the sources of the data, assigned to the variables by the* copy *constructs, are "*opaque*". The opaqueness mechanism allows for presenting the external world with an abstract view of the business logic, which hides the portions that the designer does not intend to disclose, and which is robust to changes with respect to the actual way in which the internal business logic is defined (e.g., calls to*

*specific data bases of the flight reservation company). Once the offer has been sent to the client (*invoke *instruction corresponding to operation* offer*), the FlightReservation service suspends (instruction* pick*), waiting for the customer either to acknowledge the acceptance of the flight offer (*onMessage *specification corresponding to operation* ack*; we remark that this message carries the information on the client booking the flight), or to refuse the offer (*onMessage *specification corresponding to operation* nack*). Only in the former case the interaction with the service is successful, and the flight has been booked.*

Notice that the process described in Figure 3 is only one of the possible interfaces. Different flight providers could adopt different approaches, e.g., requiring that the information on the client is given at the beginning of the process rather than during the acknowledgement, or providing to the user a second choice if the first flight offer is refused.

A BPEL specification provides a very detailed description of the interactions that need to be carried out with a web service in order to exploit it. However, this is still not sufficient to allow for the purpose of automatically composing such web service with other services. Indeed, it is necessary to describe also the "semantic" aspects of such interactions. We do this by extending the BPEL specification with "semantic annotations" (the **semann** attributes in Figure 3).

In our example, it is necessary first of all to associate concepts in the ontology to the (parts of the) input and output messages exchanged by the process. This is the role, for instance of the semantic annotations "/req/start : Location, /req/dest : Location, /req/date : Date" of the receive activity for operation request, at the beginning of the BPEL process. Moreover, it is necessary to express "semantic" relations among the input and output data values exchanged during the interaction with the web service, e.g., between the start and destination locations and dates requested by the client and the flight returned by the reservation service. This is done in annotation "/offer/fl.start = /req/start, /offer/fl.destination = /req/dest, /offer/fl.date = /req/date" of the opaque assignment. A further usage of semantic annotations is to define the outcome of an interaction with a web service. In our example it is clear that a flight has been booked only if a flight is available, the reservation service sends an offer, and the user acknowledges the acceptance of the offer. To express this in the BPEL specification, we add annotation "/offer/status = booked" to the activity corresponding to the reception of the acknowledgement.

The semantic annotations are necessary to compensate the specificities of the interface at hand, and to put it in relation with the common ontology. We remark, however, that the semantic annotations that have to be added to this purpose are very limited if compared to processes defined in languages such as OWL-S or WSMO. As we will see, they are sufficient for the automated composition task we are interested in.

3 BPEL Processes as Annotated STSs

We encode BPEL processes (extended with semantic annotations) as *annotated state transition systems* which 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*. We distinguish actions in *input actions*, *output actions*, and τ. *Input actions* represent the reception of messages, *output actions* represent messages sent to external services, and τ is a special action, called *internal action*, that represents internal evolutions that are not visible to external services. In other words, τ represents the fact that the state of the system can evolve without producing any output, and without consuming any inputs. A *transition relation* describes how the state can evolve on the basis of inputs, outputs, or of the internal action τ. Concerning the *states*, we associate to each state a set of *concept assertions* and *role assertions*. This configures a state as the assertional component (or ABox) of a knowledge representation system based on a given description logic where the ontology plays the role of the terminological component (or TBox). Therefore, *concept assertions* are formulas of the form $a : C$ (or $C(a)$) and state that a given individual a belongs to (the interpretation) of the concept C. *Role assertions* are formulas of the form $a.R = b$ (or $R(a, b)$) and state that a given individual b is a value of the role R for a. As a consequence, each action can be viewed as a transition from a state consisting in an ABox to a different state consisting in a different ABox.

