An Approach for the Automated Composition of BPEL Processes

M. Pistore
University of Trento
Via Sommarive 14
38050 Povo (TN), Italy
pistore@dit.unitn.it

P. Traverso, P. Bertoli, A. Marconi
ITC-IRST
Via Sommarive 18
38050 Povo (TN), Italy
[traverso,bertoli,marconi]@itc.it

Abstract

We describe a technique for the automated synthesis of new composite web services. Given a set of component services described as abstract BPEL4WS *processes enriched with semantic annotations, and given a composition requirement, we automatically generate an executable* BPEL4WS *process that, once deployed, is able to interact with the components to satisfy the requirement. We focus in particular on the description of the annotations that we have to add to the abstract* BPEL4WS *processes in order to capture the "semantic" aspects of their execution, and of the role that these semantic annotations play in the automated composition task.*

1. Introduction

Web services provide the basis for the development and execution of business processes that are distributed over the network and available via standard interfaces and protocols. *Service composition* [10] is one of the most promising ideas underlying web services: *composite web services* perform new functionalities by interacting with pre-existing services, called *component web services*. Service composition has the potential to reduce development time and effort for new applications by re-using published services.

Different standards and languages have been proposed to develop composite web services. BPEL4WS (Business Process Execution Language for Web Services) [2] is one of the emerging standards for describing the behaviour of the services. In BPEL4WS components are represented as stateful processes that publish an interaction protocol with external web services. Such an interaction flow is published as BPEL4WS *abstract* process specifications, while BPEL4WS *executable* processes are used to implement the internal processes (not published externally), which can be executed on standard business processes execution engines (see, e.g., [1, 13]). In this context, the development of a composite web service must be driven by the analysis of published abstract BPEL4WS specifications of component services and by requirements and constraints the composite service has to satisfy, and results in the generation of an executable BPEL4WS process.

The manual analysis of abstract BPEL4WS processes, and the implementation of programs that interact with them, is a very difficult, time consuming, and error prone task; efficient automated support is needed to achieve cost-effective, practically exploitable web service composition.

In previous works [16, 15, 14] we have proposed a technique for the automatic synthesis of executable composite web services from abstract BPEL4WS processes. This is achieved in the following steps.

1. We automatically translate the abstract BPEL4WS specifications of the component services into state transition systems, which formally describe their behaviors.
2. We formalize the requirements for the composite service using expressive requirement languages developed in the field of AI planning.
3. Given the state transition systems of the components and the formal requirements, we automatically generate a state transition system that encodes a process behaviour which satisfies the requirements.
4. We automatically translate the resulting state transition system into an executable BPEL4WS program.

In this paper we report the approach introduced in [16, 15, 14]. We focus in particular on a key aspect of this approach which has not been covered by the previous papers: the semantic annotations that have to be added to the abstract BPEL4WS specifications of the

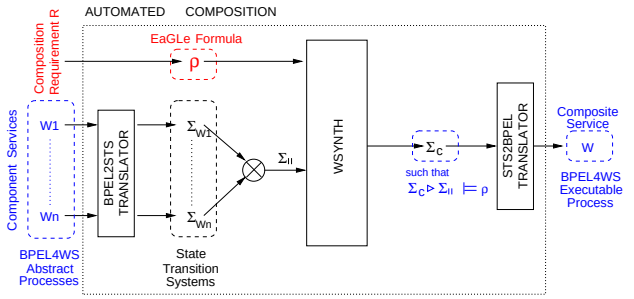

Figure 1. The Approach.

component services in order to allow for their composition. Indeed, while BPEL4WS provides a precise description of the behaviors of the component services in terms of the interactions that it establishes with its invokers, it does not capture at all the "semantic" aspects of such interactions. Additional 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).

The paper is structured as follows. In Section 2 we give an overview of the the approach. We do this by introducing an explanatory example which will be used all along the paper. In Section 3 we describe the semantic annotations that we add to the abstract BPEL4WS processes in order to capture their outcomes and effects, while in Section 4, we explain how these annotated BPEL4WS processes can be translated into state transition systems. In Section 5, we describe the language we exploit for expressing composition requirements and we show how it exploits the semantic annotations which enrich the BPEL4WS processes. We explain how we automatically generate BPEL4WS executable processes in Section 6. We conclude with a comparison with related work in Section 7.

