Process-Level Composition of Executable Web Services: "On-the-fly" Versus "Once-for-all" Composition[⋆]

Marco Pistore, Pierluigi Roberti, and Paolo Traverso

University of Trento and ITC-IRST
`pistore@dit.unitn.it`, `{roberti, traverso}@itc.it`

Abstract. Most of the work on automated composition of 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. In this paper, we address the problem of automated composition at the *process level*, i.e., a composition that takes into account that executing a web service requires interactions that may involve different sequential, conditional, and iterative steps. We define two kinds of process-level composition problems: *on-the-fly compositions* that satisfy one-shot user requests specified as composition goals, and a more general form, called *once-for-all compositions*, whose goal is to build a general composed web service that is able to interact directly with the users, receive requests from them, and propose suitable answers. We propose a solution to these two kinds of process-level compositions, and apply the solution to the case of web services described in OWL-S. As a result, we automatically generate process-level compositions as executable OWL-S process models. We show that, while executable on-the-fly compositions can be described as standard OWL-S process models, once-for-all compositions need OWL-S process models to be extended with receive and reply constructs.

1 Introduction

The automated composition of web services is one of the most promising ideas and — at the same time — one of the main challenges for the taking off of service oriented applications: services that are composed automatically can perform new functionalities by interacting with existing services that are published on the web, thus significantly reducing the time and effort needed to develop new web based and service oriented applications. It has been widely recognized that one of the key elements for the automated composition of web services is semantics: unambiguous descriptions of web services capabilities and web service processes, e.g., in standard languages like OWL-S [3] or WSMO [6], can provide the ability to reason about web services, and to automate web services tasks, like web service discovery and composition, see, e.g., [9].

Most of the work on the composition of semantic web services has focused so far on the problem of composition at the *functional level*, i.e., composition of services

[⋆] This work is partially funded 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".

which are considered “atomic” components described in terms of their inputs, outputs, preconditions, and effects, and which can be executed in a simple request-response step (see, e.g. [12, 4]). One of the key open problems for semantic web services is composition at the *process level*, i.e., the problem of generating automatically composed web services that can be directly executed to interact with the component services and achieve the composition goal. The problem of *process-level composition* is far from trivial. We need to take into account the fact that, in real cases, component services cannot in general be executed in a single request-response step. Component services are instead stateful processes, and they require to follow an interaction protocol which may involve different sequential, conditional, and iterative steps. For instance, we cannot in general interact with an “hotel booking” web service in an atomic step. The service may require a sequence of different operations including authentication, submission of a specific request for a room, possibly a negotiation of an offer, acceptance (or refusal) of the offer, and finally payment. These different steps may have conditional, or non-nominal outcomes, e.g., authentication can fail, or there may be no rooms available, and so on. Conditional outputs affect the flow of the interaction, e.g., no request can be submitted if the authentication fails, or, in the case no room is available, no request can be submitted. It may also be the case that the same operation can be repeated iteratively, e.g., in order to refine a request or to negotiate the conditions of the offer.

While the details on the exact protocol required to interact with an existing service are not important in functional-level composition, they become essential when we aim at generating composed web services that are executable. For instance, suppose we compose an “hotel booking” service and a “flight booking” service to obtain an composite service implementing a “virtual travel agency”. The composite service has to guarantee properties such as the fact that, if no available hotel is found, then we do not want to book the flights, or that the nights spent in the hotel are compatible with the arrival and departure dates of the flights. All these properties can be guaranteed only by “interleaving” in a suitable way the interaction protocols of “flight booking” and “hotel booking” (e.g., by booking the flights first, so that the arrival and departure dates are known, but by completing the interaction with “flight booking” and confirming the flight only after a suitable hotel room has been found). This example shows that process-level composition needs to deal with descriptions of web services in terms of complex, composite processes, that consist of arbitrary (conditional and iterative) combinations of atomic interactions, in the style, e.g., of OWL-S process models [3] or WSMO interfaces [6].

In this paper, we address the problem of process-level composition of OWL-S process models. Given a set of services that are available on the web and that are described with OWL-S process models, and given a composition goal that describes a set of desired requirements over the behavior of the composed web service to be generated, we generate automatically an executable composed web service, also described with an OWL-S process model. The composed web service, when executed, interacts with the available services in order to satisfy the goal.

More precisely, we address two different forms of process-level composition. In the first form, called *on-the-fly composition*, the composition is created in order to satisfy a specific request of the customer (e.g., a trip to a specific location in a specific period of time). In this first form of composition, every time the “virtual travel agency” receives a customer’s request, it generates a new composition and executes it, thus obtaining an offer that can be given back to the customer. Moreover, the interactions with the customer of the “virtual travel agency” are forced to follow a simple request/response pattern: after the customer has sent a request, the agency interacts in a suitable way with the hotel and flight services and sends an offer back to the customer.

In the second form, called *once-for-all composition*, the objective of the composition task is to construct a general web service that is able to answer to different requests from customers. For instance, rather than composing the hotel booking and the flight booking services from a request for a specific location and period, the goal is to generate a service that accepts different travel requests, interacts with the customer to propose a suitable travel offer, and finalizes the offer if accepted by the customer. While in on-the-fly compositions the interactions with the customer are restricted to a request/response pattern, in once-for-all composition these interactions can be general. For instance, it is possible to generate a composition that, after receiving a request from the customer, asks the flight and hotel services for possible offers to be combined and sent back to the customer; the customer can now inspect the offers and decide whether to accept or refuse them; the virtual travel agency can confirm or cancel the hotel and flight offers after it receives the feedback from the customer. This way, not only the interactions with the component services (the hotel and flight services) are complex protocols, but also the interactions with the customer become complex, possibly conditional and iterative protocols. Technically speaking, the customer becomes one of the existing component services in input to the composition task.

