Artificial Intelligence 174 (2010) 316–361

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Automated composition of Web services via planning in asynchronous domains

Piergiorgio Bertoli [*], Marco Pistore, Paolo Traverso

Fondazione Bruno Kessler, via Sommarive 18, 38100 Povo (Tn), Italy

A R T I C L E   I N F O

Article history:
Received 9 April 2008
Received in revised form 13 November 2009
Accepted 26 November 2009
Available online 16 December 2009

Keywords:
Planning
Web services
Automated program synthesis

A B S T R A C T

The service-oriented paradigm promises a novel degree of interoperability between business processes, and is leading to a major shift in way distributed applications are designed and realized. While novel and more powerful services can be obtained, in such setting, by suitably orchestrating existing ones, manually developing such orchestrations is highly demanding, time-consuming and error-prone. Providing automated service composition tools is therefore essential to reduce the time to market of services, and ultimately to successfully enact the service-oriented approach.

In this paper, we show that such tools can be realized based on the adoption and extension of powerful AI planning techniques, taking the "planning via model-checking" approach as a stepping stone. In this respect, this paper summarizes and substantially extends a research line that started early in this decade and has continued till now. Specifically, this work provides three key contributions.

First, we describe a novel planning framework for the automated composition of Web services, which can handle services specified and implemented using industrial standard languages for business processes modeling and execution, like WS-BPEL. Since these languages describe stateful Web services that rely on asynchronous communication primitives, a distinctive aspect of the presented framework is its ability to model and solve planning problems for asynchronous domains.

Second, we formally spell out the theory underlying the framework, and provide algorithms to solve service composition in such framework, proving their correctness and completeness. The presented algorithms significantly extend state-of-the-art techniques for planning under uncertainty, by allowing the combination of asynchronous domains according to behavioral requirements.

Third, we provide and discuss an implementation of the approach, and report extensive experimental results which demonstrate its ability to scale up to significant cases for which the manual development of WS-BPEL composed services is far from trivial and time consuming.

1. Introduction

Since its inception, the Web has maintained a fast growth rate in terms of quantity and variety of contained information, becoming a reference information source for billions of users and business entities world-wide. In particular, in the last few years, the economical impact of the Web has grown substantially, due to the fact that the Web is not used anymore just to present static information, but, more and more, to expose services with which a Web user (or a different Web-exposed ser-

[*] Corresponding author.
E-mail addresses: bertoli@fbk.eu (P. Bertoli), pistore@fbk.eu (M. Pistore), traverso@fbk.eu (P. Traverso).

0004-3702/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2009.12.002

vice) can actively interact. This has enacted a range of Web-based solutions for commercial, learning and health-care activities, and gave a substantial push to e-Commerce, e-Learning, and e-Health initiatives (see e.g. [99,51,35,55,92,80,52,33,68]).

This has been the starting point for the emergence of a Service Oriented Computing paradigm, which envisages the adoption of standards for the publication and access of services over the Web, so to allow the interoperability of independently developed procedures over the Web. In such a setting, existing services can be suitably combined by means of Web-based "orchestration" services, so to realize novel and more complex procedures that satisfy some given user or business requirement. For instance, different services taking care of specific aspects related to the organization of a trip (e.g. flight booking, lodging, bank payment, and so on) can be suitably coordinated by an integrated "trip adviser" service, whose adoption may save a customer considerable time and effort in setting up a trip. Indeed, being able to build new services by composing existing ones is crucial to the actual enactment of the service-oriented paradigm. However, the task of manually developing such orchestrations is extremely difficult, time-consuming and error-prone, even for experienced designers and programmers. This calls for the design of effective support techniques and automated tools capable of synthesizing service orchestrations starting from suitable high-level composition requirements.

In this context, planning has proved to be one of the most promising techniques for the automated composition of Web services. Several works in planning have addressed different aspects of this problem, see, e.g., [100,59,34,65,83,17,73,8,84,4,43]. In these works, automated composition is described as a planning problem: services that are available and published on the Web, the *component services*, are used to construct the planning domain, *composition requirements* can be formalized as planning goals, and planning algorithms can be used to generate *composed services*, i.e., plans that compose the component services. These works, which provide different technical solutions, share the conception of services as stateless entities, which enact simple query–response protocols.

An even more difficult challenge for planning is the *automated composition of Web services at the process level*, i.e., the composition of component services that consist of stateful business processes, capable to establish complex multi-phase interactions with their partners. Indeed, in the large majority of real cases, services cannot be considered simply as atomic components, which, given some inputs, return some outputs, in a single request–response step. On the contrary, in most application domains, they need to be represented as stateful processes that realize interaction protocols which may involve different sequential, conditional, and iterative steps. For instance, we cannot in general interact with a "flight booking" service in an atomic step. The service may require a sequence of different operations including an authentication, a submission of a specific request for a flight, the possibility to submit iteratively different requests, acceptance (or refusal) of the offer, and finally, a payment procedure. In these cases, the process, i.e. the published interaction flow, is the key aspect to be considered when (automatically) composing services.

The planning problem corresponding to the automated composition of services that are published as processes is far from trivial. First, component services cannot be simply represented as atomic actions of the planning domain. As a consequence, it is not obvious, like in the case of atomic component services, which is the planning domain that corresponds to the composition problem. Second, in realistic cases, component services publish nondeterministic and partially observable behaviors, since, in general, the outputs of a service cannot be predicted *a priori* and its internal status is not fully available to external services. For instance, whether a payment transaction will succeed cannot be known a priori of its execution, and whether there are still seats available on a flight cannot be known until a specific request is submitted to the service. Third, the plan that coordinates the component services cannot be simply a sequence of actions that call atomic components; rather, it needs to interleave the (partial execution of) component services with typical programming language constructs such as conditionals and loops. Finally, Web service interactions are typically asynchronous: each process evolves independently and with unpredictable speed, and interacts with the other processes only through asynchronous message exchanges. Message queues are used in practical implementations to guarantee that processes do not lose messages that they are not ready to receive.

As a consequence of all these characteristics of Web services, it is far from obvious how their automated composition can be adequately represented as a planning problem. Moreover, their nondeterministic, partially observable, and asynchronous behavior poses strong requirements and introduce novel problems for the planning techniques that can be used. This has led the authors of this paper and their colleagues to investigate a research line on service oriented composition, that started with [71,95], and continued with [78,79,74,61,62], up to date [60,63,72,64,21]. While distinct for the technical solutions and degree of maturity, these works share two general ideas. The first consists in taking, as an algorithmic and technological baseline, the "planning via model checking" approach devised by the authors of this paper together with other colleagues [16,77,32,15]. Such approach combines state-of-the-art performance with the ability to deal with general forms of nondeterminism, therefore tackling effectively one of the critical aspects implied by the nature of Web services. The second idea is to face actual composition problems by pragmatically considering services expressed using de-facto standard languages such as WS-BPEL [1,31]. While this choice renders the problem further complex, it is strongly motivated by the objective to provide usable tools.

This paper summarizes and significantly extends a large portion of the corpus of work presented in [71,95,78] and [79,74,61,62,60,63,72,64], providing for the first time both a comprehensive survey of the framework underlying the approach, and a thorough account of the formal and empirical features of the underlying algorithms. In particular, in this paper, we focus on a form of automated process-level composition of Web services where services can be represented as finite state automata, and composition requirements command the finite termination of (the execution of) component services. While extensions to consider infinite-state services and infinite component iterations are possible, and indeed considered in [74, 62,60,21], this form of composition is already expressive enough to cover a large variety of relevant service composition scenarios. Focusing on this setting allows us to keep the presentation compact and self-contained, while comprehensively surveying all the formal and empirical aspects at a sufficient level of detail.

In detail, this work provides the following results:

- We formalize the automated process-level composition of services (considering *reachability* composition requirements, and finite-state representations of services) as a planning problem. The planning domain is constructed from processes that describe component services and results in a nondeterministic, partially observable, and asynchronous domain.
- We devise a novel planning approach that is able to work in nondeterministic, partially observable, asynchronous domains. We prove formally its properties, its correctness and its completeness.
- We embed the planning framework in a real industry-wide adopted environment for describing, composing, and executing Web services. In this environment, Web services are described using WS-BPEL (Business Process Execution Language for Web Services [1,31]), one of the most used industrial standard languages for describing and executing Web services. We generate automatically the planning domain from the WS-BPEL specification of the component services. We then automatically translate back the generated plans into executable WS-BPEL processes that implement the composed service and that can be run on standard execution engines.
- We discuss our implementation of the proposed framework and perform an extensive experimental evaluation. We show that our approach can be used in practice to automatically compose Web services described in WS-BPEL. The technique can scale up to significant cases in which the manual development of WS-BPEL composite services is not trivial and time consuming.

1.1. Roadmap

Our roadmap for the paper is following. First, in Section 2, we describe informally the automated composition problem, introducing an explanatory example that will be used all along the paper, and showing how the component services, and the intended orchestration, can be expressed using the WS-BPEL language and its complementary WSDL interface definitions.

In Section 3, we show how WS-BPEL specifications of component services can be interpreted as state transition systems (STSs), that is as automata that describe the behavior of services, and we discuss how it is possible to perform automatically such a translation.

Based on these grounds, in Section 4 we provide the formal definition of the composition problem. In particular, we first introduce the key notion of deadlock-free controller of an STS, which identifies the adequacy requirements that an STS must satisfy in order to act as a suitable orchestrator for some other STSs. Then, we clarify what kind of composition requirements are considered in our setting, and what does it mean to satisfy them. This allows us to model the composition problem as one of synthesizing a deadlock-free controller for the set of concurrently executing STSs associated to the component services, such that its behaviors satisfy a given composition requirement. Once the formal definition of the composition problem is set up, in Section 5, we describe an algorithm for the automated composition and prove its properties. To do so, we first rephrase the problem in a way that makes it explicit how it can be solved by constructing and visiting a (portion of a) specific search space. In particular, the elements of such a search space must represent *beliefs* of the orchestration service over the current status of the (STSs associated with the) component services. Our approach consists in first pre-computing such belief-level search space, and then searching it by means of effective techniques inspired by model-checking based planning, for which we provide and discuss algorithmic descriptions.

In Section 6, we discuss the implementation of the planning algorithm, and we describe the experimental evaluation. For this purpose, on top of our reference scenario, we introduce a variety of scalable scenarios, report data on experiments run on all of them, and wrap up by synthesizing our empirical findings about the features of our approach. Finally, Sections 7 and 8 discuss in detail work related to ours, and draw conclusions about our approach, as well as about possible directions of future work.

2. The problem

In this section, we describe the service composition problem, introducing a scenario which will be then used as a reference throughout the paper. The scenario will allow us to better discuss the way services taking part to a composition can be expressed by means of a standard language, and to highlight some relevant assumptions underlying our approach.

Taking a general view, the service composition problem amounts to construct a new service (the *composed service*) that performs some desired functionality by interacting with available services (the *component services*).

In our approach, we consider the case where stateful component services require complex interaction protocols to perform their task. This is a very expressive setting which enables capturing a large variety of real-life scenarios. However, a single paper would not be enough to expose and solve the problem in its full width, considering fully generic components and behavioral requirements. We therefore take two key assumptions that serve to constrain the discussion, and we leave to Section 7 an analysis on how to relax such assumptions.

The first key assumption concerns the nature of composition requirements. In this work, we consider composition requirements that constrain the data exchanged during the orchestration and the *final* admissible states of component services.

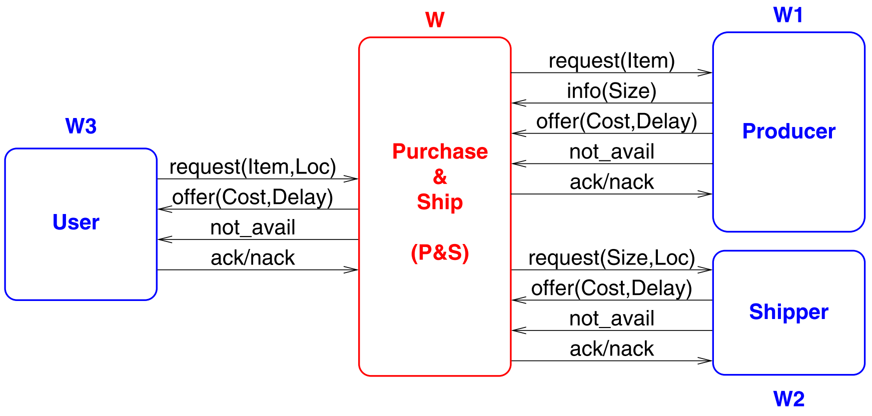


Fig. 1. The P&S scenario.

Such requirements, therefore, imply that the execution of each component service terminates in a finite number of steps; that is, in our setting, we can only admit component services whose loops (if any) can be proven to exit after a finite number of iterations. This has the further consequence that also the orchestration terminates in a finite number of steps, and therefore, without lack of generality, we will be able to focus on the synthesis of acyclic orchestration services.

The second assumption is the ability to represent services in terms of finite-state automata. Of course, this assumption holds immediately for services whose number of internal states is finite. However, this does not rule out services whose internal states are infinite, insofar it is possible to identify an adequate finite-state abstraction, as we will discuss in more detail in Section 7. This is indeed the case in many realistic composition scenarios, such as the one described in the following reference example.

Example 1. Our goal is to implement a composed service for furniture purchasing and delivering, the P&S service, by composing two independent existing services: a furniture purchase service Producer, and a delivery service Shipper. The P&S service should allow a User to ask for desired products that should be delivered at a desired location, see Fig. 1. The component services are not atomic, i.e., they cannot be executed in a single request–response step; they are stateful processes, and require establishing an interaction protocol that involve different sequential and conditional steps. In our example, the Producer accepts requests for given products. If the requested product is available, it provides some information about it, among which for instance its size; then, if the requester acknowledges the interest to buy, the Producer makes an offer with a cost and a production time. This offer can be accepted or refused by the requester. In both cases the Producer terminates execution, with success or failure, respectively.

Also the behavior of the Shipper is stateful. Once it receives a request to deliver an object of a given size to a certain location, the Shipper may refuse to process the request, or may produce an offer where the cost and the time to deliver are specified. In the latter case, the invoker can either confirm the order, or cancel it.

Similarly, the User performs a two-step interaction with the P&S: first it sends his/her request, then it gets either a refusal or an offer, and finally (in the latter case) it either confirms or disconfirms the request. Notice that, as shown in Fig. 1, the user is perceived analogously to the Shipper and Producer, as one of the components with which P&S will need to interact; as such, we assume the user's behavior to be modeled as a service, representing a software front-end to a human entity.

The goal of the P&S is to sell a product at a destination, as requested by a customer. To achieve this, the P&S has to interact with the customer on the one side and with Producer and Shipper on the other side, trying to reach a situation where the three interactions reach a successful completion, i.e., three final confirmations are obtained. Clearly, the goal of selling a product at destination may be not always achievable by the P&S, since this depends on decisions taken by third parties that are out of its control: the Producer and the Shipper may refuse to provide the service for the requested product and location (e.g., since the product is not available or the location is out of the area of service of the shipper), and the customer may refuse the offer by the P&S (e.g., since it is too expensive). If this happens, the P&S should step back from both orders and it should not commit to the customer. Indeed, we do not want the P&S to buy something that cannot be delivered, as well as we do not want it to promise a product at destination that it will not be able to buy.

The order in which the interactions with the different services are interleaved in the implementation of the P&S is critical. For instance, when the P&S gets a request for a given item from a customer, it has to obtain the size of the item from the producer before it can call the shipper. Two offers, from the Producer and from the Shipper, are necessary to the P&S in order to make an overall offer to the customer. Moreover, the offers from Producer and Shipper can be accepted by the P&S only after the customer has accepted the offer from the P&S. The necessity to figure out and realize all these constraints makes the implementation of the P&S a complex task, also in simple scenarios like the one described above.

Table 1
Synopsis table for WSDL.

Construct	Meaning
types	The types element encloses data type definitions that are relevant for the exchanged messages. In particular, types are used via the type system in the WSDL XML schema definition (XSD for short).
message	Messages consist of one or more logical parts. Each part is associated with a type from some type system. WSDL defines two different message-typing attributes: element, which refers to an XSD element using a qualified XML name, or type, which refers to an XSD simpleType or complexType (using, again, a qualified XML name).
portType	A portType is a named set of operations.
operation	An operation is a named entity that specifies, through the output and input elements, the abstract message format for the solicited request and response, respectively. WSDL defines four types of operation: • One-way, the operation can receive a message but will not return a response • Request–response, the operation can receive a request and will return a response • Solicit–response, the operation can send a request and will wait for a response • Notification, the operation can send a message but will not wait for a response
port	While portTypes define abstract functionality by using abstract messages, ports provide actual access information, including communication endpoints and (by using extension elements) other deployment-related information, such as public keys for encryption. These are associated by means of so-called bindings.
service	A service is viewed simply as a group of ports.

2.1. *Specifying services: the* WS-BPEL *and* WSDL *languages*

In this paper, we assume that the component and composed services are expressed using the WS-BPEL language. WS-BPEL (the "Business Process Execution Language") [1,31] is the de-facto standard for describing the stateful behavior of Web services. In WS-BPEL, a set of atomic communication operations (i.e., invoke, receive, and reply activities) are combined within a workflow that defines the process implemented by the stateful service. The atomic communications correspond to atomic Web service operations, and are defined in a WSDL (Web Service Description Language [28]) specification. WSDL is an extensible markup language (XML) format, and is the standard language for describing operations implemented as Web services along with the input and output data of these operations. Table 1 reports a synopsis of the fundamental WSDL keywords, together with their informal semantics.

Table 2 reports the key WS-BPEL constructs, providing their informal semantics. It is important to remark that there are two flavors of WS-BPEL, namely *executable* WS-BPEL programs, that are used to implement the process defining a service, and can be run by standard engines such as the Active BPEL Open Engine or the Oracle BPEL Process Manager [3,69], and *abstract* WS-BPEL specifications, which are used to publish the interaction protocol with external Web services. That is, an abstract WS-BPEL process is meant to describe a public interface, whereas an executable process defines an actual implementation. WS-BPEL programs are identified as being abstract or executable by appropriately setting an abstractProcess attribute, and share the vast majority of the constructs; however, only in abstract programs it is possible to hide details over the assignment of variables and the flow control conditions (by an opaque construct).

Given our choice for the specification of services, our general statement of the composition problem can be described as follows: "*Let W be a set of component services, whose interactions are described as abstract* WS-BPEL *specifications, and let ρ be some composition requirement that describes the desired functionalities of a composed service. The problem of service composition for W and ρ requires constructing a new executable* WS-BPEL *that, when executed, satisfies the requirements ρ by interacting with the component services in W*".

In the following, prior to detail our view of composition requirements, we go more in detail about our reference scenario, discussing the component service specifications that serve as input to the composition problem in that case.

Example 2. Fig. 2 presents the WSDL specification for the Producer, abridged from technical details irrelevant to our discussion. The WSDL specification starts with the definition of the data types used in the interactions. In the case of the Producer, they are the requested `Item` and its `Size`, `Cost`, and production `Delay`. The actual definition of these data types is not relevant to our purposes, and is also omitted from the WSDL specification.

The WSDL specification then describes the structure of the messages relevant for the interactions with the Producer. According to the specification, a `requestMsg` message contains the requested article, `art`. The `infoMsg` and `offerMsg` messages contain, respectively, the `size` and the production `cost` and `delay` for an article. The other three messages (`unavailMsg`, `ackMsg`, `nackMsg`) do not carry data values.

Then, the WSDL specification defines the invocation and reply operations provided by the service. Operations are collected in port types that are associated to different communication channels of the producer service with its partners. In our example, we define two port types, namely `P_PT` and `PC_PT` (for "Producer Port Type" and "Producer Callback Port Type"

Table 2
Synopsis table for WS-BPEL.

Construct	Meaning
partnerLink	In WS-BPEL a Web service that is involved in the process is always modeled as a partnerLink. Every partnerLink is characterized by a partnerLinkType which is defined in the WSDL definition. The role of the process in the communication is specified by a myRole attribute, and the role of the partner is specified by the attribute partnerRole.
variable	Variables offer the possibility to store data. The messages that get stored are most of the time either coming from partners or going to partners. Variables also offer the possibility to store data that is only state based and never send to partners. There are three types of variables: WSDL message type, XML Schema simple type and XML Schema element.
assign	It allows to copy the data from one source (a variable or an expression written using XPath, the XML path language, to address and combine parts of variables) to a target variable. Only in abstract processes, an assignment source can be opaque, in which case the target variable is meant as being nondeterministically assigned with any value in its domain range. This may reflect into a nondeterministic service behavior, as the variable can be used within control constructs. Finally, the assign activity can be to perform *dynamic binding* of partners, by copying endpoint references to and from partner links.
correlation set	Each correlation set in WS-BPEL is a named group of properties that, taken together, serve to define a way of identifying an application-level conversation within a business protocol instance. After a correlation set is initiated, the values of the properties for a correlation set must be identical for all the messages in all the operations that carry the correlation set and occur within the corresponding scope until its completion.
receive	A receive activity specifies the partner link it expects to receive from, and the port type and operation that it expects the partner to invoke, and a variable that is to be used to store the message data received. In addition, receive activities play a role in the life-cycle of a business process. The only way to instantiate a business process in WS-BPEL is to annotate a receive activity with the createInstance attribute set to "yes".
reply	A reply activity is used to send a response to a request previously accepted through a receive activity. Such responses are only meaningful for synchronous interactions. An asynchronous response is always sent by invoking the corresponding one-way operation on the partner link.
invoke	This construct is used to invoke a service operation. Such an operation can be a synchronous request/response or an asynchronous one-way operation. WS-BPEL uses the same basic syntax for both with some additional options for the synchronous case. An asynchronous invocation requires only the input variable of the operation because it does not expect a response as part of the operation. A synchronous invocation requires both an input variable and an output variable.
sequence	A sequence activity contains one or more activities that are performed sequentially, that is, in the order in which they are listed within the corresponding XML element.
switch	The switch activity consists of an ordered list of one or more conditional branches defined by case elements, followed optionally by an otherwise branch. The case branches of the switch are considered in the order in which they appear. The first branch whose condition holds true is taken and provides the activity performed for the switch. If no branch with a condition is taken, then the otherwise branch is taken.
while	The while activity supports repeated performance of a specified iterative activity. The iterative activity is performed until the given Boolean loop condition no longer holds true.
pick	The pick activity awaits the occurrence of one of a set of events and then performs the activity associated with the event that occurred. After a pick activity has accepted an event for handling, the other events are no longer accepted by that pick. Possible events are the arrival of some message in the form of the invocation of an operation, which is handled within a correspondent onMessage sub-activity analogous to a receive, or a "timer alarm", handled within an onTimer sub-activity. Similarly to the receive activity, a pick can initiate a process by setting the createInstance attribute (in which case no timer events are permitted).
flow	The flow activity is the most visible remnant of the WSFL language, which was into the WS-BPEL specification. Its most fundamental effect is to enable concurrency of activities: a flow completes when all of the activities it contains have completed. Furthermore, to allow for complex concurrency scenarios, the flow construct allows the expression of synchronization dependencies between the enclosed activities. In particular, execution of an activity can be dependent on other activities and certain conditions. The synchronization dependencies are expressed by means of link constructs A named link is defined as a sub-element of the flow activity, and then attached to a source and a target activities. As a consequence, the target activity will only be executed upon termination of the source activity.

respectively), which will be used for the incoming requests and the outgoing messages by the Producer service.[1] Finally, the WSDL specification defines bi-directional links between the service and its partners. In our case, there is only one of such links, `P_PLT`, between the Producer and the customer invoking it.

Fig. 3 shows the abstract WS-BPEL of the Producer (again, abridged from some low-level technical details). It starts with a declaration of the links with external parties that are exploited within the process. In this case, only one external partner exists, the `client` of the producer. The type of the link is `P_PLT` (see the WSDL file), for "Producer Partner Link Type",

[1] In our example, we only exploit one-way operations, i.e., operations that consist of a communication from the sender to the receiver. WSDL and WS-BPEL also support invoke-and-response operations, which define an input message and an output message (plus a failure message to manage non-nominal outcomes). Since the invocation has a non-blocking behavior in WS-BPEL, invoke-and-response operations can always be realized as pairs of unidirectional operations. Therefore, in the paper we can limit to discuss unidirectional operations without giving up generality, based on the possibility of such a translation.

```
(1)  ⟨definitions name = "Producer"⟩
(2)    ⟨types⟩
(3)      ⟨schema targetNamespace  = "http://www.astroproject.org/Producer"⟩
(4)        ⟨element  name = "Item"⟩
(5)          .......
(6)        ⟨/element⟩
(7)        ⟨element  name = "Size"⟩
(8)          .......
(9)        ⟨/element⟩
(10)       ⟨element  name = "Location"⟩
(11)         .......
(12)       ⟨/element⟩
(13)       ⟨element  name = "Cost"⟩
(14)         .......
(15)       ⟨/element⟩
(16)       ⟨element  name = "Delay"⟩
(17)         .......
(18)       ⟨/element⟩
(19)     ⟨/schema⟩
(20)   ⟨/types⟩
(22)   ⟨message  name = "requestMsg"⟩
(23)     ⟨part name = "art" type  = "Item"/⟩
(24)   ⟨/message⟩
(25)   ⟨message  name = "infoMsg"⟩
(26)     ⟨part name = "size" type  = "Size"/⟩
(27)   ⟨/message⟩
(28)   ⟨message name = "offerMsg"⟩
(29)     ⟨part name = "cost" type  = "Cost"/⟩
(30)     ⟨part name = "delay" type  = "Delay"/⟩
(31)   ⟨/message⟩
(32)   ⟨message  name = "ackMsg"⟩
(33)   ⟨message  name = "nackMsg"⟩
(34)   ⟨message  name = "unavailMsg"⟩
(36)   ⟨portType  name = "P_PT"⟩
(37)     ⟨operation  name = "request"⟩
(38)       ⟨input  message  = "requestMsg"/⟩
(39)     ⟨/operation⟩
(40)     ⟨operation  name = "ack"⟩
(41)       ⟨input  message  = "ackMsg"/⟩
(42)     ⟨/operation⟩
(43)     ⟨operation  name = "nack"⟩
(44)       ⟨input  message  = "nackMsg"/⟩
(45)     ⟨/operation⟩
(46)   ⟨/portType⟩
(48)   ⟨portType  name = "PC_PT"⟩
(49)     ⟨operation  name = "info"⟩
(50)       ⟨input  message  = "infoMsg"/⟩
(51)     ⟨/operation⟩
(52)     ⟨operation  name = "unavail"⟩
(53)       ⟨input  message  = "unavailMsg"/⟩
(54)     ⟨/operation⟩
(55)     ⟨operation  name = "offer"⟩
(56)       ⟨input  message  = "offerMsg"/⟩
(57)     ⟨/operation⟩
(58)   ⟨/portType⟩
(60)   ⟨plnk:partnerLinkType  name = "P_PLT"⟩
(61)     ⟨plnk:role  name = "P_Service"⟩
(62)       ⟨plnk:portType  name = "P_PT"/⟩
(63)     ⟨plnk:role⟩
(64)     ⟨plnk:role  name = "P_Customer"⟩
(65)       ⟨plnk:portType  name = "PC_PT"/⟩
(66)     ⟨plnk:role⟩
(67)   ⟨plnk:partnerLinkType⟩
(69) ⟨/definitions⟩
```

Fig. 2. The WSDL for the Producer process.

and the two port types associated with it (`P_PC` and `P_PT`) are given separate roles that correspond to the direction of the messages, following their definition.