2. Overview

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 specifically (see Figure 1) we assume that component services are described as BPEL4WS abstract processes. Given n BPEL4WS abstract processes $(W_1, \ldots, W_n)$, the BPEL2STS module automatically translates each of them into a *state transition system* $(\Sigma_{W_1}, \ldots, \Sigma_{W_n})$. Intuitively, each Σ_{W_i} is a compact representation of all the possible behaviors of the component service W_i.

These $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ can be combined into a *parallel state transition system* $\Sigma_{\|}$ that represents all the possible behaviors of the component services, and which is one of the inputs to the WSYNTH tool that performs the synthesis.

The second input to WSYNTH consists of the requirements for the composite service. They are formalized as a formula ρ of the requirement language EAGLE (see Section 5). Given $\Sigma_{\|}$ and ρ, WSYNTH generates a new state transition system Σ_c. Σ_c encodes the new service W, which dynamically receives and sends invocations from/to the component services $W_1, \ldots, W_n$ and behaves depending on responses received from these services. Σ_c is such that $\Sigma_c \triangleright \Sigma_{\|}$ satisfies the requirement ρ (written $\Sigma_c \triangleright \Sigma_{\|} \models \rho$), where $\Sigma_c \triangleright \Sigma_{\|}$ represents all the evolutions of the component services as they are controlled by the composite service. The state transition system Σ_c is then given in input to the STS2BPEL module which translates it into an executable BPEL4WS process, implementing the composite service.

In the rest of the paper, we will describe the steps in the synthesis process through the following example.

Example 1 *Our reference example consists in providing a furniture purchase & delivery service, say the P&S service, which satisfies some user request, by combining two separate, independent, and existing services: a furniture producer Producer, and a delivery service Shipper. The idea is that of combining these two services so that the user may directly ask the composed service P&S to purchase and deliver a given item at a given place. To do so, we exploit a description of the expected interaction between the P&S service and the other actors. In the case of the Producer and of the Shipper, the interactions are defined in terms of the service requests that are accepted by the two actors. In the case of the User, we describe the interactions in terms of the requests that the user can send to the P&S. As a consequence, the P&S service should interact with three available services: Producer, Shipper, and User. These are the three available services which are described as BPEL4WS abstract processes and translated to state transition systems by the BPEL2STS module in Figure 1. The problem is to automatically generate the concrete BPEL4WS implementation of the P&S service.*

In the following, we describe informally the three available services. Producer accepts requests for providing information on a given product and, if the product is available, it provides information about its size. The Producer also accepts requests for buying a given product, in which case it returns an offer with a cost and production time.

This offer can be accepted or refused by the external service that has invoked the Producer. The Shipper service receives requests for transporting a product of a given size to a given location. If delivery is possible, Shipper provides a shipping offer with a cost and delivery time, which can be accepted or refused by the external service that has invoked Shipper. The User sends requests to get a given item at a given location, and expects either a negative answer if this is not possible, or an offer indicating the price and the time required for the service. The user may either accept or refuse the offer. Thus, a typical interaction between P&S and the component services would go as follows:

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

Of course this is only the nominal case, and other interactions should be considered, e.g., for the cases the producer and/or delivery services are not able to satisfy the request, or the user refuses the final offer.