In this paper we propose a solution to the problem of process-level composition, both in the case of on-the-fly and of once-for-all composition. We define a theoretical framework for automated process-level composition that can be used for both kinds of composition. We compare the on-the-fly composition and once-for-all composition problems, and show that the former is actually a simpler case of the latter. We apply the framework to the case of web services described in OWL-S. In the case of on-the-fly composition, we generate OWL-S process models that can be executed in standard execution engines for OWL-S, e.g., the Mindswap engine [10]. In the case of once-for-all composition, we show that standard OWL-S process models are not adequate to represent executable composed web services. We propose an extension to OWL-S that allows for the execution of web services generated by the automated once-for-all composition task.

2 Motivating Example

In our reference example we have two separate, independent, and existing component services: a “hotel booking” and a “flight booking” service. We aim at composing them into a “virtual travel agency” service that, according to a user travel request, books both hotels and flights.

The HotelBooking service accepts requests for booking a room for a given period of time and location, and if at least one hotel is available, it proposes an offer for a given hotel at a certain cost. This offer can be accepted or refused by the external service that

```
<process:CompositeProcess rdf:ID="HotelBooking">
  <process:Sequence>
    <process:AtomicProcess rdf:about="#HotelRequest">
      <process:Input rdf:ID="Period"><process:parameterType rdf:resource="#Period"/></process:Input>
      <process:Input rdf:ID="Location"><process:parameterType rdf:resource="#Location"/></process:Input>
      <process:ConditionalOutput rdf:ID="Cost">
        <process:coCondition rdf:resource="#HotelBookingPossible"/>
        <process:parameterType rdf:resource="#Cost"/>
      </process:ConditionalOutput>
      <process:ConditionalOutput rdf:ID="Hotel">
        <process:coCondition rdf:resource="#HotelBookingPossible"/>
        <process:parameterType rdf:resource="#Hotel"/>
      </process:ConditionalOutput>
      <process:ConditionalOutput rdf:ID="NA">
        <process:coCondition rdf:resource="#NotHotelBookingPossible"/>
        <process:parameterType rdf:resource="#NotAvailable"/>
      </process:ConditionalOutput>
    </process:AtomicProcess>
    <process:CompositeProcess>
      <process:IfThenElse>
        <process:ifCondition rdf:resource="#HotelBookingPossible"/>
        <process:then>
          <process:CompositeProcess>
            <process:Choice>
              <process:AtomicProcess rdf:ID="#AcceptHotelOffer"/>
              <process:AtomicProcess rdf:ID="#RefuseHotelOffer"/>
            </process:Choice>
          </process:CompositeProcess>
        </process:then>
      </process:IfThenElse>
    </process:CompositeProcess>
  </process:Sequence>
</process:CompositeProcess>

<process:condition rdf:ID="HotelBookingPossible">
  <expr:expressionBody>
    #AvailableHotel(#HotelRequest.Period,#HotelRequest.Location) != UNDEF
  </expr:expressionBody>
</process:condition>
...
<process:sameValues rdf:parseType="Collection">
  <process:ValueOf>
    <process:theParameter rdf:resource="#Hotel"/>
    <process:atProcess rdf:resource="#HotelRequest"/></process:ValueOf>
  <process:valueData>
    <expr:expressionBody>
      #AvailableHotel(#HotelRequest.Period,#HotelRequest.Location)
    </expr:expressionBody>
  </process:ValueData>
</process:sameValues>
...
```

Fig. 1. OWL-S Process model for the hotel booking service

has invoked the hotel service. The OWL-S process model for the hotel service is show in Figure 1. It is a composite service consisting of the atomic process `HotelRequest`, `AcceptHotelOffer`, and `RefuseHotelOffer`. `HotelRequest` receives in input a request for a given `Period` and `Location`. The conditional outputs models the fact the service returns an offer including the price (`Cost`) and the name of the hotel (`Hotel`) only if there exists a hotel with available rooms; a "not available" (`NA`) message is returned otherwise. The decision whether there exists a hotel with available rooms is taken according to condition `HotelBookingPossible`, which is defined in term of function `AvailableHotel`(p, l): this function returns a hotel name if the booking is possible for a period p and location l, and `UNDEF` otherwise (see definition of condition `HotelBookingPossible`). Function `AvailableHotel`(p, l) is used also to decide the value of output parameter `Hotel` of service `HotelRequest` (see the `process:sameValues` declaration). Similarly, output parameter `Cost` is defined in terms of function `CostOfRoom`(p, h), returning the cost of a room for period p in hotel h (this declaration has been omitted in Figure 1). If the booking is possible (see the control construct `IfThenElse` and the condition `HotelBookingPossible`), the hotel service waits for a nondeterministic external decision (control construct `Choice`) that either accepts (`AcceptHotelOffer`) or refuses (`RefuseHotelOffer`) the booking offer.

The FlightBooking service is conceptually similar to the hotel booking one: it accepts requests for booking a flight for a given period and location, and if the flight is available, returns an offer with a cost and a flight. This offer can be accepted or refused by the external service that has invoked the flight booking service. Its OWL-S process model has a similar structure too. It is a composition of the following atomic processes: `FlightRequest`, `AcceptFlightOffer` and `RefuseFlightOffer`. For lack of space we omit the OWL-S description.

Let us now consider an example of *on-the-fly composition*. Our goal is to construct a composed service (say H&F) that, given in input a specific location and time period, interacts with the hotel and the flight service, and books both a flight and an hotel room whenever possible. A possible OWL-S process model for the H&F service is described in Figure 2. The OWL-S specification starts with a declaration of the input (`Period` and `Location`) and output (`Cost`, `Hotel`, and `Flight`, or `NA`) parameters of the composition. Then the body of the composition is defined as a suitable interleaving of the atomic services of the HotelBooking and FlightBooking services. In this example, we choose a H&F service that interacts first with the hotel and then with the flight, therefore, the atomic process `HotelRequest` is called first.[1] If the booking is possible, the condition `HotelBookingPossible` holds and the H&F process starts interacting with the flight service (`FlightRequest`). If a flight is available (condition `FlightBookingPossible`) the H&F process accepts both the offer of the hotel booking service (`AcceptHotelOffer`) and the one of the flight booking service (`AcceptFlightOffer`). If the flight is not available, H&F needs to refuse just the hotel offer (`RefuseHotelOffer`) previously received. We remark that condition `HotelBookingPossible` is defined in terms of the contents of the answer received from atomic service `HotelRequest`. Indeed, the hotel booking is possible if and only if the `NA` does not appear in the answer.