Definition 1 (State transition system [16]). *A* state transition system Σ *is a tuple* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$ *where:*

- $\mathcal{S}$ *is the finite set of states;*
- $\mathcal{S}^0 \subseteq \mathcal{S}$ *is the set of initial states;*
- $\mathcal{I}$ *is the finite set of input actions;*
- $\mathcal{O}$ *is the finite set of output actions;*
- $\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{S}$ *is the transition relation.*

Definition 2 (Annotated state transition system). *An* annotated state transition system *is a tuple* $\langle \Sigma, \mathcal{T}, \Lambda \rangle$ *where:*

- Σ *is the state transition system,*
- $\langle \mathcal{T}, \Lambda \rangle$ *is the annotation,*
- $\mathcal{T}$ *is the terminology (TBox) of the annotation,*
- $\Lambda : \mathcal{S} \rightarrow 2^{\mathcal{A}_\mathcal{T}}$ *is the annotation function, where* $\mathcal{A}_\mathcal{T}$ *is the set of all the concept assertions and role assertions defined over* $\mathcal{T}$.

Example 4. Figure 4 shows a textual description of the annotated STS corresponding to the annotated BPEL *code of Figure 3. The set of states* $\mathcal{S}$ *(the section* STATE *in Figure 4) models the steps of the process and the evolution of the concept and the role assertions.* pc *is a variable that ranges over the set of states* $\mathcal{S}$ *and thus holds the current execution step of the service (e.g.,* pc = checkAvailability *when it is ready to check whether the flight is available). The set of initial states* $\mathcal{S}^0$ *is represented by the section* INIT *in Figure 4.*

The concepts used in the annotated STS are listed in the section CONCEPT *of Figure 4. They must be defined in the terminology* $\mathcal{T}$.

According to the formal model, we distinguish among three different kinds of actions (see the sections INPUT *and* OUTPUT *of Figure 4). The input actions* $\mathcal{I}$ *model all the incoming requests to the process and the information they bring (e.g.,* request *is used for the receiving of the flight reservation request). The output actions* $\mathcal{O}$ *represent outgoing*

```
PROCESS FlightReservation;
STATE pc : { START, receive_request, checkAvailability, isNotAvailable, isAvailable, invoke_not_available,
        prepareOffer, invoke_offer, waitAcknowledge, END_NA, END_ACK, END_NACK };
INIT pc = {START};
CONCEPT Flight; Location; Date; Client; Status;
INPUT request(Location, Location, Date); ack(Client); nack();
OUTPUT flightOffer(Flight); not_avail();
TRANS
    pc = START -[TAU]-> pc = receive_request;
    pc = receive_request -[INPUT request(req_start,req_dest,req_dat)]-> pc = checkAvailability
    pc = checkAvailability -[TAU]-> pc = isNotAvailable;
    pc = checkAvailability -[TAU]-> pc = isAvailable;
    pc = isNotAvailable -[TAU]-> pc = invoke_not_available;
    pc = invoke_not_available -[OUTPUT not_avail()]-> pc = END_NA;
    pc = isAvailable -[TAU]-> pc = prepareOffer;
    pc = prepareOffer -[TAU]-> pc = invoke_offer,
    pc = invoke_offer -[OUTPUT offer(offer_fl)]-> pc = waitAcknowledge;
    pc = waitAcknowledge -[INPUT ack(pax_client)]-> pc = END_ACK;
    pc = waitAcknowledge -[INPUT nack()]-> pc = END_NACK;
ANNOTATION FUNCTION
   LAMBDA( checkAvailability) = { req_start : Location, req_dest : Location, req_date : Date };
   LAMBDA( END_NA) = { offer_fl : Flight, offer_fl.status = notAvailable } ∪ LAMBDA( checkAvailability);
   LAMBDA( invoke_offer) = { offer_fl : Flight, offer_fl.date = req_date, offer_fl.start = req_start,
        offer_fl.destination = req_dest } ∪ LAMBDA( checkAvailability);
   LAMBDA( END_ACK) = { pax_client : Client, offer_fl.pax = pax_client, offer_fl.status = booked}
        ∪ LAMBDA( invoke_offer);
   LAMBDA( END_NACK) = {offer_fl.status = cancelled} ∪ LAMBDA( invoke_offer);
```

Fig. 4. The annotated STS corresponding to the FlightReservation process

messages (e.g., flightOffer is used to bid a flight). The action τ is used to model internal evolutions of the process, such as assignments and decision making.

