Automated Synthesis of Composite BPEL4WS Web Services

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

In this paper we propose a technique for the automated synthesis of new composite web services. Given a set of abstract BPEL4WS *descriptions of component services, and 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 implement the proposed approach exploiting efficient synthesis techniques, and experiment with some case studies taken from real world applications and with a parameterized domain. We show that the technique can scale up to cases in which the manual development of* BPEL4WS *composite services is not trivial and is time consuming.*

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* [14] 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 with 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, 18]). 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 this paper we propose 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 EAGLE [11], an expressive requirement language with a clear semantics.
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.

It is widely recognized that, in general, automated synthesis is a hard problem, both in theory and in practice (see, e.g. [21]). In this paper, beyond showing that automated synthesis from abstract BPEL4WS processes is conceptually possible, we also address and investigate the important issue of the practicality of our approach. The automated synthesis we propose exploits and extends a plan synthesis technique [8, 5, 11], known as *planning via symbolic model checking*, that has been shown to be able to scale up

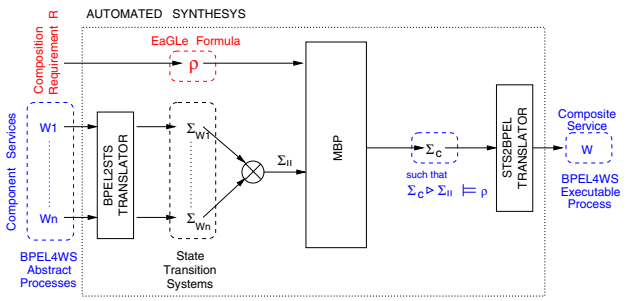


Figure 1. The Approach.

to rather large domains for the problem of planning under uncertainty. This synthesis technique inherits the symbolic mechanisms that are used in *symbolic model checking* [16], a verification technique that has been shown to deal with extremely large systems [6], and which has been used in real-world industrial applications [9].

In this paper, we extend the framework to deal with the state transition systems that are generated by the translation from BPEL4WS processes. We implement the proposed approach in MBP [4], a system built on top of the state-of-the-art symbolic model checker NUSMV [7], and experiment with some case studies taken from a real world application and with a set of parameterized domains. We 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. We also show that, in the considered cases, 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. These experimental results, far from being a proof that automated synthesis can be used in general, provide a witness of the potentials of our approach for the specific case of the synthesis of executable composite web services.

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 explain how (a subset of) BPEL4WS abstract processes can be translated into state transition systems. In Section 4, we describe the EAGLE language for expressing composition requirements. We explain how we automatically generate BPEL4WS executable processes in Section 5. We report the results of our experimental evaluation in Section 6, and 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 MBP tool that performs the synthesis.

The second input to MBP consists of the requirements for the composite service. They are formalized as a formula ρ of the requirement language EAGLE. Given $\Sigma_{\|}$ and ρ, MBP generates the 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.

We have implemented the described approach into a prototype tool, called WS-GEN, available from `http://astroproject.org`.

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 BPEL4WS processes as STSs

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.

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*. 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.

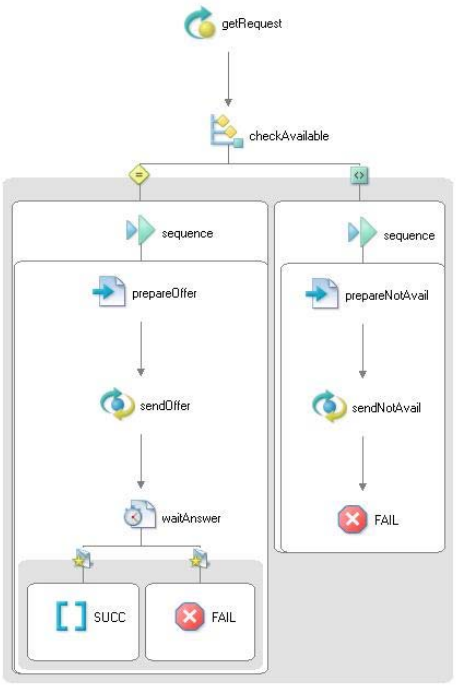
Definition 2 (State transition system)
A state transition system Σ *is a tuple* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$ *where:*

- $\mathcal{S}$ *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 assume that infinite loops of τ-transitions cannot appear in the system, since they are associated to divergent, interaction-free system behaviors.

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. This feature is not exploited within this paper, as all assignments appearing into the abstract BPEL4WS are "opaque", i.e. they model nondeterministic choices by leaving the assigned value unspecified. We omit the formal definition of the translation, which can be found at `http://astroproject.org`, and we illustrate it by means of an example.