Then the variables that are used in input/output messages are declared (see lines 5–13), together with their typing, referring either to the types defined in the WSDL file, or to standard WSDL types (e.g. `xsd:boolean`). The rest of the abstract WS-BPEL specification (lines 14–55) describes the interaction flow.

The Producer is activated by a customer request of information for a specified product (see the `receive` instruction at line 15). The item specified by the requester is stored in variable `reqVar`. The `operation="request"` identifies which WSDL operation is performed. The `partnerLink="client"` and `portType="P_PT"` fields serve to identify the link

```
(1)  ⟨process  name = "Producer" abstractProcess = "yes"⟩
(2)    ⟨partnerLinks⟩
(3)      ⟨partnerLinkname = "client" partnerLinkType = "P_PLT" myRole  = "P_PT" partnerRole  = "PC_PT"/⟩
(4)    ⟨/partnerLinks⟩
(5)    ⟨variables⟩
(6)      ⟨variable name = "reqVar" messageType = "requestMsg"/⟩
(7)      ⟨variable name = "unavailVar" messageType = "unavailMsg"/⟩
(8)      ⟨variable name = "infoVar" messageType = "infoMsg"/⟩
(9)      ⟨variable name = "offerVar" messageType = "offerMsg"/⟩
(10)     ⟨variable name = "ackVar" messageType = "ackMsg"/⟩
(11)     ⟨variable name = "nackVar" messageType = "nackMsg"/⟩
(12)     ⟨variable name = "availVar" messageType = "xsd:boolean"/⟩
(13)   ⟨/variables⟩
(14)   ⟨sequence name = "main"⟩
(15)     ⟨receive name = "getRequest" operation = "request" variable = "reqVar" partnerLink = "client"
(16)              portType = "P_PT"/⟩
(17)     ⟨assign name = "setAvail"⟩
(18)       ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "availVar"/⟩⟨/copy⟩
(19)     ⟨/assign⟩
(20)     ⟨switch name = "checkAvail"⟩
(21)       ⟨case condition = "bpws:getVariableData('availVar') = xsd:False"⟩
(22)         ⟨invoke name = "sendUnavail" operation = "unavail" inputVariable = "unavailVar"
(23)                  partnerLink = "client" portType = "PC_PT"/⟩
(24)       ⟨/case⟩
(25)       ⟨otherwise
(26)         ⟨assign name = "prepareInfo"⟩
(27)           ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "infoVar" part = "size"/⟩⟨/copy⟩
(28)         ⟨/assign⟩
(29)         ⟨invoke name = "sendInfo" operation = "info" inputVariable = "infoVar" partnerLink = "client"
(30)                  portType = "PC_PT"/⟩
(31)         ⟨pick⟩
(32)           ⟨onMessage name = "getNack" operation = "nack" variable = "nackVar" partnerLink = "client"
(33)                      portType = "P_PT"/⟩
(34)           ⟨/onMessage⟩
(35)           ⟨onMessage name = "getAck" operation = "ack" variable = "ackVar" partnerLink = "client"
(36)                          portType = "P_PT"⟩
(37)             ⟨assign name = "prepareOffer"⟩
(38)               ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "offerVar" part = "delay"/⟩⟨/copy⟩
(39)               ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "offerVar" part = "cost"/⟩⟨/copy⟩
(40)             ⟨/assign⟩
(41)             ⟨invoke operation = "sendOffer" inputVariable = "offerVar"
(42)                      partnerLink = "client" portType = "PC_PT"⟩
(43)             ⟨pick⟩
(44)               ⟨onMessage name = "getAccepted" operation = "ack" variable = "ackVar"
(45)                          partnerLink = "client" portType = "P_PT"/⟩
(46)               ⟨/onMessage⟩
(47)               ⟨onMessage name = "getRefused" operation = "nack" variable = "nackVar"
(48)                          partnerLink = "client" portType = "P_PT"/⟩
(49)               ⟨/onMessage⟩
(50)             ⟨/pick⟩
(51)           ⟨/onMessage⟩
(52)         ⟨/pick⟩
(53)       ⟨/otherwise⟩
(54)     ⟨/switch⟩
(55)   ⟨/sequence⟩
(56) ⟨/process⟩
```

Fig. 3. The abstract WS-BPEL for the Producer process.

along which the request is received (of course, these fields become useful when there are more than one partners for the WS-BPEL process).

At lines 17–19, Producer decides on the availability of the requested product. We remark that in this case, by marking the source of the data as "opaque", the abstract WS-BPEL hides the actual implementation of the internal logic that governs the Producer's choice. This capability of abstract WS-BPEL is crucial whenever, as it is often the case, a business entity does not want to disclose details about internal procedures and choice criteria to the world via its Web exposure. Then, in the switch activity named checkAvail, the service decides on the basis of the availability.

If the Producer is not available, it signals this to its partner (see activity invoke on line 22) and terminate; otherwise, it prepares and send the information regarding the size of the required item (lines 26–29).

The information regarding the size is (internally and opaquely) computed within the assign statement named prepareInfo.

After sending the information, the Producer suspends (instruction pick at line 31), waiting for the customer either to acknowledge or to refuse to proceed to buy the specified product. If the customer refuses to proceed (statement onMessage on line 32), the Producer terminates the execution. If, otherwise, a message is received that corresponds to operation ack (line 35), meaning that the customer confirms its interest in the item, then the Producer prepares and sends and offer to

```
(1)  ⟨process name = "User" abstractProcess = "yes"⟩
(2)    ⟨partnerLinks⟩
(3)      ⟨partnerLinkname = "client" partnerLinkType = "U_PLT" myRole  = "U_PT" partnerRole  = "UC_PT"/⟩
(4)    ⟨/partnerLinks⟩
(5)    ⟨variables⟩
(6)      ⟨variable name = "reqVar" messageType = "requestMsg"/⟩
(7)      ⟨variable name = "unavailVar" messageType = "unavailMsg"/⟩
(8)      ⟨variable name = "offerVar" messageType = "offerMsg"/⟩
(9)      ⟨variable name = "ackVar" messageType = "ackMsg"/⟩
(10)     ⟨variable name = "nackVar" messageType = "nackMsg"/⟩
(11)     ⟨variable name = "acceptsVar" messageType = "xsd:boolean"/⟩
(12)   ⟨/variables⟩
(13)   ⟨sequence name = "main"⟩
(14)     ⟨assign name = "prepareRequest"⟩
(15)       ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "reqVar" part = "art"/⟩⟨/copy⟩
(16)       ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "reqVar" part = "loc"/⟩⟨/copy⟩
(17)     ⟨/assign⟩
(18)     ⟨invoke name = "sendRequest" operation = "request" inputVariable = "reqVar" partnerLink = "client"
(19)             portType = "U_PT"⟩
(20)     ⟨pick⟩
(21)       ⟨onMessage name = "getUnavail" operation = "unavail" variable = "unavailVar" partnerLink = "client"
(22)                   portType = "UserC_PT"⟩
(23)       ⟨/onMessage⟩
(24)       ⟨onMessage name = "getOffer" operation = "offer" variable = "offerVar" partnerLink = "client"
(25)                  portType = "UserC_PT"⟩
(27)         ⟨assign name = "prepareAcceptOrReject"⟩
(28)           ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "acceptsVar" ⟩⟨/copy⟩
(29)         ⟨/assign⟩
(30)         ⟨switch name = "checkAcceptance"⟩
(31)           ⟨case condition = "bpws:getVariableData('acceptsVar') = xsd:False" ⟩
(32)             ⟨invoke name = "refused" operation = "nack" inputVariable = "nackVar" partnerLink = "client"
(33)                     portType = "U_PT"⟩
(34)           ⟨/case⟩
(35)           ⟨otherwise
(36)             ⟨invoke name = "accepted" operation = "ack" inputVariable = "ackVar" partnerLink = "client"
(37)                     portType = "U_PT"⟩
(38)           ⟨/otherwise⟩
(39)         ⟨/switch⟩
(40)       ⟨/onMessage⟩
(41)     ⟨/pick⟩
(42)   ⟨/sequence⟩
(43) ⟨/process⟩
```

Fig. 4. The abstract WS-BPEL for the User process.

the customer, which contains a cost, and an expected delivery time (lines 35–41). Again, opaqueness is used to hide the actual way in which the production time (line 38) and the cost (line 39) are computed.

After sending the offer, the Producer suspends waiting for a final acknowledgment of refusal of the offer by the client. and terminates after receiving it either with success or failure, respectively (lines 43–50).

The WSDL and abstract WS-BPEL specifications for the Shipper are similar to that of the Producer, so we do not report them.

For the purposes of composition, we assume to start from a situation where the abstract WS-BPEL of the Producer and of the Shipper are published, and available as input to potential users of the services, as well as to the composition task. On top of this, we also assume the availability of a description of the interaction protocol that the User is intended to carry out with the composite service: the implementation of the P&S will depend on the protocols of all of the interacting partners, i.e., the Producer, the Shipper, and the User.

Example 3. Fig. 4 shows the abstract WS-BPEL of the User. After determining the content of its request (lines 14–17), the user invokes the composite service (line 18) and suspends for a reply. The reply can signal that the request cannot be satisfied, or propose an offer. In the first case, the service simply terminates (line 23). In the second case, the user then decides whether to accept or not such offer, and signals his decision before terminating (lines 24–40).

2.2. Goal and solution

Considering our reference example, the goal of the composition task is to generate the executable code of the P&S service in such a way that the following criteria are obeyed: (i) sell a product at destination whenever possible, and (ii) whenever this is not possible, guarantee that no pending commitments with the partners are left, that is, handle consistently all the cases where some partner prevents the successful selling to take place. Of course, in achieving this goal, the P&S has to respect the protocols described by the three abstract WS-BPEL specifications of Producer, Shipper, and User. Furthermore, we impose additional crucial constraints, capturing the way data exchanged between the services are related as well as the nature of the business: namely, P&S must obey a constraint that associates the cost paid by the User with the prices proposed by the Producer and Shipper, including a profit margin.

Example 4. Fig. 5 shows a possible implementation of the P&S as an executable WS-BPEL process.

As we see, the service features three `partnerLink`s, one for each of the three component services it has to interact with. Its variables refer to the typed declared in the WSDL files associated to each component; for instance, the `U_RequestVar` has the `requestMsg` type declared for the User, which contains an item and a location, while `S_RequestVar` has the `requestMsg` type declared for the Shipper, and contains a size and a location.

The P&S is activated by a request from the customer — the request specifies the desired product and delivery location (line 28). The P&S asks the Producer for information about the product, namely its size (line 35). If the item can be produced, then the Producer gives the P&S information about the size, the cost, and the time required for the service (line 45). The P&S then asks the delivery service the price and time needed to transport an object of such a size to the desired locations (line 58). If the Shipper makes an offer to deliver the item, then the P&S sends an aggregated offer to the customer. This offer takes into account the overall production and delivery costs and time (line 93). If the customer sends a confirmation, the final acknowledgment is sent both to the Producer and to the Shipper (lines 99–108), via a `flow` construct that specifies that the associated `invoke` activities can be executed concurrently. As a consequence, all three component services terminate successfully.

While we have described only the nominal flow of interaction, the executable WS-BPEL of the P&S takes also into account the cases in which the service cannot succeed, i.e., the cases in which either the Producer cannot provide the product, or the Shipper cannot deliver, or the customer does not accept the offer, by canceling the pending orders (see lines 41, 66–69, and 114–123) (see Figs. 5 and 6).

Notice that the executable WS-BPEL for the P&S is much more complex than any of the abstract components. Indeed, while the abstract WS-BPEL s only describe the protocols exported by the partners, the executable P&S must implement all the communication with the three component services (the User, the Producer and the Shipper), as well as all the computations over the internal variables, e.g., those computing the total cost and time for the offer to the customer.

It is important to remark that the component and composed services in our P&S example comply with the assumptions discussed at the start of this section. First, every component service is guaranteed to terminate in a finite number of steps; this is trivially established, in this case, since the User, Producer and Shipper protocols are completely loop-free. Second, the requirement expressed for the P&S service implies the termination of all partners activated by the orchestration, either by succeeding in the transaction, or by unrolling all commitments. That is, the kind of composition requirement is one of *reachability*: upon termination of the orchestration, a certain set of properties must hold. As a consequence of these two assumptions, also the composed P&S service is loop-free. Finally, although the types of the variables of the component services are not explicitly reported and may be of infinite range, we remark that, insofar as the variables are many-valued, the structure of a P&S service that orchestrates them according to the requirements is independent of their ranges and concrete values. Indeed, we observe that if one considers components with single-valued variables, it is possible to devise a P&S orchestrator which, while satisfactory in that specific case, may not work in the general case: since the values produced by component services are predictable regardless of their inputs, P&S might exploit this, and route foreseen results a priori from receiving them. Vice versa, as soon as one considers components whose variables have two or more values, the orchestration must actually ask component services to operate and produce outputs which then can be routed accordingly, considering all possible cases. That is, in our scenario, considering binary (disjoint) types for the variables of the component services is enough to separate all conceptually distinct behaviors of the components, and as such, to obtain generic P&S orchestrator. In general, in this paper, we assume that, in case it is needed, a suitable finitization of component services is identified in a pre-processing phase, and we leave a more detailed discussion of finitization techniques in Section 8.

3. Processes as state transition systems

In this section, we discuss how services expressed using the standard WS-BPEL language can be recasted into a well-known formal model which can be more easily manipulated to solve the composition problem, and we show how our approach works on the reference example. Specifically, we encode WS-BPEL processes as *finite state transition systems*. While our choice to adopt *finite* STSs will require particular care to appropriately encode (possibly infinite-state) services, it guarantees the possibility to build upon effective synthesis and representation techniques which are not available for infinite-state transition systems. In the following, we first formally define the notion of STS, and then describe the way services can be translated into STSs and back.

3.1. State transition systems

A *finite state transition system* (STS) describes a dynamic system that can be in one of its possible *states* (some of which are marked as *initial states*). A state can be understood as a collection of *propositional properties* that hold in that state, and it can evolve to new states as a result of performing some *actions*. In a state transition system, actions are distinguished in

```
⟨process name = "PandS_Executable"⟩
  ⟨partnerLinks⟩
    partnerLinkmyRole  = "PandS_Service" name = "client" partnerLinkType = "tns:PandS_PLT"
              partnerRole  = "PandS_Customer"⟩
    partnerLinkmyRole  = "Shipper_Customer" name = "PandS_Shipper" partnerLinkType = "Shipper:S_PLT"
              partnerRole  = "Shipper_Service"⟩
    partnerLinkmyRole  = "Producer_Customer" name = "PandS_Producer" partnerLinkType = "Producer:P_PLT"
              partnerRole  = "Producer_Service"⟩
  ⟨/partnerLinks⟩
  ⟨variables⟩
    ⟨variable name = "U_RequestVar" messageType = "User:requestMsg"/⟩
    ⟨variable name = "U_OfferVar" messageType = "User:offerMsg"/⟩
    ⟨variable name = "U_AckVar" messageType = "User:ackMsg"/⟩
    ⟨variable name = "U_NackVar" messageType = "User:nackMsg"/⟩
    ⟨variable name = "P_RequestVar" messageType = "Producer:requestMsg"/⟩
    ⟨variable name = "P_UnavailVar" messageType = "Producer:unavailMsg"/⟩
    ⟨variable name = "P_InfoVar" messageType = "Producer:infoMsg"/⟩
    ⟨variable name = "P_NackVar" messageType = "Producer:nackMsg"/⟩
    ⟨variable name = "P_AckVar" messageType = "Producer:ackMsg"/⟩
    ⟨variable name = "P_OfferVar" messageType = "Producer:offerMsg"/⟩
    ⟨variable name = "S_AckVar" messageType = "Shipper:ackMsg"/⟩
    ⟨variable name = "S_NackVar" messageType = "Shipper:nackMsg"/⟩
    ⟨variable name = "S_RequestVar" messageType = "Shipper:requestMsg"/⟩
    ⟨variable name = "S_UnavailVar" messageType = "Shipper:unavailMsg"/⟩
    ⟨variable name = "S_OfferVar" messageType = "Shipper:offerMsg"/⟩
  ⟨/variables⟩
  ⟨sequence⟩
    ⟨receive operation = "request" variable = "U_RequestVar" partnerLink = "client" portType = "PandS_PT"⟩
    ⟨assign ⟩
      ⟨copy⟩
        ⟨from variable = "U_RequestVar" part = "art"⟩
        ⟨to variable = "P_RequestVar" part = "art"/⟩
      ⟨/copy⟩
    ⟨/assign⟩
    ⟨invoke operation = "request" inputVariable = "P_RequestVar" partnerLink = "PandS_Producer"
            portType = "Producer:P_PT"⟩
    ⟨pick⟩
      ⟨onMessage operation = "unavail" variable = "P_UnavailVar" partnerLink = "PandS_Producer"
                  portType = "Producer:PC_PT"⟩
        ⟨sequence ⟩
          ⟨invoke operation = "unavail" inputVariable = "P_UnavailVar" partnerLink = "client"
                  portType = "PandSC_PT"⟩
        ⟨/sequence⟩
      ⟨/onMessage⟩
      ⟨onMessage operation = "getP_Info" operation = "info" variable = "P_InfoVar"
                  partnerLink = "PandS_Producer" portType = "Producer:PC_PT"⟩
        ⟨sequence⟩
          ⟨assign ⟩
            ⟨copy⟩
              ⟨from variable = "P_InfoVar" part = "size"⟩
              ⟨to variable = "S_RequestVar" part = "size"/⟩
            ⟨/copy⟩
            ⟨copy⟩
              ⟨from variable = "U_RequestVar" part = "loc"⟩
              ⟨to variable = "S_RequestVar" part = "loc"/⟩
            ⟨/copy⟩
          ⟨/assign⟩
          ⟨invoke operation = "request" inputVariable = "S_RequestVar" partnerLink = "PandS_Shipper"
                  portType = "Shipper:S_PT"⟩
          ⟨pick⟩
            ⟨onMessage operation = "unavail" variable = "S_UnavailVar" partnerLink = "PandS_Shipper"
                        portType = "Shipper:ShipperC_PT" ⟩
              ⟨sequence⟩
                ⟨flow⟩
                  ⟨sequence⟩
                    ⟨invoke operation = "unavail" inputVariable = "U_Unavail" partnerLink = "client"
                            portType = "PandSC_PT"⟩
                    ⟨invoke operation = "nack" inputVariable = "P_NackVar" partnerLink = "PandS_Producer"
                            portType = "Producer:P_PT"⟩
                  ⟨/sequence⟩
                ⟨/flow⟩
              ⟨/sequence⟩
            ⟨/onMessage⟩
            ⟨onMessage operation = "offer" variable = "S_OfferVar" partnerLink = "PandS_Shipper"
                        portType = "Shipper:ShipperC_PT"⟩
              ⟨sequence⟩
                ⟨invoke operation = "ack" inputVariable = "P_AckVar" partnerLink = "PandS_Producer"
                        portType = "Producer:P_PT"⟩
```

Fig. 5. The executable WS-BPEL for the P&S process (pt.1).

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

```
................
                ⟨receive operation = "offer" variable = "P_OfferVar" partnerLink = "PandS_Producer"
                      portType = "Producer:PC_PT"⟩
                ⟨assign⟩
                  ⟨copy⟩
                    ⟨from expression = "bpws:getVariableData('P_OfferVar', 'cost') +
                                 bpws:getVariableData('S_OfferVar', 'cost')"⟩
                    ⟨to variable = "U_OfferVar" part = "cost"/⟩
                  ⟨/copy⟩
                  ⟨copy⟩
                    ⟨from expression = "bpws:getVariableData('P_OfferVar', 'delay') +
                                  bpws:getVariableData('S_OfferVar', 'delay')"⟩
                    ⟨to variable = "U_OfferVar" part = "delay"/⟩
                  ⟨/copy⟩
                ⟨/assign⟩
                ⟨invoke operation = "offer" inputVariable = "U_OfferVar" partnerLink = "client"
                      portType = "PandSC_PT"⟩
                ⟨pick⟩
                  ⟨onMessage operation = "ack" variable = "U_AckVar" partnerLink = "client"
                            portType = "PandS_PT"⟩
                    ⟨sequence⟩
                      ⟨flow⟩
                        ⟨sequence⟩
                          ⟨invoke operation = "ack" inputVariable = "S_AckVar"
                                partnerLink = "PandS_Shipper" portType = "Shipper:S_PT"⟩
                        ⟨/sequence⟩
                        ⟨sequence⟩
                          ⟨invoke operation = "ack" inputVariable = "P_AckVar"
                                partnerLink = "PandS_Producer" portType = "Producer:P_PT"⟩
                        ⟨/sequence⟩
                      ⟨/flow⟩
                    ⟨/sequence⟩
                  ⟨/onMessage⟩
                  ⟨onMessage operation = "nack" variable = "U_NackVar"partnerLink = "client"
                           portType = "PandS_PT"⟩
                    ⟨sequence
                      ⟨flow⟩
                        ⟨sequence⟩
                          ⟨invoke operation = "nack" inputVariable = "S_NackVar"
                                  partnerLink = "PandS_Shipper" portType = "Shipper:S_PT"⟩
                        ⟨/sequence⟩
                        ⟨sequence⟩
                          ⟨invoke operation = "nack" inputVariable = "P_NackVar"
                                  partnerLink = "PandS_Producer" portType = "Producer:P_PT"⟩
                        ⟨/sequence⟩
                      ⟨/flow⟩
                    ⟨/sequence⟩
                  ⟨/onMessage⟩
                ⟨/pick⟩
              ⟨/sequence⟩
            ⟨/onMessage⟩
          ⟨/pick⟩
        ⟨/sequence⟩
      ⟨/onMessage⟩
    ⟨/pick⟩
  ⟨/sequence⟩
⟨/process⟩
```

Fig. 6. The executable WS-BPEL for the P&S process (pt.2).

the internal action τ. Finally, a *labeling function* associates to each state the set of properties that hold in the state. These properties will be used to define the composition requirements.

Definition 5 *(State transition system).* A *state transition system* Σ defined over a set of propositions $\mathcal{P}rop$ 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;
- $\mathcal{S}^0 \subseteq \mathcal{S}$ 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;
- $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{P}rop}$ is the labeling function.

We require that $\mathcal{I} \cap \mathcal{O} = \emptyset$ and that $\tau \notin \mathcal{I} \cup \mathcal{O}$.

We now recall some standard definitions on state transition systems, see e.g. [56,7,32]. We say that an action $a \in (\mathcal{I} \cup \mathcal{O} \cup \{\tau\})$ is *applicable* on a state $s \in \mathcal{S}$, denoted with $\mathrm{Appl}(a, s)$, if there exists a state $s' \in \mathcal{S}$ s.t. $(s, a, s') \in \mathcal{R}$. A state of an STS is *final* if no action $a \in (\mathcal{I} \cup \mathcal{O} \cup \{\tau\})$ is applicable in s, i.e., if there is no transition leaving s. The execution of an STS is represented by its set of possible *runs*, i.e., of sequences of states $s_0, s_1, \ldots$ such that $s_0 \in \mathcal{S}^0$ and $(s_i, a, s_{i+1}) \in \mathcal{R}$ for some $a \in \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$. In general, such executions may be finite or infinite.[2] A run is said to be *completed* if it is finite, and if its last state is final. A state $s \in \mathcal{S}$ will be said *reachable* if and only if either $s_0 \in \mathcal{S}^0$, or there exists a state $s_0 \in \mathcal{S}^0$ and a sequence $s_0, s_1, \ldots, s_n$ such that $s_n = s$, and for all $i = 0, \ldots, n-1$ there is some $a \in (\mathcal{I} \cup \mathcal{O} \cup \{\tau\})$ such that $\langle s_i, a, s_{i+1} \rangle \in \mathcal{R}$. We will denote the set of reachable states of the STS Σ with $\mathit{Reachable}(\Sigma) \subseteq \mathcal{S}$.

Finally, we say that a state $s \in \mathcal{S}$ is *divergent* if there exists an infinite sequence of τ transitions starting from s, i.e., if there is an infinite sequence of states $s = s_1, s_2, s_3, \ldots$ such that $(s_i, \tau, s_{i+1}) \in \mathcal{R}$ for all $i \geqslant 1$. Infinite τ-sequences starting from divergent states correspond to behaviors where the service never interacts with the domain, and thus cannot be controlled or accessed in any way. For instance, in a WS-BPEL implementation of a service, this might originate from having an infinite while loop whose body never performs any communication. Having in mind that the main purpose of services is to interoperate with external entities, it is clear such a behavior is undesirable and corresponds to a bad design of the service. Divergent states may as well appear due the adoption of inappropriate abstractions when encoding services in terms of STSs. For instance, consider a service containing a while loop that repeatedly increments a variable v and exits as soon as v gets a fixed value. If the encoding of the service abstracts away that variable, the entry point of that loop is mapped into a divergent state. We assume both that services are designed correctly, and that, in case an abstraction step is needed to encode a service into an STS, such abstraction does not originate divergent states. As such, in the rest of this paper, we will assume that STSs will not have divergent states.

Properties can be combined using the standard propositional logical connectives, and we adopt a standard notion of satisfaction to identify whether a formula corresponding to a combination of properties is valid onto a given state.