The evolution of the process is modelled through a set of possible transitions (the section TRANS in Figure 4). Each transition defines its applicability conditions on the source state, its firing action, and the destination state. For instance, pc = checkAvailability -[TAU]-> pc = isNotAvailable states that an action τ can be executed in state checkAvailability and leads to the state isNotAvailable; this transition models the decision of the reservation service that no flight is available.

The annotation function Λ (see the section ANNOTATION FUNCTION in Figure 4) models how the assertions vary depending on the states. For instance, LAMBDA(END_NACK) = {req_start : Location, req_dest : Location, req_date : Date, offer_fl : Flight, offer_fl.start = req_start, offer_fl.destination = req_dest, offer_fl.date = req_date, offer_fl.status = cancelled} represents the fact that state END_NACK contains, among others, the concept assertions fl : Flight (i.e., fl is an individual that belongs to the concept Flight) and the role assertions offer_fl.start = req_start and offer_fl.status = cancelled (the roles start and destination of the individual fl are filled with the individuals req_start and cancelled).

We remark that each TRANS clause and each LAMBDA clause of Figure 4 corresponds to different elements in the transition relation $\mathcal{R}$ and in the annotation function Λ, respectively. For example, the transition and the LAMBDA clause described above generate different elements of $\mathcal{R}$ and Λ depending on which individuals req_date, req_start, req_dest we have in the destination state. Concerning cancel, it has been defined in Figure 2 and, thus, it denotes the same individual in all the states (i.e., all the ABoxes).

The definition of the state transition system provided in Figure 4 is parametric w.r.t. the individuals that can be associated to concepts Flight, Location, Date, Client. In order to obtain a concrete state transition system (a set of concrete ABoxes) and to apply the automated synthesis techniques described in this paper, finite set of individuals have to be assigned concepts Flight, Location, Date, Client. A possible approach to assign these individuals consists in defining appropriate concept assertions in the common part of the ABoxes (e.g., the part of ABoxes depicted in Figure 2). Another, better technique is to use knowledge level techniques such as the ones in [13] to avoid an explicit enumeration of the individuals of Flight, Location, Date, Client.

We have formally defined a translation that associates an annotated state transition system to each component service, starting from its annotated BPEL specification. In Figure 4 we have reported the translation for the specific case of the flight booking service, with minor changes (e.g., in the order of the clauses and in some automatically generated names) to improve the readability. We omit the formal definition of the translation, which can be found at http://www.astroproject.org/.

According to the above definitions, when we have to check if a given assertion p is true in a given state $s \in \mathcal{S}$ we have to apply instance checking denoted as $\langle \mathcal{T}, \Lambda(s) \rangle \models p$. On the other hand, ABoxes play no active role when checking subsumption [17], therefore subsumption can be checked without considering what is the current state (i.e., the current ABox). For example, when we have to check $\langle \mathcal{T}, \Lambda(s) \rangle \models C \sqsubseteq D$, we only need to check $\langle \mathcal{T}, \emptyset \rangle \models C \sqsubseteq D$. Furthermore, let us assume to use $\mathcal{ALN}$ as description logic and a generalized acyclic TBox as $\mathcal{T}$ [2]. This language is expressive enough to describe non-trivial examples as the VTA domain. The computational complexity of subsumption w.r.t. an acyclic terminology is NP-complete, while the computational complexity of instance checking is P [6], which makes the reasoning problems tractable, and, e.g., less complex than those of OWL-S.

4 Web Process Composition Problem

According to our approach, the inputs of the composition problem are (1) a global ontology $\langle \mathcal{T}, \mathcal{A} \rangle$, (2) a set of annotated BPEL processes, or, equivalently, of annotated STSs $\langle \Sigma_i, \mathcal{T}, \Lambda_i \rangle$, defining the component services, and (3) a composition requirement ρ, that formalizes these desired properties of the composite service to be synthesized. Inputs (1) and (2) have been already described. We now focus on the definition of the composition requirement.

