Automated Composition of Web Services by Planning in Asynchronous Domains *

M. Pistore[1] and **P. Traverso**[2] and **P. Bertoli**[2]
[1] pistore@dit.unitn.it - DIT - University of Trento - via Sommarive 14, 38050, Trento, Italy
[2] [traverso,bertoli]@irst.itc.it - ITC-IRST - via Sommarive 18, 38050, Trento, Italy

Abstract

We propose a novel planning framework for the automated composition of web services. We consider services that are specified and implemented in industrial standard languages for business processes modeling and execution, like BPEL4WS. These languages describe web services whose behavior is intrinsically asynchronous. For this reason, the key aspect of our framework is the modeling of asynchronous planning problems. In the paper we describe the framework and propose a planning approach that is based on state of the art techniques for planning under uncertainty. Our experiments show that this approach can scale up to significant cases, i.e., to cases in which the manual development of BPEL4WS composed services is not trivial and is time consuming.

Introduction

Planning is one of the most promising techniques for the automated composition of web services. Several recent works in planning have addressed different aspects of this problem, see, e.g., (Blythe, Deelman, & Gil 2003; Wu *et al.* 2003; Dermott 1998; Sheshagiri, desJardins, & Finin 2003; McIlraith & Son 2002; McIlraith & Fadel 2002). In these works, automated composition is described as a planning problem: existing services can be used to construct the planning domain, composition requirements can be formalized as planning goals, and planning algorithms can be used to generate plans that compose the published services.

A challenge for planning is the automated composition of services that are specified and implemented in industrial standard languages for business process modeling and execution, such as BPEL4WS (Andrews *et al.* 2003). These languages have been designed specifically for web service composition (Khalaf, Mukhi, & Weeravarana 2004). BPEL4WS, for instance, is used both for the publishing and for the execution of compositions. More precisely, *abstract* BPEL4WS specifications are used to publish the interaction protocol with external web services, while *concrete* BPEL4WS programs can be used to implement the internal process, i.e., the part that is not visible to external services, and that can be executed by standard engines, such as the Active BPEL Open Engine or the Oracle BPEL Process Manager. In this context, automated composition amounts to generating automatically concrete BPEL4WS programs that compose web services published with abstract BPEL4WS specifications, thus reducing significantly development efforts, time, and errors.

Unfortunately, the planning problem corresponding to the automated composition of BPEL4WS processes is far from trivial, since it poses strong requirements on the kind of planning techniques that can be used. Web services must be modeled with nondeterministic and partially observable behaviors, and composition requirements must be expressed with extended goals[1] (Koehler & Srivastava 2002; Koehler, Tirenni, & Kumaran 2002; Pistore *et al.* 2004; Hull *et al.* 2003; Berardi *et al.* 2003). A preliminary solution taking into account these aspects is presented in (Pistore *et al.* 2004).

A further crucial characteristic of web services has been again widely recognized (Fu, Bultan, & Su 2004; Bultan *et al.* 2003; Foster *et al.* 2003), but has never been addressed by planning for web service composition: web services interactions are intrinsically *asynchronous*. Indeed, each BPEL4WS process evolves independently and with unpredictable speed, synchronizing with the other processes only through asynchronous message exchanges. Message queues are used in practical implementations to guarantee that processes don't loose messages that they are not ready to receive.

In this paper, we address the problem of the automated composition of web services by means of *planning in asynchronous domains*. More precisely, given a set of BPEL4WS abstract specifications of published web services, and given a composition requirement, we generate automatically a BPEL4WS concrete process that interact asynchronously with the published services. We deploy the gen-

*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".

[1] In this paper, by extended goals we mean goals that are not limited to reachability goals, i.e., sets of states, but that can express, e.g., temporal conditions, like CTL (Emerson 1990), and/or preference conditions, like EAGLE (Dal Lago, Pistore, & Traverso 2002).

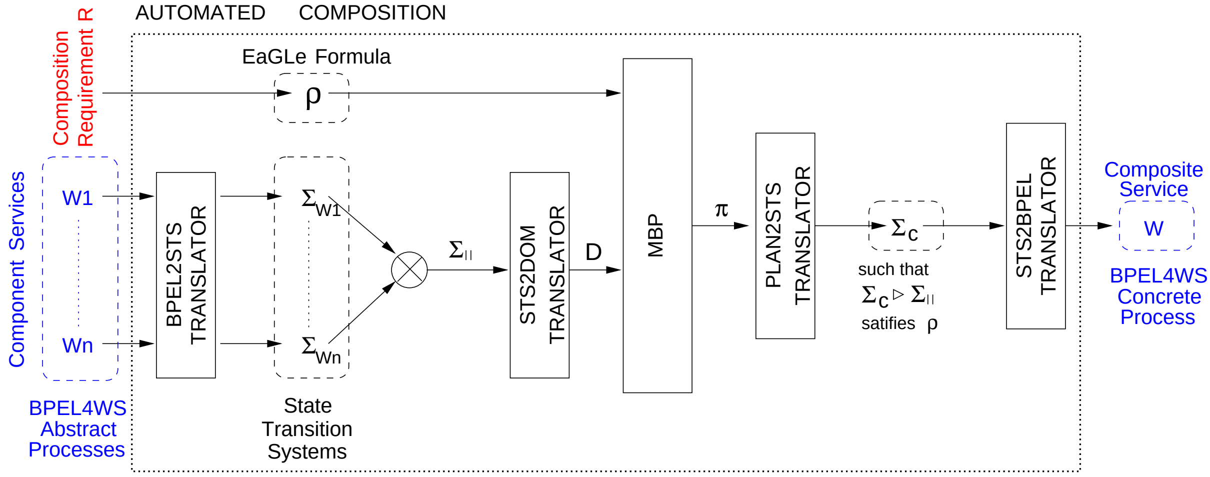


Figure 1: The approach.

erated BPEL4WS process and execute it on an available execution engine, thus integrating the planning for web service composition task in the software development cycle.