3. Annotated BPEL4WS processes

BPEL4WS [2] provides an operational description of the (stateful) behavior of web services on top of the service interfaces defined in their WSDL specifications. An abstract BPEL4WS 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 2 *In Figure 2 we report the abstract* BPEL4WS *specification of the Shipper in the scenario discussed in Example 1.*[1] *We start with a declaration of the variables that are used in input/output messages:* `req` *is the input variable that specifies the size of the item to ship and the destination;* `offer` *is the Shipper offer to the client, including the cost and the time for delivery. The* `messageType` *declaration specifies the structure of a variable used for sending/receiving messages. Such structure, along with the definition of the* `messageType` *structures involved in the different web service invocations, are detailed in the* WSDL *specification associated to the* BPEL4WS *code, which we omit for lack of space.*

The rest of the abstract BPEL4WS *specification describes the interaction flow. The Shipper is activated by a request from a client (*`receive` *instruction corresponding to operation* `request`*). The information on the size and of the destination location submitted by the client are stored in variable* `req`*.*

Depending on its internal availability (`switch` *instruction named* `checkAvailability`*), the shipper can either send an answer refusing the request (*`invoke` *instruction corresponding to operation* `not_avail`*), or prepare and send the information regarding an offer for the delivery. In the latter case, the cost and delay of the delivery are computed and assigned to variable* `offer` *within the* `assign` *statement named* `prepareOffer`*. The way in which the information are obtained are not disclosed and published by the abstract* BPEL4WS*: 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 that is robust to changes with respect to the actual way in which the internal business logic is defined.*

Once the offer has been sent to the client (`invoke` *instruction corresponding to operation* `offer`*), the Shipper suspends (instruction* `pick`*), waiting for the customer either to acknowledge the acceptance of the shipping offer (*`onMessage` *specification corresponding to operation* `ack`*), or to refuse the offer (*`onMessage` *specification corresponding to operation* `nack`*). Only in the former case the interaction with the service is successful, and a shipping agreement is established.*

A BPEL4WS specification describes in a very detailed way 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 BPEL4WS specification with "semantic annotations".[2]

1 The specification contains some annotations in boldface: they are not part of the BPEL4WS language, and we will explain their meaning later on in this section.

2 In Figure 2 we have adopted a concrete syntax for these an-

```
<process name="Shipper">
  <partnerLinks>
    <partnerLink name="client" partnerLinkType="Shipper_PLT" myRole="Shipper_Server" partnerRole="Shipper_Client" />
  </partnerLinks>

  <variables>
    <variable name="req" messageType="shippingRequest" />
      <!-- "req" has two parts: "/req/size" and "/req/location" -->
    <variable name="offer" messageType="shippingOffer" />
      <!-- "offer" has two parts: "/offer/cost" and "/offer/delay" -->
  </variables>

  <sequence name="main">
    <receive partnerLink="client" portType="Shipper_PT" operation="request" variable="req" />
    <switch name="checkAvailability">
      <case name="isNotAvailable">
        <invoke partnerLink="client" portType="Producer_Callback_PT" operation="not_avail" />
        <empty semeffect="not shippingAgreement">
      </case>
      <otherwise name="isAvailable">
        <assign name="prepareOffer">
          <copy><from opaque="yes" semvalue="costOf(/req/size,/req/location)" />
            <to variable="offer" part="cost" /></copy>
          <copy><from opaque="yes" semvalue="delayOf(/req/size,/req/location)" />
            <to variable="offer" part="delay" /></copy>
        </assign>
        <invoke partnerLink="client" portType="Shipper_Callback_PT" operation="offer" inputVariable="offer" />
        <pick name="waitAcknowledge">
          <onMessage partnerLink="client" portType="Producer_PT" operation="ack">
            <empty semeffect="shippingAgreement" />
          </onMessage>
          <onMessage partnerLink="client" portType="Producer_PT" operation="nack">
            <empty semeffect="not shippingAgreement" />
          </onMessage>
        </pick>
      </otherwise>
    </switch>
  </sequence>
</process>
```

Figure 2. The annotated BPEL4WS process of the Shipper.

It is necessary first of all to express "semantic" relations among the input and output data values exchanged during the interaction with the web service. In our example, it is necessary to define a correspondence between the size and destination location requested by the client and the cost and delay returned by the shipper. This is achieved through the two `semvalue` annotations which appear in the opaque assignment: while we do not specify explicitly how cost and delay are computed, we annotate the fact that these correspond to `costOf` and `delayOf` applied to the requested size and location. We will see later on how `costOf` and `delayOf` will be used to express the composition goal.