Definition 6 *(Satisfaction of property formula).* Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be an STS defined over a set of propositions $\mathcal{P}rop$. Let ρ, ϕ, ψ be propositional formulas over $\mathcal{P}rop$. Then state $s \in \mathcal{S}$ satisfies ρ, denoted $s \models \rho$, if and only if one of the following holds:

- if $\rho \in \mathit{Prop}$, then $\rho \in \mathcal{L}(s)$;
- if $\rho = \neg\phi$, then $s \models \phi$ does not hold;
- if $\rho = \phi \vee \psi$, then $s \models \phi$ holds, or $s \models \psi$ holds;
- if $\rho = \phi \wedge \psi$, then $s \models \phi$ holds and $s \models \psi$ holds;
- if $\rho = \phi \Longrightarrow \psi$, then either $s \models \phi$ does not hold, or $s \models \psi$ holds.

3.2. *Translating* WS-BPEL *services as STSs*

We have implemented a translation that associates a STS to each component service, starting from its abstract WS-BPEL specification. The translation currently covers a significant subset of the WS-BPEL language, which we report in Table 2. More precisely, we support all WS-BPEL *basic* and *structured activities*, like invoke, reply, receive, sequence, switch, while, pick, onMessage, terminate, empty and flow (without links). Moreover we support restricted forms of assignments (specifically, we restrict the expressions that can appear in the from part of the copy statements) and of correlations.[3]

The translation works in two steps, which are both conceptually simple, and is discussed at length in [49,57]; we give here a compact, yet comprehensive summary of its workings. In a nutshell, the first step translates a service into an automa whose states are represented by means of variables whose range may not be determined; as such, this is in general an infinite-state automa. In the second step, all variables are associated to finite ranges, therefore obtaining a finite-state transition system. For both steps, we adopt a convenient, compact and fairly standard symbolic representation, where states are represented using a fixed set of variables, and actions and transitions are represented by means of schemata. Specifically, our representation has the following features:

(1) The states of the STS correspond to complete assignments to a set of typed variables, corresponding to the parts of the structured WS-BPEL variables; on top of these, a special variable `pc` (for "program counter") is used to represent the current execution step of the service.

[2] We remark that we do not rely on a bisimulation semantics, and we do not exploit any bisimilarity relation to define equivalent STSs based on runs. We will use runs exclusively to identify infinite and finite executions of an STS (which may or may not traverse some states more than once).

[3] The considered subset does not deal at the moment with WS-BPEL constructs like scopes, fault, event and compensation handlers, which are used to deal with run-time events into *concrete* WS-BPEL specifications. This choice keeps the translation simple, and is pragmatic. In fact, in our experience, we never found cases where some of the abovementioned constructs appears in an available abstract WS-BPEL specification. As such, the considered subset appears expressive enough for describing the way business processes are actually exposed using *abstract* WS-BPEL. Nevertheless, extensions are of course possible, as we discuss in Section 8.

(2) The input and output actions of an STS are represented in terms of schemata, composed of an action name and a (possibly empty) set of argument types. In general, an action schema corresponds to several possible action instantiations inside the STS, one for each combination of ground values associated to the arguments.
(3) It is possible to declare functions, taking as input some typed arguments and returning one typed argument.
(4) The transitions are also represented schematically. A transition schema corresponds to a set of transitions of the STS, and defines a set of preconditions, at most one input or output action, and a set of post-conditions. Pre-conditions identify source states by postulating over the value of the `pc` variable and, possibly, over values of other (non-`pc`) state variables. An action is described by an action name, and as many (non-`pc`) variables as specified in the argument list of the action declaration. Similar to pre-conditions, post-conditions define the target states by postulating over the values of the `pc` and, possibly, non-`pc` variables, which can be assigned with values, variables, or functional expressions.

This schematic representation is of great practical help for two main reasons. First, its schematic nature makes it very compact. Second, it allows a simple translation of the declaration of WS-BPEL variables, functions and input/output actions into corresponding declarations inside the schematic STS.

More in detail, in the first step of the translation, we convert the service into such a schematic representation, using both enumerative types and 'abstract' types to describe variables: enumerative types correspond to enumerative WS-BPEL types, while 'abstract' types, whose range is not yet defined at this stage, correspond to WS-BPEL types whose range is infinite.

In this step, the translation of the workflow body of the WS-BPEL specification takes place in a compositional way: for each activity, a translation pattern is defined, and it is enough to recursively analyze the structure of the process, and to translate each WS-BPEL activity in turn.

More specifically, a basic WS-BPEL activity (i.e. invoke, reply, receive, onMessage, assign, empty, terminate) is directly translated into a single loop-free STS featuring one initial state, one final state, and one or more transitions. Table 3 reports the translation schemata for the receive, empty (synchronous and asynchronous) invoke activities, and for assignments of variables, using a graphical notation to depict the schematic STS representation explained above. In the graphical notation, a transition schema is represented as an arc between two nodes. The nodes are labeled with the conditions over the `pc` variable, while pre- and post-conditions on non-`pc` variables appear as labels at the start and end of the transition. In particular, for the sake of easier understanding, we use curly braces to denote pre-conditions, and square brackets to denote post-conditions. Input and output actions label the arcs, and are identified by prepending them with a "?" and a "!" marks respectively. Transitions that feature neither input nor output actions are marked with the internal action τ. We remark here that the STS translation for the basic reply, onMessage, terminate activities correspond to those for invoke, receive, empty activities respectively: at the level of STSs, reply and invoke both map to output actions, receive and onMessage both map to input actions, and terminate and empty both map to τ actions. The differences between the respective pairs of activities reside exclusively at the syntactic WS-BPEL level and therefore have no impact at the level of composition of STSs. Of course, when translating the composed service back from STS into WS-BPEL, some care must be taken so to translate correctly e.g. an STS output either into an invoke or a reply, by recovering some information which was abstracted away during their conversion as STSs. Concretely, this is realized by storing some simple 'action correspondence' maps when stepping from WS-BPEL to STS, and recovering information from them when going back to WS-BPEL.

Based on the above translations of basic activities, a structured WS-BPEL activity (sequence, pick, switch, while, flow) is translated by first translating its sub-activities, and then linking the resulting STSs (namely their initial and final states) to obtain a single STS which corresponds to the structured activity, and which features a single initial state (the entry point to the activity) and a single final state (its exit point). Table 4 gives an account of such a translation mechanism, using again a graphical notation (and, for the flow activity, showing the case of two activities; a generic flow of n activities results into an STS allowing every possible interleaving combination of these n activities).

This first phase of the translation is concluded by a simple reduction phase, where transitions that do not perform any input/output, nor affect the values of variables, are eliminated, and their corresponding entry and exit states are unified.

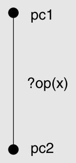
Then, the second step of the translation consists of the instantiation of the schematic representation obtained so far, to get to a finite STS model. In the representation we adopted, this boils down to identify finite ranges that replace the abstract types, and, concerning possibly existing interpreted functions, to describe their behavior on ground values.

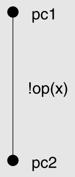
In this phase, it is crucial that all the relevant behaviors of the original service are nonetheless captured by the groundized service; as we will discuss in Section 8, it is easy to devise such an instantiation in a naive way, but this can lead to problems related to the large model size; on the contrary, looking for size-optimal instantiations is harder. In the following, we will work under the hypothesis that either the starting WS-BPEL specification is finite-state, or that such a suitable finitization is identified as a pre-processing step.

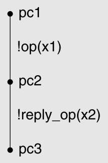
Once finite variable ranges and interpreted function behaviors are made explicit, our schematic representation can be very easily mapped into the formal model of the STS. First, interpreted functions are compiled away as constants, while uninterpreted functions are mapped into sets of variables corresponding to their applications on ground values, whose values are unknown but remain constant at each transition. Then, the set of states simply consists of all the combinatorial assignments of variables, while the transitions correspond to the instantiations of assignments and input/output actions with the concrete values of the variables and of the input/output arguments. Finally, the set $\mathcal{P}rop$ of properties over which the STS is defined consists simply of all the possible value assignments to each typed variable and ground function term, and the labeling function is the obvious one. It is important to remark, however, that while this groundization is required

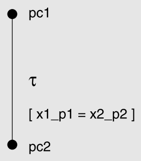
Table 3
Translating basic WS-BPEL activities to STSs.

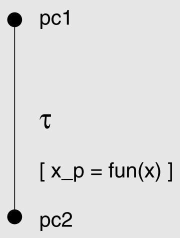
WS-BPEL activity	STS

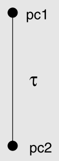
`receive`
`operation="op" variable="x"`
	

`invoke`
`operation="op"`
`inputVariable="x"`
	

`invoke`
`operation="op"`
`inputVariable="x1"`
`outputVariable="x2"`
	

`assign`
`copy from variable="x1"`
`part="p1"`
`to variable="x2" part="p2"`
	

`assign`
`copy from expression="fun(y)"`
`to variable="x" part="p"`
	

`empty`	

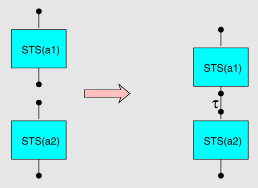
for our theoretical development, our implementation will largely operate over schematic STS representations, to leverage on their compactness.

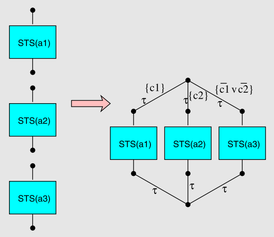
We illustrate the translation through the example of the Producer abstract WS-BPEL introduced in the previous section, breaking it into the two translation phases.

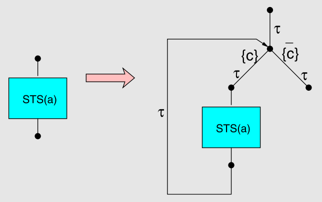
Example 7. Fig. 7 shows the STS for the abstract WS-BPEL process of the Producer, represented in the internal language that is used by the WS-BPEL to STS translator. The specification starts with a list of `TYPE`s (`Article`, `Size`, `Cost`, `Delay`) exploited in the STS, which are derived from the WSDL specification of the Producer. Then, the `STATE` of the STS is defined, using a set of typed variables, among which the `pc` "program counter" variable. Notice that the values of `pc` are closely related to the start points of activities. For example, `pc` has value `getRequest` when the process is waiting to receive

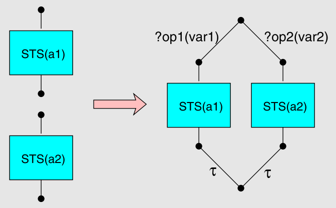
Table 4
Translating structured WS-BPEL activities into STSs.

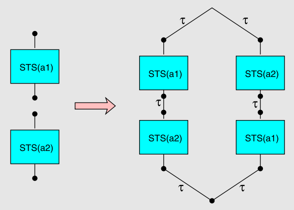
WS-BPEL activity	STS translation

`sequence`
`activity a1`
`activity a2`
	

`switch`
`case condition="c1"`
`activity a1`
`case condition="c2"`
`activity a2`
`otherwise`
`activity a3`
	 STS(a1)
	STS(a2)
	STS(a3)
	{c1}
	{c2}
	{c̄1 ∨ c̄2}
	τ

`while condition="c"`
`activity a`
	

`pick`
`onMessage`
`operation="op1"`
`variable="var1"`
`activity a1`
`onMessage`
`operation="op2"`
`variable="var2"`
`activity a2`
	

`flow`
`activity a1`
`activity a2`
	

a request for an item, and value `checkAvail` when it is ready to check whether the item is available, and these values correspond to the receive and switch statements at lines 15 and 20 of the WS-BPEL in Fig. 3, respectively. In fact, whenever WS-BPEL associates a name to an activity, our translation uses that name as the value attained by the program counter at the start of that activity; for instance, line 51 is associated to the activity named `prepareInfo` in the code. Otherwise, a unique `pc` value is created by the translator (e.g. in the case of the ⟨/pick⟩ activity at line 31). If an activity takes place immediately prior to termination, its name (if defined) is also used to describe the final value of the Program Counter, prepending a `DONE_` prefix (see for instance line 59 and line 60).

The other variables that define the state of the STS (e.g., `offerVar_delay`, `offerVar_cost`) correspond to the parts of the WS-BPEL message variables. In the `INIT`ial states, all the variables are undefined, except `pc` which is set to the initial WS-BPEL activity `main`.

The `INPUT` declarations represent schematically the input actions of the STS, modeling all the incoming requests to the process and the information they bring (i.e., `request` is used for the receiving of the product request, and has a single parameter of type `Article`). Similarly, the `OUTPUT` declarations model the outgoing messages (e.g., `unavail` is used

```
(1)  PROCESS Producer;
(2)  TYPE
(4)     Article;
(5)     Size;
(6)     Cost;
(7)     Delay;
(8)     Boolean: {True,False};
(9)  STATE
(11)    pc : { main, getRequest, setAvail, checkAvail, sendUnavail, seq1, prepareInfo, sendInfo,
(14)         pick1, seq2, prepareOffer, pick2, sendOffer, DONE_sendUnavail,
(15)         DONE_getNack, DONE_getAccepted, DONE_getRefused};
(18)    infoVar_size: Size;
(19)    offerVar_delay: Delay;
(20)    offerVar_cost: Cost;
(21)    reqVar_article: Article;
(22)    availVar : Boolean;
(23) INIT
(25)    pc := main;
(26)    infoVar_size := UNDEF;
(27)    offerVar_delay := UNDEF;
(28)    offerVar_cost := UNDEF;
(29)    reqVar_article := UNDEF;
(30)    availVar := UNDEF;
(32) INPUT
(34)    request (Article);
(35)    nack ();
(36)    ack ();
(38) OUTPUT
(40)    unavail ();
(41)    offer (Cost, Delay);
(42)    info (Size);
(43) TRANS
(45)    pc = main -[TAU]-> pc := getRequest;
(46)    pc = getRequest -[INPUT request(reqVar_article)]-> pc := setAvail;
(47)    pc = setAvail -[TAU] -> pc = checkAvail, availVar in {True,False};
(48)    pc = checkAvail   ,  availVar = False -[TAU]-> pc := sendUnavail;
(49)    pc = sendUnavail -[OUTPUT unavail()]-> pc := DONE_sendUnavail;
(50)    pc = checkAvail   ,  availVar = True -[TAU]-> pc := seq1;
(51)    pc = seq1 -[TAU]-> pc := prepareInfo;
(52)    pc = prepareInfo -[TAU]-> pc := sendInfo, infoVar_size in Values(Size);
(53)    pc = sendInfo -[OUTPUT info(infoVar_size)]-> pc := pick1;
(54)    pc = pick1 -[INPUT nack()]-> pc := DONE_getNack;
(55)    pc = pick1 -[INPUT ack()]-> pc := seq2;
(56)    pc = seq2 -[TAU]-> pc := prepareOffer;
(57)    pc = prepareOffer -[TAU]-> pc := sendOffer, offerVar_cost in Values(Cost), offerVar_delay in Values(Delay);
(58)    pc = sendOffer -[OUTPUT offer(offerVar_cost, offerVar_delay)]-> pc := pick2;
(59)    pc = pick2 -[INPUT ack()]-> pc := DONE_getAccepted;
(60)    pc = pick2 -[INPUT nack()]-> pc := DONE_getRefused;
```

Fig. 7. The STS for the Producer process.

when the article is not available in stock, and `offer` is used to bid the production of an item at a particular price and with a particular production time).

The schematic presentation of the transitions of the STS is given in terms of a set of `TRANS` statements. Each `TRANS` statements essentially corresponds to an arc in the representation of Tables 3 and 4; as discussed above, it defines its applicability conditions on a set of source states, its firing action, and the corresponding destination states. For instance, "`pc = main -[TAU]-> pc = getRequest`" (line 45) states that an action τ can be executed in any state where the property `pc = main` holds, leading to a state where `pc = getRequest` holds (and all other properties have retained their former value). Transitions such as the one above are directly associated to WS-BPEL activities, e.g. in this case the starting `main` activity; other transitions are inserted by the translation mechanism, e.g. to link two activities that must be performed in sequence (see for instance "`pc = seq2 -[TAU]-> pc = prepareOffer`", line 56). WS-BPEL assignments are mapped into transitions whose effects also affect variables different from the program counter; for instance, this is the case for the assignment at line 52, corresponding to line 27 in the WS-BPEL code. Notice that at this stage, opaque WS-BPEL assignments correspond to giving a variable any possible value in the range defined by its type, denoted by the "`Values`" notation for abstract types, and with sets of concrete values for finite types. Fig. 8 reports the graphical representation of the STS for the Producer, corresponding to its schematic textual representation of Fig. 7.

Once the types (`Article`, `Size`, `Cost`, and `Delay`) are instantiated with finite ranges, as required by our finiteness assumption, we easily obtain a concrete finite-state STS. States $\mathcal{S}$ correspond to assignments to the variables, inputs $\mathcal{I}$ and outputs $\mathcal{O}$ to the instantiations of the messages defined in the `INPUT` and `OUTPUT` sections. Each `TRANS` clause of Fig. 7 corresponds to the different elements in the transition relation $\mathcal{R}$: e.g. "`pc = checkAvail, availVar = False -[TAU]-> pc = sendUnavail`" generates different elements of $\mathcal{R}$, depending on the values of variables `reqVar_article`, `offerVar_cost` and so on. The properties associated to the STS are expressions of the form `<vari-`

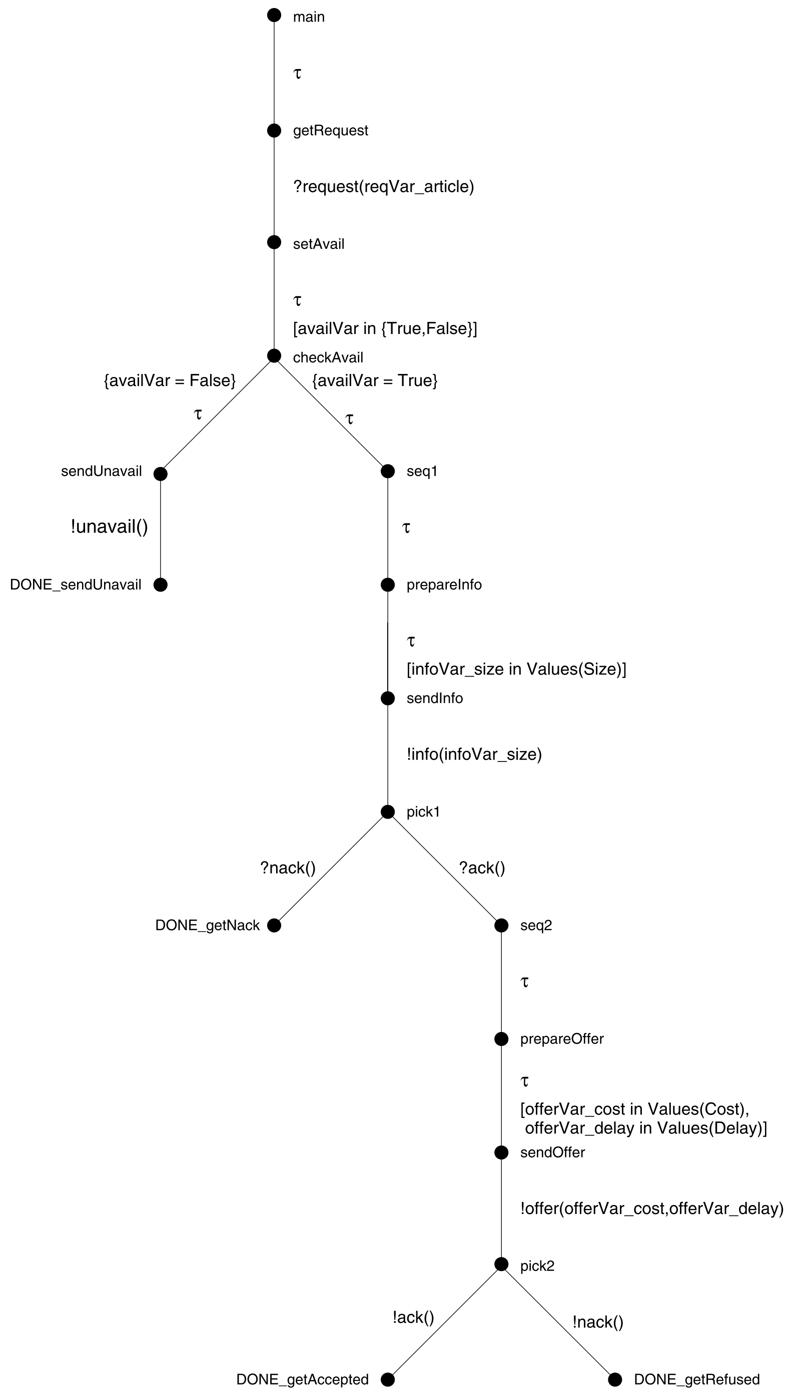


Fig. 8. The graphical representation of the STS for the Producer process.

`able> = <value>`, where `<variable>` is one of the variables in the `STATE` declaration, and `<value>` one of its possible values. The labeling function is the obvious one. As remarked, however, our implementation will never explicitly produce a groundized STS represented with a human-readable language such as the one described so far. Rather, as we will discuss more extensively in Section 5, it will leave to low-level symbolic mechanisms based on Binary Decision Diagrams [26] the enactment and representation of the groundization.

A similar mapping from WS-BPEL process to STS holds also for the Shipper and the User processes.

3.3. Translating STSs as WS-BPEL *services*

Our composition approach also requires the ability to convert back an STS into WS-BPEL, since it extracts the desired orchestration as an STS. This process is actually easier than that of obtaining STSs from WS-BPEL specifications, due to a key observation on the nondeterministic nature of services.

Namely, in general, STSs such as those resulting from WS-BPEL conversion may feature nondeterminism of two different natures, often referred to as "external" and "internal" nondeterminism. The first, "external" nondeterminism, is related to the fact that the domain interacting with an STS may behave in ways which the STS may not predict a priori. For instance, in our example, an offer given by the Producer may be answered positively or not, and this is outside the control of the Producer. In these cases, an STS must be able to handle different requests, responding adequately to the possible behaviors of the partners that interact with it. This kind of nondeterminism occurs for instance in state `pc = pick1`, where messages `ack` and `nack` can be received. The second, "internal" nondeterminism, refers to the fact that an STS may evolve in different ways according to internal decisions that are not explicit in the STS. For instance, the fact that a variable can be assigned one of several values (e.g. `availVar` in state `pc = setAvail`) models the presence of a decision on the variable value in an "abstract" way, without describing the logics behind such decision within the STS. Of course, this can reflect in different STS behaviors driven on the basis of the value of the `availVar` variable. Similar considerations hold for states where different output transitions can be performed. Example 7 shows that both forms of nondeterminism are necessary to model the published services which we intend to compose (in our case, Producer, Shipper and User).

However, it is important to remark that, for what concerns the executable orchestration services which we aim to synthesize, we shall focus on STS that only feature the "external" nondeterminism which is needed to respond correctly to various messages coming from the component services. Indeed, we need to avoid generating composite services featuring "internal" nondeterminism, since, in order for a service to be executable on an engine, it must not contain any "abstract" choices which would make its run-time behavior not uniquely determined.

We will capture this distinction by calling an STS *deterministic* if it does not contain "internal" nondeterminism (while it may contain "external" nondeterminism). Basically, a deterministic STS has a single initial state and admits, for each state, either a unique τ-transition, or a unique output transition, or a nonempty set of input transitions — as formally spelled out in the following definition:

Definition 8 *(Deterministic STS).* An STS $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ is deterministic if and only if:

(1) $|\mathcal{S}^0| = 1$;
(2) for each state $s \in \mathcal{S}$, if $(s, \tau, s') \in \mathcal{R}$, then no other transition from s belongs to $\mathcal{R}$;
(3) for each state $s \in \mathcal{S}$, if $(s, a, s') \in \mathcal{R}$, with $a \in \mathcal{O}$, then no other transition from s belongs to $\mathcal{R}$.

The fact that, in our framework, we aim to synthesize deterministic automata implies a significant simplification for what concerns the translation of such synthesized STSs to (executable) WS-BPEL. In fact, due to the deterministic and loop-free nature of the orchestration STSs we generate, such a translation turns out to be conceptually simple. The extraction of input/output WS-BPEL operations is immediate, as well as that of a set of WS-BPEL variables that represent the STS variables (apart from the `pc` and the variables associated to uninterpreted functions). For what concerns the reconstruction of the WS-BPEL workflow starting from the set of STS transitions, we first perform a simple optimization step that eliminates all τ-transitions which only affect the program counter. Then, it is enough to perform a recursive top-down visit of the transition tree starting from the initial state, applying transformation patterns that essentially invert those in Tables 3 and 4. For instance, output transitions are converted into invoke or reply (depending on whether their corresponding operations in the WS-BPEL components are asynchronous or synchronous). Assignments are converted into assign activities. An input transition which originates alone from a state is converted into a receive activity; tuples of input transitions sharing their starting state correspond to a pick and a tuple of onMessage sub-activities, and so on. Transitions that feature preconditions over non-`pc` variables originate switch activities. The sequence activity is inserted at the start, and inside any branch of a structured activity (a case within a switch, or an onMessage within a pick).