We achieve these results as follows. We model BPEL4WS abstract processes as state transition systems that can change state either by asynchronously receiving messages (called *input actions*), or by sending messages (*output actions*), or by evolving internally (by means of an *internal action*). Input and output actions define the protocol that is published, while internal actions represent the internal behavior that is not visible to external parties.

We model asynchronous interactions by abstracting away the specific mechanism of input queues. To this purpose, 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.

We then devise a formal framework for planning in asynchronous domains under this modeling assumption, and show how planning can be applied to generate executable and deployable BPEL4WS code. We finally implement the proposed framework and experiment with it, showing that the proposed approach improves dramatically over the performance of previous solutions for synchronous domains reported in (Pistore *et al.* 2004).

The paper is structured as follows. We first give an overview of the approach and introduce an explanatory example. We then describe the modeling of abstract BPEL4WS processes and of composition requirements. Next we describe the formal framework and discuss the planning problem and its solution. We finally report the results of our experimental evaluation and discuss a comparison with related work.

Overview

Our goal is to automatically generate a new service W (called the *composite service*) that interacts with a set of published web services $W_1, \ldots, W_n$ (called the component services) and satisfies 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* (STS from 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. Each Σ_{W_i} is described in terms of states, input and output actions, and internal actions.

We then construct a *parallel STS* $\Sigma_\parallel$ that combines $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. Formally, this combination is a parallel product, which allows the n services to evolve concurrently. $\Sigma_\parallel$ represents therefore all the possible behaviors, evolutions of the different component services, without any control by and interaction with the composite service that will be generated, i.e., W. From $\Sigma_\parallel$, we generate a planning domain $\mathcal{D}$ that is passed in input to the planner (module STS2DOM).

The second kind of input to the planner consists of the requirements for the composite service. They are formalized as a goal ρ in EAGLE, a language for expressing extend planning goals (Dal Lago, Pistore, & Traverso 2002). While the framework presented in this paper is general, and works with other kinds of extended goals, e.g., with CTL goals (Emerson 1990), we choose EAGLE since, as we will see, it is better suited to express composition requirements.

Given $\mathcal{D}$ and ρ, MBP generates a plan π that is then translated into a STS Σ_c. Σ_c encodes the new service W that has to be generated, which dynamically receives and sends invocations from/to the composite services $W_1, \ldots, W_n$ and behaves depending on responses received from the external services. Σ_c is such that $\Sigma_c \triangleright \Sigma_\parallel$ satisfies the requirement ρ, where $\Sigma_c \triangleright \Sigma_\parallel$ represents all the evolutions of the component services as they are controlled by the composite service.

The STS Σ_c is then given in input to the STS2BPEL module which translates it into a concrete BPEL4WS process that implements the desired composite web service.

Running Example

In the rest of the paper, we will describe our approach through the following example.

Example 1 *Our reference example consists in providing a furniture purchase & delivery service, say the P&S service. We do so 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 (see Figure 2). These are the three available services* W_1, W_2, *and* W_3, *which are described as* BPEL4WS *abstract processes and translated to STSs by the* BPEL2STS *module in Figure 1. The problem is to automatically generate the concrete* BPEL4WS *implementation of the P&S service, i.e.,* W *in Figure 1.*

In the following, we describe informally the three available services. The 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, the Shipper provides a shipping offer with a cost and delivery time, which can be accepted or refused by the external service that has invoked the 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 the user, the combined purchase & delivery service P&S, the producer, and the shipper would go as follows:

1. *the user asks P&S for an item* i, *that he wants to be delivered 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 provides the user an offer which takes into account the overall cost (plus an added cost for P&S) and time to produce and deliver the item;*
5. *the user sends a confirmation of the order, which is dispatched by P&S to the delivery and producer.*

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

At a high level, Figure 2 describes the data flow amongst our integrated web service, the two services composing it, and the user. □

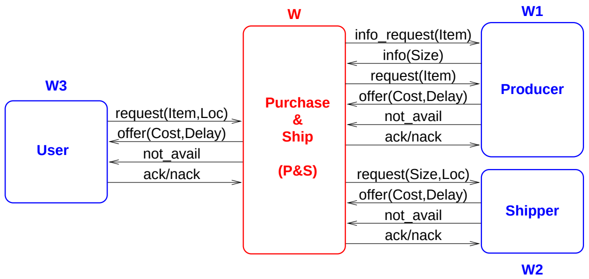


Figure 2: The Purchase & Ship Example.

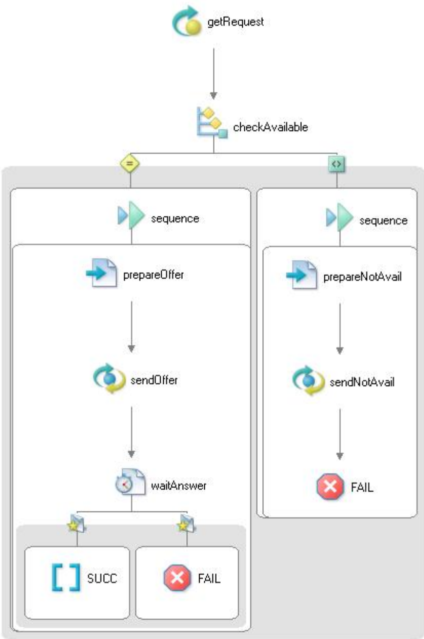
Abstract processes as state transition systems

BPEL4WS (Andrews *et al.* 2003) 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.