Example 3 *Figure 2 shows a graphical representation of the abstract* BPEL4WS *process of the* Shipper *service and the corresponding state transition system. 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* `checkAvailable` *when*




```
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(s,l)]-> pc = checkAvailable,
                                            customer_req_size = s,
                                            customer_req_loc = l;
    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 = waitAsnwer -[INPUT nack]-> pc = FAIL;
    pc = waitAsnwer -[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 2. The Shipper BPEL4WS **process and the corresponding STS.**

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, 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 = checkAvailable -[TAU]-> pc = _sequence_1`*" states that an action* τ *can be executed in state* `checkAvailable` *and leads to the state* `_sequence_1`*.*

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 2 is parametric w.r.t. the types `Size`*,* `Location`*,* `Cost`*, and* `Delay` *used in the messages. 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. 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 Section 6.*

4 Composition Requirements

Our aim is to to generate a specific composite service W that satisfies some desired properties. Composition requirements state the desired properties of the composite service to be automatically synthesized. 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 [12], that have been used in other frameworks to formalize requirements for automated synthesis, but that are unable to describe such requirements. Thus, we adopt EAGLE, a requirement language designed with this specific purpose; EAGLE operators are similar to CTL operators, but their semantics, formally defined in [11], 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 5 *The* EAGLE *formalization of the requirement in Example 4 is the following.*

TryReach
`user.pc=SUCC` $\wedge$ `prod.pc=SUCC` $\wedge$ `ship.pc=SUCC`
$\wedge$ `user.offer_delay=`
`add_delay(prod.offer_delay,ship.offer_delay)`
$\wedge$ `user.offer_cost=`
`add_cost(prod.offer_cost,ship.offer_cost)`
Fail DoReach
`user.pc=FAIL` $\wedge$ `prod.pc=FAIL` $\wedge$ `ship.pc=FAIL`

5 Automated synthesis

MBP has two inputs (see Figure 1): 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 6 (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 7 (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 [20], 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 8 (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 [20]. Given this, we can characterize formally a (web service) composition problem.

Definition 9 (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 [20] we show how to adapt to this task the "Planning as Model Checking" approach, which is able to deal with nondeterminism, partial observability, and with requirements expressed in EAGLE. We exploit the MBP platform that implements such approach for the synthesis of Σ_c.

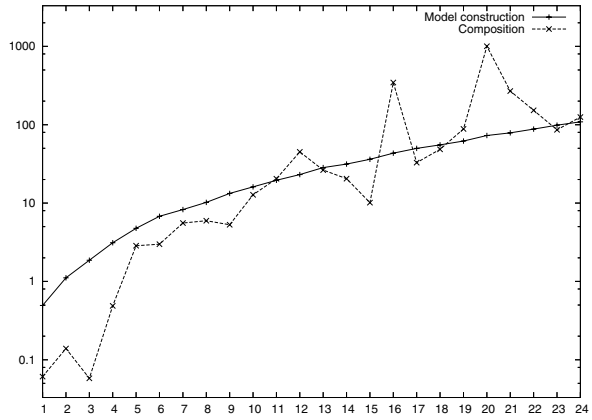
Once the state transition system Σ_c has been generated, it is translated into BPEL4WS by component STS2BPEL of Figure 1. The translation is conceptually simple, but particular care has been put in the implementation of this module in order to guarantee that the generated BPEL4WS is of good quality,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

6 Experimental Evaluation

In order to test the performance of the proposed technique, we have conducted some experiments. All experiments have been done over a 1.8 GHz Pentium machine, equipped with 1 GByte memory, and running a Linux 2.6.7 operating system.

In the first set of experiments, we test automated synthesis w.r.t. the number of services to be composed. Each component is represented by a very simple abstract BPEL4WS process that is requested to provide a service and can respond either positively or negatively. The composition requirement is similar to the one described in the explanatory

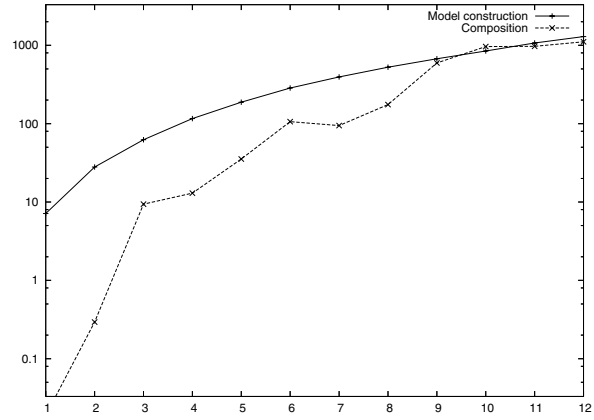
example, i.e., either all services end successfully or a failure is reported to the invoker of the composed service. The results are shown in the following graph.



In the horizontal axis we have the number of components. In the vertical axis, we report in seconds the time for *model construction* (i.e., the time to build the parallel state transition system $\Sigma_{\|}$ in Figure 1 starting from BPEL4WS) and the time for automated *composition* (i.e., the time spent by MBP to generate the composite BPEL4WS process).

As expected, the time for model construction increases regularly with the number of components. Also the composition time increases with a similar trend, but less regularly. This depends on the symbolic mechanisms underlying MBP, which are responsible of constructing a compact internal representations of the search space for the composition phase. This internal representation can be more of less efficient, depending on the specific problem instance, with strong impacts on the performance of the composition. With these examples, the time required for the automated synthesis increases less than exponentially and manages to deal with a rather high number of components in a rather short time. The case with the worst performance among the considered experiments is that of 20 components, where model construction takes about 70 seconds, while automated composition takes about 1000 seconds.

We remark that the component web services used in the previous experiment are very elementary, as they implement essentially an invoke-response protocol. Such kind of services is very common in the domain of web services. To model even more complex situations, we have complicated the parameterized domain by imposing a composition that requires a high degree of interleaving between components. Here, the interactions with each component are more complex than a single invoke-response step, and, to achieve the goal, it is necessary to carry out interactions with all components in an interleaved way. Such interleaving is common in the P&S example where, e.g., the P&S cannot confirm the order to the producer if shipping is not available or if the user does not accept the offer. As shown in the following graph, automated synthesis in this case is more difficult than in the previous set of experiments.



While in the previous experiment model construction and automated composition with 12 components took, respectively, 25 and 45 seconds, now they take both about 1200 seconds. Still, we expect that realistic BPEL4WS compositions will not include more than 12 components, which the technique is able to manage in an acceptable time.

To validate the results of this experimental evaluation, we also conducted some experiments of automated composition on problems extracted from realistic web service domains. The results are reported in the following table.