4. Modeling the composition problem

In this section, based on our interpretation of services in terms of STSs, we formalize the problem of composing Web services, and we describe its solution by means of planning techniques. Our modeling and solution rely on the composition framework represented in Fig. 9. Its inputs consist of a set of component services $W_1, \ldots, W_n$ and a reachability composition requirement ρ, and its output is a novel process W that can be deployed and run to coordinate $W_1, \ldots, W_n$ in order to achieve ρ. Internally, the framework goes through two crucial steps: constructing a domain $\Sigma_\|$, which is an STS that represents all the possible behaviors of the component services that need to be controlled, and identifying a suitable "controller" Σ_c for $\Sigma_\|$, that is an STS which interacts with $\Sigma_\|$ to make the requirement ρ satisfied, and which as discussed in the previous chapter, is deterministic and loop-free. Fig. 9 also highlights that, to break down the complexity of this second step, a pre-processing phase transforms $\Sigma_\|$ into an intermediate STS $\textsc{Bel}(\Sigma_\|)$, which intuitively represents the behaviors

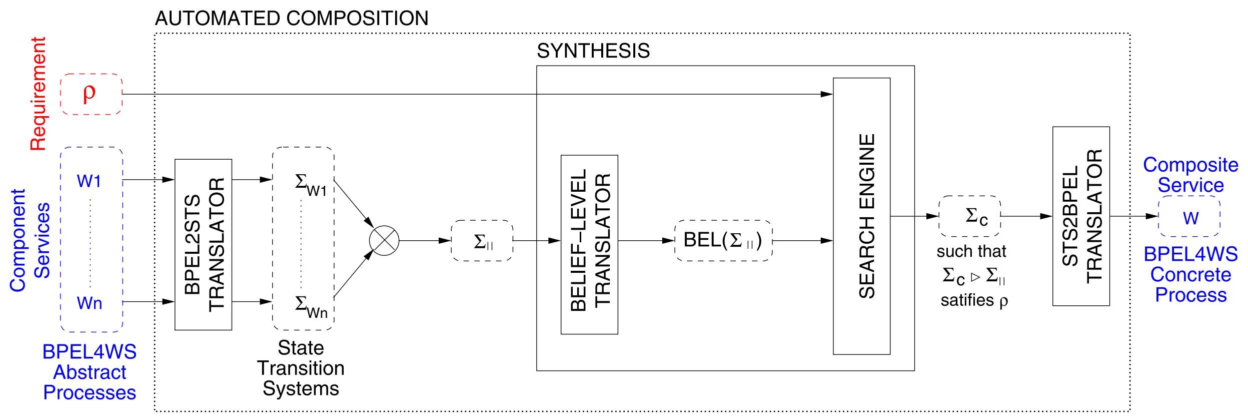


Fig. 9. The approach.

of the component services "as recognizable by an external observer", and which can be analyzed more easily than $\Sigma_\|$ to extract Σ_c.

The next two subsections introduce the key formal elements for performing the steps discussed above. First, in Section 4.1, we discuss the notion of domain in terms of a *parallel product* $\Sigma_\|$, showing the way it models the independently evolving components $\Sigma_1, \ldots, \Sigma_n$. Based on this, we can define what it means to control $\Sigma_\|$ by an STS Σ_c, and we represent the behaviors of the controlled system as an STS $\Sigma_c \triangleright \Sigma_\|$.

Then, in Section 4.2, we define the notions of composition problem and solution. To do so, we first formalize composition requirements, and the conditions to satisfy them. This allows stating the conditions under which a controller satisfies some requirements, and consequently, to spell the composition problem as that of finding a controller Σ_c such that $\Sigma_c \triangleright \Sigma_\|$ satisfies a given composition requirement ρ.

4.1. Domain and controller

The automated composition problem has two inputs (see Fig. 9): the formal composition requirement ρ and the set of component services $W_1, \ldots, W_n$, associated to STSs $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. The components $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ evolve independently; together, they build up an environment whose behaviors need to be driven, and represent, in planning terms, the domain which we aim to control. Such domain $\Sigma_\|$ is obtained as the first step of the composition, by combining $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ by means of a *parallel product* operation which essentially follows the notion of *asynchronous product* of [45].

Definition 9 *(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 $(\mathcal{I}_1 \cup \mathcal{O}_1) \cap (\mathcal{I}_2 \cup \mathcal{O}_2) = \emptyset$, and $\mathcal{L}_1 \cap \mathcal{L}_2 = \emptyset$. The parallel product $\Sigma_1 \| \Sigma_2$ 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 Fig. 9 is formally defined as $\Sigma_\| = \Sigma_{W_1} \| \cdots \| \Sigma_{W_n}$.

We remark that our definition of parallel product requires that the inputs/outputs of Σ_1 and those of Σ_2 are disjoint. This means that the component services may not communicate directly with each other. While an extension to the case where such component services may directly communicate is possible (and rather simple), we will not consider it, since in our scenario, we intend to compose independent existing services, which we assume not to be aware of each other. Also notice that, in order to obtain a consistent labeling of $\Sigma_\|$, Σ_1 and Σ_2 must be defined over disjoint proposition sets. This means, in essence, that the corresponding services must refer to different internal variables. This is actually the case, since service variables have a local scope, and even when name clashes take place, homonym variables must be made not ambiguous within the models (by applying appropriate renamings).

The automated composition problem consists in generating a STS Σ_c that controls $\Sigma_\|$ so that its executions satisfy the requirement ρ (according to a formal notion of requirement satisfaction).

In particular, since we aim at generating controllers which are executable, we can safely restrict to consider them as being internally deterministic, according to Definition 8.

Moreover, we require that the inputs of a controller coincide with the outputs of its corresponding domain, and vice versa: situations where some I/O only appears on either the controller or the domain are useless, and may cause deadlock situations.

Definition 10 *(Controller).* A controller for $\Sigma_\| = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L}\rangle$ is a deterministic STS $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_\emptyset\rangle$, where $\forall s \in \mathcal{S}_c : \mathcal{L}_\emptyset(s) = \emptyset$.

The behaviors of a STS Σ when driven by a controller Σ_c can be represented as an STS, which we call the *controlled system*, and whose definition, essentially, extends the standard notion of *synchronous product* of STSs [45] with an appropriate handling of τ transitions:

Definition 11 *(Controlled system).* Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L}\rangle$ be a state transition system, and let $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_\emptyset\rangle$ be a controller for Σ. The STS $\Sigma_c \triangleright \Sigma$, describing the behaviors of system Σ when controlled by Σ_c, is defined as:

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

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, 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 labeled 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.

In general, an STS Σ_c may not be adequate to control a system Σ. In particular, deadlocks may occur in case Σ_c performs an output at a point in time where Σ is not able to accept it, or vice versa. Of course, we want to only consider controllers that prevent these situations — guaranteeing that, whenever Σ_c performs an output transition, then Σ is able to accept it, and vice versa. Moreover, for sake of generality, we need to do so in a way which is independent from low-level, engine-dependent implementation details, and in particular from the way message queuing mechanisms are realized. To achieve this, we follow a careful, conservative approach that guarantees that deadlocks are avoided regardless of what specific message queuing mechanisms are put in place by the run-time engine.

We do so by defining a deadlock–freedom condition that assumes that messages are associated to a buffer, but does not rely on the existence of any specific message queuing/buffering mechanism. In particular, we assume that s can accept a message a if there is some successor $s' \in \mathcal{S}$ of s, reachable from s through a chain of τ transitions, such that s' can perform an input transition labeled 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.[4]

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) the union of the τ-closures on all the states of a set $S \subseteq \mathcal{S}$ (remember that, due to the absence of divergent states, chains of τ transitions are finite).

Definition 12 *(τ-closure).* Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L}\rangle$ be a STS, and $s \in \mathcal{S}$. Then τ-closure$(s) = \{t\colon\ \exists n \in \mathbb{N} : \exists s^0, s^1, \ldots, s^n : s = s^0, t = s^n, \forall i \in [0, n-1] : \langle s^i, \tau, s^{i+1}\rangle \in \mathcal{R}\}$. Moreover if $S \subseteq \mathcal{S}$, then τ-closure$(S) = \bigcup_{s \in S} \tau$-closure$(s)$.

Definition 13 *(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 *deadlock free* for Σ if all states $(s_c, s) \in \mathit{Reachable}(\Sigma_c \triangleright \Sigma)$ satisfy the following conditions:

(1) 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 $\langle s'_c, a, s''_c\rangle \in \mathcal{R}_c$ for some $s''_c \in \mathcal{S}_c$; and
(2) 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 $s'' \in \mathcal{S}$.

We remark that it is also possible to adopt a different formulation that maintains requirement (1), and replaces (2) with a condition which closely reflects the idea that all τ-successors of states in $\mathcal{R}$ must be considered:

(2′) *if* $\langle s_c, a, s'_c\rangle \in \mathcal{R}_c$ with $a \in \mathcal{I}$ then $\forall s' \in \tau$-closure(s) there is a $s'' \in \mathcal{R}$ such that $\langle s', a, s''\rangle \in \mathcal{R}$.

[4] 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''.

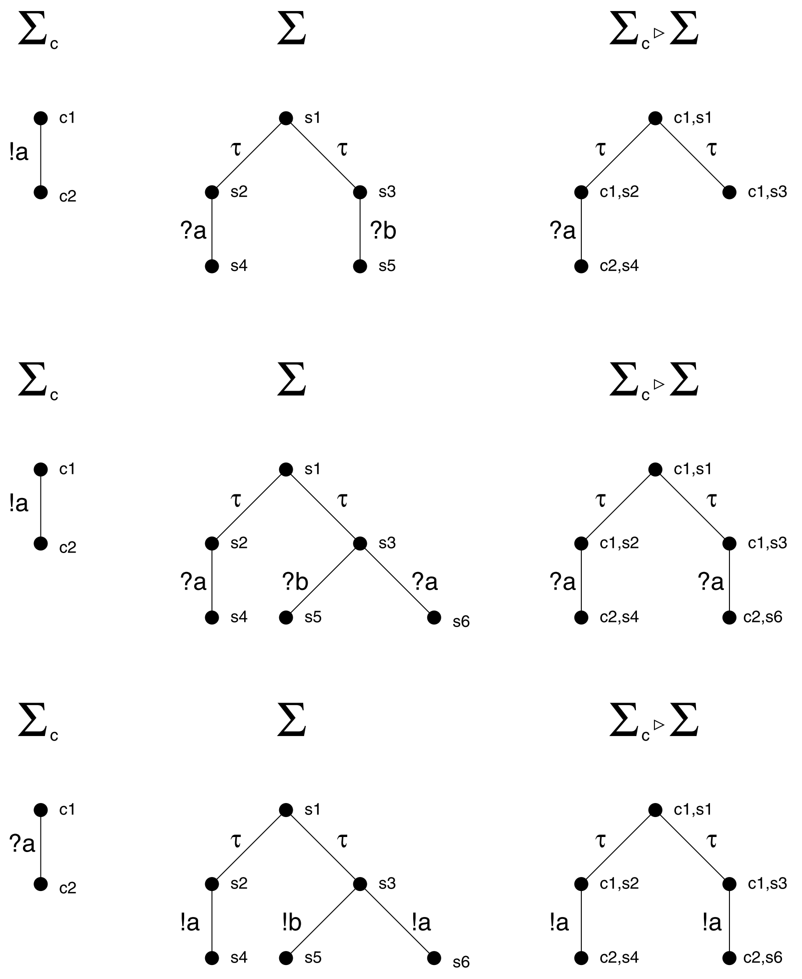


Fig. 10. Some examples of deadlock-prone and deadlock-free controllers.

The two formulations are equivalent due to the fact that, in Definition 13, (1) and (2) are both applied to all states $(s_c, s) \in$ *Reachable*$(\Sigma_c \rhd \Sigma)$.

Independently from its formulation, our deadlock–freedom definition corresponds to the minimal requirement that allows for asynchronous execution of WS-BPEL services, and as such is guaranteed to be supported by any Web-service execution engine, regardless of their specific implementation of queuing and buffering mechanisms.

Example 14. The top section of Fig. 10 shows an example of a controller Σ_c which is not deadlock-free for an STS Σ. This is due to the fact that Σ may nondeterministically evolve, from its initial state $s1$, into one of $s2$ or $s3$, and only in the former state it can handle the output produced by the controller Σ_c. That is, in the state $(c1, s3)$ of the controlled system $\Sigma_c \rhd \Sigma$, the requirement (1) of Definition 13 is violated.

Vice versa, as shown in the middle section of the figure, if the output of the controller can also be handled in the state $s3$, then no deadlock may occur — that is, Σ_c is deadlock-free for Σ.

The bottom part of the figure stresses the fact that inputs and outputs do not play symmetric roles in the definition of deadlock–freedom. Namely, in this case, we consider a variation of the STSs Σ_c and Σ of the middle section, where inputs are turned into outputs and vice versa. The resulting controlled system is just the same of that in the middle section; however, in this case, the controller is *not* deadlock-free, since in the situation corresponding to the state $(c1, s3)$ of the controlled system, it cannot handle the output b.

Notice that our definitions of controller and deadlock–freedom rule out the cases where, in a controlled system, both input and output actions can be executed on the same state, or on states connected by just a sequence of τ-transitions. This is because, at each step of execution, the controller, which we recall is a *deterministic* STS, may uniquely perform, as its next

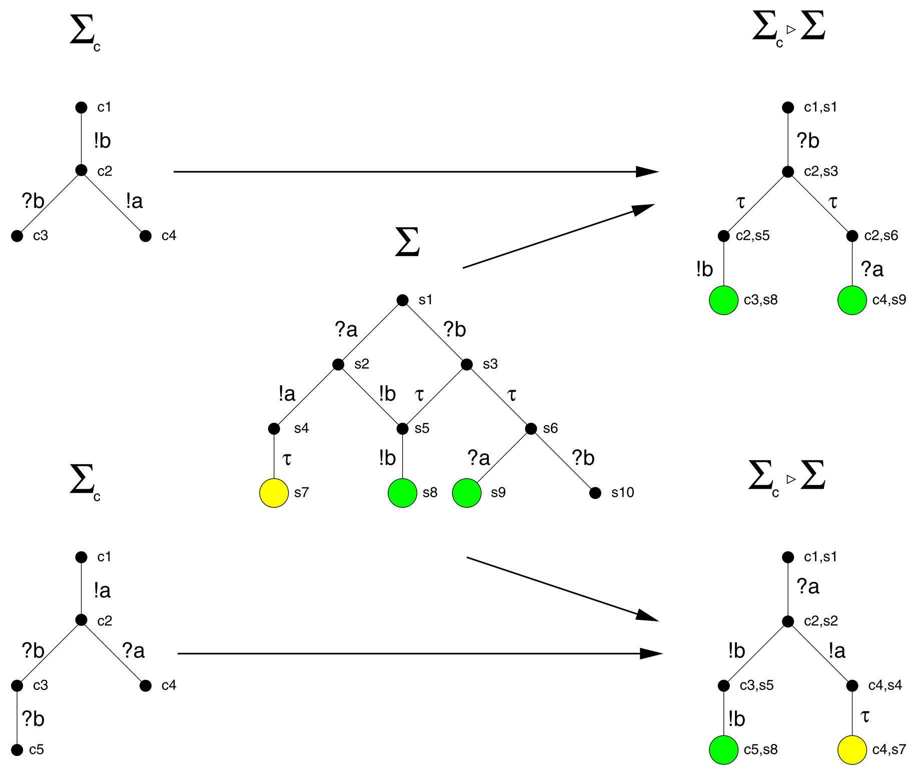


Fig. 11. Controllers and input/output.

action, either some input or a single output. Indeed, it is highly desirable to avoid situations where both input and outputs can be executed on the same state (or on states connected just by a sequence of τ-transitions): in those cases, the relative speeds of the I/Os would decide whether the controller gets deadlocked or not. This is shown by the following example.

Example 15. Consider Fig. 11; we intend to control Σ in such a way that some dark shaded states (namely, $s8$ or $s9$) are reached.

Driving Σ to always get to $s8$ or $s9$ is not possible without incurring into the possibility of a deadlock. Indeed, the controller STS Σ_c in the upper part of the figure (left) is designed in order to attempt driving Σ to either $s8$ or $s9$, by allowing either an input or an output while in state $c2$. However, as a result of such behavior, the corresponding controlled system $\Sigma_c \triangleright \Sigma$ is nondeterministic, and critical runs may occur in states $(c2, s5)$ and $(c2, s6)$, thus breaking the deadlock-freedom requirements in Definition 13.

Let us now relax our requirement, and also admit reaching the bright shaded state $s7$. The deterministic controller in the bottom part of the figure satisfies such a requirement. The associated controlled system is deadlock-free, and as one can see, at each step of its execution, it is fully determined whether (apart from internal evolutions) the next step will be an input or an output operation.

4.2. Requirements and problem statement

To define the notion of composition problem, we first discuss which kind of properties can be required for the orchestration of services, by means of an example within the context of our reference scenario.

Example 16. A reasonable requirement for the P&S is to sell the product at destination as requested by the customer. This means to conclude successfully a transaction started by a customer request. In terms of the interaction flows specified by the three available abstract WS-BPEL, this means to reach the situation where the User, the Producer, and the Shipper receive a final confirmation. This can be put in correspondence with WS-BPEL activities; for instance, concerning the User, this means it should get to execute line 36 in Fig. 4.

However, this requirement may be not always satisfiable by the composed service, since it depends on decisions taken by third parties that are out of its control: the Producer and the Shipper may refuse to provide the service for the requested product and location (e.g., since the product is not available or the location is out of the area of service of the shipper), and the User may refuse the offer by the P&S (e.g., since it is too expensive). If this happens, we require that the P&S should step back from both orders, since we do not want the P&S to buy something that cannot be delivered, as well as we do not want it to spend money for delivering an item that we cannot buy. Again, in terms of abstract WS-BPEL, this can be put in correspondence with WS-BPEL activities; for instance, concerning the User, this means it should terminate execution either at line 32 or at line 21 in Fig. 4.

Requirements like the one above can be formulated in terms of reachability over sets of states of the component services. In particular, we can structure our requirements as the conjunction of requirements over the flow of data amongst the components, and requirements over the control logics.

In the control part, we specify that, in case all partners are available, they should all terminate in a “successful” state, i.e. a state where the final agreement to buy or sell has been achieved with the P&S. Otherwise, each partner must either remain inactive, or terminate in a “failure” state where (a) the partner is aware that no agreement to buy/sell is possible, and (b) the partner no commitment to buy or sell.

In order to more compactly define this part of the requirements, we semantically annotate WS-BPEL activities using two annotations, *Fail* and *Succ*. These identify the final failure and success states for each of the partners, and are interpreted as the sets of program counter values associated to failing and successful terminations respectively. For instance, $Producer.Succ := (Producer.pc := DONE\_getAccepted)$, and $Producer.Fail := (Producer.pc\ \ IN\ \ \{DONE\_getRefused, DONE\_Nack\})$. Similarly, we introduce an *Init* annotation that indicates, for each component, that the program counter for that component has its initial value. This said, the control flow part ρ_c of the requirement goes as follows:

$$\begin{aligned}
&\big((Prod.AvailVar \wedge Ship.AvailVar \wedge User.AcceptsVar) \rightarrow \\
&(Prod.Succ \wedge Ship.Succ \wedge User.Succ)\big) \\
&\qquad \wedge \\
&\big(\neg(Prod.AvailVar \wedge Ship.AvailVar \wedge User.AcceptsVar) \rightarrow \\
&\big((Prod.Fail \vee Prod.Init) \wedge (Ship.Fail \vee Ship.Init) \wedge (User.Fail \vee User.Init)\big)\big)
\end{aligned}$$

The data flow part of the requirements identifies the data dependencies amongst the partners. More specifically, we expect that the Producer and Shipper receive (from P&S) the request data sent by the User, that the Producer is the one to provide the size information to the Shipper, and that costs and times are coherently computed and communicated to the User by P&S. Of course there may be cases where some data are not actually produced nor circulated, e.g., if the Producer is unavailable to provide the item, he will not communicate any location or article data to the other partners. This is captured by stating that “only in case the a component responsible for generating some data actually provides it, then it has to be communicated to the partner” — i.e., by a logical implication. The data section of the requirement, ρ_d, is thus as follows, where $DEF(d)$ is a condition that holds if d has a value different from a default ‘indefiniteness‘ value:

$$\begin{aligned}
&DEF(Producer.reqVar\_loc) \rightarrow Producer.reqVar\_loc = User.reqVar\_loc \wedge \\
&DEF(Producer.reqVar\_art) \rightarrow Producer.reqVar\_art = User.reqVar\_art \wedge \\
&DEF(Shipper.reqVar\_size) \rightarrow Shipper.reqVar\_size = Producer.infoVar\_size \wedge \\
&DEF(Shipper.reqVar\_loc) \rightarrow Shippher.reqVar\_loc = User.reqVar\_loc \wedge \\
&DEF(Shipper.reqVar\_art) \rightarrow Shippher.reqVar\_art = User.reqVar\_art \wedge \\
&DEF(User.offerVar\_delay) \rightarrow \\
&User.offerVar\_delay = Shipper.offerVar\_delay + Producer.offerVar\_delay \wedge \\
&DEF(User.offerVar\_cost) \rightarrow \\
&User.offerVar\_cost = Shipper.offerVar\_cost + Producer.offerVar\_cost \wedge
\end{aligned}$$

Intuitively, a controller is a *solution* for the requirement $\rho = \rho_c \wedge \rho_d$ iff it guarantees that ρ is achieved. We can formally express this by requiring that every execution of the controlled system $\Sigma_c \triangleright \Sigma_\|$ ends up in a state where ρ holds. This is immediate to do, since the requirement ρ in fact refers to values of the program counter, and values of the variables, possibly linked by means of functional expressions. As such, taking into account the finiteness of types, ρ can be easily compiled, reasoning by cases in the case of uninterpreted functions, into a formula over the propositions associated to $\Sigma_\|$.

Definition 17 *(Satisfiability).* We say that an STS Σ satisfies a propositional formula p, denoted with $\Sigma \models p$, if and only if

- there exists no infinite run of Σ;
- every final state of Σ satisfies p according to Definition 6.

Definition 18 *(Solution controller).* A controller Σ_c is a solution for goal ρ iff $\Sigma_c \triangleright \Sigma_\| \models \rho$.

We can now formally characterize a (Web service) composition problem as follows:

Definition 19 *(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 deadlock-free controller Σ_c such that $\Sigma_c \triangleright (\Sigma_1 \| \cdots \| \Sigma_n) \models \rho$.

We stress once more that, due to our definition of satisfiability, we do not admit, as solutions to our problem, controllers that may lead to infinite runs, entering into interaction loops with no guarantee to terminate. This kind of solution, which is strictly connected to the fact that requirements are formulated in terms of reachability conditions, is fully general when composing services which are guaranteed to terminate, as we assume to have as a starting point. Of course, several extensions are possible and meaningful, relaxing our assumption over the finite termination of components, considering composition requirements of a different nature, and correspondingly relaxing the definition of satisfiability. However, in order not to dilute the presentation, and to provide a self-contained and detailed discussion of the approach, we will stick to the above definitions, and restrict to (loop-free) controllers which do not imply infinite runs. We stress that, at the same time, this does not imply that we restrict to composing loop-free services. Indeed, for scenarios where (some) components feature loops, and where the composition requirement can be reached after they traverse such loops a finite number of times, our approach is capable to build an appropriate controller; specifically, the controller will handle each loop traversal into a different portion of its loop-free structure.

5. The synthesis algorithms

The statement of composition problem given by Definition 19 can be directly exploited to verify whether a given controller Σ_c can be used to satisfy a requirement ρ, e.g. by applying model-checking techniques. However, for the purposes of automatically generating a controller that solves the problem, we need to conveniently rephrase the problem in a way that makes explicit how such a controller can be identified by constructing and visiting a (prefix of a) search space. This will allow us to tackle the problem by first realizing the construction of such search space, and then by making use of effective search techniques taken from the planning area. In the following, we first discuss such reformulation, and then provide the algorithms for solving the reformulated problem.

5.1. Belief-level search reformulation

We start by observing that $\Sigma_\| = \Sigma_1 \| \cdots \| \Sigma_n$ is a (nondeterministic) STS 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 $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. This uncertainty has two different sources. The first is the presence of nondeterministic transitions in a situation where direct observation of the domain state is impossible; this is something also considered by several approaches to planning. The second source of uncertainty is due to the fact that we are modeling an asynchronous framework, where entities may evolve internally through τ-transitions. While asynchronous behaviors are well known and studied in process algebra, and have been consequently considered in different approaches to program synthesis, this aspect has never been considered so far in planning. All classical approaches to planning rely over the assumption that the domain of discourse has a synchronous nature, i.e. actions take place, possibly in concurrence, on the basis of a unique discrete timeline. This cannot model our situation, where independent actors evolve with unrelated and uncontrollable speed. Neither this can be modeled using recent planning approaches and languages that consider actions whose effects are continuous in time, since the triggering of such actions is still synchronous.

In this setting, the incomplete knowledge on the runtime state of the environment under control must be taken into account by grouping together the executions which may not be perceived as distinct by an external observer of the environment. In particular, at each step of the execution, we need to consider that a set of different states may be 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*. While several approaches in planning rely on this kind of representation to model finite-state uncertainty, see e.g. [18,81,93,24], we remark again that their use of beliefs is different, due to the absence of asynchronicity in their models.

Recalling Definition 12, it is easy to see that the initial belief for the execution is the τ-closure of the initial states $\mathcal{S}^0$ of $\Sigma_\|$, since it must consider all the states that each component may have reached, since their activation, only by means of internal transitions. This belief is updated whenever $\Sigma_\|$ 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 applied, then the new belief $B' = \mathit{Evolve}(B, a)$ is defined as the set of states reachable by first applying a, and then considering every (possibly empty) sequence of internal transitions on the resulting states. We remark that a has to be applicable over the belief; however, such notion of applicability cannot simply be defined by requiring the applicability of a over all states of the belief, due to two reasons. First of all, a state in B may evolve within B due to τ-transitions, and during such internal evolution (which we recall is finite, due to the absence of divergent states) it is perfectly admissible that transient states are traversed where the action a is not handled. Moreover, since output actions are executed just on some states, depending on the choice performed by the STS under exam, we shall not require that they are executable on every state of B, but just on some state of B.

Definition 20 *(Belief applicability and evolution).* Let $B \subseteq \mathcal{S}$ be a belief of some state transition system Σ. We say that an action $a \in \mathcal{I} \cup \mathcal{O}$ is applicable on a belief B, denoted $\mathrm{Appl}(B, a)$, iff:

- $a \in \mathcal{O}$, and $\exists s \in B : \mathrm{Appl}(s, a)$, or
- $a \in \mathcal{I}$, and $\forall s \in B : \exists s' \in \tau\text{-closure}(s) : \mathrm{Appl}(s', a)$.

The evolution of B under action a is the belief $B' = \mathit{Evolve}(B, a)$, where

- if $\mathrm{Appl}(B, a)$, then $\mathit{Evolve}(B, a) = \tau\text{-closure}(\{s'. \exists s \in B, a \in \mathcal{I} \cup \mathcal{O}\colon \langle s, a, s'\rangle \in \mathcal{R}\})$;
- otherwise, $\mathit{Evolve}(B, a) = \emptyset$.