We encode BPEL4WS processes 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*. 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 requirements.

Definition 2 (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 finite set of states;*




```
PROCESS
    Shipper;
TYPE
    Size; Location; Cost; Delay;
STATE
    pc: { START, getRequest, checkAvailable, _end_checkAvailable,
          _sequence_1, _sequence_2, prepareOffer, sendOffer, waitAnswer,
          _end_waitAnswer, _empty_1, prepareNotAvail, sendNotAvail,
          SUCC, FAIL };
    customer_req_size: Size ∪ { UNDEF };
    customer_req_loc: Location ∪ { UNDEF };
    offer_delay: Delay ∪ { UNDEF };
    offer_cost: Cost ∪ { UNDEF };
INIT
    pc = START;
    offer_delay = UNDEF;
    offer_cost = UNDEF;
    customer_req_size = UNDEF;
    customer_req_loc = UNDEF;
INPUT
    request(s: Size, l: Location);
    ack;
    nack;
OUTPUT
    offer(d: Delay, c: Cost);
    not_avail;
TRANS
    pc = START -[TAU]-> pc = getRequest;
    pc = getRequest -[INPUT request(customer_req_size,
          customer_req_loc)]-> pc = checkAvailable;
  pc = checkAvailable -[TAU]-> pc = _sequence_1;
  pc = checkAvailable -[TAU]-> pc = _sequence_2;
  pc = _sequence_1 -[TAU]-> pc = prepareOffer;
  pc = prepareOffer -[TAU]-> pc = sendOffer,
                              offer_cost IN Cost,
                              offer_delay IN Delay;
  pc = sendOffer -[OUTPUT offer(offer_cost, offer_delay)]->
                                          pc = waitAnswer;
  pc = waitAnswer -[INPUT nack]-> pc = FAIL;
  pc = waitAnswer -[INPUT ack]-> pc = _empty_1;
  pc = _empty_1 -[TAU]-> pc = _end_waitAnswer;
  pc = _end_waitAnswer -[TAU]-> pc = _end_checkAvailable;
  pc = _end_checkAvailable -[TAU]-> pc = SUCC;
  pc = _sequence_2 -[TAU]-> pc = prepareNotAvail;
  pc = prepareNotAvail -[TAU]-> pc = sendNotAvail;
  pc = sendNotAvail -[OUTPUT not_avail]-> pc = FAIL;
```

Figure 3: The STS for the Shipper process.

- *$\mathcal{S}^0 \subseteq \mathcal{S}$ 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;*
- *$\mathcal{L} : \mathcal{S} \rightarrow 2^{\mathcal{P}rop}$ is the labeling function.*

We assume that infinite loops of τ-transitions cannot appear in the system. Indeed, an infinite τ-loop would describe a divergent behavior of the system, i.e., a behavior where the service is not interacting with the environment. We also assume that there is no state which originates both input and output transitions.

We have formally defined a translation that associates a STS to each component service, starting from its abstract BPEL4WS specification. This translation is performed automatically by the BPEL2STS module in Figure 1. For the moment, the translation is restricted to a subset of BPEL4WS processes: we support all BPEL4WS *basic* and *structured activities*, like invoke, receive, sequence, switch, while, flow (without links) and pick; moreover we support assignments and correlation. Our next steps will be dealing with scopes and with fault, event and compensation handlers. We omit the formal definition of the translation, since it is outside the scope of the paper.

Example 3 *Figure 3 shows the graphical representation (using Active BPEL) of the abstract* BPEL4WS *process of the* Shipper*, and its corresponding STS. 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* getRequest *when the process is waiting to receive a shipping request, and value* checkAvailable *when it is ready to check whether the shipping is possible). The other variables (e.g.,* offer_delay*,* offer_cost*) correspond to those used by the process to store significant information. In the initial states $\mathcal{S}^0$ all the variables are undefined but* 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 = checkAvailable -[TAU]-> pc = _sequence_1*" states that an action τ can be executed in state* checkAvailable *and leads to the state* _sequence_1*. We remark that each* TRANS *clause of Figure 3 corresponds to different elements in the transition relation $\mathcal{R}$: e.g., "*pc = checkAvailable -[TAU]-> pc = _sequence_1*" generates different* *elements of $\mathcal{R}$, depending on the values of variables* `customer_req_size` *and* `customer_req_loc`.

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., `request` *is used for the receiving of the shipping request, while* `ack` *models the confirmation of the order and* `nack` *its cancellation). The output actions $\mathcal{O}$ represent the outgoing messages (i.e.,* `not_avail` *is used when the shipping is not supported by the process, while* `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, as for instance assignments and decision making (e.g., when the Shipper process is in the state* `checkAvailable` *and performs internal activities to decide whether the shipping is possible, or when, in the state* `prepareOffer`*, it must obtain the shipping price and delay).*

Finally, the properties of the STS are expressions of the form `<variable> = <value>`*, and the labeling function is the obvious one.*

The definition of STS provided in Figure 3 is parametric w.r.t. the types `Size`*,* `Location`*,* `Cost`*, and* `Delay` *used in the messages. In order to obtain a concrete STS and to apply the automated synthesis techniques described later in this paper, finite ranges have to be assigned to these types. We are currently adopting the approach of assigning very small ranges to the types, namely ranges with only two elements. We will discuss the impacts of this choice in the discussion of the experimental evaluation.* □

Composition requirements as extended goals

Consider the following example.