Let us consider now an example of *once-for-all composition*. This time our goal is to construct a composed service (say H&Fservice) that interacts with the customer and with the component services as follows. H&Fservice gets from the user a travel request for a given location and period. It then interacts with the hotel and flight services to discover whether a hotel and a flight are available; if this is the case, H&Fservice interacts again with the user by offering the available hotel and flight at a certain price. At this point, if the user accepts the offer, H&Fservice books both the hotel and the flight. If the offer is refused, we assume that the H&Fservice service simply cancels both the hotel and the flight requests.

The main conceptual difference of "once-for-all" composition w.r.t. "on-the-fly" composition is the fact that the composed service H&Fservice has to interact in a

[1] We can notice that the `Period` parameter passed to the `HotelRequest` process coincides with the `Period` input parameter of the composed process (see the `process:sameValues` declaration at the end of the figure). Similar correspondences, omitted in Figure 2, are also defined for the input and output parameters of the other atomic processes that appear in the composition.

```
<process:CompositeProcess rdf:ID="H&F">
  <process:hasInput rdf:resource="#Period"/>
  <process:hasInput rdf:resource="#Location"/>
  <process:hasResult rdf:resource="#Cost"/>
  <process:hasResult rdf:resource="#Hotel"/>
  <process:hasResult rdf:resource="#Flight"/>
  <process:hasResult rdf:resource="#NA"/

  <process:Sequence>
    <process:AtomicProcess rdf:about="#HotelRequest"/>
    <process:CompositeProcess>
      <process:IfThenElse>
        <process:ifCondition rdf:resource="#HotelBookingPossible"/>
        <process:then>
          <process:CompositeProcess>
            <process:Sequence>
              <process:AtomicProcess rdf:about="#FlightRequest"/>
              <process:CompositeProcess>
                <process:IfThenElse>
                  <process:ifCondition rdf:resource="#FlightBookingPossible"/>
                  <process:then>
                    <process:CompositeProcess>
                      <process:Sequence>
                        <process:AtomicProcess rdf:about="#AcceptHotelOffer"/>
                        <process:AtomicProcess rdf:about="#AcceptFlightOffer"/>
                      </process:Sequence>
                    </process:CompositeProcess>
                  </process:then>
                  <process:else>
                    <process:AtomicProcess rdf:about="#RefuseHotelOffer"/>
                  </process:else>
                </process:IfThenElse>
             </process:CompositeProcess>
           </process:Sequence>
          </process:CompositeProcess>
        </process:then>
      </process:IfThenElse>
    </process:CompositeProcess>
  </process:Sequence>
</process:CompositeProcess>

<process:condition rdf:ID="HotelBookingPossible">
  <expr:expressionBody>
     #HotelRequest.NA = UNDEF
  </expr:expressionBody>
</process:condition>
...
<process:sameValues rdf:parseType="Collection">
  <process:ValueOf>
    <process:theParameter rdf:resource="#Period"/>
    <process:atProcess rdf:resource="#HotelAndFlight"/>
  </process:ValueOf>
  <process:ValueOf>
    <process:theParameter rdf:resource="#Period"/>
    <process:atProcess rdf:resource="#HotelRequest"/>
  </process:ValueOf>
</process:sameValues>?
...
```

Fig. 2. OWL-S Process model for on-the-fly composition: the H&F service

complex way also with the user. More precisely, we can assume that the interactions between the user and the virtual travel agency are modeled as a composite OWL-S process model. According to this process model, the user has to call first a `H&FRequest` atomic web service, passing period and location as parameters; if the requested booking is possible, an answer including hotel name, flight number and cost is returned, otherwise a "not available" answer is returned. In case of positive answer, the user can decide whether to confirm the offer (calling service `AcceptH&FOffer`) or to cancel it (calling service `RefuseH&FOffer`).

The OWL-S process model for the H&Fservice service is described in Figure 3. It is defined as a suitable combination of the atomic web services corresponding to HotelBooking, FlightBooking, and to the process defining the virtual travel agency service (`H&FRequest`, `AcceptH&FOffer`, and `RefuseH&FOffer`).

```
<process:CompositeProcess rdf:ID="HotelAndFlightGeneration">
  <process:Sequence>
    <process:Receive rdf:about="#H&FRequest">
    <process:AtomicProcess rdf:about="#HotelRequest"/>
    <process:CompositeProcess>
      <process:IfThenElse>
        <process:ifCondition rdf:resource="#HotelBookingPossible"/>
        <process:then>
          <process:Sequence>
            <process:AtomicProcess rdf:about="#FlightRequest"/>
            <process:CompositeProcess>
              <process:IfThenElse>
                <process:ifCondition rdf:resource="#FlightBookingPossible"/>
                <process:then>
                  <process:CompositeProcess>
                    <process:Sequence>
                      <process:Reply rdf:about="#H&FRequest">
                      <process:Choice>
                        <process:CompositeProcess>
                          <process:Sequence>
                            <process:Receive rdf:about="#AcceptH&FOffer"/>
                            <process:AtomicProcess rdf:about="#AcceptHotelOffer"/>
                            <process:AtomicProcess rdf:about="#AcceptFlightOffer"/>
                            <process:Reply rdf:about="#AcceptH&FOffer"/>
                          </process:Sequence>
                        </process:CompositeProcess>
                        <process:CompositeProcess>
                          <process:Sequence>
                            <process:Receive rdf:about="#RefuseH&FOffer"/>
                            <process:AtomicProcess rdf:about="#RefuseHotelOffer"/>
                            <process:AtomicProcess rdf:about="#RefuseFlightOffer"/>
                            <process:Reply rdf:about="#RefuseH&FOffer"/>
                          </process:Sequence>
                        </process:CompositeProcess>
                      </process:Choice>
                    </process:Sequence>
                  </process:CompositeProcess>
                </process:then>
                <process:else>
                  <process:CompositeProcess>
                    <process:Sequence>
                      <process:Reply rdf:about="#H&FRequest"/>
                      <process:AtomicProcess rdf:about="#RefuseHotelOffer"/>
                    </process:Sequence>
                  </process:CompositeProcess>
                </process:else>
              <process:IfThenElse>
            </process:CompositeProcess>
          </process:Sequence>
        </process:then>
        <process:else>
           <process:Reply rdf:about="#H&FRequest"/>
        </process:else>
      </process:IfThenElse>
    </process:CompositeProcess>
  </process:Sequence>
</process:CompositeProcess>
```