By starting from the τ-closure of $\mathcal{S}^0$, and iteratively evolving the belief via every applicable action, we obtain the set of beliefs $\mathit{Reachable}_{\mathcal{B}}(\Sigma)$ reachable from $\mathcal{S}^0$:

- $\tau\text{-closure}(\mathcal{S}^0) \in \mathit{Reachable}_{\mathcal{B}}(\Sigma)$;
- if $B \in \mathit{Reachable}_{\mathcal{B}}(\Sigma)$ and $\exists a \in \mathcal{I} \cup \mathcal{O} : \emptyset \neq B' = \mathit{Evolve}(B, a)$, then $B' \in \mathit{Reachable}_{\mathcal{B}}(\Sigma)$.

To evaluate whether a system satisfies a given reachability goal, we must not consider "transient" states of beliefs, i.e. those that can be traversed while the components are performing their internal transitions. This is because of two reasons: first, we may not understand whether the system is in one of those states (since this depends on unobservable events), and second, we cannot control the system so to execute (or prevent) some internal transitions. Rather, we need to consider exclusively those "stable" states of beliefs that are reached after every possible internal evolution has been performed, and prior to any further execution of I/Os. This is what we call the *τ-frontier* of B:

Definition 21 *(τ-frontier).* Let $B \subseteq \mathcal{S}$ be a belief; we call the τ-frontier of B the set $\tau\text{-frontier}(B) = \{s \in B\colon \exists s' \in \mathcal{S}, a \in \mathcal{I} \cup \mathcal{O} : (s, a, s') \in \mathcal{R} \vee \forall s' \in \mathcal{S} : (s, \tau, s') \notin \mathcal{R}\}$.

Since we deal with reachability goals, we can state that a belief satisfies a propositional property p by just considering its τ-frontier: it satisfies p iff every state in its τ-frontier does.

Definition 22 *(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 and only if for any state $s \in \tau\text{-frontier}(B)$, $p \in \mathcal{L}(s)$.

We can now define a "belief-level" STS for $\Sigma_\|$, whose states are the beliefs that may be traversed by the executions of $\Sigma_\|$, and whose transitions describe belief evolutions:

Definition 23 *(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 $\textsc{Bel}(\Sigma) = \langle \mathcal{S}_{\mathcal{B}}, \{s^0_{\mathcal{B}}\}, \mathcal{I}, \mathcal{O}, \mathcal{R}_{\mathcal{B}}, \mathcal{L}_{\mathcal{B}}\rangle$, where:

- $\mathcal{S}_{\mathcal{B}} = \mathit{Reachable}_{\mathcal{B}}(\Sigma)$;
- $s^0_{\mathcal{B}} = \tau\text{-closure}(\mathcal{S}^0)$;
- 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 $\langle B, a, B'\rangle \in \mathcal{R}_{\mathcal{B}}$;
- $\mathcal{L}_{\mathcal{B}}(B) = \{p \in \mathcal{P}rop : B \models_\Sigma p\}$.

Therefore, the belief-level system for $\Sigma_\|$ is an STS, exempt from τ transitions and with a single initial state, which collects together all the possible executions of $\Sigma_\|$ which are equivalent from the point of view of an external observer of $\Sigma_\|$: all the states that are reached by means of equivalent transitions (under such viewpoint) are collected into one state of $\textsc{Bel}(\Sigma_\|)$. This allows us to recast the definition of composition problem by imposing conditions not on the controller, but on the observable induced executions of the controlled domain, represented in $\textsc{Bel}(\Sigma_\|)$. This means that we can search for a suitable controller by identifying a subset (namely, a prefix) of $\textsc{Bel}(\Sigma_\|)$ such that every possible execution satisfies the composition requirement. More specifically, as justified in the previous Section, since we consider reachability goals, we can restrict our attention to controllers whose execution is loop-free; this corresponds to searching just for DAG-structured prefixes of $\textsc{Bel}(\Sigma_\|)$. Moreover, the fact that we intend to generate deterministic controllers, and the requirement of deadlock–freedom, imply two further constraints on the kind of subsets of $\textsc{Bel}(\Sigma_\|)$ we are interested into. First, the determinism of controllers imposes that, in our subtree, at each point, we will need to consider at most one input transition. Second, deadlock–freedom implies that such controllers never discard outputs of the controlled system; that is, if our subset contains a state B that originates some output transitions in $\textsc{Bel}(\Sigma_\parallel)$, then the subset must contain *every* output transitions of $\textsc{Bel}(\Sigma_\parallel)$ from B.

Putting together all the above constraints, we conclude that we only need to consider portions of $\textsc{Bel}(\Sigma_\parallel)$ structured as follows:

Definition 24 *(Execution subtree).* Let $\textsc{Bel}(\Sigma) = \langle \mathcal{S}_\mathcal{B}, \mathcal{S}^0_\mathcal{B}, \mathcal{I}, \mathcal{O}, \mathcal{R}_\mathcal{B}, \mathcal{L}_\mathcal{B}\rangle$ be the belief-level system for a domain $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L}\rangle$. We say that $\Sigma'_\mathcal{B} = \langle \mathcal{S}'_\mathcal{B}, \mathcal{S}^0_\mathcal{B}, \mathcal{I}, \mathcal{O}, \mathcal{R}'_\mathcal{B}, \mathcal{L}'_\mathcal{B}\rangle$ is an *execution subtree* of $\Sigma_\mathcal{B}$ if and only if:

(1) $\mathcal{S}'_\mathcal{B} \subseteq \mathcal{S}_\mathcal{B}$ and $\mathcal{R}'_\mathcal{B} \subseteq \mathcal{R}_\mathcal{B}$, that is, $\Sigma'_\mathcal{B}$ is a subgraph of $\textsc{Bel}(\Sigma)$.
(2) $\forall \langle B, o, B'\rangle \in \mathcal{R}_\mathcal{B} : ((o \in \mathcal{O} \wedge B \in \mathcal{S}'_\mathcal{B}) \to (B' \in \mathcal{S}'_\mathcal{B} \wedge \langle B, o, B'\rangle \in \mathcal{R}'_\mathcal{B}))$, that is, a belief of the subtree must originate all the outputs it originates in the belief-level STS.
(3) $\forall B \in \mathcal{S}'_\mathcal{B} : (\exists B' \in \mathcal{S}_\mathcal{B}, a \in \mathcal{I} : (B, a, B') \in \mathcal{R}'_\mathcal{B}) \to (\neg \exists B'' \in \mathcal{S}'_\mathcal{B}, a' \in \mathcal{I} : (B, a', B'') \in \mathcal{R}'_\mathcal{B} \wedge (a \neq a' \vee B' \neq B''))$, that is. If some input transition starts from a belief B, only one such transition may start from B in the subtree.
(4) No run of $\Sigma'_\mathcal{B}$ can traverse the same belief more than once, i.e. any belief may only appear once in any path that can be constructed by concatenating transitions of $\Sigma'_\mathcal{B}$. This implies that all such paths have finite length.
(5) $\mathcal{L}'_\mathcal{B}$ is the function restriction of $\mathcal{L}_\mathcal{B}$ to the states in $\mathcal{S}'_\mathcal{B} \subseteq \mathcal{S}_\mathcal{B}$ (that is, $\mathcal{L}'_\mathcal{B} = \mathcal{L}_\mathcal{B}|_{\mathcal{S}'_\mathcal{B}}$).

We will denote with $\Sigma'_\mathcal{B} \sqsubseteq \textsc{Bel}(\Sigma)$ the fact that $\Sigma'_\mathcal{B}$ is an execution subtree of $\textsc{Bel}(\Sigma)$. (Notice that, while the subtree relation can applied to generic STSs, we will make use of it only by referring to subtrees of belief-level systems.)

It can be seen that:

- conditions (1) and (4), and the fact that the initial belief of the subtree is the same of the originating belief-level system, ensure that the subtree is a DAG-structured portion of the belief-level system;
- condition (2) requires that no output generated from the environment is disregarded;
- condition (3) states that, in the subtree, we only need to consider a uniquely determined way of driving the environment, since the controller is deterministic;
- since the controller is deterministic, it is never the case that both an input and an output transitions start from the same state B.

Given its acyclic structure, a subtree can be associated to a notion of *subtree depth*: the maximal length of the sequences $B_0, B_1, \ldots, B_n$ such that $B_i, a, B_{i+1} \in \mathcal{R}_\mathcal{B}$. Intuitively, the depth of a subtree identifies the number of steps needed, in the worst case, to terminate the execution of the controlled system.

A subtree π of $\textsc{Bel}(\Sigma_\parallel)$ can be associated to a controller that, once connected to $\Sigma_\parallel$, drives it in such a way that only the beliefs of π can be traversed during the execution of the controlled system. Amongst the controllers that satisfy this requirement, we can focus on a synchronous controller for which we can give an immediate constructive definition: the structure of the controller mirrors the one of the subtree, featuring one state for each state of the subtree and contraposing one-to-one its I/O operations with those of the subtree.

Definition 25 *(Controller associated to a subtree).* Let $\Sigma'_\mathcal{B} = \langle \mathcal{S}'_\mathcal{B}, \mathcal{S}^0_\mathcal{B}, \mathcal{I}, \mathcal{O}, \mathcal{R}'_\mathcal{B}, \mathcal{L}'_\mathcal{B}\rangle \sqsubseteq \textsc{Bel}(\Sigma)$ be an execution subtree for $\textsc{Bel}(\Sigma)$. We say that the STS $Contr(\Sigma'_\mathcal{B}) = \langle \mathcal{S}'_\mathcal{B}, \mathcal{S}^0_\mathcal{B}, \mathcal{O}, \mathcal{I}, \mathcal{R}'_\mathcal{B}, \mathcal{L}_\emptyset\rangle$ is the controller associated to $\Sigma'_\mathcal{B}$, where $\mathcal{L}_\emptyset$ denotes the empty labeling function.

Lemma 26. *Let Σ be an STS, and $\Sigma'_\mathcal{B} \sqsubseteq \textsc{Bel}(\Sigma)$. The STS $Contr(\Sigma'_\mathcal{B})$ is a controller for Σ.*

Proof. Since the labeling of $Contr(\Sigma'_\mathcal{B})$ is empty by definition and the input/outputs correspond to those of Σ, we only need to show that $Contr(\Sigma'_\mathcal{B})$ is a deterministic STS:

- Since the belief-level system $\textsc{Bel}(\Sigma)$ has a single initial state, and so its execution subtree $\Sigma'_\mathcal{B}$, clause (1) of Definition 8 is satisfied.
- Since the belief-level system $\textsc{Bel}(\Sigma)$ contains no τ-transition, also the subtree $\Sigma'_\mathcal{B}$ and its associated controller contain no τ-transition. Thus clause (2) of Definition 8 is satisfied.
- From clause (3) of the definition of subtree, it is immediate that at most one output transition can originate from any state of the controller associated to a subtree. Thus also clause (3) of Definition 8 is satisfied. $\square$

Thus, if $\Sigma'_\mathcal{B} \sqsubseteq \textsc{Bel}(\Sigma)$, then $Contr(\Sigma'_\mathcal{B})$ can be used to control Σ. However, to prove that such a controller is also deadlock-free for Σ, we need to first show that the execution of $Contr(\Sigma'_\mathcal{B}) \rhd \Sigma_\parallel$ traverses states (B, s) such that $s \in B$ (remember that the states of $Contr(\Sigma'_\mathcal{B})$ are beliefs of Σ).

Lemma 27. *Let Σ be an STS. Let $\Sigma'_{\mathcal{B}} \subseteq \text{Bel}(\Sigma) = \langle \mathcal{S}_{\mathcal{B}}, \mathcal{S}^0_{\mathcal{B}}, \mathcal{I}, \mathcal{O}, \mathcal{R}_{\mathcal{B}}, \mathcal{L}_{\mathcal{B}} \rangle$. Every state $(B, s) \in Reachable(Contr(\Sigma'_{\mathcal{B}}) \triangleright \Sigma_{\|})$ is such that $s \in B$.*

Proof. The proof goes by induction on the number i of input/output actions traversed to reach (B, s) (which equals the length of the sequence of beliefs of the subtree traversed to reach B).

Basis: $i = 0$. Thus B is the τ-closure of $\mathcal{S}^0$. Since no input nor output have been executed, s must be reachable by a chain of τ transitions from an initial state, so it belongs to B.

Induction step. We assume the theorem holds for (B, s) with $i = k$, and we prove it for $i = k + 1$. Then, there is an action $a \in \mathcal{I} \cup \mathcal{O}$ such that $(B, a, B') \in \mathcal{R}'_{\mathcal{B}} \subseteq \mathcal{R}_{\mathcal{B}}$, with $B' = Evolve(B, a)$. By the definition of *Evolve*, the states of B' consist of the τ-closure of the states of B that can be progressed via the action a. That is, considering a state $s' \in B'$, there is a least a state $\bar{s} \in B$ such that (a) $\bar{s}$ can be progressed via a, i.e. $\exists s'' : (\bar{s}, a, s'') \in \mathcal{R}_{\mathcal{B}}$, and (b) there is a (possibly empty) sequence of τ-transitions in $\mathcal{R}_{\mathcal{B}}$ that leads from s'' to s'. This means that $(B', s') \in Reachable(Contr(\Sigma'_{\mathcal{B}}) \triangleright \Sigma_{\|})$. In particular, since, by the inductive hypothesis, (B, s) can be reached in k input/output actions (from some state $(\mathcal{S}^0_{\mathcal{B}}, s_0)$, with $s_0 \in \mathcal{S}^0_{\mathcal{B}}$), (B', s') is reachable in $k+1$ input/output actions. Then, the set of states $\{(B', s') : s' \in B' \wedge B' = Evolve(B, a)\}$ are all the states reachable in $k + 1$ input/output actions starting from states of the form $(\mathcal{S}^0_{\mathcal{B}}, s_0)$ (with $s_0 \in \mathcal{S}^0_{\mathcal{B}}$). $\square$

Lemma 28. *Let Σ be an STS. Let $\Sigma'_{\mathcal{B}} \subseteq \text{Bel}(\Sigma)$ be an execution subtree of (the belief-level system for) Σ. Then $Contr(\Sigma'_{\mathcal{B}})$ is deadlock-free for Σ.*

Proof. To prove the statement, we prove in turn that $\Sigma_c = Contr(\Sigma'_{\mathcal{B}})$ obeys clauses (1) and (2) of Definition 13.

Concerning clause (1), we recall that every output transition from a reachable state has a correspondent transition within the belief-level system, and that if a belief is part of the subtree, every output transition from it must also appear into the subtree. Thus, if $(s, a, s') \in \mathcal{R}$ for some output $a \in \mathcal{O}$ and $(B, s) \in Reachable(\Sigma_c \triangleright \Sigma)$, since $s \in B$, some transition (B, a, B') must be part of the subtree. But then, it also belongs to $Contr(\Sigma'_{\mathcal{B}})$, which proves clause (1) of Definition 13 (where $\tau\text{-closure}(s_c) = s_c$ given the absence of τ-transitions in the controller).

Concerning clause (2) of Definition 13, we observe that, if $(B, a, B') \in \mathcal{R}'_{\mathcal{B}}$ for some $(B, s) \in Reachable(\Sigma_c \triangleright \Sigma)$, then two cases are possible:

- the only input from the controller is applicable on $s \in B$, i.e. s is in the τ-frontier of B; then, clause (2) is trivially true.
- s is not in the τ-frontier of B. In this case, since (B, a, B') belongs to the belief-level system of Σ, there must be a sequence of τ-transitions from s, such that every state in the sequence is still in B, and a is applicable on the last state — i.e. $(s', a, s'') \in \mathcal{R}$ for some $s' \in \tau\text{-closure}(s)$. This shows clause (2) holds. $\square$

We are finally in a position to recast the notion of solution controller so to refer to it as an execution subtree, i.e. a subgraph of the constructively defined $\text{Bel}(\Sigma_{\|})$: it is enough to require that every final state of the subtree is a belief satisfying ρ.

Definition 29 *(Solution subtree).* Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be a domain defined over propositions $\mathcal{P}rop$. Let ρ be a propositional formula over $\mathcal{P}rop$. The execution subtree $\Sigma'_{\mathcal{B}} \subseteq \text{Bel}(\Sigma)$ is a solution for the requirement ρ if every final state B of $\Sigma'_{\mathcal{B}}$ is such that $B \models_{\Sigma} \rho$.

Indeed, a solution subtree π is associated to a solution controller $Contr(\pi)$, and vice versa, as shown by the following lemma:

Lemma 30. *Let π be a solution subtree of $\text{Bel}(\Sigma_{\|})$ for the problem of service composition over the requirement ρ. Then $Contr(\pi)$ is a solution controller for the same problem.*

Proof. The statement is easily proved based on Lemma 27. If the subtree is a solution, then every state of every final state of π is a goal state; but then, due to Lemma 27, every final state reachable by execution $Contr(\pi)$ is a goal state, so $Contr(\pi)$ is a solution controller. $\square$

To associate a subtree of $\text{Bel}(\Sigma_{\|})$ with a controller Σ_c, we observe that, following Definition 11, Σ_c constrains the behaviors of $\Sigma_{\|}$: only certain states and transitions of $\Sigma_{\|}$ can ever be traversed during the execution of $\Sigma_c \triangleright \Sigma_{\|}$. That is, Σ_c *induces* a constrained subset of $\Sigma_{\|}$, which we denote with $\Sigma_{\|}(\Sigma_c)$. We now show that $\Sigma_{\|}(\Sigma_c)$ is a solution subtree if Σ_c is a solution controller:

Definition 31 *(Induced STS).* Let $\Sigma_{\|} = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be an STS describing a set of component services, and let $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}^0_c, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_{\emptyset} \rangle$ be a controller for $\Sigma_{\|}$.

The STS $\Sigma_{\|}(\Sigma_c) = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}', \mathcal{L} \rangle$, where $\mathcal{R}' = \{(s, a, s') : \exists s_c, s'_c : ((s_c, s) \in Reachable(\Sigma_c \triangleright \Sigma_{\|}) \wedge ((s_c, s), a, (s'_c, s')) \in \mathcal{R}_c \triangleright \mathcal{R})\}$, is called the STS induced by Σ_c on $\Sigma_{\|}$.

Lemma 32. *If $\Sigma_c \triangleright \Sigma_\parallel \models \rho$, then the execution subtree corresponding to the controlled system, $\textsc{Bel}(\Sigma_\parallel(\Sigma_c)) \subseteq \textsc{Bel}(\Sigma_\parallel)$ is a solution subtree for the composition problem.*

Proof. The proof relies on the fact that we restrict to controllers that induce finite, loop-free executions over the controlled system; i.e. a state of the controlled system is never traversed more than once. Under that proviso, according to Definition 17 and, $\Sigma_c \triangleright \Sigma_\parallel \models \rho$ means that every execution of $\Sigma_c \triangleright \Sigma_\parallel$ is finite, and its final state satisfies ρ (according to Definition 6). In turn, considering Definition 31 above, this means that the final states of the STS $\Sigma_\parallel(\Sigma_c)$ induced by Σ_c on $\Sigma_\parallel$ satisfy ρ. Due to the definition of belief-level system, such final states will be states in the τ-frontiers of the final beliefs of the induced subtree $\textsc{Bel}(\Sigma_\parallel(\Sigma_c))$. That is, each final belief $\textsc{Bel}(\Sigma_\parallel(\Sigma_c))$ satisfies ρ, according to Definition 22. Following Definition 29, this means that $\textsc{Bel}(\Sigma_\parallel(\Sigma_c))$ is a solution subtree for the composition problem, proving the statement. $\square$

The above lemmas can be combined to associate in a one-to-one way the existence of a solution with that of a solution subtree:

Theorem 33 *(Composition problem at the belief-level). Let $\Sigma_1, \dots, \Sigma_n$ be a set of state transition systems, defined over propositions $\mathcal{P}rop_1, \dots, \mathcal{P}rop_n$ respectively. Let ρ be a composition requirement, that is a propositional formula over $\mathcal{P}rop_1, \dots, \mathcal{P}rop_n$. A solution controller for the problem exists if and only if a solution subtree of $\textsc{Bel}(\Sigma_1 \parallel \cdots \parallel \Sigma_n)$ exists.*

Proof. This theorem descends directly from Lemmas 30 and 32. Namely: if there exists a solution subtree π, Lemma 30 guarantees that there exists a solution controller $Contr(\pi)$. Vice versa, if there exists a solution controller Σ_c, Lemma 32 guarantees that $\textsc{Bel}(\Sigma_\parallel(\Sigma_c))$ is a solution subtree. $\square$

Relevant to the purposes of automatically constructing a controller, Lemma 30 states that if a solution subtree π has been found, a solution controller $Contr(\pi)$ can be obtained: a controller Σ_c that induces a solution subtree of $\textsc{Bel}(\Sigma_1 \parallel \dots \parallel \Sigma_n)$ is a solution to the composition problem. Thus we can recast the composition problem as that of identifying a solution subtree $\pi \subseteq \textsc{Bel}(\Sigma_1 \parallel \cdots \parallel \Sigma_n)$, taking the loop-free controller $Contr(\pi)$ as the solution of the problem.

Based on these considerations, our approach will be structured as follows:

(1) Construct, for each component service, an STS Σ_i;
(2) Construct the parallel product $\Sigma_\parallel$ of $\Sigma_1, \dots, \Sigma_n$, and then its belief-level counterpart $Bel(\Sigma_\parallel)$;
(3) Search the execution structure in $Bel(\Sigma_\parallel)$ to identify one execution tree associated to a solution controller;
(4) Extract, from the execution tree, the controller Σ_c, and emit it as a WS-BPEL program.

In the following subsections, we detail the core of this approach, i.e. the algorithm used for constructing $Bel(\Sigma_\parallel)$, and the one for searching a solution subtree inside it.

5.2. Algorithms

Following the ideas stated in the previous section, we solve the composition problem by first constructing the belief-level machine $\textsc{Bel}(\Sigma_\parallel)$, and then searching for a solution subtree π inside it. Once such a solution is found, the (loop-free) controller $Contr(\pi)$ is extracted and constitutes a solution to the composition problem.

To construct the belief-level machine $\textsc{Bel}(\Sigma_\parallel)$, we rely on an algorithm that implements Definition 23 by a simple fix-point algorithm that performs a forward search of the space of beliefs, starting from the initial belief. At each iteration of the fix-point, a τ-closure operator is used that realizes Definition 12, so that indistinguishable states of the STS are grouped together.

While it is possible to apply the algorithm directly on $\Sigma_\parallel$, we can do better. In particular, we exploit the fact that $\textsc{Bel}(\Sigma_1 \parallel \Sigma_2) = \textsc{Bel}(\Sigma_1) \parallel \textsc{Bel}(\Sigma_2)$. This stems from the way states and transitions of the two individual STSs cohabit independently within $\Sigma_1 \parallel \Sigma_2$, therefore allowing for projecting them back when constructing the belief-level system. Therefore, we can separately compute the belief-level system for each component service, and to represent the overall belief-level system as a modular composition of the various $\textsc{Bel}(\Sigma_i)$, where modules evolve on the basis of a parallel product semantics.

The impact of this approach on the effectiveness of the belief-level construction is clear: since the algorithm, given an STS Σ, explicitly enumerates the reachable beliefs of $\textsc{Bel}(\Sigma)$, and since they can be up to $2^{|\mathcal{S}|}$, the monolithic approach may have to enumerate $2^{|\mathcal{S}_1|+\cdots+|\mathcal{S}_n|}$ beliefs, while the partitioned approach only $2^{|\mathcal{S}_1|} + \cdots + 2^{|\mathcal{S}_n|}$.

Once the belief-level system is built, we exploit a variant of the algorithm for strong planning presented in [32] to extract a solution subtree from it. Such algorithm receives as input the STS representing $\textsc{Bel}(\Sigma_\parallel)$, produced by the belief-level construction algorithm, and performs a symbolic regression search, where frontiers of states of $\textsc{Bel}(\Sigma_\parallel)$ are manipulated at once. We remark that a state of $\textsc{Bel}(\Sigma_\parallel)$ corresponds to a belief, i.e. a set of states of the component services; that is, symbolically manipulating sets of states of $\textsc{Bel}(\Sigma_\parallel)$ means symbolically manipulating sets of sets of states of the component services. This is an extremely powerful technique, that avoids handling enumeratively large sets of sets of service states.

In difference to the original planning algorithm presented in [32], our algorithm must explicitly deal with the constraints coming from the way subtrees are defined, and from the requirement that the controlled system is deadlock-free.

```
(1)  function $plan(I, G)$
(2)  $OldSA := Fail$
(3)  $SA := \emptyset$
(4)  while $(OldSA \neq SA \wedge I \notin (G \cup \text{StatesOf}(SA)))$
(5)      $Pr := \text{StrongPreImage}(G \cup \text{StatesOf}(SA))$
(6)      $NewSA := \text{PruneStates}(Pr, G \cup \text{StatesOf}(SA))$
(7)      $OldSA := SA$
(8)      $SA := SA \cup NewSA$
(9)      done
(10) if $(I \in (G \cup \text{StatesOf}(SA)))$
(11)     return $SA$
(12) else
(13)         return $Fail$
(14) endif
```

Fig. 12. The search algorithm.

This specifically affects the key primitive over which the algorithm is built, namely the so-called strong pre-image primitive StrongPreImage.

Given a set S of states of $\text{Bel}(\Sigma_\parallel)$, the pre-image primitive returns a set of pairs $\{\langle s, a\rangle\}$, associating a state s with an input/output action a. Such pairs identify all the predecessors of the states in S via some input/output action, such that the execution of the action a associated to s is guaranteed to lead to states inside S, regardless of nondeterminism. The set of state–action pairs constitutes a so-called *state–action table* (see [32]), which represents a (partial) function between states of the domain and actions to be executed on them. Our pre-image primitive is defined as follows:

$$\begin{aligned}\text{StrongPreImage}(S) \doteq {} & \{(s,i) : i \in \mathcal{I} \wedge \forall s' : (s,i,s') \in \mathcal{R} \to s' \in S\} \cup \\ & \{(s,o) : o \in \mathcal{O} \wedge \exists s' : (s,o,s') \in \mathcal{R} \to s' \in S \wedge \\ & \forall s' \in \mathcal{S}, o' \in \mathcal{O} : (s,o',s') \in \mathcal{R} \to s' \in S\}.\end{aligned}$$

Here, due to the deadlock–freedom requirement, input and output actions are treated differently. In particular, since outputs correspond to uncontrollable choices taken by the component services, once we consider an output from a service, we have to consider *every* possible outputs from it, and guarantee that every outcome is in the target set.