Example 4 *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 (Emerson 1990), that have been used in other frameworks to formalize requirements for automated synthesis, but that are unable to describe such requirements. We use instead the EAGLE language, which has been designed with the purpose to satisfy such expressiveness[2]. A detailed definition and a formal semantics for the EAGLE language can be found in (Dal Lago, Pistore, & Traverso 2002). Here we just explain how EAGLE can express the composition requirement of the running example.

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

TryReach
```
user.pc=SUCC ∧ producer.pc=SUCC∧
shipper.pc=SUCC ∧
user.offer_delay =
  add_delay(producer.offer_delay,
          shipper.offer_delay)∧
user.offer_cost =
  add_cost(producer.offer_cost,
    shipper.offer_cost)
```
Fail DoReach
```
user.pc=FAIL ∧ producer.pc=FAIL∧
shipper.pc=FAIL
```

*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 items at home". During the execution of the service, a state may be reached from which it is not possible to reach c, e.g., since the product is not available. When such a state is reached, the requirement **TryReach** c fails and the recovery condition **DoReach** d, in our case "never a single commit" is considered.* □

Composition Problem

The automated composition problem has two inputs (see Figure 1): the formal composition requirement ρ and the parallel STS $\Sigma_\|$, which represents the services $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. We now formally define the *parallel product* of two STSs, which models the fact that both systems may evolve independently, and which is used to generate $\Sigma_\|$ from the component web services.

[2]However, the results presented in this paper are valid also in the case a temporal logic like CTL or LTL is chosen to formalize requirements.

Definition 6 (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 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 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 to the more general case where Σ_1 and Σ_2 can send messages to each other (i.e., $(\mathcal{I}_1 \cup \mathcal{O}_1) \cap (\mathcal{I}_2 \cup \mathcal{O}_2) \neq \emptyset$) by modifying in a suitable way the definition of parallel product.

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

Definition 7 (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 the plan 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 according to our asynchronous model, which abstracts away queues. 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.[3]

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

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 ρ, which is defined 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, while if $\rho =$ **DoMaint** q then we need to check that condition q is satisfied in all "configurations" reached by all executions of $\Sigma_c \triangleright \Sigma_\|$.

In order to define formally when $\Sigma_c \triangleright \Sigma_\|$ satisfies ρ, we need to define first the executions of $\Sigma_c \triangleright \Sigma_\|$. In doing this, we need to take into account that the state transition system $\Sigma_\|$ models a domain that is only partially observable by Σ_c. That is, 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 $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. Consider for instance the STS corresponding to the Shipper (see Figure 3). The composite P&S service has no access to the values of the shipper's internal variables, and can only deduce their values from the messages exchanged with the Shipper. This uncertainty has two different sources. The first one is the standard source of uncertainty in planning, namely the presence of non-deterministic transitions (e.g., the two τ transitions of the Shipper from "`pc = checkAvailable`", which model the fact the shipping may be possible or not). The second source of uncertainty is due to the fact that we are modeling an asynchronous framework and that, therefore, it is not possible for the P&S to know when internal τ transitions are performed in the Shipper. Due to this uncertainty, after a message "`request(s,l)`"

[3] We remark that, if there is such a successor s' of s, a deadlock can still occur. This can happen if a different chain of τ transitions is executed from s that leads to a state s'' for which a cannot be executed anymore. In this case, the deadlock is recognized in s''.

has been sent to the Shipper, it is impossible to distinguish whether the shipper is still checking whether the delivery is possible (pc = checkAvailable), or whether this task has terminated positively (pc is _sequence_1, prepareOffer, or sendOffer) or negatively (pc is _sequence_2, prepareNotAvail, or sendNotAvail). This uncertainty disappears only when an "offer" or a "not_avail" message is received by the P&S.

In the definition of the executions of $\Sigma_c \triangleright \Sigma_\|$ (and, more in general, of a state transition system Σ) we take into account this uncertainty by considering, at each step of the execution, a set of possible states, each equally plausible given the partial knowledge that we have of the system. Such a set of states is called a *belief state*, or simply *belief*. The initial belief for the execution is the set of initial states $\mathcal{S}^0$ of Σ. This belief is updated whenever Σ performs an observable (input or output) transition. More precisely, if $B \subseteq \mathcal{S}$ is the current belief and an action $a \in \mathcal{I} \cup \mathcal{O}$ is observed, then the new belief $B' = \mathit{Evolve}(B, a)$ is defined as follows: $s \in \mathit{Evolve}(B, a)$ if, and only if, there is some state s' reachable from B by performing a (possibly empty) sequence of τ transitions, such that $\langle s', a, s\rangle \in \mathcal{R}$. That is, in defining $\mathit{Evolve}(B, a)$ we first consider every evolution of states in B by internal transitions τ, and then, from every state reachable in this way, their evolution caused by a.