	number of components	model construction	composition
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.

The first case is the P&S example explained in the previous sections. Automated composition is very fast, since in spite of the interleaving required, we have just three components (shipper, producer, and user of the P&S). We have then experimented with a more complex version of P&S. We have added a further service, taking into account that, in realistic cases, the composite service may require the payment to be dealt by a third party, i.e., a Bank, that is delegated to receiving the money from the client. In this example, the interleaving of interactions is increased by the necessity of receiving a payment confirmation from the Bank before the order can be confirmed to Producer and Shipper. The experimental results confirm that the problem is more difficult than in the original P&S example, and automated composition time increases of one order of magnitude.

Finally, we experimented with a case study taken from a real e-government application we are developing for a pri-

vate 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. According to the existing Italian laws, such a request involves the interaction of different offices of the public administration, such as a Protocol Office, a Technical Committee, a Province Board, a Citizen Service and a Secretary Service. The composition requirement here can be described with a set of constraints on the order of execution of different procedural steps performed by different offices. We considered three variants of this domain, corresponding to an increasing interleaving between the different services, getting in all cases a very good performance.

In all the realistic examples, automated synthesis has shown to be feasible within a reasonable time, surely much faster than manual development of BPEL4WS composite processes. Moreover, the times required for the synthesis confirm the trends of the experiments with parameterized domains.

An important question is the quality of the generated BPEL4WS processes. To evaluate this aspect, we have asked one of our experienced programmers to develop manually the BPEL4WS program for the basic P&S case and we have compared the automatically generated and the hand-written solutions. As a result, we discovered that the two solutions implement the same strategy and have a similar structure. The main difference is in the way they implement the preparation of the offer for the user. The automatically generated process performs a case analysis of the various combinations of costs and delays within one large switch statement; each branch of the switch handles one specific combination. The program developed manually includes a number of temporary variables and a computation that allows to perform the preparation without branching. As a consequence, the automatically generated code is larger than the manual one (22 KBytes vs. 11 KBytes). Except for this problem, that could be solved by optimizing the generation of the branches, the automatically generated code is reasonable, and rather easy to read and understand.

We have seen in Section 3 that a final (and possibly small) range has to be defined for the different types defining the messages exchanges by the component web services. In all the experiments described in this section, ranges are defined to contain only two values. 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.

7 Related Work and Conclusions

In this paper, we have shown how concrete and executable BPEL4WS processes that compose web services can be automatically generated from abstract BPEL4WS descriptions of components and from composition requirements. Our preliminary evaluation shows the potential of our approach and witnesses the feasibility of automatically performing composition of web services to solve significant tasks. In this paper we do not provide the full technical details of the approach, which are presented in the companion paper [20].

As far as we know, the results and the approach presented in this paper are new, and have never been proposed before. 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, [15]). 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 [17], 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., [27, 23]. 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, [22, 24]. These techniques do not take into account behavioral descriptions of web services like abstract BPEL4WS.

In [13], 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 the future, we will address the problem of associating finite ranges to the data types in the generation of the state transition systems from the BPEL4WS web services. In particular, we intend to investigate techniques for deciding the right size of these ranges, so that the generated web service implements a general solution that can be adopted independently on the actual ranges. We also intend to investigate the so called "knowledge-level" techniques [19] for the synthesis of web services: these techniques prevent the necessity of fixing a finite range and allow for the synthesis of a general solution. We plan also to extend the work to the automated synthesis of semantic web services, e.g., described in OWL-S [10] or WSMO [26], along the lines of [25].

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".

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, M. Roveri, and P. Traverso. MBP: a Model Based Planner. In *Proc. of IJCAI-2001 workshop on Planning under Uncertainty and Incomplete Information*, 2001.

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

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[7] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), 2000.

[8] 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.

[9] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

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

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

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

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

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

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

[16] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

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

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

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

[20] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. ICAPS'05*, 2005.

[21] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. of IEEE Symp. of Foundations of Computer Science*, 1990.

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

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

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

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

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

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