Fig. 3. OWL-S Process model for once-for-all composition: the H&Fservice service

Notice that the H&Fservice process needs to get the location and period from the user, and then to invoke the hotel and flight service. Similarly, the H&Fservice process needs to propose to the user an offer, and then to either cancel or confirm the bookings of the hotel and flight. These kinds of interactions cannot be modeled with usual OWL-S atomic processes. While OWL-S process models allow for defining a suitable combination of invocations of atomic operations provided by the component services, they do not allow for intermixing these invocations with interactions with the user, since OWL-S process models do not allow to model a process that is both "invoked by" and "invoker of" external services. In order to solve this problem, we introduce new constructs that allow the process to `Receive` messages and `Reply` to messages corresponding to atomic services that the composition should provide to the user.

Indeed, the first step of the H&Fservice process is to wait for a user request (`process:Receive rdf:about="#H&FRequest"`). The rest of the process is the same as the one of the H&F service, but the last steps in the sequence: if the hotel and flight booking is possible, the process replies to the booking request of the user with an offer (`Process:Reply rdf:about="#H&FRequest"`). At this point the H&Fservice waits for a nondeterministic external decision (control construct `Choice`) that either accepts or refuses the offer. If the offer is accepted by the user, the atomic process `AcceptH&FOffer` is activated and then the hotel and flight booking are confirmed. The `Reply` statement communicates to the user that the booking is completed. If the offer is not accepted, the atomic process `RefuseH&FOffer` is activated, and the hotel and flight bookings are refused as well. In this case the `Reply` statement communicates to the user that the booking has been cancelled.

3 Theoretical Framework

Our goal is to automatically generate a new service W (called the *composed service*) that interacts with a set of published web services $W_1, \ldots, W_n$ (called the *component services*) and satisfies a given composition goal. More specifically, we start from n process-level descriptions of web services $W_1, \ldots, W_n$, e.g., their process models in OWL-S, and we automatically translate each of them into a *state transition system* (STS form now on): $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. Intuitively, each Σ_{W_i} is a compact representation of all the possible behaviors, evolutions of the component service W_i. We then construct a *parallel STS* $\Sigma_{\|}$ that combines $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ and represents all the possible evolutions of these component services, without any control by and interaction with the composed service that will be generated. We also formalize the requirements for the composed service as a composition goal, say ρ. Intuitively, ρ describes the functionality that the composed service should have. Given $\Sigma_{\|}$ and ρ, we automatically generate a STS Σ_c that encodes the new service W that has to be generated. Σ_c represents a service that dynamically receives and sends messages from/to the component services $W_1, \ldots, W_n$ and behaves depending on the responses received from $W_1, \ldots, W_n$. Σ_c is such that $\Sigma_c \triangleright \Sigma_{\|}$ satisfies the composition goal ρ, where $\Sigma_c \triangleright \Sigma_{\|}$ represents all the evolutions of the component services $\Sigma_{\|}$ as they are controlled by the composed service Σ_c. The STS Σ_c is then automatically translated into an executable web service, e.g., described as an (extension of a) OWL-S process model.