A second usage of semantic annotations is to define the outcome of an interaction with a web service. In our example it is clear that a shipping agreement has been established only if the shipping is available, the shipper sends an offer, and the client acknowledge the acceptance of the offer. However, to express this in the BPEL4WS specification, we need to differentiate the possible terminal states of the interactions with the shipper. In Figure 2 this is obtained through the `semeffect` annotations associated to the "dummy" `empty` activities which mark the three possible terminal states: the semantic attribute `shippingAgreement` is true if and only if the interaction has been successful.

We remark 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. Still, they are sufficient for the automated composition task we are interested in.

notations. However, the focus here is not on the specific notations that we have adopted but on the information carries by these annotations. Other notations, e.g., inspired to standard languages such as OWL-S, WSMO, or WSDL-S, could be used instead.

4. BPEL4WS processes as STSs

We encode BPEL4WS processes (extended with semantic annotations) as *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*. Following the standard approach in process algebras, 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 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.

Definition 3 (State transition system) *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.*

We have formally defined a translation that associates a state transition system to each component service, starting from its abstract BPEL4WS specification. Currently, the translation is restricted to a subset of BPEL4WS processes: we support all BPEL4WS *basic* and *structured activities*, like assign, invoke, receive, sequence, switch, while, flow (without links), pick, and a restricted form of correlation. The translator also supports a subset of XPATH expressions for assignments and conditions. We omit the formal definition of the translation, which can be found at `http://astroproject.org`, and we illustrate it by means of the P&S example.

Example 4 *Figure 3 shows a textual description of the STS corresponding to the annotated* BPEL4WS *code if Figure 2. The set of states $\mathcal{S}$ models the steps of the evolution of the process, the values of its variables (taking into account that these variables can consist of different parts, e.g.,* `req_size` *and* `req_location` *for* BPEL4WS *variable* `req`*), and the value of the semantic attributes (*`shippingAgreement` *in our case). The special variable* `pc` *implements a "program counter" that holds the current execution step of the service (e.g.,* `pc` *has value* `checkAvailability` *when it is ready to check whether the shipping is possible). In the initial states $\mathcal{S}^0$ all the variables are undefined, apart from* `pc` *that is set to* `START`*.*

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 = checkAvailability -[TAU]-> pc = isNotAvailable`*" states that an action τ can be executed in state* `checkAvailability` *and leads to the state* `isNotAvailable`*. We remark that each* `TRANS` *clause of Figure 2 corresponds to different elements in the transition relation $\mathcal{R}$: e.g., "*`pc = checkAvailability -[TAU]-> pc = isNotAvailable`*" generates different elements of $\mathcal{R}$, depending on the values of variables* `req_size` *and* `req_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 (e.g., `request` *is used for the receiving of the shipping request) The output actions $\mathcal{O}$ represent the outgoing messages (e.g.,* `offer` *is used to bid the transportation of an item at a particular price). The action τ is used to model internal evolutions of the process, such as assignments and decision making.*

The definition of state transition system provided in Figure 3 is parametric w.r.t. the types `Size`*,* `Location`*,* `Cost`*, and* `Delay` *used in the messages and the functions* `costOf` *and* `delayOf`*. In order to obtain a concrete state transition system and to apply the automated synthesis techniques described later in this paper, finite ranges have to be assigned to these types and definitions have to be provided for the functions. Two approaches are possible to define these ranges. The first approach consists in assigning very small ranges to the types, and arbitrary functions compatible with these ranges are chosen for* `costOf` *and* `delayOf` *— this approach has been adopted in [15, 16]. The second, more general approach consists in giving a knowledge-level representation of types and functions: instead of describing the values of variables and functions the STS describes whether these values are* known *of* not known *to the client. As shown in [14] this information is sufficient for achieving an automated composition of web services.*[3]

5. Composition requirements

Our aim is to generate a specific composite service W that satisfies some desired properties. The compo-