Definition 9 (belief evolution)
Let $B \subseteq \mathcal{S}$ be a belief on some state transition system Σ. We define the evolution of B under action a as the belief $B' = \mathit{Evolve}(B, a)$, where

$$\mathit{Evolve}(B, a) = \{s' : \exists s \in \tau\text{-}\mathit{closure}(B). \langle s, a, s'\rangle \in \mathcal{R}\}.$$

In the definition of goal satisfaction, we use beliefs to describe the different "configurations" reached during execution. In order to characterize goal satisfaction, we need to define when a belief B satisfies a given state property p. In planning under partial observability, B is said to satisfy p simply if all states $s \in B$ satisfy p. The definition becomes more complex in our asynchronous setting, due to the presence of τ transitions. Let us consider again the Shipper. After a request message has been sent to the shipper, it is not yet possible to predict whether the shipping can be fulfilled or not; therefore, we expect that conditions pc = prepareOffer and pc = notAvailable are both false in B. On the other hand, after an acknowledge message has been received by the shipper, it is unavoidable to reach a successful state; therefore, we want to be able to conclude that condition pc = SUCC is true in the corresponding belief. Informally, we are assuming that the execution of τ transitions cannot be postponed forever. Therefore, if the execution of τ transitions is guaranteed to reach a state satisfying a given condition, then we can assume that the condition holds also in belief state B. Conversely, if there is some sequence of τ transitions that does not contain states satisfying p and that cannot be further extended with other τ transitions so that p is reached, then the property p is not satisfied in B.

Definition 10 (belief satisfying a property)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L}\rangle$ be a STS, $p \in \mathcal{P}rop$ be a property for Σ, and $B \subseteq \mathcal{S}$ be a belief. We say that B satisfies p, written $B \models_\Sigma p$, if the following condition holds. Let $s_0, s_1, \ldots, s_n$ be such that $s_0 \in B$, $\langle s_i, \tau, s_{i+1}\rangle \in \mathcal{R}$ and either s_n has no outgoing transitions or there exists s_{n+1} such that $\langle s_n, a, s_{n+1}\rangle$ with $a \neq \tau$. Then $p \in \mathcal{L}(s_i)$ for some $0 \leq i \leq n$.

We are now ready to define the STS that defines the executions of $\Sigma_c \triangleright \Sigma_\|$ and, more in general, of a STS Σ. We call it "belief-level" STS, since its states are beliefs of Σ and its transitions describe belief evolutions.

Definition 11 (belief-level system)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L}\rangle$ be a STS. The corresponding belief-level STS is $\Sigma_\mathcal{B} = \langle \mathcal{S}_\mathcal{B}, \mathcal{S}^0_\mathcal{B}, \mathcal{I}, \mathcal{O}, \mathcal{R}_\mathcal{B}, \mathcal{L}_\mathcal{B}\rangle$, where:

- *$\mathcal{S}_\mathcal{B}$ are the beliefs of Σ reachable from the initial belief $\mathcal{S}^0$;*
- *$\mathcal{S}^0_\mathcal{B} = \{\mathcal{S}^0\}$;*
- *transitions $\mathcal{R}_\mathcal{B}$ are defined as follows: if $\mathit{Evolve}(B, a) = B' \neq \emptyset$ for some $a \in \mathcal{I} \cup \mathcal{O}$, then $\langle B, a, B'\rangle \in \mathcal{R}_\mathcal{B}$;*
- *$\mathcal{L}_\mathcal{B}(B) = \{p \in \mathcal{P}rop : B \models_\Sigma p\}$.*

We remark that a belief-level STS is a very restricted case of STS, since it only has one initial state, there are no τ transitions, and, for all beliefs B and actions a there is at most one belief B' such that $\langle B, a, B'\rangle \in \mathcal{R}_\mathcal{B}$. For these reasons, it is straightforward to re-interpret on $\Sigma_\mathcal{B}$ the definitions of goal satisfaction proposed in the literature for (fully observable) non-deterministic domains (e.g., strong and strong cyclic reachability goals (Cimatti *et al.* 2003), LTL and CTL goals (Pistore & Traverso 2001), and the most relevant for our purposes, EAGLE goals (Dal Lago, Pistore, & Traverso 2002)). In the following, we write $\Sigma_\mathcal{B} \models \rho$ whenever the belief-level STS $\Sigma_\mathcal{B}$ satisfies goal ρ.

We can now characterize formally a (web service) composition problem.

Definition 12 (composition problem)
Let $\Sigma_1, \ldots, \Sigma_n$ be a set of state transition systems, and let ρ be a composition requirement. 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 $\Sigma_\mathcal{B} \models \rho$ where $\Sigma_\mathcal{B}$ is the belief-level STS of $\Sigma_c \triangleright (\Sigma_1 \| \ldots \| \Sigma_n)$.

Once the STS Σ_c has been generated, it is translated into BPEL4WS. This translation is performed by component STS2BPEL of Figure 1.

Planning domain, problem and solution

We now show how we can define the composition problem in terms of a planning problem for extended goals and with nondeterminism. We choose to translate the composition problem to a planning problem under full observability in the belief space. This allows us to use efficient planning algorithms based on symbolic model checking.

The definitions of planning domains and plans are taken, with minor modifications, from (Pistore & Traverso 2001). We recall that a fully observable non-deterministic planning domain is defined as a tuple $\mathcal{D} = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{A}, \mathcal{T}, \mathcal{L}\rangle$, where $\mathcal{S}$ is the finite set of states and $\mathcal{S}^0 \subseteq \mathcal{S}$ is the subset of initial states, $\mathcal{A}$ is the finite set of actions, $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation and $\mathcal{L} : \mathcal{S} \rightarrow \mathcal{P}rop$ is the labelling function.

We now formally define the planning domain associated to a STS.