The formal definition of the process-level composition problem is based on the notion of *state transition system* (STS). STSs are general models for defining dynamic systems that can be in different *states* (some of which are marked as *initial states*) and can evolve to new states as a result of performing some *actions*. Actions are distinguished in *input actions*, which represent the reception of messages, *output actions*, which represent messages sent to external services, and a special action τ, called *internal action*. The action τ is used to represent internal evolutions that are not visible to external services, i.e., the fact that the state of the system can evolve without producing any output, and independently from the reception of inputs. A *transition relation* describes how the state can evolve on the basis of inputs, outputs, or of the internal action τ. Finally, a *labeling function* associates to each state the set of properties $\mathcal{P}rop$ that hold in the state. These properties will be used to define the composition goal ρ.

```
PROCESS HotelBooking;
TYPE    Period;  Location;  Hotel;  Cost;  NA;
STATE   pc: { start, receiveHotelBooking, checkHotelBookingPossible, isHotelBookingPossible,
              isNotHotelBookingPossible, replyHotelBookingPossible, replyHotelBookingNotPossible,
              choiceAcceptHotelOfferRefuseHotelOffer, replyAcceptHotelOffer, replyRefuseHotelOffer,
              endHotelBookingNotPossible, endRefuseHotelOffer, endAcceptHotelOffer};
        period: Period ∪ { UNDEF };
        loc: Location ∪ { UNDEF };
        hotel: Hotel ∪ { UNDEF };
        cost: Cost ∪ { UNDEF };
        na: NA ∪ { UNDEF };
        AvailableHotel[Period,Location]: Hotel ∪ { UNDEF };
        CostOfRoom[Period,Hotel]: Cost;
INIT    pc = start;
        req_period = UNDEF;
        req_loc = UNDEF;
        offer_hotel = UNDEF;
        offer_cost = UNDEF;
INPUT   HotelBooking(Period, Location);
        AcceptHotelOffer();
        RefuseHotelOffer();
OUTPUT  HotelBookingAnswer(Hotel ∪ {UNDEF}, Cost ∪ {UNDEF}, NA ∪ {UNDEF});
        AcceptHotelOfferAnswer();
        RefuseHotelOfferAnswer();
TRANS   pc = isHotelBookingPossible —[TAU]—> pc = replyHotelBookingPossible,
        pc = start —[TAU]—> pc = receiveHotelBooking;
        pc = receiveHotelBooking —[INPUT HotelBooking(period,location)]—> pc = checkHotelBookingPossible;
        pc = checkHotelBookingPossible ∧ AvailableHotel[period,location] ≠ UNDEF
                                              —[TAU]—> pc = isHotelBookingPossible;
        pc = checkHotelBookingPossible ∧ AvailableHotel[period,location] = UNDEF
                                              —[TAU]—> pc = isNotHotelBookingPossible;
        pc = isHotelBookingPossible —[TAU]—> pc = replyHotelBookingPossible,
                                              hotel = AvailableHotel[period,location],
                                              cost = CostOfRoom[period,hotel];
        pc = replyHotelBookingPossible —[OUTPUT HotelBookingAnswer(hotel,cost,na)]—>
                                              pc = choiceAcceptHotelOfferRefuseHotelOffer;
        pc = choiceAcceptHotelOfferRefuseHotelOffer —[INPUT AcceptHotelOffer]—> pc = replyAcceptHotelOffer;
        pc = replyAcceptHotelOffer —[OUTPUT AcceptHotelOfferAnswer]—> pc = endAcceptHotelOffer;
        pc = choiceAcceptHotelOfferRefuseHotelOffer  —[INPUT RefuseHotelOffer]—> pc = replyRefuseHotelOffer;
        pc = replyRefuseHotelOffer —[OUTPUT RefuseHotelOfferAnswer]—> pc = endRefuseHotelOffer;
        pc = isNotHotelBookingPossible —[TAU]—> pc = replyHotelBookingNotPossible, na ∈ NA;
        pc = replyHotelBookingNotPossible —[OUTPUT HotelBookingAnswer(hotel,cost,na)]—>
                                              pc = endHotelBookingNotPossible;
```

Fig. 4. The STS for the HotelBooking process

Definition 1 (State transition system (STS)). *A* state transition system Σ *is a tuple* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ *where* $\mathcal{S}$ *is the set of states,* $\mathcal{S}^0 \subseteq \mathcal{S}$ *is the set of initial states,* $\mathcal{I}$ *is the set of input actions,* $\mathcal{O}$ *is the set of output actions,* $\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{S}$ *is the transition relation, and* $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{P}rop}$ *is the labeling function.*

We assume that infinite loops of τ-transitions cannot appear in the system (i.e., the service has to interacting with the environment after a finite number of steps). We also assume that there is no state which originates both input and output transitions.

Figure 4 shows a description of the STS corresponding to the HotelBooking web service (see Figure 1). The set of states $\mathcal{S}$ models the steps of the evolution of the process and the values of its variables. The special variable `pc` implements a "program counter" that holds the current execution step of the service (e.g., `pc` has value `receiveHotelBooking` when the process is waiting to receive a booking request, and value `checkHotelBookingPossible` when it is ready to check whether the booking is possible). Other variables like `location` or `cost` correspond to those used in message exchanges. Finally, arrays like `AvailableHotel` or `CostOfRoom` describe predicates and functions expressing properties of the web service (e.g., the fact that there is an hotel available for a given period and in a given location, or the cost of a room in a given hotel for a given period). In the initial states $\mathcal{S}^0$ the `pc` is set to `START`, while all the other basic variables are undefined. The initial values of the arrays `AvailableHotel` and `CostOfRoom` are unspecified, since they can assume any value in the domain.

The evolution of the process is modeled through a set of possible transitions. Each transition defines its applicability conditions on the source state, its firing action, and the destination state. For instance,

```
pc = checkHotelBookingPossible &
AvailableHotel[period,location] ≠ UNDEF −[TAU]−>          (1)
pc = isHotelBookingPossible
```

asserts that an action τ can be executed in state `checkHotelBookingPossible`, leading to the state `isHotelBookingPossible`, if there is some hotel with available rooms for the specified period and location. We remark that each `TRANS` clause of Figure 4 corresponds to different elements in the transition relation $\mathcal{R}$: e.g., clause (1) generates different elements of $\mathcal{R}$, depending on the values of variables `period` and `location`.

According to the formal model, we distinguish among three different kinds of actions. The input actions $\mathcal{I}$ model all the incoming requests to the process and the information they bring (i.e., `HotelBooking` is used for the receiving of the booking request). The output actions $\mathcal{O}$ represent the outgoing messages (in our case, the replies to the input requests). The action τ is used to model internal evolutions of the process, as for instance assignments and decision making.

Finally, the set of properties $\mathcal{P}rop$ of the STS are expressions of the form `<variable> = <value>` or `<array>[idx_1,...,idx_n] = <value>`, and the labeling function is the obvious one.

A remark is in order. The definition of STS provided in Figure 4 is parametric w.r.t. the types `Period`, `Location`, `Hotel`, `Cost`, and `NA` used in the messages. In order to obtain a concrete STS and to apply the automated composition techniques described in Section 4, specific ranges have to be assigned to these types. Different approaches are possible for defining these ranges. The simpler approach is to associate finite (and possibly small) ranges to each type. This approach, makes the definition of the STS easy (and allows for an efficient automated composition), however, it has the disadvantage of imposing unrealistic assumptions on the data types handled by the web services. A more realistic (but more complex) approach consists of using abstract models for the data types, avoiding an enumeration of all concrete values that these types can assume, and representing explicitly only those aspects of the data type that are relevant for the task at hand. This second approach will be addressed in future work (see Section 5).