Example 5. We want the VTA to define and book a vacation package according to the request of a client. This means we want the VTA to reach the situation where a trip has been booked from the start location to the destination specified by the user, and for the dates specified by the user; moreover, an accommodation has been booked for the same destination and dates. However, this goal of the VTA may be impossible to achieve. It might be impossible to book the trip or the accommodation for the given destination or, in more realistic descriptions of the VTA, the package defined by the VTA may be too expensive. We cannot avoid these situations, and we therefore cannot ask the composite service to guarantee that a vacation package is always defined and

booked. Nevertheless, we would like the VTA to try *(do whatever is possible) to satisfy it. Moreover, in the case the "*define and book a vacation package*" requirement is not satisfied, we do not want to book the trip only or the accommodation only. That is, either trip and accommodation are both booked, or none of them has to be booked. Let us call this requirement "*no booking is pending*". Our global composition requirement would therefore be something like:*

try to define and book a vacation package;
upon failure, guarantee that no booking is pending.

We remark that, when composing web services, it is often the case that composition requirements have the structure described in the previous example, i.e., they define a "primary condition" to be achieved whenever possible, and a "recovery condition" that has to be achieved in all the cases the main condition fails.[3]

Besides the principal and the recovery conditions, a composition requirement also defines two sets of concept assertions. The first one, that we call *input concept assertions*, can be seen as input parameters for the composition requirements, such as desired trip destination and dates, which can be assumed to exist in the ABox of the global ontology. The second set, called *output concept assertions*, describes the elements that have to be defined by the web service composition, in our case the trip and the accommodation.

We now give the formal definition of composition requirement.

Definition 3 (Composition requirement). *Let $\mathcal{T}$ be the terminology for the composition problem. A composition requirement is a tuple $\rho = \langle i, o, p, r \rangle$, where:*

- *i is a set of* input concept assertions *for $\mathcal{T}$;*
- *o is a set of* output concept assertions *for $\mathcal{T}$;*
- *p is a* goal condition *on $\langle \mathcal{T}, i \cup o \rangle$ specifying the* primary condition*;*
- *r is a* goal condition *on $\langle \mathcal{T}, i \cup o \rangle$ specifying the* recovery condition*.*

Goal conditions p *and* r *are expressions in the following grammar:*

$$p ::= a : C \mid a.R = b \mid p \ \textsf{OR}\ p \mid p \ \textsf{\&}\ p \mid \textsf{NOT}\ p$$

where $a : C$ is a concept assertion and $a.R = b$ is a role assertion defined w.r.t. $\mathcal{T}$.

Example 6. The composition requirement of the VTA scenario is based on the following input concept assertions: start : Location *(the starting location for the travel),* dest : Location *(the destination of the travel),* date : Date *(the dates of the travel), and* client : Client *(the client who is booking the travel). The output concept assertions are:* tr : Trip *(the trip returned by the VTA) and* ac : Accommodation *(the accommodation returned by the VTA).*

As discussed in Example 5, the principal goal requires to book a suitable trip and a suitable accommodation:

[3] In [15, 16] we consider a more general language for specifying composition requirements. Composition requirements consisting of a main and of a recovery condition are enough for the purposes of the paper.

tr.date = date & tr.start = start & tr.destination = dest & tr.pax = client
& ac.date = date & ac.location = dest & ac.pax = client &
& tr.status = booked & ac.status = booked

The recovery condition of the composition requirement specifies that neither the trip not the accommodation has to be booked, and can be represented as follows:

tr.status $\neq$ booked & ac.status $\neq$ booked

In the composition requirements, the semantic annotations introduced in the BPEL processes play a fundamental role for defining conditions on the outcomes of web service executions. For instance, the semantic annotations defining correspondences between the input and output messages of the flight reservation service make it clear which values have to be passed to that service in order to book the flight. Moreover, the value assigned to offer_fl.status in the final activities of the different branches of the process make it possible to distinguish successful executions (with the flight booked) from failures (due to non-available flights or to a reservation cancellation).

Now that we have defined all the inputs of the composition, we are ready to provide a formal definition of composition problem. In the following, we re-use the definitions already proposed in [15, 16], adapted to the case of annotated STSs.