3 For lack of space, we cannot discuss more on this issue. The interested reader can find more details in [14].

```
PROCESS Shipper;
TYPE
    Size; Location; Cost; Delay;
FUNCTION
    costOf(Size,Location): Cost;
    delayOf(Size,Location): Delay;
STATE
    pc: { START, receive_request, checkAvailability, isNotAvailable,
          isAvailable, invoke_not_available, empty_1, prepareOffer,
          invoke_offer, waitAcknowledge, empty_2, empty_3, END };
    req_size: Size ∪ { UNDEF };
    req_location: Location ∪ { UNDEF };
    offer_delay: Delay ∪ { UNDEF };
    offer_cost: Cost ∪ { UNDEF };
    shippingAgreement: { TRUE, FALSE, UNDEF };
INIT
    pc = START;
    offer_delay = UNDEF;
    offer_cost = UNDEF;
    customer_req_size = UNDEF;
    customer_req_location = UNDEF;
    shippingAgreement = UNDEF;
INPUT
    request(Size, Location);
    ack();
    nack();
OUTPUT
    offer(Delay, Cost);
    not_avail();
TRANS
    pc = START -[TAU]-> pc = receive_request;
    pc = receive_request -[INPUT request(req_size,req_location)]-> pc = checkAvailablility;
    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 = empty_1;
    pc = empty_1 -[TAU]-> pc = END, shippingAgreement = FALSE;
    pc = isAvailable -[TAU]-> pc = prepareOffer;
    pc = prepareOffer -[TAU]-> pc = invoke_offer,
                               offer_cost IN Cost,
                               offer_delay IN Delay;
    pc = invoke_offer -[OUTPUT offer(offer_cost, offer_delay)]-> pc = waitAcknowledge;
    pc = waitAcknowledge -[INPUT ack]-> pc = empty_2;
    pc = waitAcknowledge -[INPUT nack]-> pc = empty_3;
    pc = empty_2 -[TAU]-> pc = END, shippingAgreement = TRUE;
    pc = empty_3 -[TAU]-> pc = END, shippingAgreement = FALSE;
```

Figure 3. The STS corresponding to the Shipper process.

sition requirement formalizes these desired properties of the composite service to be automatically synthesized. Consider the following example.

Example 5 *We want the composite P&S service to "sell items at home". This means we want the P&S service to reach the situation where the user has confirmed his order, and the service has confirmed the corresponding (sub)orders to the producer and shipper services. However, the product may not be available, the shipping may not be possible, the user may not accept the total cost or the total time needed for the production and delivery of the item... We cannot avoid these situations, and we therefore cannot ask the composite service to guarantee this requirement. Nevertheless, we would like the P&S service to try (do whatever is possible) to satisfy it. Moreover, in the case the "sell items at home" requirement is not satisfied, we would like that the P&S service does not commit to an order for production or for delivery, since we do not want the service to buy an item that will be never delivered, as well we do not want to spend money for a delivering service when there is no item to deliver. Let us call this requirement "never a single commit". Our global requirement would therefore be something like:*

try to "sell items at home";
upon failure,
do "never a single commit".

Notice that the secondary requirement ("never a single

commit") has a different strength w.r.t. the primary one ("sell items at home"). 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 as those of the previous example, including conditions of different strengths (like "try" and "do"), and preferences among different (e.g., primary and secondary) requirements. For this reason, we cannot use well known temporal logics like LTL or CTL [8], that have been used in other frameworks to formalize requirements for automated synthesis, but that are unable to describe such requirements. In [16, 15] we have adopted EAGLE, a requirement language designed with this specific purpose; EAGLE operators are similar to CTL operators, but their semantics, formally defined in [7], take into account the notion of preference and the handling of failure when subgoals cannot be achieved.