We now formally define the *parallel product* $\Sigma_1 \parallel \Sigma_2$ of the two STSs Σ_1 and Σ_2. It models the fact that the systems may evolve independently (i.e., without direct communications), and is used to generate $\Sigma_\parallel$ from the component web services $W_1, \ldots, W_n$: formally, $\Sigma_\parallel = \Sigma_{W_1} \parallel \ldots \parallel \Sigma_{W_n}$.

Definition 2 (parallel product). *Let* $\Sigma_1 = \langle \mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{R}_1, \mathcal{L}_1 \rangle$ *and* $\Sigma_2 = \langle \mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{R}_2, \mathcal{L}_2 \rangle$ *be two STSs with* $(\mathcal{I}_1 \cup \mathcal{O}_1) \cap (\mathcal{I}_2 \cup \mathcal{O}_2) = \emptyset$. *The parallel product* $\Sigma_1 \parallel \Sigma_2$ *of* Σ_1 *and* Σ_2 *is defined as:*

$$\Sigma_1 \| \Sigma_2 = \langle \mathcal{S}_1 \times \mathcal{S}_2, \mathcal{S}_1^0 \times \mathcal{S}_2^0, \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{R}_1 \| \mathcal{R}_2, \mathcal{L}_1 \| \mathcal{L}_2 \rangle$$

where:

- $\langle (s_1, s_2), a, (s'_1, s_2)\rangle \in (\mathcal{R}_1 \| \mathcal{R}_2)$ *if* $\langle s_1, a, s'_1\rangle \in \mathcal{R}_1$*;*
- $\langle (s_1, s_2), a, (s_1, s'_2)\rangle \in (\mathcal{R}_1 \| \mathcal{R}_2)$ *if* $\langle s_2, a, s'_2\rangle \in \mathcal{R}_2$*;*

and $(\mathcal{L}_1 \| \mathcal{L}_2)(s_1, s_2) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$.

The automated composition problem consists in generating a STS Σ_c that controls $\Sigma_\|$ by satisfying the composition requirement ρ. We now define formally the STS describing the behaviors of a STS Σ when controlled by Σ_c.

Definition 3 (controlled system). *Let* $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L}\rangle$ *and* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_\emptyset\rangle$ *be two state transition systems, where* $\mathcal{L}_\emptyset(s_c) = \emptyset$ *for all* $s_c \in \mathcal{S}_c$*. The STS* $\Sigma_c \triangleright \Sigma$*, describing the behaviors of system* Σ *when controlled by* Σ_c*, is defined as:*

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

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* $\langle s, a, s'\rangle \in \mathcal{R}$*.*

Notice that we require that the inputs of Σ_c coincide with the outputs of Σ and vice-versa. Notice also that, although the systems are connected so that the output of one is associated to the input of the other, the resulting transitions in $\mathcal{R}_c \triangleright \mathcal{R}$ are labelled by input/output actions. This allows us to distinguish the transitions that correspond to τ actions of Σ_c or Σ from those deriving from communications between Σ_c and Σ. Finally, notice that we assume that Σ_c has no labels associated to the states.

A STS Σ_c may not be adequate to control a system Σ. Indeed, we need to guarantee that, whenever Σ_c performs an output transition, then Σ is able to accept it, and vice-versa. We define the condition under which a state s of Σ is able to accept a message. We assume that s can accept a message a if there is some successor s' of s in Σ, reachable from s through a chain of τ transitions, such that s can perform an input transition labelled with a. Vice-versa, if state s has no such successor s', and message a is sent to Σ, then a deadlock situation is reached. In the following definition, and in the rest of the paper, we denote by τ-closure(s) the set of the states reachable from s through a sequence of τ transitions, and by τ-closure(S) with $S \subseteq \mathcal{S}$ the union of τ-closure(s) on all $s \in S$.

Definition 4 (deadlock-free controller). *Let* $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L}\rangle$ *be a STS and* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_\emptyset\rangle$ *be a controller for* Σ*.* Σ_c *is said to be* deadlock free *for* Σ *if all states* $(s_c, s) \in \mathcal{S}_c \times \mathcal{S}$ *that are reachable from the initial states of* $\Sigma_c \triangleright \Sigma$ *satisfy the following conditions:*

- *if* $\langle s, a, s'\rangle \in \mathcal{R}$ *with* $a \in \mathcal{I}$ *then there is some* $s'_c \in \tau$*-closure*(s_c) *such that* $\langle s'_c, a, s''_c\rangle \in \mathcal{R}$ *for some* $s''_c \in \mathcal{S}_c$*; and*
- *if* $\langle s_c, a, s'_c\rangle \in \mathcal{R}_c$ *with* $a \in \mathcal{O}$ *then there is some* $s' \in \tau$*-closure*(s) *such that* $\langle s', a, s''\rangle \in \mathcal{R}$ *for some* $s'' \in \mathcal{S}$*.*

The automated composition task needs to generate a deadlock-free Σ_c that guarantees the satisfaction of a composition goal ρ. This is formalized by requiring that the controlled system $\Sigma_c \triangleright \Sigma_{\|}$ must satisfy ρ, written $\Sigma_c \triangleright \Sigma_{\|} \models \rho$. The definition of this requirement is technical: it depends on the definition of the executions of $\Sigma_c \triangleright \Sigma_{\|}$, and these have to be defined taking into account the special role of τ actions, which describe internal, "unobservable" evolutions of the system. For lack of space, we omit the formal definition of $\Sigma_c \triangleright \Sigma_{\|} \models \rho$: the interested reader can find it in [14].