The first step in the definition of composition problem consists in merging the annotated STSs $\Gamma_i = \langle \Sigma_i, \mathcal{T}, \Lambda_i \rangle$ corresponding to the different component services into a single STS $\Gamma_\parallel = \langle \Sigma_\parallel, \mathcal{T}, \Lambda_\parallel \rangle$ defining the combined behavior of the component services. More precisely, $\Sigma_\parallel = \Sigma_1 \parallel \Sigma_2 \parallel \cdots \parallel \Sigma_n$ is the STS defining the *parallel product* of the Σ_i, where each component evolves independently from the others, that is, each transition of $\Sigma_\parallel$ corresponds to a transition of one of the components (see [15, 16] for a formal definition). Moreover, $\Lambda_\parallel$ associates to each state of $\Sigma_\parallel$ all the annotations of the corresponding states of Σ_1, Σ_2,..., Σ_n.

The automated synthesis of the composite service consists in generating a new state transition system Σ_c that, once connected to $\Sigma_\parallel$, satisfies the composition requirement. We now define formally the state transition system describing the behaviors of Σ when connected to Σ_c.

Definition 4 (Controlled system [16]). *Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$ and $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{I}_c, \mathcal{O}_c, \mathcal{R}_c \rangle$ be two state transition systems such that $\mathcal{I} = \mathcal{O}_c$ and $\mathcal{O} = \mathcal{I}_c$. The state transition system $\Sigma_c \triangleright \Sigma$, describing the behaviors of system Σ when controlled by Σ_c, is defined as follows:*

$$\Sigma_c \triangleright \Sigma = \langle \mathcal{S}_c \times \mathcal{S}, \mathcal{S}_c^0 \times \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}_c \triangleright \mathcal{R} \rangle$$

where:

- $\langle (s_c, s), \tau, (s'_c, s) \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s_c, \tau, s'_c \rangle \in \mathcal{R}_c$;
- $\langle (s_c, s), \tau, (s_c, s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s, \tau, s' \rangle \in \mathcal{R}$;
- $\langle (s_c, s), a, (s'_c, s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$, *with* $a \neq \tau$, *if* $\langle s_c, a, s'_c \rangle \in \mathcal{R}_c$ *and* $\langle s, a, s' \rangle \in \mathcal{R}$.

This definition can be easily extended to the case of an annotated STS. Indeed, the composite service Σ_c has no annotations, and hence the annotations of a state of $\Sigma_c \triangleright \Gamma$ are those of the corresponding state in Γ.

In a web service composition problem, we need to generate a Σ_c that guarantees the satisfaction of a composition requirement ρ. This is formalized by requiring that the controlled system $\Sigma_c \triangleright \Gamma_{\|}$ must satisfy ρ, written $\Sigma_c \triangleright \Gamma_{\|} \models \rho$.

Definition 5 (goal satisfaction). *Let $\Gamma = \Sigma_c \triangleright \Gamma_{\|}$, and let $\rho = \langle i, o, p, r \rangle$ be a composition requirement.*

- *We say that Γ strongly satisfies ρ, written $\Gamma \models_s \rho$ if all final states s of Γ are such that $\langle \mathcal{T}, \Lambda(s) \cup i \cup o \rangle \models p$.*
- *We say that Γ weakly satisfies ρ, written $\Gamma \models_w \rho$ if all final states s of Γ are such that $\langle \mathcal{T}, \Lambda(s) \cup i \cup o \rangle \models p$ or $\langle \mathcal{T}, \Lambda(s) \cup i \cup o \rangle \models r$.*
- *We say that Γ satisfies ρ, written $\Gamma \models \rho$ if:*
  - *$\Gamma \models_s \rho$, or*
  - *$\Gamma \models_w \rho$, and there is no $\Gamma' = \Sigma'_c \triangleright \Gamma$ such that $\Gamma' \models_s \rho$.*

According to this definition, a controlled system satisfies a composition goal if: either (1) all the states reached at the end of the computation satisfy the principal condition (in this case we say that the goal is satisfied in a strong way), or (2) all the states reached at the end of the computation satisfy either the principal or the recovery condition (in this case we say that the goal is satisfied in a weak way) and no strong satisfying controller can be defined (i.e., a weak satisfaction is the best we can achieve).