Let propositional formulas $p \in \mathcal{P}rop$ define conditions on the states of a given state transition system. In EAGLE, a *composition requirement* ρ over $\mathcal{P}rop$ is defined as follows:

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

DoReach p specifies that condition p has to be eventually reached in a mandatory way, for all possible evolutions of the state transition system. Similarly, **DoMaint** q specifies that property q must mandatorily be maintained true. **TryReach** p and **TryMaint** q are weaker versions of the previous requirements, where the composite service is required to do "everything that is possible" to achieve condition p or maintain condition q, but failure is accepted if unavoidable. Construct ρ_1 **Fail** ρ_2 is used to model preferences among requirements and recovery from failure. More precisely, requirement ρ_1 is considered first. Only if the achievement or maintenance of this requirement fails, then ρ_2 is used as a recovery or second-choice requirement. Consider for instance the requirement "**TryReach** c **Fail DoReach** d". **TryReach** c requires a service that tries to reach condition c. During the execution of the service, a state may be reached from which it is not possible to reach c. When such a state is reached, the requirement **TryReach** c fails and the recovery condition **DoReach** d is considered. Constructs **Then**, **Repeat** and **And** are used to model sequencing, infinite iteration, and conjunction of goals respectively.

Example 6 *The* EAGLE *formalization of the requirement in Example 5 is the following.*

```
TryReach
     user.p&sAgreement
   ∧ producer.productionAgreement
   ∧ shipper.shippingAgreement
   ∧ user.offer_delay =
       producer.delayOf(user.req_item) +
        shipper.delayOf(producer.sizeOf(user.req_item),
                        user.req_location)
   ∧ user.offer_delay =
       producer.costOf(user.req_item) +
        shipper.costOf(producer.sizeOf(user.req_item),
                        user.req_location)
Fail DoReach
     ¬ user.p&sAgreement
   ∧ ¬ producer.productionAgreement
   ∧ ¬ shipper.shippingAgreement
```

It specifies that we should try to reach a situation where production and shipping agreements have been established with Producer and Shipper, and a delivery at home agreement has been established with the User. In this case, we have also to guarantee that the User gets an offer corresponding to the required item and destination location: the cost charged to the user should correspond to the cost of producing the given item plus the cost of shipping an object of the size of the requested item — and similarly for the delay. If the establishment of any of these agreements is not possible, then we require to reach a recovery situation in which none of these agreements has to be established.

In the composition requirements, the semantic annotations introduced in the BPEL4WS abstract processes play a fundamental role for defining conditions on the outcomes of web service executions. In Example 6, the semantic attributes `p&sAgreement`, `productionAgreement` and `shippingAgreement`, of User, Producer and Shipper respectively, are used to guarantee that either all the three services in establishing agreements, or none is successful. Moreover, functions like `costOf` and `delayOf`, which are used in the semantic annotations of the Shipper, are exploited to guarantee the consistency between the data exchanged among the web services.

6. Automated synthesis

The synthesizer in Figure 1 has two inputs: the formal composition requirement ρ and the parallel state transition system $\Sigma_{\|}$, which represents the services $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. We now formally define the *parallel product* of two state transition systems, which models the fact that both systems may evolve independently,

and which is used to generate $\Sigma_{\|}$ from the component web services.

Definition 7 (Parallel product) *Let* $\Sigma_1 = \langle \mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{R}_1 \rangle$ *and* $\Sigma_2 = \langle \mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{R}_2 \rangle$ *be two state transition systems 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:*

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

where:

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

The system representing (the parallel evolutions of) the component services $W_1, \ldots, W_n$ of Figure 1 is formally defined as $\Sigma_{\|} = \Sigma_{W_1} \parallel \ldots \parallel \Sigma_{W_n}$. We remark that this definition only applies to the specific case where the inputs/outputs of Σ_1 and those of Σ_2 are disjoint. This is a reasonable assumption in the case of web service composition, where the different components are independent (e.g., in the P&S domain, there is no direct communication between user, producer, and shipper). It is however possible to extend the approach described in the following to cover the cases where Σ_1 and Σ_2 can send messages between each other by modifying in a suitable way the definition of parallel product.

The automated synthesis problem consists in generating a state transition system Σ_c that, once connected to $\Sigma_{\|}$, satisfies ρ. We now define formally the state transition system describing the behaviors of Σ when connected to Σ_c.

Definition 8 (Controlled system) *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* $\langle s, a, s' \rangle \in \mathcal{R}$.