Definition 5 (composition problem). *Let $\Sigma_1, \ldots, \Sigma_n$ be a set of state transition systems, and let ρ be a composition goal. The composition problem for $\Sigma_1, \ldots, \Sigma_n$ and ρ is the problem of finding a controller Σ_c that is deadlock-free and such that $\Sigma_c \triangleright (\Sigma_1 \parallel \ldots \parallel \Sigma_n) \models \rho$.*

4 Automated Process-Level Composition

The automated process-level composition of web services is obtained in four steps:

1. **From Web Services to State Transition Systems.** The process-level descriptions of the available component web services are translated into STSs.
2. **Expressing Composition Goals.** This step consists in the formalization of the goal ρ that defines the functionality that the composed web service should provide.
3. **Synthesis of the Composition.** During this step, the STS implementing the composed web services is automatically generated starting from the STSs constructed in step 1 and from the goal specified in step 2.
4. **Deployment and Execution of the Composed Service.** In this step, the STS generated in step 3 is translated into an executable web service.

In this paper we consider web services described in OWL-S. The translation from OWL-S process models to STS (step 1) is performed along the lines described in Section 3 (see also [16]). More precisely, the STS of Figure 4 is obtained from the OWL-S process model in Figure 1 (the STS has been slightly edited for improving readability).

The definition of the composition goal (step 2) depends on the kind of process-level composition we are interested in. In the case of *on-the-fly composition*, we have to generate a service that satisfies a specific customer's request. For instance, in the example of the hotel and flight services, the customer's request specifies a given location l to be visited in a given period of time p. In order to satisfy this request, the composed service has to find a flight and a hotel room compatible with the request of the customer. The composition goal could be the something like "if a travel offer is available, then sell a travel offer to the customer". In our example, the offer is possible if it is possible to book a flight and a hotel for the period and for the location specified by the customer. The fact that the offer is sold is described by requiring that the HotelBooking and the FlightBooking processes reach the end state corresponding to an accepted booking offer. The goal condition can hence be specified by the formula:

```
    HotelBooking.AvailableHotel[p,l] ≠ UNDEF ∧
       FlightBooking.AvailableFlight[p,l] ≠ UNDEF
 →  HotelBooking.pc = endAcceptHotelBooking ∧
       FlightBooking.pc = endAcceptFlightBooking ∧
       h = HotelBooking.AvailableHotel[p,l] ∧
       f = FlightBooking.AvailableFlight[p,l] ∧
       c = HotelBooking.CostOfRoom[p,h] +
          FlightBooking.CostOfFlight[f]
```

where `c`, `f`, and `h` describe, respectively, the cost of the offer, and the specific flight and hotel information. This goal has to be interpreted as a condition that must hold at the end of the execution of the composed service.

The case of *once-for-all composition* is more complex. In our reference example, we want to automatically generate a composed service whose requirement is to "sell a travel offer to the customer" that satisfies a generic customer request. This means we want the composed service to reach the situation where an offer has been made to the customer, the customer has confirmed this offer, and the service has confirmed the corresponding (sub-)offers to the HotelBooking and FlightBooking services. However, the hotel may have no available rooms, the flight may not be possible, the user may not accept the offer due to its cost... We cannot avoid these situations, and we cannot therefore ask the composed service to guarantee this requirement. Nevertheless, we would like the composed service to try (do whatever is possible) to satisfy the request. If it is not possible to satisfy the main requirement, i.e., to sell the offer, our requirement for the composed service is to book neither a room nor a flight. Indeed we do not want the service to book and pay for rooms when flights are not available, or to book flights when rooms are not available, or to pay for rooms and flights that will not be accepted by the customer. Our composition goal is therefore a combination of the main requirement ("sell a travel offer to the customer") and of a secondary requirement ("book neither a room nor a flight"), i.e., something like: "try to sell a travel offer to the customer; upon failure, do book neither a room nor a flight". Notice that the secondary requirement ("book neither a room nor a flight") has a different strength w.r.t. the primary one ("sell a travel offer to the customer"). We write "do" satisfy, rather than "try" to satisfy. Indeed, in the case the primary requirement is not satisfied, we want the secondary requirement to be guaranteed.

We need a formal language that can express requirements of this kind, including conditions of different strengths (like "try" and "do"), preferences among different (e.g., primary and secondary) requirements, and failure recovery conditions. For this reason, we cannot simply use a state formula as we did for the case of on-the-fly composition. We use instead the EAGLE language, which has been designed with the purpose to satisfy such expressiveness. A detailed definition and a formal semantics for the EAGLE language can be found in [5]. Here we just explain how EAGLE can express the composition requirement of the running example. The EAGLE formalization of the requirement is the following:

TryReach
```
   HotelBooking.pc = endAcceptHotelBooking ∧
   FlightBooking.pc = endAcceptFlightBooking ∧
   Customer.pc = endAcceptHandFBooking ∧
   Customer.h = HotelBooking.AvailableHotel[Customer.p,Customer.l] ∧
   Customer.f = FlightBooking.AvailableFlight[Customer.p,Customer.l] ∧
   Customer.c = HotelBooking.CostOfRoom[Customer.p,Customer.h] +
         FlightBooking.CostOfFlight[Customer.f]
```

Fail DoReach

```
    HotelBooking.pc ≠ endAcceptHotelBooking ∧
    FlightBooking.pc ≠ endAcceptHotelBooking ∧
    Customer.pc ≠ endAcceptHandFBooking
```

The goal is of the form "**TryReach** c **Fail DoReach** d". **TryReach** c requires a service that tries to reach condition c, in our case the condition "sell a travel offer to the customer". During the execution of the service, a state may be reached from which it is not possible to reach c, e.g., since the flight is not available. When such a state is reached, the requirement **TryReach** c fails and the recovery condition **DoReach** d, in our case "book neither a room nor a flight" is considered.