Definition 6 (Composition problem [16]). *Let $\Gamma_1, \ldots, \Gamma_n$ be a set of annotated state transition systems on the same terminology $\mathcal{T}$, and let ρ be a composition requirement. The composition problem for $\Gamma_1, \ldots, \Gamma_n$ and ρ is the problem of finding a state transition system Σ_c such that*[4]

$$\Sigma_c \triangleright (\Gamma_1 \parallel \ldots \parallel \Gamma_n) \models \rho.$$

5 Automated Synthesis of the Process Composition

In [15, 16], an algorithm is described that automatically generates the composite service Σ_c starting from the component services Σ_1,..., Σ_n and the composition requirement ρ. Moreover, the algorithm has been implemented within the ASTRO toolset (see http://www.astroproject.org/) and applied to different domains, showing that it is able to compose complex services in a very small amount of time, much smaller than the time required for a manual implementation of the composite process.

However, in [15, 16] the component STSs did not exploit semantic annotations, and the composition requirement was based on propositional logic instead of description logic and ontologies. Our goal is to extend the approach of [15, 16] to the case of process composition with semantic annotations. There are different ways to achieve this.

[4] The definition of composition problem in [15, 16] takes into account a further requirement for the composite process, that is, it should be deadlock free. Intuitively, this means that the system should never reach a state where both the component services and the composite services are blocked waiting for inputs. For simplicity, we omitted this property from the definition reported in this paper.

Here, we adopt a very simple approach, consisting in transforming the constraints in the composition requirement into propositional formulas through a *grounding* process. Once this has been done, the semantic annotations of the component STSs can be interpreted as "syntactic" annotations, that are not subject to subsumption, and the algorithm of [15, 16] can be reused. We now describe in detail the grounding process.

The aim of the composition process is to define in a suitable way the output individuals of the composition goal, so that the principal or recovery conditions are satisfied. The grounding process consists in looking in the ontology for all the concepts that are subsumed by the concepts of the output concept assertions in the goal, and to reformulate the conditions in the goal using the union of all these concepts for the output individuals, as shown in the following example.

Example 7. The goal in Example 6 defines two output concept assertions, tr: Trip *and* ac: Accommodation*. In the terminology of Figure 1, there are two concepts subsumed by* Trip*, namely* Flight *and* Train*, and two concepts subsumed by* Accommodation*, namely* Hotel *and* GuestHouse*. Taking this into account, the principal goal can be grounded as follows:*

tr: (Trip ⊔ Flight ⊔ Train) & tr.status = booked
  & tr.date = date & tr.start = start & tr.destination = dest & tr.pax = client
& ac: (Accommodation ⊔ Hotel ⊔ GuestHouse) & ac.status = booked
  & ac.date = date & ac.location = dest & ac.pax = client

Similarly, the recovery condition can be grounded as:

tr: (Trip ⊔ Flight ⊔ Train) & tr.status ≠ booked
& ac: (Accommodation ⊔ Hotel ⊔ GuestHouse) & ac.status ≠ booked

The following result shows the correctness of re-using the existing algorithms of [15, 16] on the ground requirement.

Theorem 1 (Soundness and completeness w.r.t. composition). *Let ρ be a composition requirement and ρ_g be the corresponding grounded requirement w.r.t. terminology $\mathcal{T}$. Then*

$$\Sigma_c \triangleright (\Gamma_1 \parallel \ldots \parallel \Gamma_n) \models \rho \qquad \text{iff} \qquad \Sigma_c \triangleright (\Sigma_1 \parallel \ldots \parallel \Sigma_n) \models_g \rho_g$$

where $\models_g$ is satisfiability with goal conditions interpreted as propositional formulas.

6 Related Work and Conclusions

In this paper we propose a practical approach to the composition of semantic web services. We keep separated the procedural and the ontological description of services, and link them through semantic annotations. We then integrate reasoning mechanisms at the ontological and at the process level.

This approach is novel with respect to existing literature. Form the one side, several works propose approaches to process-level composition that do not address explicitly the need for (reasoning about) semantic descriptions of web services [7, 3, 15, 13]. From

the other side, most of the work on automated composition of semantic web services has focused so far on the problem of composition at the functional level, i.e., composition of atomic services that can be executed in a single request-response step (see, e.g. [11, 5]).