We remark that interactions among web services are asynchronous; however, in the definition of controlled system, we assumed synchronous interactions, motivated by performance considerations. To guarantee that Σ_c works also in an asynchronous setting, we require that, when a message is sent to a process, either it can be received immediately, or the process will be able to consume it after a sequence of internal action executions. This way, we assume that processes rely on a machinery that prevents messages from being lost, but we are independent from any specific implementation of message buffers. In particular, according to [15], a state s is able to accept a message a if there exists some successor s' of s, reachable from s through a (possibly empty) sequence of τ transitions, such that an input transition labeled with a can be performed in s'. This intuition is captured in the following definition, where we denote by τ-closure(s) the set of states reachable from s through a chain of τ transitions.

Definition 9 (deadlock-free controller) *Let* $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$ *be a STS and* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c \rangle$ *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* $a \in \mathcal{O}$ *then there is some* $s_c' \in \tau$*-closure*(s_c) *s.t.* $\langle s_c', a, s_c'' \rangle \in \mathcal{R}_c$ *for some* $s_c'' \in \mathcal{S}_c$;
- *if* $\langle s_c, a, s_c' \rangle \in \mathcal{R}_c$ *with* $a \in \mathcal{I}$ *then there is some* $s' \in \tau$*-closure*(s) *s.t.* $\langle s', a, s'' \rangle \in \mathcal{R}$ *for some* $s'' \in \mathcal{S}$.

In a web service composition problem, we need to generate a Σ_c that guarantees the satisfaction of a composition requirement ρ (see Figure 1). This is formalized by requiring that the controlled system $\Sigma_c \triangleright \Sigma_{\|}$ must satisfy the EAGLE formula ρ, written $\Sigma_c \triangleright \Sigma_{\|} \models \rho$. The definition of whether ρ is satisfied is given in terms of the executions that $\Sigma_c \triangleright \Sigma_{\|}$ can perform. So, for instance, if $\rho =$ **DoReach** p with p a state condition, then we need to check that all executions of $\Sigma_c \triangleright \Sigma_{\|}$ eventually reach a "configuration" that satisfies condition p. We omit the formal definition of $\Sigma_c \triangleright \Sigma_{\|}$, which can be found in [15]. Given this, we can characterize formally a (web service) composition problem.

Definition 10 (Composition) *Let* $\Sigma_1, \ldots, \Sigma_n$ *be a set of state transition systems, and let* ρ *be an* EAGLE *formula defining a composition requirement. The composition problem for* $\Sigma_1, \ldots, \Sigma_n$ *and* ρ *is the problem of finding a state transition system* Σ_c *such that*

$$\Sigma_c \triangleright (\Sigma_1 \parallel \ldots \parallel \Sigma_n) \models \rho.$$

To synthesize Σ_c, we need to take into account that $\Sigma_{\|}$ is only partially observable by Σ_c, that is, due to the τ-transitions and the nondeterministic behaviors, at execution time the composite service Σ_c cannot in general get to know exactly what is the current state of the component services modeled by $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. In

[15] we have show how to adapt to this task the "Planning as Model Checking" approach, which is a well known plan synthesis technique able to deal with nondeterminism, partial observability, and with requirements expressed in EaGLe [5, 4, 7].

Once the state transition system Σ_c has been generated, it is translated into BPEL4WS by component STS2BPEL of Figure 1. The translation is conceptually simple, but particular care has been put in the implementation of this module in order to guarantee that the generated BPEL4WS is of good quality, e.g. it is emitted as a structured program rather than being based on jumps and labels.

7. Related work and conclusions

In this paper we have shown how executable BPEL4WS processes that compose web services can be automatically generated from descriptions of component services consisting of abstract BPEL4WS specifications enriched with semantic annotations.

In other papers [16, 15, 14] we have reported our experiments with some case studies taken from real world applications and with a set of parameterized domains. These experiments show that the technique can scale up to significant cases, where the manual development of BPEL4WS composite services is not trivial and is time consuming. They show also that it is possible to generate BPEL4WS programs that are easy to read and analyze by a programmer, and the size of the program is reasonable compared with the one that can be programmed by hand.