This is necessary to guarantee conditions (1) and (2) in Definition 13, avoiding situations such as that shown in Fig. 10 (low), where some output generated by a component may not be handled by its controller. Indeed, the definition of StrongPreImage is such that its application consists in generating a portion of execution subtree; the asymmetry between the handling of inputs and outputs closely corresponds to requirements (2) and (3) of Definition 24, which in turn derive themselves from Definition 13.

We remark that the deadlock–freedom requirement has no correspondence in the formulation of strong planning of [32], where only the controller is responsible for taking actions. Indeed, [32] relies on a different StrongPreImage primitive, where actions are handled uniformly (and analogously to how we treat inputs).

The other important primitive for the algorithm is the so-called pruning primitive PruneStates, and is used after executing a pre-image, to remove, from a state–action table π, all the pairs $\langle s, a\rangle$ such that a solution is already known for s and stored into S. It is defined as:

$$\text{PruneStates}(\pi, S) \doteq \{\langle s,a\rangle \in \pi : s \notin S\}.$$

This pruning is important to guarantee that only the shortest solution from any state appears in the state–action table.

The search algorithm is depicted in Fig. 12, and basically consists of a fix-point iteration that incrementally constructs a state–action table. The state–action table SA is initialized as empty, and enriched, at each iteration, with states from which it is possible to control the system to achieve states in SA. The algorithm terminates either when no more states can be added to the state–action table, or when the current state–action table already includes the initial states, in which case a solution has been found.

The arguments for proving that the algorithm always terminates, and it is correct and complete, are similar to those presented in [32]. In particular, termination is proved by observing that the number of states in the visited domain is finite, and that the number of states in the state–action table grows up monotonically.

Theorem 34 *(Termination). 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 a (belief-level) STS; let $I \in \mathcal{S}_\mathcal{B}$ and $G \subseteq \mathcal{S}_\mathcal{B}$. The execution of $plan(I, G)$ on $\Sigma_\mathcal{B}$ terminates.*

Proof. We prove that the main loop of the algorithm (lines 4–9) exists after a finite number of iterations. To do so, we first recall that $\mathcal{S}_\mathcal{B}$ contains a finite number of states. Moreover, we observe that the state–action table SA grows up monotonically (see line 8 of the algorithm). This implies that, after at most $|\mathcal{S}_\mathcal{B}|$ iterations of the main loop, no new state–action pair will be added to SA at line 8. But, as soon as no new state–action pair is added to SA at line 8, the main loop exits, since then, at line 4, $OldSA = SA$, and the "while" condition becomes false. That is, the main loops exits after at most $|\mathcal{S}_\mathcal{B}|$ iterations. □

To prove correctness and completeness, we rely on proving that, at each iteration of the main loop at lines 4–9, the state–action table contains states for which a solution tree exists, and such tree is indeed represented by (a portion of) the state–action table. Indeed, starting from a set of initial states, a state–action table induces an STS defined as follows:

Definition 35 *(STS induced by state–action table).* Let $SA = \{\langle s, a\rangle\colon\ s \in \mathcal{S}, a \in \mathcal{I} \cup \mathcal{O}\}$ be a state–action table referring to the actions and states of an STS $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L}\rangle$. Let $I \subseteq \{s\colon\ \exists a \in \mathcal{I} \cup \mathcal{O}.\langle s, a\rangle \in SA\}$ be a set of initial states, and let $G \subseteq \mathcal{S}$ be a set of goal states.

The STS induced by I on SA, denoted $\mathit{STSof}(SA, I, G) = \langle \mathcal{S}_{SA}, I, \mathcal{I}, \mathcal{O}, \mathcal{R}_{SA}, \mathcal{L}_{SA}\rangle$, is defined as follows:

(1) $I \subseteq \mathcal{S}_{SA}$;
(2) if $s \in \mathcal{S}_{SA}$, and there exists a sequence $\langle s_0, a_0\rangle, \ldots, \langle s_n, a_n\rangle$ such that $s_0 = s$, $\langle s_i, a_i\rangle \in SA$, $\forall i \in [0, n-1] : \langle s_i, a_i, s_{i+1}\rangle \in \mathcal{R}$, then $s_n \in \mathcal{S}_{SA}$;
(3) $\mathcal{S}_{SA}$ contains no other state than those induced by the above conditions (1) and (2);
(4) if $s \in \mathcal{S}_{SA}$ and $\langle s, a, s'\rangle \in \mathcal{R}$, then $\langle s, a, s'\rangle \in \mathcal{R}_{SA}$;
(5) $\mathcal{L}_{SA}$ is the function restriction of $\mathcal{L}$ to the states in $\mathcal{S}_{SA}$, i.e. $\mathcal{L}_{SA} = \mathcal{L}|_{\mathcal{S}_{SA}}$.

Lemma 36 *(Invariant property of the state–action table). At the i-th iteration of the main loop at lines* 4–9, *$G \cup$ StatesOf(SA) contains all the states for which a solution subtree of depth up to i exists. In particular, if $s \in G \cup$ StatesOf(SA), then STSof(SA, I, G) is a solution subtree rooted at s.*

Proof. The proof goes by induction on the number i of iterations of the main loop of the algorithm (lines 4–9).

Basis ($i = 0$). The state–action table is empty, and $G \cup$ StatesOf$(SA) = G$ consists of exactly those states for which a solution exists that requires 0 steps to achieve the goal (i.e. for which a solution subtree of depth 0 exists).

Induction step. We assume that the theorem holds at the i-th iteration, and prove it for the $i + 1$-th iteration. At line 5, due to the definition of StrongPreImage, every tuple of states in $G \cup$ StatesOf(SA) is considered as a possible target starting from any state s, either by some input action or by all possible output actions. Due to the inductive hypothesis and the definition of StrongPreImage, the produced state–action pairs Pr will be such that the goal will be achievable from their states in at most $i + 1$ steps, that is, if $\langle s, a\rangle \in Pr$, then $\mathit{STSof}(SA \cup Pr, s, G)$ is a subtree of depth up to $i + 1$ (in other words, given the inductive hypothesis and the correspondence between the definition of StrongPreImage and requirements (2) and (3) in Definition 24, the application of StrongPreImage extracts exactly the states and transitions corresponding to a subtree of depth $i + 1$). At line 6, only state–action pairs in Pr for which the goal is reachable in *exactly* $i + 1$ steps will be stored in *NewSA*, since due to the inductive hypothesis, all those for which the goal can be reached in up to i steps are in SA already. At line 8, such state–action pairs are added to SA, which as a consequence will hold all of the state–action pairs from which the goal is reachable in up to $i + 1$ steps. □

Theorem 37 *(Correctness and completeness). If the algorithm returns a state–action table SA, then SA contains a solution subtree to the composition problem. If the algorithm returns Fail, then no solution subtree to the composition problem exists.*

Proof. This is a direct consequence of Lemma 36: if and only if $I \in G \cup$ StatesOf(SA), a (finite) solution subtree for I exists, and in that case, it is contained in *SA*. Depending on such condition, either SA or *Fail* is returned (lines 10–14). □

The implementation of the algorithms for belief-level domain construction, and for searching the solution subtree, rely on a symbolic machinery to effectively represent and manipulate sets of states, where Binary Decision Diagrams (BDDs [26]) are used as a canonical representation for formulas over state variables. This symbolic representation technology has been largely investigated in the areas of Model Checking, Diagnosis, Program Synthesis and Planning (see e.g. [13,58,32,39,88,47]), and is at the core for the improved performances of state-of-the-art tools in these areas (see e.g. [27,14,48]).

While, within the two phases (belief-level construction and search), the states of the symbolic representation assume very different meanings, we uniformly adopt the SMV language to describe STSs: in belief construction, both for the domain given in input and the outputted belief-level STSs; in search, as the input domain. SMV is the native language used in model-checking systems such as NuSMV [27]. It is a rich language, allowing a variety of constructs and mechanisms to compactly represent STSs. On top of a fairly standard representation of the states and actions by means of state and action variables, it allows representing transitions by combining an explicit enumeration of corresponding TRANS statements, and a description of the evolution of variables by means of (possibly conditional) ASSIGNments. Indeed, the representation of STSs via SMV is schematic, and very close to the one depicted in Section 3. Moreover, a very relevant feature for our purpose stands in the possibility to define independent "modules"; semantically, independent modules are equivalent to their parallel product, while internally, a disjunctively partitioned transition relation is stored so to save significant space and computation time. As mentioned, this is very useful in belief-level construction: given the disjointness of input/outputs amongst component services, it holds that $\textsc{Bel}(\Sigma_1 \parallel \cdots \parallel \Sigma_n) = \textsc{Bel}(\Sigma_1) \parallel \cdots \parallel \textsc{Bel}(\Sigma_n)$, and as such, we can build the belief-level independently for each component, and implicitly combine them by representing them as modules.

Thus, in belief-level construction, the states of the input domain correspond to states of the STS associated to a component Web service; in this case, a BDD represents a set of indistinguishable states of one service, i.e. a belief. Based on such representation, belief-level construction explicitly enumerates the beliefs for a single component service, but symbolically evolves the sets of states they represent, by internal or input/output actions. The resulting belief-level domain for a component service is emitted as a MODULE in the SMV language, featuring input/output variables that correspond to those in the STS (which, in turn, are connected to parts of input/output operations in the original WS-BPEL input). The state of such belief-level system consists of a single enumerative SMV variable that names all the possible beliefs traversed by the component. The dynamics of such belief-level system is represented via a list of TRANS statements describes the way input/output actions connect beliefs. Then, the modules corresponding to the components services are then simply glued together within a unique SMV file, which represents the overall domain to be searched upon, and whose states correspond to beliefs of $\textsc{Bel}(\Sigma_1 \parallel \cdots \parallel \Sigma_n)$, i.e. to tuples $\langle B_1, \ldots, B_n \rangle$ where $B_i \in \textsc{Bel}(\Sigma_i)$. This means that symbolically evolving a set of states, as performed by the search algorithm, corresponds to evolving a set of state sets of the component services; this would be extremely demanding, and practically unfeasible, by an explicit enumeration of the states, making our symbolic factored representation technique virtually necessary.

We remark that, as it is well known from the literature on binary decision diagrams, e.g. [26,13], symbolic representations, in principle, may blow-up as badly as enumerative ones. However, as witnessed in the huge literature on model-checking, diagnosis and monitoring, BDDs are capable, in the vast majority of cases, to represent formulas in a size polynomial in the number of propositions, and therefore, due to the polynomial complexity of the key BDD operations, to manipulate them effectively. This result is due in good part to the usage of advanced heuristics and optimizations inside the realization of software BDD packages, and it holds independently from the fact that formulas may (and often do) feature exponentially many models in the number of propositions. That is, BDD representations are in most cases compact and effective, even when a corresponding model-enumerative representation may not be handled in practice. Naturally, it is possible to design "pathological" formulas that cause the blow-up of BDD representations, and it is known that also for a very limited set of practical cases (e.g. combinatorial multiplier circuits in hardware verification), BDD representations are not polynomial. In the following section, we empirically test the approach, and we will therefore be able to discuss whether the good practical results on BDDs shown in the literature on model-checking and the related areas of planning as model-checking, symbolic diagnosis and monitoring also apply to the problem of Web service composition.

6. Experimental evaluation

In order to test the effectiveness of the proposed technique, we have implemented the architecture proposed in Fig. 9, realizing the translations between WS-BPEL and STS described in Section 3, the belief-level construction of Definition 23 and the search algorithm in Fig. 12. Our platform makes use of CUDD [85], an effective BDD package that offers effective primitives including highly advanced memory and space optimizations [25,91].

On our platform, we conducted a set of experiments, executing them over a 1.2 GHz Pentium machine, equipped with 4 GByte memory, running Linux. All of our tests have been run setting a timeout at 3600 seconds of CPU usage.

Our analysis aims at identifying which structural and dimensional features of Web services impact on the performance of our composition platform, and in which way. This allows us to discuss the scalability and limits of our approach.

More specifically, our analysis focuses on the composition of loop-free services, for which the complexity of orchestrating protocols can be more easily analyzed observing their structure, and, before tackling more general scenarios, starts by considering a set of symmetric scenarios where all services play the same role. These symmetric scenarios allow evaluating the scalability over the number of asynchronous component services, and are organized as follows:

(1) We first consider "simple" services, which encode question-and-answer procedures that simply receive a message, produce an output and terminate. We will study both deterministic services, whose answer is a function of the input, and nondeterministic ones, which may also return a failure message based on an internal choice.
(2) Then, we consider more complex services, encoding procedures that are more lengthy, and which feature sources of internal nondeterminism. This means that, in order to satisfy a requirement, a composed service will be forced to take complex choices, depending on the behavior of the components. In particular, in order to study the impact of structural factors over the composition, we will consider in turn:
    (a) "binary unbalanced procedures", where, at each step, a conclusive failure may be signaled by the procedure; in fact, these procedures correspond to "concatenations" of simple nondeterministic services;
    (b) "binary balanced procedures", where two possible answers are possible at each interaction, none of which signaling a conclusive failure;
    (c) "n-ary balanced procedures", which (at a high level) generalize binary balanced procedures in terms of number of possible answers.

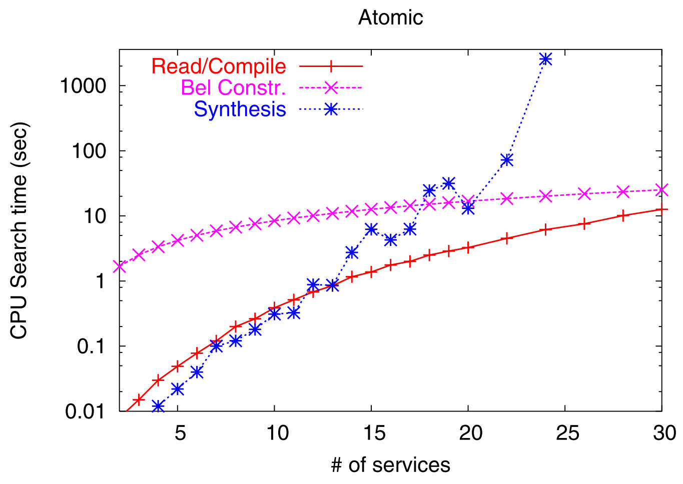


Fig. 13. Combining N atomic services.

For both sets, we will vary the size of the sets of services that need to be composed, analyzing the performance of the various phases in the composition task: construction of the belief-level domain, internalization and search. This will allow us to draw some considerations about the relationship of the results in the various settings and the structural characteristics of the involved services. Finally, as an example of a non-symmetric, more realistic scenario, we will consider the performance over the P&S example, discussing its relationship with the symmetric scenarios introduced above.

We remark that, to have experiments comparable, we will keep a fixed arity for the types of the objects handled by the component services of all the scenarios. In particular, all services will rely on disjoint binary types, since this is sufficient to guarantee, for all the cases we consider, that the synthesized orchestration enacts a fully generic protocol. We leave to Section 8 a further discussion on different ways to ground variable types.

6.1. *Simple services*

We start by considering the simpler services that may be thought of, which model a (fully deterministic, predictable) function computation: each component W_i receives an input D, computes a (non-interpreted) function $f_i(D)$ and returns it as an output D', terminating into a state s_f.

Given a set of such components, the composition requirement consists of having the n-th service return the nested function $f_n(f_{n-1}(\ldots(f_1(D))))$ (and have each component terminate successfully). In terms of elements of the finite STSs according to Section 3, the goal consists of a formula that (a) for all services, requires the state to be s_f, and (b) describes the nested computation as a disjunction of equalities between service variables and ground function terms.

To achieve the goal, insofar functions refer to many-valued ranges, a composed procedure must suitably interleave the invocations to the components, passing at each step the current result; we remark again that this would not be the case, should functions feature unary domains or co-domains.

The results are shown in Fig. 13. The three lines report the timings required in the respective phases of our approach to service composition: *Bel constr.* refers to the construction of the belief-level search space; *Read/compile* refers to the time spent for reading and internalizing the representation of such space; and *Synthesis* refers to the search and emission of the orchestration service. We are able to combine up to 20 services within a very reasonable time. We observe that belief-level construction has a significant relative cost; however, as expected, the partitioned representation helps out, and the time spent in this phase grows only linearly with the number of component services.

Once the belief-level system has been represented, it has to be internalized; constructing its internal representation, which implies computing the parallel product of each module, has a marginal cost, which however grows polynomially in the number of components.

For large sets of components, the most relevant cost is due to the search phase, which seems to grow up quasi-exponentially, with some notable oscillations.

We remark that, in the vast majority of cases, exponentially large sets of states can be represented by polynomially large BDD structures, and thus manipulated within polynomial time. Since we adopt a breadth-first backward search style, given the domain under exam, composing N services will require computing a number of layers proportional to N; the number of nodes in each layer can be thought, at a high level of approximation, as growing polynomially at each backward iteration. All this would then result in an overall search time polynomial in N.

However, breadth-first search is very memory-demanding, and when N is high, the garbage collection mechanism of the BDD package enters into play to rearrange data structures, and remove unused ones. This is what influences most the performance of the search for high values of N, in particular with $N > 15$.

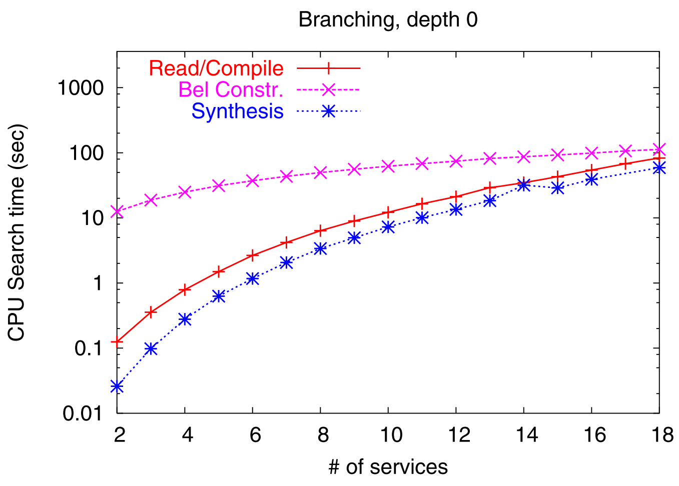


Fig. 14. Combining N branching services.

We then step to the smaller possible Web services that encode a possibly failing procedure; i.e., each Web service will accept a request, and either compute a function, or fail. In the former case, the service will wait for an ack/nack signal, depending on which it finally fails or terminates successfully. Thus such services will feature two branching points; the first one in the procedure encodes a form of "internal" nondeterminism, while the second represents and "external" nondeterminism which is necessary in order to be capable to drive the service to fail or to succeed. The requirement for combining a number of such services consists in asking for the set of services to compute a nested function $f_n(f_{n-1}(\ldots(f_1(D))))$ of a datum D, and have each component terminate successfully. In sub-order, we require to have every started component terminate with failure. We remark that allowing the externally commanded refusal is necessary so to have services "controllable" enough to be able to compose them according to the requirement. That is, the goal is similar to the one considered in the previous case, but adds as alternative (that is, "in or") a formula stating that, for all services, the state is either the initial one, or the one associated to failure. The composed procedure must implement the following behavior, which combines transferring data "in the right order" between services, and detecting and handling failure situations by appropriately driving each suspended service to fail.

- invoke the service W_{i_n} computing f_{i_n}, passing it D
- if the service answers with failure, terminate
- receive D_n from W_{i_n}, and invoke the service $W_{i_{n-1}}$ computing $f_{i_{n-1}}$, passing it D_n
- if $W_{i_{n-1}}$ answers with failure, send a *nack* to W_{i_n} and terminate
- ...
- receive D_1 from W_{i_1}, and invoke the service W_{i_0} computing f_{i_0}, passing it D_1
- if W_{i_0} answers with failure, send a *nack* to every service $W_{i_1}, \ldots, W_{i_n}$ and terminate
- send *ack* to every service $W_{i_0}, \ldots, W_{i_n}$ and terminate.

The results for this test are reported in Fig. 14.

We observe that it is possible to achieve, in a reasonable time, the composition of a rather high number of such services — approximately 20. In this case, the harder phase in the composition consists in the construction of the belief-level system $\Sigma_\|$. Again, thanks to the modular modeling allowed by the SMV language, the weight of constructing and emitting $\Sigma_\|$ grows only linearly with the number of component services — while its interpretation to build the internal representation of $\Sigma_\|$ is polynomial. Both costs are about an order of magnitude higher than in the case of deterministic components.

The complexity of searching for a controller grows according to a polynomial curve, not unlikely the one for the predictable services. Indeed, the cost of the composition is only marginally higher than the one for predictable services.

This result is somehow to be expected: the cost of the search basically depends on the number of states to be represented in the number N of layers generated by the backward search, and N amounts to the length of the longer interaction needed to compose the services. But, in this test, given a set of n components, N is the same regardless of whether the components are predictable or not, and the number of visited states do not differ significantly.

These first tests on simple services hint at the fact that our approach might be particularly suited to deal with nondeterministic services: not only our representation is general enough to capture nondeterminism, but also the performance of composition seems to be affected only marginally by the need of handling different contingencies. In fact, for the specific case of fully predictable atomic services, it is possible to adopt simpler formalisms (such as STRIPS [38]) and more focused search techniques (e.g. forms of classical heuristic planning, see [44]) to obtain a better scalability. We remark, however,

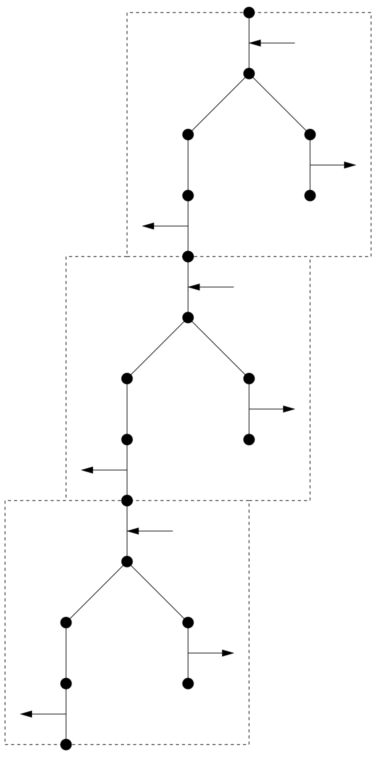


Fig. 15. A 3-cell binary unbalanced component.

that we provide an approach that is more general, and encompasses practical situations where services obey to an internal logic and expose a nondeterministic behavior.

6.2. Complex services

We now consider services that encode multiple interactions, where each interaction may possibly lead to a failure of the service, and such that a large variety of functions may be computed.

6.2.1. Binary unbalanced procedures

We start by analyzing services which generalize the kind of nondeterministic service used in the "simple" test, by encoding a "sequential" pattern of nondeterministic interaction. Each "cell" of the pattern represents one interaction analogous to the one explained in the previous subsection: a request is received, and either a refusal message is given back, or a function f_i is computed, and the service suspends for either an ack or a nack message. As such, each cell may fail (because it refuses computing f_i, or because it receives a *nack*) or succeed. Failure of the cell corresponds to failure of the service; success of the cell, instead, activates the next "cell", and the success of the service corresponds to the success of the last cell. We call such components "binary unbalanced"; the number of their branches grows linearly with the number of cells, and so does the length of the longer possible sequence of query–response interactions, which we name the "depth" of the service. Fig. 15 schematically show a component service with 3 cells.

We first consider the combination of a number of services containing $1, 2, 3, 4$ cells each, where the goal is, once more, to compute a nested function, or to have every component fail. The results are shown in Fig. 16, again distinguishing between time spent for building the search space, time spent for internalizing it, and time spent for searching it.

In general, we observe that, similarly to the case of simple nondeterministic services, (a) the performance of belief-level construction degrades quasi-linearly with the number of the involved services, (b) its internalization and the search degrade polynomially, and (c) the search degrades polynomially (with a higher polynomial factor).

On top of this, we can observe the following:

- Regarding belief-level construction and interpretation, as expected, their costs grow up when the components are more complex: approximately, of one order of magnitude for each additional cell. Similarly to the previous tests, belief-level construction grows up linearly with the number of components, and belief-level internalization grows up polynomially.
- The performance of the search phase of our composition chain depends mainly on the total number of "cells" in the set of components, and quite marginally on whether the cells are glued together in few, large component services,

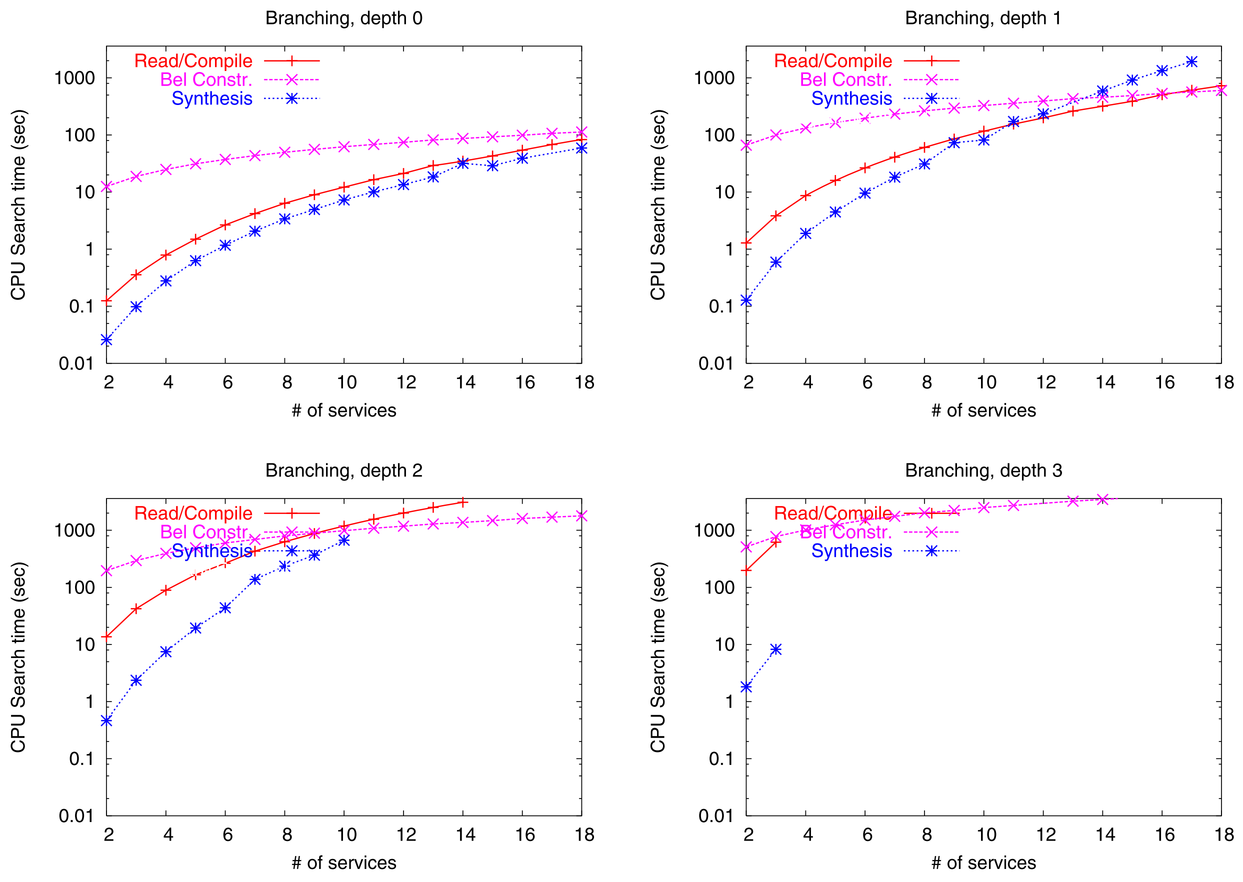


Fig. 16. Combining N binary unbalanced services of depth 1, 2, 3, 4.

or distributed over several, small ones. For instance, we observe that to compose 12 cells, it takes about 10 seconds, regardless of whether we use 12 components with 1 cell each, 6 with two cells, or 4 with three cells. For large numbers of cells, fewer complex services are somehow preferable; this is due to the fact that in this case, the search is more constrained, since a smaller variety of input/output actions can be performed at each interaction step, and the cell ordering inside a component W_i also constrains the ordering of the input/outputs concerning the functions computed by W_i.