The work on WSDL-S and METEOR-S [18, 12, 21] provides semantic annotations for WSDL. It is close in spirit to ours, but does not deal with semantically annotated (BPEL) process-level descriptions of web services. The work in [8] is also close in spirit to our general objective to bridge the gap between the semantic web framework and languages proposed by industrial coalitions. However, [8] focuses on a different problem, i.e., that of extending BPEL with semantic web technology to facilitate web service interoperation, while the problem of automated composition is not addressed.

Recently, an increasing amount of work is dealing with the problem of composing semantic web services taking into account their behavioral descriptions [10, 23, 20, 14, 22]. In this context, the research community is following two related but different main approaches: OWL-S [4] and WSMO [22]. Approaches based on OWL-S [10, 23, 20, 14] are different from the one proposed in this paper, since, in OWL-S, even processes are described as ontologies, and therefore there is no way to separate reasoning about processes and reasoning about ontologies. The approach undertaken in WSMO is closer in spirit to ours: processes are represented as Abstract State Machines, a well known and general formalism to represent dynamic behaviors. The idea underlying WSMO is that the variables of Abstract State Machines are all defined with terms of the WSMO ontological language. Our processes work instead on their own state variables, some of which can be mapped to a separated ontological language, allowing for a minimalist and practical approach to semantic annotations and for effective reasoning to compose services automatically. Indeed, the aim of the work on WSMO is to propose a general language and representation mechanism for semantic web services, while we focus on the problem of providing effective techniques for composing automatically semantic web services.

It would be interesting to investigate how our approach can be applied to WSMO Abstract State Machines rather than BPEL processes, and how the idea of minimalist semantic annotations can be extended to work with WSMO orchestration languages and mechanisms, such that we could exploit our automated composition techniques effectively in this framework. In this context, we plan also to integrate our proposal for automated composition with techniques for web service discovery, a problem that we do not address in this paper.

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. F. Baader and W. Nutt. Basic Description Logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *The Description Logic Handbook*, pages 43–95. Cambridge University Press, 2003.
3. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of E-Services that export their behaviour. In *Proc. ICSOC'03*, 2003.
4. The OWL Services Coalition. OWL-S: Semantic Markup for Web Services, 2003.

5. I. Constantinescu, B. Faltings, and W. Binder. Typed Based Service Composition. In *Proc. WWW'04*, 2004.
6. F. M. Donini. Complexity of Reasoning. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *The Description Logic Handbook*, pages 96–136. Cambridge University Press, 2003.
7. R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: A Look Behind the Curtain. In *Proc. PODS'03*, 2003.
8. D. Mandell and S. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In *Proc. of ISWC'03*, 2003.
9. D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, 2004.
10. S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW'02*, 2002.
11. M. Paolucci, K. Sycara, and T. Kawamura. Delivering Semantic Web Services. In *Proc. WWW'03*, 2002.
12. A. Patil, S. Oundhakar, A. Sheth, and K. Verma. METEOR-S Web Service Annotation Framework. In *Proc. WWW'04*, 2004.
13. M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI'05*, 2005.
14. M. Pistore, P. Roberti, and P. Traverso. Process-level compositions of executable web services: on-the-fly versus once-for-all compositions. In *Proc. ESWC'05*, 2005.
15. M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. ICAPS'05*, 2005.
16. M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *Proc. ICWS'05*, 2005.
17. A. Schaerf. *Query Answering in Concept-Based Knowledge Representation Systems: Algorithms, Complexity, and Semantic Issues*. Dottorato di Ricerca in Informatica, Università degli Studi di Roma "La Sapienza", Italia, 1994.
18. A. Sheth, K. Verna, J. Miller, and P. Rajasekaran. Enhacing Web Service Descriptions using WSDL-S. In *EclipseCon*, 2005.
19. S. Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, RWTH Aachen, 2001.
20. P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *Proc. ISWC'04*, 2004.
21. K. Verma, A. Mocan, M. Zarembra, A. Sheth, and J. A. Miller. Linking Semantic Web Service Efforts: Integrationg WSMX and METEOR-S. In *Proc. SDWP'05*, 2005.
22. SDK WSMO working group. The Web Service Modeling Framework - http://www.wsmo.org/.
23. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S Web Services Composition using SHOP2. In *Proc. ISWC'03*, 2003.