Definition 13 (planning domain)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be a STS and let $\Sigma_{\mathcal{B}} = \langle \mathcal{S}_{\mathcal{B}}, \mathcal{S}^0_{\mathcal{B}}, \mathcal{I}, \mathcal{O}, \mathcal{R}_{\mathcal{B}}, \mathcal{L}_{\mathcal{B}} \rangle$ be the corresponding belief-level state transition system according to Definition 11. The planning domain corresponding to Σ is $\mathcal{D} = \langle \mathcal{S}_{\mathcal{D}}, \mathcal{S}^0_{\mathcal{D}}, \mathcal{A}_{\mathcal{D}}, \mathcal{T}_{\mathcal{D}}, \mathcal{L}_{\mathcal{D}} \rangle$, where:

- $\mathcal{S}_{\mathcal{D}} = \mathcal{S}_{\mathcal{B}} \times (\mathcal{O} \cup \{\star\})$ *and* $\mathcal{S}^0_{\mathcal{D}} = \mathcal{S}^0_{\mathcal{B}} \times \{\star\}$;
- $\mathcal{A}_{\mathcal{D}} = \mathcal{I} \cup \{\star\}$;
- *transitions $\mathcal{T}_{\mathcal{D}}$ are defined as follows:*
  - *if $\langle B, i, B' \rangle \in \mathcal{R}_{\mathcal{B}}$ with $i \in \mathcal{I}$ then:* $\langle (B, o), i, (B', \star) \rangle \in \mathcal{T}_{\mathcal{D}}$;
  - *if $\langle B, o', B' \rangle \in \mathcal{R}_{\mathcal{B}}$ with $i \in \mathcal{I}$ then:* $\langle (B, o), \star, (B', o') \rangle \in \mathcal{T}_{\mathcal{D}}$;
- $\mathcal{L}_{\mathcal{D}}((B, o)) = \mathcal{L}_{\mathcal{B}}(B)$.

When transforming a STS into a planning domain, we first have to move to the belief level, that is, each state of the domain corresponds to a whole belief state of the starting state transition system. This is necessary to handle partial observability in the STS, and to allow for applying algorithms working in fully observable domains. A second transformation applied in the construction of the planning domain is that outputs are moved from the transitions into the states of the domain. This is obtained by defining a state of a planning domain as a pair (B, o), where o is the output received during the last transition (a special mark $\star$ is used if the last transition has been an input).

Existing planning algorithms can thus be used to obtain plans for a domain $\mathcal{D}$ and for a given goal. We recall that a plan $\pi = \langle C, c^0, \alpha, \epsilon \rangle$ is defined by a set of execution contexts C, one of which, c^0, is the initial one, by an action evolution (partial) function $\alpha : C \times \mathcal{S} \rightarrow \mathcal{A}$ and by a context evolution (partial) function $\epsilon : C \times \mathcal{S} \rightarrow C$.

The execution of a plan $\pi = \langle C, c^0, \alpha, \epsilon \rangle$ on a domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{A}, \mathcal{T}, \mathcal{L} \rangle$ is defined in term of configurations, i.e., of pairs $(c, s) \in C \times \mathcal{S}$. The initial configurations are those of the form (c^0, s^0), with $s^0 \in \mathcal{S}^0$. In configuration (c, s) the plan executes action $a = \alpha(c, s)$, if defined, and evolves into the set of configurations (c', s'), where $\langle s, a, s' \rangle \in \mathcal{T}$ and $c' = \epsilon(c, s)$. We say that a plan is executable if, for all configurations (c, s) reachable from the initial ones, if $a = \alpha(c, s)$ is defined, then also $\epsilon(c, s)$ is defined, and a is executable in s.

The *execution structure* of a plan π over a domain $\mathcal{D}$ is the automaton consisting of all reachable configurations, where each state (c, s) is labelled according to $\mathcal{L}(s)$. A plan is said to satisfy the goal if the goal is true on the execution structure.

The final step is then to transform a plan π into a STS. The following definition explains how this is done.

Definition 14 (system of a plan)
Let Σ, $\Sigma_{\mathcal{B}}$, and $\mathcal{D}$ be defined as in Definition 13, and let $\pi = \langle C, c^0, \alpha, \epsilon \rangle$ be a plan for $\mathcal{D}$. The STS corresponding to π is defined as follows: $\Sigma_\pi = \langle \mathcal{S}_\pi, \mathcal{S}^0_\pi, \mathcal{O}, \mathcal{I}, \mathcal{R}_\pi, \mathcal{L}_\emptyset \rangle$, where:

- $\mathcal{S}_\pi = C \times \mathcal{S}_{\mathcal{B}} \times (O \cup \{\star\})$
- $\mathcal{S}^0_\pi = \{c^0\} \times \mathcal{S}^0_{\mathcal{B}} \times \{\star\}$
- *$\mathcal{R}_\pi$ is defined as follows:*
  - *if $\alpha(c, (B, o)) = i \neq \star$ and $c' = \epsilon(c, (B, o))$, then $\langle (c, B, o), i, (c', B', \star) \rangle \in \mathcal{R}_\pi$, where $B' =$ Evolve(B, i);*
  - *if $\alpha(c, (B, o)) = \star$ and $c' = \epsilon(c, (B, o))$, then $\langle (c, B, o), o', (c', B', o') \rangle \in \mathcal{R}_\pi$ for all o' and $B' \neq \emptyset$ such that $B' =$ Evolve(B, o').*

According to this definition, the STS corresponding to the plan mimics very strictly the execution structure for $\mathcal{D}$ and π, except for the necessity to describe the outputs on the transitions.

We remark that, in general, the STS obtained from a plan according to Definition 14 may not be a deadlock free controller for the starting STS Σ (see Definition 8). However, there are conditions on Σ that guarantee not only that Σ_π is deadlock free, but also that Σ_π satisfies a given composition requirement ρ on Σ whenever π is a solution to planning goal ρ on planning domain $\mathcal{D}$. The conditions that guarantee these properties are that there is no belief in which both inputs and outputs are possible, and that, when it is the controller's turn, we need to be sure that the domain will be able to process all messages the controller can decide to send.