- When composing the more complex amongst these services, memory consumption becomes a major issue, mainly due to the breadth-first style adopted by our search algorithm. Indeed, instances where services have depth $d \geqslant 3$, as well as the composition of large sets of services with depth $d = 2$, are terminated not due to timeouts, but due to exceeding the 4 GByte memory bound.

To study in more detail the relationships sketched above, we also analyze the combination of a fixed set of such services, where we vary the average number of cells in the component services. Fig. 17 reports the results for the composition of 2 to 5 services. These graphics show clearly that both the belief-level construction and internalization, and the search times, grow up exponentially with the number of cells in the components. Concerning belief-level construction and internalization, this is due to the fact that the size of the belief-level system of a component is (potentially) exponential in the number of states, and thus in the number of cells. Concerning search, this is due to the fact that the length of the composed service is, in this case, also proportional to the number of cells. Also, we can observe a sort of "step" discontinuity in the curves that refer to belief-level construction and internalization times. This comes from the fact that the time required for the larger component in the set dominates the overall performance in such phases.

6.2.2. Binary balanced procedures

The pattern above is asymmetric, and such that there exists only one successful path for each service. As a consequence, similar to what happens for simple services, the combination of the services fails as soon as one of the involved services fails.

To model a more general situation, where different success paths exist for the combination of services, and to analyze the combination of services with a non-trivial branching factor, we generalize the pattern above into a "balanced binary

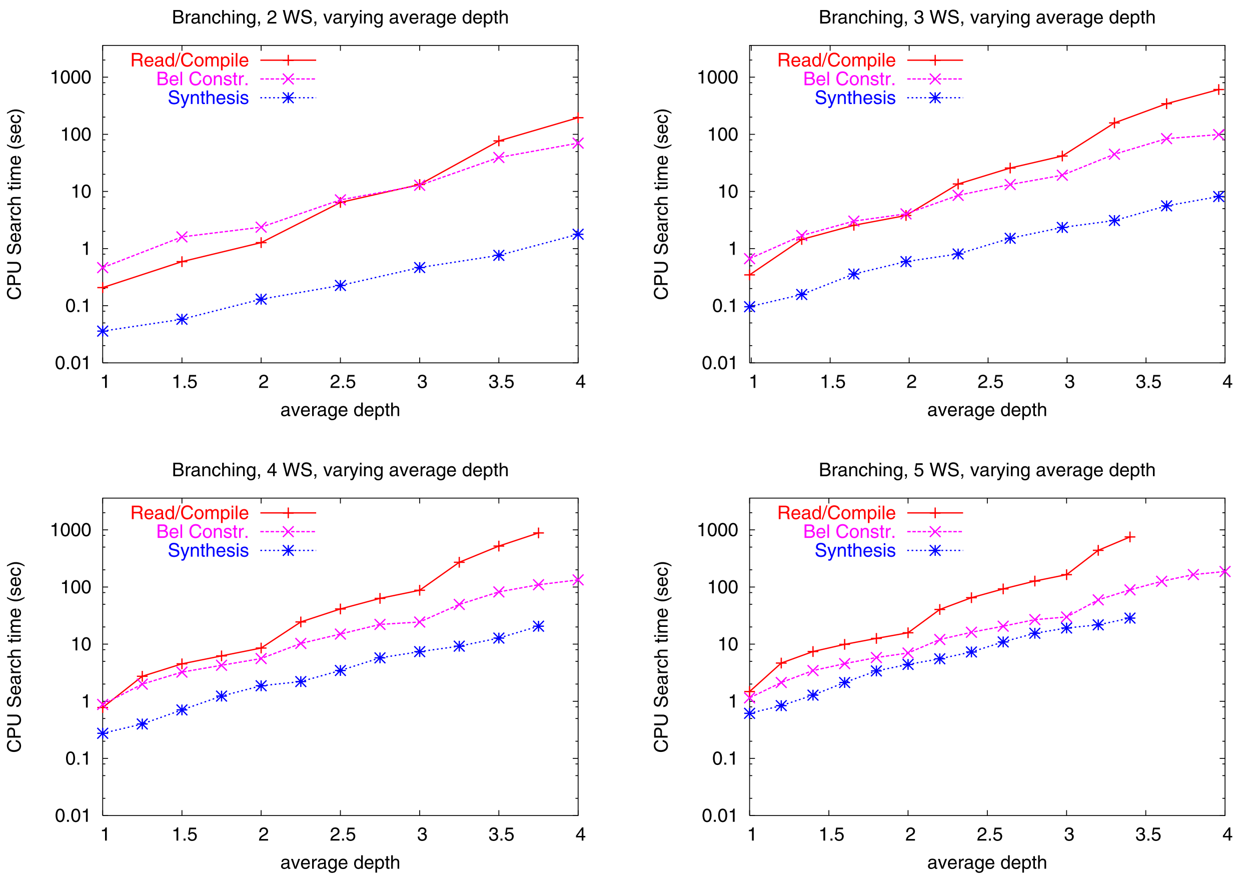


Fig. 17. Combining 2, 3, 4, 5 branching services of increasing depth.

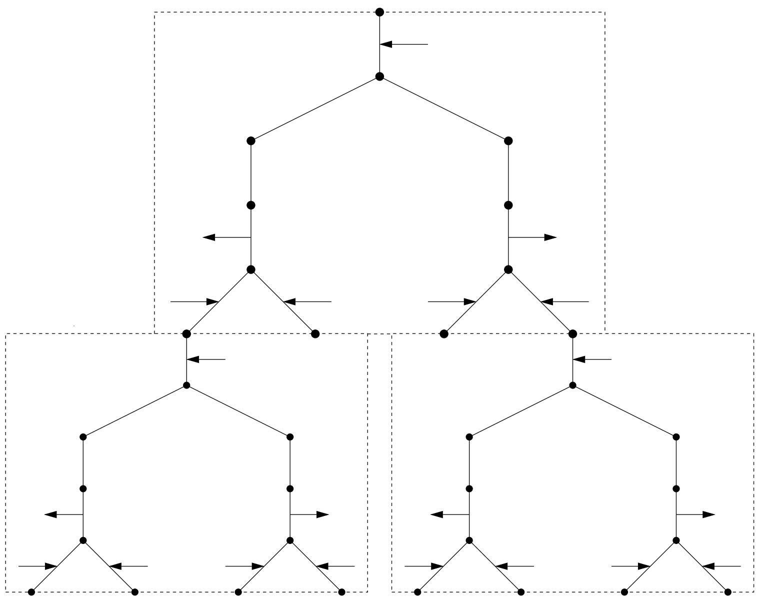


Fig. 18. A 2-depth "balanced branch" component.

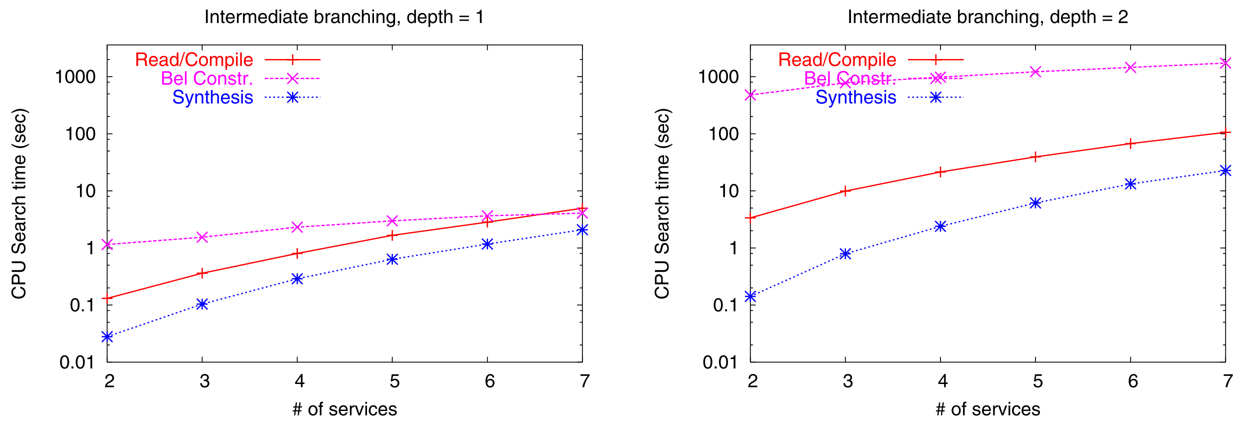


Fig. 19. Combining N binary balanced services of fixed depth $(1, 2)$.

tree" of nondeterministic interaction cells. In this case, an interaction cell computes one of two different functions, sending out two different messages, and in each case awaiting for an ack/nack message to either terminate with success or fail. A balanced tree of such cells is such that the each success state of a cell coincides with the start cell of another one; the success of such a tree is the success of a leaf cell of the tree. Fig. 18 shows a schema of such a balanced component of depth 2.

A goal for a set of such Web services consists in computing one of several possible combinations of functions $f_{i_{0_j}}(f_{i_{1_j}}(\ldots(f_{i_{n_j}}(D))))$ and having every component service succeed, or in sub-order to have every service fail. Again, in terms of finite STS, this goal is compiled into a disjunctive formula that represents the control flow as a union of possible combinations of states of the components, and the data flow as a union of possible combinations of equalities of values with variables and ground function terms. To achieve this goal, the composed service must decide "on the fly" in which order to invoke the component services, since the answer of one service may not only make it impossible to achieve success, but also determine which of the possible combinations of functions can still be pursued.

We first consider combining increasingly large sets of balanced components of a fixed depth. The results are shown in Fig. 19.

Analogously to what we have seen for the composition of unbalanced nondeterministic components, the computational cost of performing the belief-level construction grows up quasi-linearly with the number of the component services, and similarly, the cost of the search seems to grow up only polynomially. The cost of belief-level construction is largely predominant over that of the search unless many services are combined, or services are small — but even for the smaller balanced components possible, the cost of the search becomes significant only with $N > 7$.

Similarly to what we did for unbalanced components, we also combine sets of fixed size of balanced branching components, and we vary the their average depth (i.e. their average number of "cells"). The results are presented in Fig. 20. Again, we witness an exponential dependency of the performance of both belief-level composition and search upon the depth of the component services. The fact the time for belief-level construction grows by steps, being dominated by the one of the larger component, is even more evident in this case, where the size of a component is exponential (rather than linear) in its depth. Indeed, it is just because of the "step" introduced by components of depth 3 that tests fail for larger instances of the problems.

6.2.3. N-ary balanced procedures

Finally, we remark that, while the services considered above generalize the structure of simple ones by combining sets of "interaction cells", they all share the fact that only binary choices are involved: either two input messages can be received, or two different branches can be nondeterministically taken by the service.

We consider a variation in this aspect which generalizes in the structural dimension of width. In particular, we combine two services which implement a short nondeterministic procedure admitting a variety of choices, either because of internal nondeterminism, or because different external controls are possible.

The first service S_1, upon receiving an input datum D, performs an internal choice and decides which function $f_i(D)$, out of a set of n, it will compute. After computing it, it returns the results and suspends for an ack/nack signal that leads it to success or failure states respectively. The second service S_2 may receive one of m different input signals, together with a datum D; depending on which signal did it receive, it computes one of m functions $g_i(D)$, and returns the result. After this, it suspends for an ack/nack signal that leads it to success or failure states respectively. These two services can be composed by requiring that one of the possible function combinations $\bigvee g_i(f_j(D))$ is computed; similarly to the case of balanced branching tree, this requires an "on-the-fly" decision, on the part of the composite service, that depends on the internal determinism of S_1.

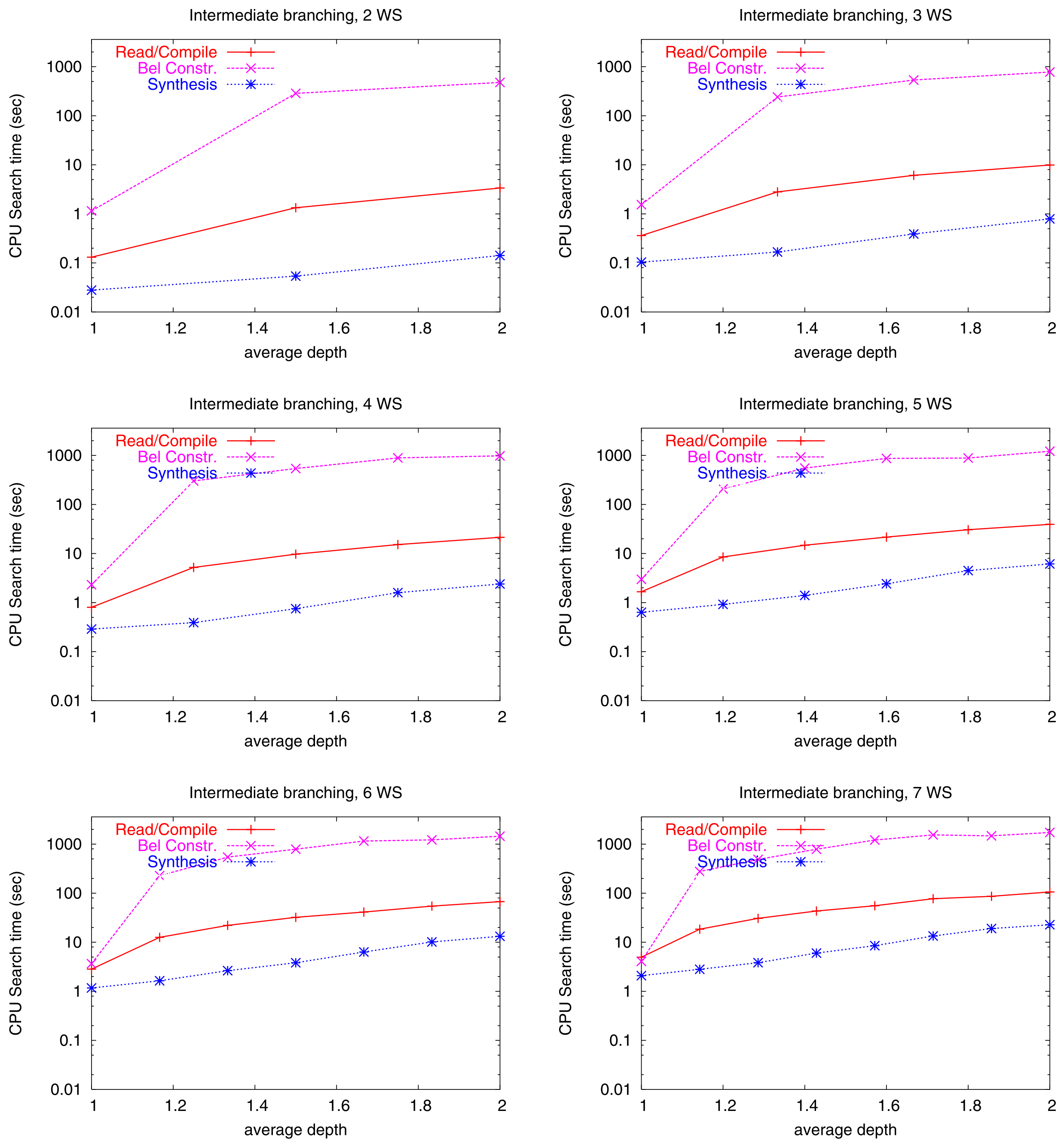


Fig. 20. Combining sets of 2 to 7 balanced branching services of varying average depth.

The results are shown in Fig. 21, and highlight that belief-level construction is the crucial bottleneck for such a kind of composition, growing exponentially with the branching factor and dominating the search.

6.3. *The P&S scenario*

We also tested our architecture considering the P&S scenario introduced in Example 1, considering the WS-BPEL specifications and composition requirements discussed throughout the paper. We remark that the P&S scenario is not symmetric, and therefore cannot be perceived as a direct instantiation of one of the scenarios tested so far. At the same time, some parts of the User, Shipper and Producer components exhibit patterns similar to those featured by the symmetric scenarios. Fig. 22 shows schematically the P&S scenario, highlighting a connection with some patterns from the "binary balanced"

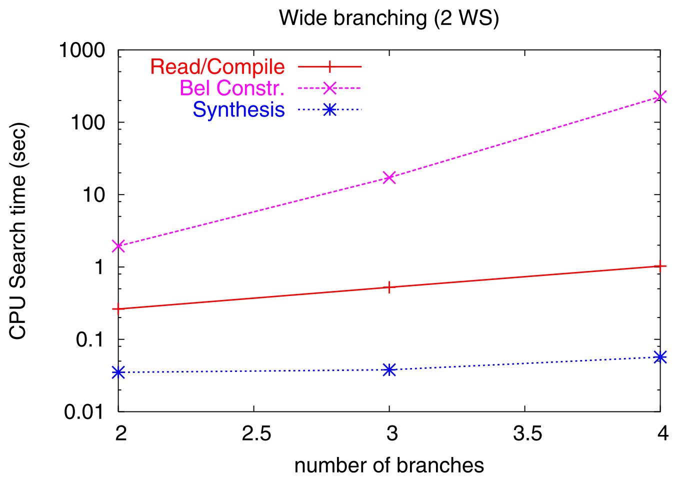


Fig. 21. Combining two branching services of increasing branching factor.

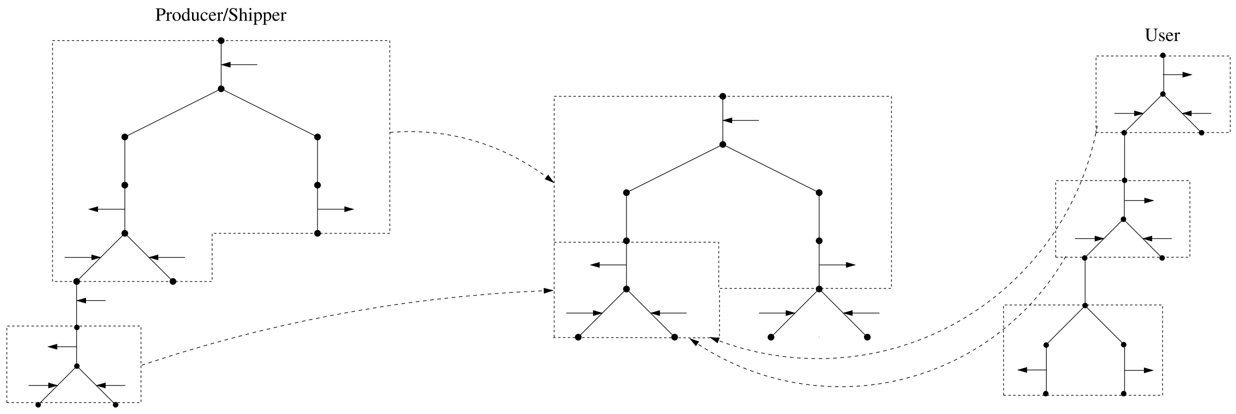


Fig. 22. Schematic comparison of the P&S components and a binary balanced service.

scenario. As one can see, the Producer and Shipper look like a slightly more complex (and not fully symmetric) version of a binary balanced service, whereas the User is fairly different but reuses, twice, a query–response pattern also appearing there.

Our platform is capable to perform the composition for the P&S scenario in a very reasonable time: it takes 0.8 seconds to read the models, 8.4 seconds to perform the belief-level construction, and 1.0 seconds to synthesize the P&S controller. The resulting executable WS-BPEL code appears to be of very reasonable quality and size: as an experiment, we asked an experienced WS-BPEL programmer to hand-code a P&S controller for the scenario, and the resulting code follows the same lines, only reducing the number of internal variables by intelligently re-using them for different purposes at different stages of the service.

6.4. Comparative analysis of results

Given the variety of scenarios adopted in the tests, we find it convenient to provide here a synthetic analysis of our results, considering the impact of the various features of the domains over the performance of the composition task, decomposed in the various phases: belief-level construction, internalization, and search. This will allow us to draw some conclusions on the applicability of our approach to different settings of the composition problem. In particular:

- Construction of the belief-level STS $\text{Bel}(\Sigma_1 \parallel \cdots \parallel \Sigma_n)$ scales up nicely, in a linear way, with the number n of components into play. This is essentially due to our ability to handle the belief-level construction of each $\text{Bel}(\Sigma_i)$ independently, representing their product in terms of modules of the obtained representation. Vice versa, belief-level construction times grow up quasi-exponentially with the size of the components, represented by the number of their states (see e.g. Fig. 17). This is explained by the potentially exponential size of the belief-level counterpart of an STS.

The presence of nondeterminism in the components has only a minor impact, due to the need, during belief-level construction, to consider a larger number of possible transitions, see e.g. Figs. 13 and 14.

- Concerning the internalization phase, which converts the modularized representation of $\text{BEL}(\Sigma_1 \parallel \cdots \parallel \Sigma_n)$ into an internal format of a unique STS, performance is polynomial with the number n of components, as shown e.g. in Fig. 13. Again, size of modules has a stronger impact, and performance degrades quasi-exponentially with it, see Fig. 17. In this phase, the adoption of symbolic representation techniques essentially annihilates the impact of having nondeterminism in the services being composed.
- The performance of synthesis appears to degrade sub-exponentially with the size of the components in all the domains we considered; and in the vast majority of cases, it degrades only polynomially with the number of components. This comes essentially from the breadth-first search style adopted during the search, in conjunction with the fact that it operates on a symbolic representation where whole frontiers of states of $\text{BEL}(\Sigma_1 \parallel \cdots \parallel \Sigma_n)$ are regressed at once. Again, nondeterminism in the components does not seem to have a significant impact on the performance.

The above analysis on the results indicates that the approach lends particularly well to treat large numbers of Web services whose description, taken in isolation, is not particularly complex (but which may embed internal choices, presented in terms of nondeterminism). Dealing with the composition of large Web services may be practically difficult because of the complexity in obtaining their belief-level representation.

More in general, from a bird's eye view at the results over the various scenarios, we can observe that the approach proves capable to effectively tackle significantly complex composition problems, whose ad-hoc solution would have been far from trivial, time-demanding and error prone. For instance, the automated composition of a dozen nondeterministic, branching WS-BPEL services is obtained within a couple of minutes, whereas the manual design and implementation of such orchestration would take several hours. This is also shown by the performance of our platform over the P&S scenario, which improves by large the time a programmer would spend to hard-code the controller. Indeed, such a performance ranks consistently between those of the complex binary scenarios, considering the complexity of the involved components.

A good portion of the effectiveness of our approach must be acknowledged to the symbolic, BDD-based representation used by the algorithms. This confirms that, similarly to what happens in the vast majority of model-checking and diagnosis scenarios, symbolic representations are practically effective in most practical cases, in spite of their exponentially large worse-case resource requirements.

7. Related work

This paper presents a framework for composing asynchronous WS-BPEL processes which is clearly framed within the context of a long-standing research line on service composition using planning techniques, developed at the IRST institute. Such a research line started in 2004 with [71,95], continued with [78,79,74,61,62,60,63,64], and is still evolving to date [72,21]. Indeed, this paper summarizes and significantly extends a large portion of the wealth of work cited above, acting as well as a useful pointer to further possible extensions and studies of the approach.

Among the works cited above, this paper is more closely related with [71,95,78,79,62,64]; however, it significantly differs for technical choices, and for the degree of maturity, which here allows a fully automated end-to-end provision of services expressed in a de-facto standard language such as WS-BPEL. As well, this is the first work that provides a clear, full-fledged and self-contained statement of the underlying theory, and the full proofs for the correctness of the approach.

More in detail, [71,95,78] are to be regarded as preliminary contributions that provided the basis to our approach. In [71, 95], the general framework is presented, but (i) only high-level service descriptions are considered, and (ii) the construction of the belief-level system is not modular. This, together with the fact that the modeling of services as a planning domain is based on an explicit representation of message buffers, leads to rather poor practical performances. In [78], a more formal account of the representation of service composition in terms of planning is presented, but still no technical discussion is reported for what concerns the adopted algorithms, neither in terms of theoretical and practical scalability, nor in terms of the correctness of the approach. We remark that [71,95,78] stand on composition requirements based on the EAGLElanguage [36], whose high expressiveness would lead to extremely complex proofs for such properties.

The above works have been extended in various directions, which are not considered here but can be perceived as relevant pointers where to pursue a full-fledged analysis and formalization. The structuring of requirements into control and data flow sections is the core subject of [62,60], which propose and enact an advanced methodology to requirement specification, based on the idea of "data nets" which link, by means of modular forms of constraints, the inputs and outputs of the services participating to the orchestration. Again, as also reported into the more applicative papers [72,64], the underlying requirements rely on the EAGLEspecification language, and no full algorithmic account and analysis is presented. In [74], service composition is dealt with by modeling services as well as requirements at a more abstract "knowledge" level. This allows describing the relationships between exchanged data at a conceptual level which abstracts away from the need to mention concrete values. While this makes the approach attractive, encoding of the problem in this way results into a non-modular planning domain and has a strong negative impact on the scalability of the performance.