The core part of the automated composition task is step 3. A formal definition of the problem solved by this task is given in Definition 5. Starting from a set of STSs modeling the existing web services and from a composition goal, a new STS is generated which implements the composed service. We see the process level composition problem as a planning problem, where the parallel STS $\Sigma_{\|}$ is the planning domain, ρ is the planning goal, and Σ_c is the generated plan that achieves the goal ρ. Notice that the planning problem is far trivial and cannot be reduced to a classical planning problem. We need planning techniques able to deal with planning under uncertainty, and more precisely with planning in nondeterministic domains and under partial observability. Indeed, it is clear from Section 3 that the planning domain is nondeterministic, i.e., actions executed in the same state can have different outcomes, and partially observable, i.e., only part of the domain information is available at run time, and, at planning time, the planner must therefore deal with the problem that the domain might be in a set of possible states. Moreover, the generated plan Σ_c is not simply a sequence of actions, must represent conditional and iterative courses of actions. Finally, in the case of *once-for-all composition*, we need to deal with extended goals, i.e., with goals that are not simply sets of desired states, but can express complex temporal and preference conditions, as discussed for Step 2. For all these reasons, we generate the STS Σ_c by using the "planning as model checking" technique [2, 5, 1], which has been shown to provide a practical solution to the problem of planning under uncertainty, and has been shown experimentally to scale up to large state spaces. A detailed description of how "planning as model checking" can be applied to solve the composition problem can be found in [14].

In step 4, the automatically generated STS Σ_c is translated into an executable web service. In the case of on-the-fly composition, the code is immediately executed. In the case of once-for-all composition, the code is deployed, and is ready to answer to users' requests. In this paper, we translate the STS into an OWL-S process model. Examples of (excerpts from) the generated OWL-S process models are given in Figures 2 and 3 for the cases of on-the-fly and once-for-all compositions, respectively.

5 Conclusions and Related Work

There is a large amount of literature addressing the problem of automated composition of web services. However, most of the approaches address composition at the functional level (see, e.g. [12, 4]), and much less emphasis has been devoted to the problem of process-level composition. Different planning approaches have been proposed to address the problem of on-the-fly composition, from HTNs [17] to regression planning based on extensions of PDDL, to STRIPS-like planning for composing services described in DAML-S [15]. However, none of these techniques addresses the problem of composing web services with conditional outputs, non-nominal outcomes, and with process models describing interaction protocols that include conditional and iterative steps. In [8, 11, 7], the authors propose an approach to the automated composition of web services based on a translation of DAML-S to situation calculus and Petri Nets. Also in these papers, however, the automated composition is limited to sequential composition of atomic services, and composition requirements are limited to reachability conditions.

As far as we know, the only approach that deals with process-level composition of semantic web services is described in our previous work [16]. In that paper, we proposed a technique for automated composition where the component web services were described as OWL-S composite processes and the generated composed services were emitted as a BPEL4WS programs. In this paper, we provide a substantial contribution with respect to [16], namely composed services are generated as OWL-S process models. In this way, we support the automated composition fully within a semantic web framework, we generate web services enriched with semantic annotations, and we allow the re-use of the generated services as input to the automated composition task. This opens the way to the incremental development of more and more complex web services, within a service oriented development process, where new executable web services are composed and then re-used to compose further services.

As a witness of the increasing interest in executable semantic web services, recent efforts within the semantic web community are addressing the problem of providing execution engines for semantic web services. For instance, the "Mindswap" OWL-S execution engine developed at the University of Maryland [10] can execute (a subset of) OWL-S process models, while there is significant on going work in providing an execution engine, called WSMX, for the WSMO language [6].

In the future, we will address the problem of associating finite ranges to the data types in the generation of the state transition systems from the process-level descriptions of web services. In particular, we intend to investigate techniques for deciding the right size of these ranges, so that the generated web service implements a general solution that can be adopted independently of the actual ranges. We also intend to investigate the so called "knowledge-level" techniques [13] for the composition of web services: these techniques prevent the necessity of fixing a finite range and permit the generation of a general solution. Our future plans also include the integration of the automated composition task with reasoning techniques for discovery and selection of web services, and the extension of the techniques within the WSMO framework [6]. A further important extension of the framework will address how to automatically generate semantic annotations for the generated web services. A possibility for OWL-S will be to exploit logical frameworks, e.g., based on description and dynamic logic, which will be able to derive preconditions and effects of composed services.

References

1. P. Bertoli, A. Cimatti, M. Pistore, and P. Traverso. A Framework for Planning with Extended Goals under Partial Observability. In *Proc. ICAPS'03*, 2003.
2. 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.
3. The OWL Services Coalition. OWL-S: Semantic Markup for Web Services. In *Technical White paper (OWL-S version 1.0)*, 2003.
4. I. Constantinescu, B. Faltings, and W. Binder. Typed Based Service Composition. In *Proc. WWW'04*, 2004.
5. U. Dal Lago, M. Pistore, and P. Traverso. Planning with a Language for Extended Goals. In *Proc. AAAI'02*, 2002.
6. The Web Service Modeling Framework. SDK WSMO working group - http://www.wsmo.org/.
7. S. McIlraith and R. Fadel. Planning with Complex Actions. In *Proc. NMR'02*, 2002.
8. S. McIlraith and S. Son. Adapting Golog for composition of semantic web Services. In *Proc. KR'02*, 2002.
9. S. McIlraith, S. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.
10. Mindswap. Maryland Information and Network Dynamics lab Semantic Web Agents Projects - http://www.mindswap.org/.
11. S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW'02*, 2002.
12. M. Paolucci, K. Sycara, and T. Kawamura. Delivering Semantic Web Services. In *Proc. WWW'03*, 2002.
13. R. Petrick and F. Bacchus. A Knowledge-Based Approach to Planning with Incomplete Information and Sensing. In *Proc. AIPS'02*, 2002.
14. M. Pistore, P. Traverso, and P. Bertoli. Automated composition of web services by planning in asyncronous domains. In *Proc. ICAPS'05*, 2005.
15. M. Sheshagiri, M. desJardins, and T. Finin. A Planner for Composing Services Described in DAML-S. In *Proc. AAMAS'03*, 2003.
16. P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *Proc. ISWC'04*, 2004.
17. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S Web Services Composition using SHOP2. In *Proc. ISWC'03*, 2003.