Definition 15 (controllable system)
Let Σ be a STS and let $\Sigma_{\mathcal{B}}$ be the corresponding belief-level state transition system. We say that Σ is controllable if all beliefs B in $\Sigma_{\mathcal{B}}$ satisfy the following conditions:

- *if Evolve$(B, o) \neq \emptyset$ for some $o \in \mathcal{O}$, then Evolve$(B, i) = \emptyset$ for all $i \in \mathcal{I}$;*
- *if Evolve$(B, i) \neq \emptyset$ for some $i \in \mathcal{I}$, then Evolve$(B, o) = \emptyset$ for all $o \in \mathcal{O}$;*
- *if Evolve$(B, i) \neq \emptyset$ for some $i \in \mathcal{I}$, then, for each $s \in \tau$-closure(B) there is some $s' \in \tau$-closure(s) and some state $s'' \in \mathcal{S}$ such that $\langle s', i, s'' \rangle \in \mathcal{R}$.*

We remark that this constraint removes possible solutions. That is, there are service composition problems for which there exist solutions according to Definition 12, but that do not satisfy the conditions in Definition 15. Still, these conditions are very reasonable in the field of web services. Indeed, the requirement that inputs and outputs are not intermixed in the same belief state correspond to the assumption that, in the interaction with a web service, we always know if the service is waiting for an invocation, or whether it is going to send an answer to a previous invocation. The requirement that the same set of inputs is available from all states corresponds to the assumption that, in any moment, we know what are the valid invocations that we can perform on a service. All the examples of web services from real world applications that we have been working on respect these assumptions and can hence be composed using the approach outlined above.

We are now ready to state the correspondence results between planning problems and composition problems.

Lemma 16 (controller/plan executability) *Let Σ be a deadlock-free STS and let $\Sigma_{\mathcal{B}}$, $\mathcal{D}$ be defined as in Definition 13. Let π be a plan for $\mathcal{D}$ and Σ_π be the corresponding STS. If π is executable on $\mathcal{D}$, then Σ_π is executable on Σ.*

Theorem 17 (controller/plan equivalence) *Let Σ be a deadlock-free STS and let $\Sigma_{\mathcal{B}}$, $\mathcal{D}$ be defined as in Definition 13.*

If plan π is a solution for domain $\mathcal{D}$ and goal g, and Σ_π is the STS corresponding to π, then $\Sigma_\pi \triangleright \Sigma \models g$ holds.

Moreover, if there is some Σ_c such that $\Sigma_c \triangleright \Sigma \models g$, then there exists some π that is a solution for domain $\mathcal{D}$ and goal g (and hence $\Sigma_\pi \triangleright \Sigma \models g$ holds).

Theorem 17 states that we can use planning algorithms for fully-observable nondeterministic domains and extended goals (e.g., EAGLE) to do automated composition. This allows us to exploit existing techniques based on symbolic model checking that are powerful enough to solve the automated composition problem and can do it efficiently, as shown in the next section.

Conclusions and Related Work

In this paper, we have presented a novel framework for planning for the composition of asynchronous BPEL4WS processes. In order to test the performance of the proposed technique, we have conducted several experiments, considering two different artificial scalable domains, and a set of problems extracted from realistic web service domains. The whole set of results can be found at `http://astroproject.org/`; for reasons of space, here we only discuss the realistic domains.

We first considered P&S, the example explained in the previous sections, and an extension of it that introduces a third party, the Bank, that is delegated to receive the money from the client. Then we considered a case study taken from a real e-government application we are developing for a private company. We aim at providing a service that implements a (public) waste management office (WMO), i.e. a service that manages user requests to open sites for the disposal of dangerous waste. We consider three variants of this domain, corresponding to an increasing interleaving between the different services. Automated composition is very fast, in all cases, as presented in Table 4, and dramatically improves over the solution presented in (Pistore *et al.* 2004), which only is able to tackle the simpler model, with a model construction time of more than 13.000 seconds, and a planning time of more than 3.000 seconds. This improved scalability is due to our reasonable modeling of asynchronous interactions among services, which takes into account that message queues are used in real BPEL4WS engines, but which does not commit to any specific implementation. We remark that here, just as in (Pistore *et al.* 2004), data types are defined to have binary ranges. This solution guarantees a good performance and, at least in the considered examples, it is not restrictive, in the sense that the automatically synthesized web service can be easily adapted to work with different (even unbounded) ranges for these types. We are currently investigating the formal conditions that guarantee this property.

	# of components	model construction	planning
P&S	3	8.4 sec.	1.0 sec.
P&S, BANK	4	39.6 sec.	35.4 sec.
WMO1	5	187.5 sec.	31.6 sec.
WMO2	5	173.1 sec.	48.6 sec.
WMO3	5	174.9 sec.	120.6 sec.

Figure 4: Experiments with different applications.

As far as we know, the framework for planning in asynchronous domains presented in this paper is new, and has never been proposed before. Different automated planning techniques have been proposed to tackle the problem of service composition, see, e.g., (Wu *et al.* 2003; Dermott 1998; Sheshagiri, desJardins, & Finin 2003). However, none of these can deal with the problem that we address in this paper, where the planning domain is nondeterministic, partially observable, and asynchronous, and goals are not limited to reachability conditions.