In common with all the works cited above, this work relies on the expressiveness and performance provided by the "planning via model checking" approach (see [16,77,32,15]), adopting its technological baseline and taking inspiration in terms of general framework and approaches to search. In particular, the composition of services requires coordinating nondeterministic and partially observable entities, and is therefore conceptually close to approaches such as [16]. However, the specific nature of service composition (due to the asynchronicity of services, and to the fact that we orchestrate multiple independent entities, rather than control a monolithic domain), has led us to design and develop radically different algorithmic and data representation solutions. Specifically, the solution presented here takes as a starting point the strong planning approach of [32], which in turn is inspired by Schopper's notion of "universal plan" [82] (but makes use of radically different search and representation tools). On top of this, our approach applies very significant extensions which are required to confront with service composition: first, non-trivial pre- and post-processing phases are needed to recast the synthesis of an asynchronous controller into a form of belief-level search, and second, different algorithms are in order, due to the specific semantics of message passing (and deadlock) between services.

Our approach is also related, but significantly different, with the large variety of planning-based approaches that have been proposed to tackle the problem of service composition at various levels. In particular, automated (discovery and) composition of semantic Web services, e.g., based on OWL-S or WSML, is presented in [90,59,43]. A similar line is followed by [100,83], considering DAML-S [6] annotated services instead; DAML-S services are considered as well by [8], which recasts composition as a constraint satisfaction problem, by [65], which adopts an interpreter for an extended version of the ConGolog language to adapt and compose services, and by [67], which provides a framework for simulating, verifying and composing Web services, by translating them into Petri Nets.

Other works have been proposed to support forms of compositions starting from WSDL-like specifications of Web services, see, e.g., [73,84], or based on description in the Universal Service-Semantics Description Language (USDL) developed in the semantic Web community [50]. Similar issues are dealt with by [4], where the composition of data access is tackled by obtaining a workflow and making use of "adapter" services.

All of the above works, be they based on OWL-S, WSML, DAML-S, WSDL or USDL service specifications do not take into account behavioral descriptions of Web services, like our approach does with WS-BPEL. Therefore, they view automated composition as a sequential composition of atomic services. This is also true of the work in [2,29], in spite of the fact that the result of the composition is expressed as WS-BPEL. That is, none of the techniques they adopt can deal with the problem addressed in this paper, where the domain under exam consists of nondeterministic, partially observable and asynchronous component services. Recent extension to works on ConGolog [87], which consider first-order logic specifications of services, are capable to deal with more complex composition requirements, but do not analyze the connections with programming languages for services such as WS-BPEL.

Within the planning field, the need to address requirements beyond reachability is well known and established, and several approaches tackle the problem using different means. Temporal logics such as LTL and CTL [37] are used either to describe planning goals or to constrain the search [53,19,77]. The Hierarchical Task Network approach followed, e.g., in [87] envisages a procedural format to express goals. In [89], procedural and declarative goals are fused to allow better flexibility in expressing goals. While of great potential interest, all these works do not consider the problem of composing Web services. While a general connection between planning and service composition is clear, these works start from significantly different assumptions and the adoption of their approaches in the context of service composition would be far from trivial.

Planning techniques have also 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 [94] by using data integration 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 [17].

In the field of e-services, synthesis techniques have been proposed to solve problems related with the one we tackle here. In particular, the work in [46,42] 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 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.

Concerning the works in [9–11,40,41,86,12] instead, it has to be remarked that they focus on fundamental aspects, more than on application in current service oriented technologies. Moreover, while considering problems which are conceptually related to ours, they take significantly different underlying assumptions and technical choices. In [9,11,41,12], services are described as fully observable entities, and are given a different communication semantics; in [10], the authors consider services of a semantic nature, inspired by the OWL-S specification; in [40,86], the target of the composition is different and consists of a set of de-centralized orchestration entities.

Some different work tackle the composition of stateful services in a semi-automated way; this is the case of [22], that presents an approach stepping through their conversion in terms of YAWL [98] workflows. Again, no details on a possible implementation of the approach are provided, but for a constrained scenario of interface adaptation amongst pairs of processes [23].

Considering a broader perspective, our work is conceptually close to those in the field of automata-based synthesis of controllers, where a given domain (often referred to as the *environment*) can be driven by a controller, and the aim is to automatically synthesize such a controller (see, e.g., [76,97,54,66,5,75]). Indeed, the composed service can be seen as a module that controls an environment which consists of the published services. Most works in the area, however, focus on theoretical foundations and analysis, and, also due to the high theoretical and empirical complexity of the task in an unconstrained setting, do not provide practical implementations which can be useful to the automated compositions of Web services. Some efforts in this sense have been attempted by constraining to selected forms of applications. Amongst these, some work such as [48] start from assumptions, and exploit techniques, radically different from ours, while remarkably, in other cases, e.g., [96,20], the technological baseline is similar to ours, as they introduce approaches where formulas are represented and manipulated by means of Binary Decision Diagrams.

8. Conclusions

Automated composition of Web services constitutes an exciting novel applicative area for planning technologies. While the service-oriented paradigm is strongly pushed by world-level industrial players such as IBM or Oracle, and complex development environments and languages have been provided and are continuously refined, the success of such an approach will strongly depend on the development of powerful automated support functionalities which have lacked so far. Planning has been identified as a first-class candidate by which it is possible to provide such functionalities; still, the particular setting of Web services poses significant challenges that require overcoming the limitations of current planning approaches, both in terms of expressiveness and effectiveness.

In this work, we provided a clear formal modeling of the problem of composing stateful Web services according to complex behavioral requirements. As a result, we provided a framework capable of dealing with significantly difficult composition of services expressed in standard languages such as WS-BPEL. The complexity of this task has been dealt by an appropriate pre-compilation of asynchronous behaviors, and by carefully designed search algorithms which rely on symbolically represented domains. We provide an extensive testing of our approach, witnessing the ability of our system to tackle significantly complex problems, and identifying the impact of various structural aspects of the composition in terms of the performance and outcomes of the composition.

The approach presented here is a crucial step forward to introduce automated support for service composition within the standard development phases of services. The approach stands on clearly defined assumptions, which we introduced in Section 2, concerning the kind of services that can be considered for composition and the kind of composition requirements that can be applied. In turn, this reflects on the kind of orchestrations that can be achieved. As such, this paper clearly lends to several extensions that may improve the expressiveness, flexibility, effectiveness, and ultimately the usability of our approach.

A first important hypothesis underlying our work, as well as essentially all the works on the composition of stateful services developed so far, is that services can be interpreted as finite automata, in spite of being realized by means of a language such as WS-BPEL that allows infinitely-ranged variables. This implies the need to carry out a pre-processing phase that performs an abstraction over component services, by identifying finite ranges such that coordinating the resulting finitely-ranged WS-BPEL components still requires a fully general orchestrator, whose behavior is independent from the considered ranges. Indeed, in all the scenarios we encountered, and in those we test, we found that it is enough to consider each variable as belonging to a binary type, and to have ranges of different types disjoint. We remark that, since our kind of requirements lead to orchestrations whose executions are finite, it is easy to devise a finitization of the components such that their (finite-state) orchestration has a structure independent from the type ranges. Of course, identifying *effective* finitizations, that is finding the smallest type ranges which guarantee a general coordination of some components, is not trivial, and this is a topic for further investigation. This problem of grounding may as well be tackled in a conceptually different way, using abstraction techniques that encode services into finite state machines which do not directly map WS-BPEL operations over (finitely ranged) WS-BPEL variables, but rather define the behaviors of the services at a higher level. In particular, knowledge-level encodings in the style of [70] may be used to represent symbolically the way the variables of services relate to each other, therefore stemming away from the need of mentioning concrete variable values. Some preliminary results using a knowledge-level encoding (different from the one in [70]) have been presented in [74]. A full investigation in this respect, and a combination of knowledge-level representation with the ground-level representation adopted here, are in our agenda.

The second assumption we took concerns the nature of composition requirements. We focused on reachability requirements which specify admissible final configurations of the component services. A first implication of this is that we stick to component services whose execution is guaranteed to terminate in a finite number of steps; that is, we ruled out services that embed some loop which is not guaranteed to exit. A possible and somehow obvious relaxation in this sense consists in having more expressive requirements where infinite behaviors are allowed. For instance, one can consider problems where the only way to interact with components implies some iterative trial-and-error strategy. In this case, even still keeping "reachability" requirements, it is possible to relax the definition of satisfiability to consider those looping situations where a chance of terminating still exists, therefore identifying so-called "strong cyclic" solutions [32]. This is considered in [78, 74,62,60], where services with cyclic behaviors are admitted, insofar they still admit final configurations which are used as targets for the requirements. Under this setting, it is possible to rely on specific "strong cyclic" search algorithms that may produce cyclic trial-and-error strategies which indefinitely keep achieving the target configurations. A further extension may consider composition requirements of a different nature, describing constraints over the evolution in time of components services, and/or considering preferences. For instance, in [71,78], service composition is tackled using EaGLe [77,36], a language which is inspired by the LTL temporal logic [37], but (differently from LTL) allows expressing user preferences amongst subgoals, by handling a notion of "goal failure" and layering subgoals according to different user-defined strengths. Such extensions are especially interesting since services are only partially controllable, and therefore orchestrating them may require dealing with a variety of more or less preferable situations. Indeed, preferences are also tentatively addressed in [78,74,62,60], since the EAGLErequirement language [36] they adopt embeds them.

We also took some assumption on the language used to express services, restricting to a subset of the de-facto WS-BPEL standard which appears sufficient to support the abstract service specifications needed as input to the composition. Nevertheless, we plan to cover a broader portion of such a language. This investigation does not appear particularly compelling under a theoretical perspective, but it may require substantial effort to avoid huge blow-ups of the models when certain specific WS-BPEL features are exploited.

While all the extensions are extremely relevant and clearly pointed out by the assumptions taken in this work, and while some work along those directions has been conducted already, we remark that the form of composition we consider here is already expressive enough to cover a large variety of relevant service composition scenarios. Focusing on this setting allows us to keep the presentation reasonably compact and self-contained, while at the same time comprehensively surveying all the formal and empirical aspects at a sufficient level of detail.

Of course, a further clear direction of work stands in investigating techniques for further improving the scalability. In this sense, devising new requirement languages that constrain service behaviors to follow some easy-to-specify pattern seems a promising option. Finally, we intend to tackle the issue of associating semantics to the activities of stateful Web services, in order to be able to respond to composition requirements specified according to some semantics. In particular, we plan to extend the work to the automated composition of semantic Web services, e.g., described in OWL-S [30] or WSMO [101], along the lines of the work done in [95].

References

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

[2] V. Agarwal, G. Chafle, K. Dasgupta, N. Karnik, A. Kumar, S. Mittal, Biplav B. Srivastava, Synthy: A system for end to end composition of Web services, Journal of Web Semantics: Science, Services and Agents on the World Wide Web 3 (4) (2005) 311–339.

[3] ActiveBPEL, The Open Source BPEL Engine, http://www.activebpel.org.

[4] J.L. Ambite, D. Kapoor, Automatically composing data workflows with relational descriptions and shim services, in: Proc. of ISWC'07, 2007.

[5] E. Asarin, O. Maler, A. Pnueli, J. Sifakis, Controller synthesis for timed automata, in: IFAC Symposium on System Structure and Control, 1998, pp. 469–474.

[6] A. Ankolekar, DAML-S: Web service description for the semantic Web, in: Proc. ISWC'02, 2002.

[7] A. Arnold, Finite Transition Systems: Semantics of Communicating Systems, Prentice Hall International, 1994.

[8] R. Aggarwal, K. Verma, J.A. Miller, W. Milnor, Constraint driven Web service composition in METEOR-S, in: Proc. of SCC'04, IEEE Computer Society, 2004, pp. 23–30.

[9] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, Automatic composition of e-services that export their behaviour, in: Proc. ICSOC'03, 2003.

[10] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, M. Mecella, Automatic composition of transition-based semantic Web services with messaging, in: Proc. of VLDB'05, 2005, pp. 613–624.

[11] D. Berardi, D. Calvanese, G. De Giacomo, M. Mecella, Composition of services with nondeterministic observable behaviour, in: Proc. ICSOC'05, 2005.

[12] D. Berardi, F. Cheikh, G. De Giacomo, F. Patrizi, Automatic service composition via simulation, Int. J. Found. Comput. Sci. 19 (2) (2008) 429–451.

[13] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Information and Computation 98 (2) (June 1992) 142–170.

[14] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, P. Traverso, MBP: a model based planner, in: Proc. of IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information, 2001.

[15] P. Bertoli, A. Cimatti, M. Pistore, P. Traverso, A framework, for planning with extended goals under partial observability, in: Proc. ICAPS'03, 2003.

[16] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Planning in nondeterministic domains under partial observability via symbolic model checking, in: Proc. IJCAI'01, 2001.

[17] J. Blythe, E. Deelman, Y. Gil, Planning for workflow construction and maintenance on the grid, in: Proc. of ICAPS'03 Workshop on Planning for Web Services, 2003.

[18] B. Bonet, H. Geffner, Planning with incomplete information as heuristic search in belief space, in: Proc. of AIPS'00, 2000, pp. 52–61.

[19] F. Bacchus, F. Kabanza, Using temporal logic to express search control knowledge for planning, Artificial Intelligence 116 (1-2) (2000) 123–191.

[20] B. Bonakdarpour, S. Kulkarni, Exploiting symbolic techniques in automated synthesis of distributed programs with large state space, Distributed Computing Systems (2007).

[21] P. Bertoli, R. Kazhamiakin, M. Paolucci, M. Pistore, H. Raik, M. Wagner, Control flow requirements for automated service composition, in: Proc. of ICWS'09, 2009.

[22] A. Brogi, R. Popescu, Towards semi-automated workflow-based aggregation of Web services, in: Proc. ICSOC'05, 2005.

[23] A. Brogi, R. Popescu, Automated generation of BPEL adapters, in: Proc. of ICSOC'06, 2006, pp. 27–39.

[24] F. Bacchus, R. Petrick, Modeling an agents incomplete knowledge during planning and execution, in: Proc. of KR'08, 2008, pp. 432–443.

[25] K. Brace, R. Rudell, R. Bryant, Efficient implementation of a BDD package, in: Proceedings of the 27th ACM/IEEE Conference on Design Automation, 1990, pp. 40–45.

[26] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers C-35 (8) (1986) 677–691.

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

[28] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Service Definition Language (WSDL 1.1), http://www.w3.org/TR/wsdl, 2001.

[29] G. Chafle, G. Das, K. Dasgupta, A. Kumar, S. Mittal, S. Mukherjea, B. Srivastava, An integrated development environment for Web service composition, in: Proc. of ICWS'07, 2007.

[30] OWL-S Coalition, OWL-S: Semantic markup for Web services, Technical White paper (OWL-S version 1.0), 2003.

[31] OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee, Web Services Business Process Execution Language Version 2.0 – OASIS Standard, 2007.

[32] A. Cimatti, M. Pistore, M. Roveri, P. Traverso, Weak, strong, and strong cyclic planning via symbolic model checking, Artificial Intelligence 147 (1–2) (2003) 35–84.

[33] M.D. Denz, Charting the evolving eHealth ecosystem, in: Proc. of eHealth'08, 2008.
[34] D. Mc Dermott, Estimated-regression planning for interactions with Web services, in: Proc. AIPS'02, 2002, pp. 204–211.
[35] A. Dogac, G.B. Laleci, S. Kirbas, Y. Kabak, S. Sinir, A. Yildiz, Y. Gurcan, Artemis: deploying semantically enriched Web services in the healthcare domain, Information Systems 31 (4–5) (2006) 321–339.
[36] U. Dal Lago, M. Pistore, P. Traverso, Planning with a language for extended goals, in: Proc. AAAI'02, 2002.
[37] E.A. Emerson, Temporal and modal logic, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, Elsevier, 1990.
[38] R.E. Fikes, N.J. Nilsson, STRIPS: a new approach to theorem proving in problem solving, Journal of Artificial Intelligence 2 (1971) 189–208.
[39] H. Foster, S. Uchitel, J. Magee, J. Kramer, Model-based verification of Web service compositions, in: Proc. ASE'03, 2003.
[40] G. De Giacomo, M. de Leoni, M. Mecella, F. Patrizi, Automatic workflows composition of mobile services, in: Proc. of ICWS'07, 2007, pp. 823–830.
[41] G. De Giacomo, S. Sardina, Automatic synthesis of new behaviors from a library of available behaviors, in: Proc. of IJCAI'07, 2007, pp. 1866–1871.
[42] R. Hull, M. Benedikt, V. Christophides, J. Su, E-services: a look behind the curtain, in: Proc. PODS'03, 2003.
[43] J. Hoffmann, P. Bertoli, M. Pistore, Web service composition as planning, revisited: In between background theories and initial state uncertainty, in: Proc. of AAAI'07, 2007.
[44] J. Hoffmann, B. Nebel, The FF planning system: fast plan generation through heuristic search, Journal of Artificial Intelligence Research 14 (2001) 253–302.
[45] G. Holzmann, The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, November 2003, preview available at http://safari.ibmpressbooks.com/0321228626/app01?close=0.
[46] R. Hull, Web services composition: a story of models, automata, and logics, in: Proc. of ICWS'05, 2005.
[47] B. Jobstmann, R. Bloem, Optimizations for LTL synthesis, in: Proc. of Formal Methods in Computer Aided Design (FMCAD'06), 2006, pp. 117–124.
[48] B. Jobstmann, S. Galler, M. Weiglhofer, R. Bloem, Anzu: a tool for property synthesis, in: Proc. of CAV'07, 2007, pp. 258–262.
[49] R. Kazhamiakin, Formal analysis of Web service compositions, PhD thesis, International Doctorate School in ICT – DIT University of Trento, Trento, Italy, March 2007.
[50] S. Kona, A. Bansal, G. Gupta, Automatic composition of semantic Web services, in: Proc. of ICWS'07, 2007.
[51] T. Zeng, K. Kim, C.J. Bonk, Surveying the future of workplace e-learning: the rise of blending, interactivity, and authentic learning, eLearn Magazine 2005 (6) (2005), available at http://www.elearnmag.org/.
[52] W.B. Korte, General practitioner use of ICT and eHealth in Europe 2002–2007 – Results from a pan-European survey, in: Proc. of eHealth'08, 2008.
[53] F. Kabanza, S. Thiebaux, Search control in planning for temporally extended goals, in: Proc. of 15th International Conference on Automated Planning and Scheduling (ICAPS-05), 2005, pp. 130–139.
[54] O. Kupferman, M. Vardi, Synthesis with incomplete information, in: Proc. of ICTL'97, 1997.
[55] S.M. Lee, T. Hwang, J. Kim, An analysis of diversity in electronic commerce research, International Journal of Electronic Commerce 12 (1) (2007).
[56] S.S. Lam, A.U. Shankar, A relational notation for state transition systems, IEEE Transactions on Software Engineering 16 (7) (1990) 755–775.
[57] A. Marconi, Automated process-level composition of Web services: From requirements specification to process run, PhD thesis, International Doctorate School in ICT – DIT University of Trento, Trento, Italy, April 2008.
[58] K.L. McMillan, Symbolic Model Checking, Kluwer Academic Publ., 1993.
[59] S. McIlraith, R. Fadel, Planning with complex actions, in: Proc. NMR'02, 2002.
[60] A. Marconi, M. Pistore, P. Poccianti, P. Traverso, Automated Web service composition at work: the Amazon/MPS case study, in: Proc. ICWS'07, 2007.
[61] A. Marconi, M. Pistore, P. Traverso, Implicit vs. explicit data-flow requirements in Web service composition goals, in: Proc. ICSOC'06, 2006.
[62] A. Marconi, M. Pistore, P. Traverso, Specifying data-flow requirements for the automated composition of Web services, in: Proc. SEFM'06, 2006.
[63] A. Marconi, M. Pistore, P. Traverso, Process-level composition of Web services: a semi-automated iterative approach, Annals of Mathematics, Computing and Teleinformatics 1 (5) (2007).
[64] A. Marconi, M. Pistore, P. Traverso, Automated composition of Web services: the ASTRO approach, IEEE Data Eng. Bull. 31 (2008).
[65] S. McIlraith, S. Son, Adapting Golog for composition of semantic Web services, in: Proc. KR'02, 2002.
[66] R.v.D. Meyden, M. Vardi, Synthesis from knowledge-based specifications, in: Proc. of CONCUR'98, 1998, pp. 34–49.
[67] S. Narayanan, S. McIlraith, Simulation, Verification and automated composition of Web services, in: Proc. WWW'02, 2002.
[68] G. Nemirovskij, M. Wolters, E. Heuel, Distributed study: a semantic Web services approach for modelling a common educational space, in: Proc. of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2008, 2008.
[69] Oracle, Oracle BPEL Process Manager, http://www.oracle.com/products/ias/bpel/.
[70] R. Petrick, F. Bacchus, A knowledge-based approach, to planning with incomplete information and sensing, in: Proc. AIPS'02, 2002.
[71] M. Pistore, P. Bertoli, F. Barbon, D. Shaparau, P. Traverso, Planning and monitoring Web service composition, in: Proc. AIMSA'04, 2004.
[72] M. Pistore, P. Braghieri, P. Bertoli, A. Biscaglia, A. Marconi, S. Pintarelli, M. Trainotti, At Your Service: Service-Oriented Computing from an EU Perspective, MIT Press, 2009 (Chapter 8).
[73] S. Ponnekanti, A. Fox, SWORD: A developer toolkit for Web service composition, in: Proc. WWW'02, 2002.
[74] M. Pistore, A. Marconi, P. Bertoli, P. Traverso, Automated composition of Web services by planning at the knowledge level, in: Proc. of IJCAI'05, 2005.
[75] N. Piterman, A. Pnueli, Y. Saar, Synthesis of reactive designs, in: Proc. of VMCAI, 2006, pp. 364–380.
[76] A. Pnueli, R. Rosner, On the synthesis of an asynchronous reactive module, in: Proc. ICALP'89, 1989.
[77] M. Pistore, P. Traverso, Planning as model checking for extended goals, in: Proc. IJCAI'01, 2001.
[78] M. Pistore, P. Traverso, P. Bertoli, Automated composition of Web services by planning in asynchronous domains, in: Proc. ICAPS'05, 2005.
[79] M. Pistore, P. Traverso, P. Bertoli, A. Marconi, Automated synthesis of composite BPEL4WS Web services, in: Proc. ICWS'05, 2005.
[80] R. Ruffolo, The ultimate e-commerce software shopping list, Computer World on Line, available at http://www.computerworld.com.au/, 2008.
[81] T. Son, C. Baral, Formalizing sensing actions: a transition function based approach, Artif. Intell. 125 (1-2) (2001) 19–91.
[82] M. Schopper, Universal plans for robots in unpredictable environments, in: Proc. of 10th International Joint Conference on Artificial Intelligence (IJCAI'87), 1987, pp. 1039–1046.
[83] M. Sheshagiri, M. desJardins, T. Finin, A planner for composing services described in DAML-S, in: Proc. AAMAS'03, 2003.
[84] D. Skogan, R. Gronmo, I. Solheim, Web service composition in UML, in: Proc. EDOC'04, 2004.
[85] F. Somenzi, CUDD: CU decision diagram package release 2.3.1, available at http://vlsi.colorado.edu/fabio/CUDD/, 2001.
[86] S. Sardina, F. Patrizi, G. De Giacomo, Automatic synthesis of a global behavior from multiple distributed behaviors, in: Proc. of AAAI'07, 2007, pp. 1063–1069.
[87] S. Sohrabi, N. Prokoshyna, S. McIlraith, Web service composition via generic procedures and customizing user preferences, in: Proc. of ISWC'06, 2006, pp. 597–611.
[88] A. Schumann, Y. Pencolè, S. Thièbaux, Diagnosis of discrete event systems using binary decision diagrams, in: Proc. of the 15th International Workshop on Principles of Diagnosis (DX'04), 2004.
[89] D. Shaparau, M. Pistore, P. Traverso, Fusing procedural and declarative planning goals for nondeterministic domains, in: Proc. of AAAI'08, 2008, pp. 983–990.

[90] E. Sirin, B. Parsia, D. Wu, J. Hendler, D. Nau, HTN planning for Web service composition using SHOP2, Journal of Web Semantics 1 (4) (2004) 377–396.
[91] J. Sanghavi, R. Ranjan, R. Brayton, A. Sangiovanni-Vincentelli, High performance BDD package by exploiting memory hierarchy, in: Proceedings of the 33rd annual conference on Design automation, 1996, pp. 635–640.
[92] E. Sakkopoulos, E. Sourla, A. Tsakalidis, M.D. Lytras, Integrated system for eHealth advisory Web services provision using broadband networks, International Journal of Social and Humanistic Computing 1 (1) (2008) 36–52.
[93] L. Tuan, C. Baral, X. Zhang, T. Son, Regression with respect to sensing actions and partial states, in: Proc. of AAAI'04, 2004, pp. 556–561.
[94] S. Thakkar, C. Knoblock, J.L. Ambite, A view integration approach to dynamic composition of Web services, in: Proc. of ICAPS'03 Workshop on Planning for Web Services, 2003.
[95] P. Traverso, M. Pistore, Automated composition of semantic Web services into executable processes, in: Proc. ISWC'04, 2004.
[96] E. Tronci, Automatic synthesis of controllers from formal specifications, in: Proc. of Second IEEE International Conference on Formal Engineering Methods (ICFEM'98), 1998.
[97] M.Y. Vardi, An automata-theoretic approach to fair realizability and synthesis, in: Proc. CAV'95, 1995.
[98] W.M.P. van der Aalst, A.H.M. ter Hofstede, YAWL: yet another workflow language, Information Systems 30 (4) (2005) 245–275.
[99] G. Vossen, P. Westerkamp, E-learning as a Web service, in: Proc. of IDEAS03, 2003.
[100] D. Wu, B. Parsia, E. Sirin, J. Hendler, D. Nau, Automating DAML-S Web services composition using SHOP2, in: Proc. ISWC'03, 2003.
[101] WSMO, The Web service modeling framework, SDK WSMO working group, http://www.wsmo.org/, 2004.