The semantic web community has used automated planning techniques to address the problem of the automated discovery and composition of semantic web services, e.g., based on OWL-S descriptions of input/outputs and of preconditions/postconditions (see, e.g, [11]). While we do not address the problem of discovery (we assume the n component services are given), we tackle a form of automated composition that is more complex than the one considered in semantic web services. Indeed, those approaches do not take into account behavioral descriptions of web service, like our approach does with BPEL4WS. In [12], the authors propose an approach to the simulation, verification, and automated composition of web services based a translation of DAML-S to situation calculus and Petri Nets. However, the automated composition is limited to sequential composition of atomic services, and composition requirements are limited to reachability conditions. Other automated planning techniques have been proposed to tackle the problem of service composition, see, e.g., [22, 18]. However, none of these can deal with the automated synthesis problem that we address in this paper, where the planning domain and the generated plans are state transition systems, and goals capture composition requirements that are not limited to reachability conditions.

Several works have been proposed to support forms of compositions starting from WSDL-like specifications of web services, see, e.g, [17, 19]. These techniques do not take into account behavioral descriptions of web services like abstract BPEL4WS.

In [9], a formal framework is defined for composing e-services from behavioral descriptions given in terms of automata. This approach considers a problem that is fundamentally different from ours, since the e-composition problem is seen as the problem of coordinating the executions of a given set of available services, and not as the problem of generating a new composite web service that interacts with the available ones. Solutions to the former problem can be used to deduce restrictions on an existing (composition automaton representing the) composed service. We generate (the automaton corresponding to) the BPEL4WS composed service, thus addressing directly the problem of reducing time, effort, and errors in the development of composite web services. This is the main conceptual difference also with the work described in [3], where automated reasoning techniques, based on Description Logic, are used to address the problem of automated composition of e-services described as finite state machines.

In this paper, we have proposed a "minimal" set of semantic annotations: we enrich the BPEL4WS specification of the component web services only with the information which is strictly necessary for achieving the composition task. We plan to extend the work to the automated synthesis of richer "semantic" languages such as OWL-S [6] or WSMO [21], along the lines of [20].

Acknowledgments. 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". The authors want to thank all members of the Astro project (`http://astroproject.org/`) for their collaboration and their feedback.

References

[1] ActiveBPEL. The Open Source BPEL Engine - http://www.activebpel.org.

[2] 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.

[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] P. Bertoli, A. Cimatti, M. Pistore, and P. Traverso. A Framework for Planning with Extended Goals under Partial Observability. In *Proc. ICAPS'03*, 2003.

[5] 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.

[6] The OWL Services Coalition. OWL-S: Semantic Markup for Web Services, 2003.

[7] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a Language for Extended Goals. In *Proc. AAAI'02*, 2002.

[8] E. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*. Elsevier, 1990.

[9] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: A Look Behind the Curtain. In *Proc. PODS'03*, 2003.

[10] R. Khalaf, N. Mukhi, and S. Weerawarana. Service Oriented Composition in BPEL4WS. In *Proc. WWW'03*, 2003.

[11] S. McIlraith and S. Son. Adapting Golog for Composition of Semantic Web Services. In *Proc. KR'02*, 2002.

[12] S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW'02*, 2002.

[13] Oracle. Oracle BPEL Process Manager - http://www.oracle.com/products/ias/bpel/.

[14] 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.

[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] S. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. In *Proc. WWW'02*, 2002.

[18] M. Sheshagiri, M. desJardins, and T. Finin. A Planner for Composing Services Described in DAML-S. In *Proc. AAMAS'03*, 2003.

[19] D. Skogan, R. Gronmo, and I. Solheim. Web Service Composition in UML. In *Proc. EDOC'04*, 2004.

[20] P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *Proc. ISWC'04*, 2004.

[21] SDK WSMO working group. The Web Service Modeling Framework - http://www.wsmo.org/.

[22] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S Web Services Composition using SHOP2. In *Proc. ISWC'03*, 2003.