Other planning techniques have been applied to related but somehow orthogonal problems in the field of web services. The interactive composition of information gathering services has been tackled in (Thakkar, Knoblock, & Ambite 2003) by using CSP techniques. Works in the field of Data and Computational Grids are more and more moving toward the problem of composing complex workflows by means of planning and scheduling techniques (Blythe, Deelman, & Gil 2003).

Planning for the automated discovery and composition of semantic web services, e.g., based on OWL-S, is used in (McIlraith & Son 2002; McIlraith & Fadel 2002; Narayanan & McIlraith 2002). These works do not take into account behavioral descriptions of web service, like our approach does with BPEL4WS.

The work in (Hull *et al.* 2003) presents a formal framework for composing e-services from behavioral descriptions given in terms of automata. This work focuses on the theoretical foundations, without providing practical implementations. Moreover, the considered e-composition problem is fundamentally different from ours, since it is seen as the problem of coordinating the executions of a given set of available services. No concrete and executable processes can be generated with that approach. This is the main difference also with the work described in (Berardi *et al.* 2003).

More in general, our work shares some ideas with work on the automata-based synthesis of controllers (see, e.g., (Pnueli & Rosner 1989; Vardi 1995)). Indeed, the composed service can be seen as a module that controls an environment which consists of the published services. However, also in this case, the work focuses on theoretical foundations and does not provide practical implementations for the automated compositions of web services.

Some preliminary work that applies the idea of "planning via symbolic model checking" to the automated composition of web services is presented in (Pistore *et al.* 2004). We improve substantially on this work by providing an adequate modeling of asynchronous BPEL4WS processes, by generating concrete BPEL4WS processes that can be actually implement web services interactions, and by definitely improving the performances of the plan generation.

In the future, we plan to adopt techniques for planning at the knowledge level in the style of (Petrick & Bacchus 2002) to address the problem of associating finite ranges to the to data types. We also plan to investigate techniques for further improving the scalability. In particular, we are going to experiment well known techniques for avoiding the computationally complex construction of parallel state transition systems. Finally, we plan to extend the work to the automated composition of semantic web services, e.g., described in OWL-S (OWL-S 2003) or WSMO (WSMO 2004), along the lines of the work done in (Traverso & Pistore 2004).

References

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

Andrews, T.; Curbera, F.; Dolakia, H.; Goland, J.; Klein, J.; Leymann, F.; Liu, K.; Roller, D.; Smith, D.; Thatte, S.; Trickovic, I.; and Weeravarana, S. 2003. Business Process Execution Language for Web Services (version 1.1).

Berardi, D.; Calvanese, D.; Giacomo, G. D.; Lenzerini, M.; and Mecella, M. 2003. Automatic composition of E-Services that export their behaviour. In *Proc. ICSOC'03*.

Blythe, J.; Deelman, E.; and Gil, Y. 2003. Planning for workflow construction and maintenance on the grid. In *Proc. of ICAPS'03 Workshop on Planning for Web Services*.

Bultan, T.; Fu, X.; Hull, R.; and Su, J. 2003. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *Proc. WWW'03*.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence* 147(1-2):35–84.

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

Dermott, D. M. 1998. The Planning Domain Definition Language Manual. Technical Report 1165, Yale Computer Science University. CVC Report 98-003.

Emerson, E. A. 1990. Temporal and modal logic. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier.

Foster, H.; Uchitel, S.; Magee, J.; and Kramer, J. 2003. Model-based verification of Web Service Compositions. In *Proc. ASE'03*.

Fu, X.; Bultan, T.; and Su, J. 2004. Analysis of Interacting BPEL Web Services. In *Proc. WWW'04*.

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

Khalaf, R.; Mukhi, N.; and Weeravarana, S. 2004. Service Oriented Composition in BPEL4WS. In *Proc. WWW2004*.

Koehler, J., and Srivastava, B. 2002. Web service composition: Current solutions and open problems. In *Proc. of ICAPS'03 Workshop on Planning for Web Services*.

Koehler, J.; Tirenni, G.; and Kumaran, S. 2002. From business process model to consistent implementation: A case for formal verification methods. In *Proc. EDOC'02*.

McIlraith, S., and Fadel, R. 2002. Planning with Complex Actions. In *Proc. NMR'02*.

McIlraith, S., and Son, S. 2002. Adapting Golog for composition of semantic web Services. In *Proc. KR'02*.

Narayanan, S., and McIlraith, S. 2002. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW2002*.

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

OWL-S. 2003. OWL-S: Semantic Markup for Web Services. Technical White paper (OWL-S version 1.0).

Petrick, R., and Bacchus, F. 2002. A Knowledge-Based Approach to Planning with Incomplete Information and Sensing. In *Proc. AIPS'02*.

Pistore, M., and Traverso, P. 2001. Planning as model checking for extended goals. In *Proc. IJCAI'01*.

Pistore, M.; Bertoli, P.; Barbon, F.; Shaparau, D.; and Traverso, P. 2004. Planning and Monitoring Web Service Composition. In *Proc. AIMSA'04*.

Pnueli, A., and Rosner, R. 1989. On the synthesis of an asynchronous reactive module. In *Proc. ICALP'89*.

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

Thakkar, S.; Knoblock, C.; and Ambite, J. 2003. A view integration approach to dynamic composition of web services. In *Proc. of ICAPS'03 Workshop on Planning for Web Services*.

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

Vardi, M. Y. 1995. An automata-theoretic approach to fair realizability and synthesis. In *Proc. CAV'95*.

WSMO. 2004. The web service modeling framework. SDK WSMO working group - http://www.wsmo.org/.

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