Specifying Data-Flow Requirements for the Automated Composition of Web Services*

Annapaola Marconi
ITC-Irst - Trento - Italy
marconi@itc.it

Marco Pistore
University of Trento - Italy
pistore@dit.unitn.it

Paolo Traverso
ITC-Irst - Trento - Italy
traverso@itc.it

Abstract

One of the fundamental ideas of Web Services and Service Oriented Architecture is the possibility to develop new applications by composing existing services that are available on the Web. Several approaches have been proposed to tackle the problem of Web Service Composition, but little effort has been devoted so far to the problem of modeling the requirements of the composition. However, it is clear that the possibility to express requirements specifying complex interactions patterns among the component services is an essential step to cope with a wide range of composition problems. In this paper we present a new model which addresses one of the key aspects of composition requirements, namely the data flow among the component services. We develop graphical notations and a formal theory for the new model and we integrate it within an existing automated composition framework.

1 Introduction

Web services are platform-independent applications that export a description of their functionalities and are accessible using standard network technologies. Web services are able to perform a wide spectrum of activities, from simple receive-response protocols to complicated business workflows, and provide the basis for the development and execution of business processes that are distributed over the network. One of the fundamental ideas of Web services is the possibility to develop new applications by composing existing services that are available on the Web. The manual development of the new composite service is often a difficult and error-prone task, because human domain experts have to take care of all the contingencies that can happen during the service execution process. The ability to support the composition of Web services with automated and semi-automated tools is an essential step to decrease time and costs in the development, integration, and maintenance of complex services.

Several methods have been proposed to cope with the automated composition problem (see e.g. [10, 9, 11, 13, 7, 2]). In all those methods, the existing component services are used to define a domain to be controlled, composition requirements specify the desired behaviors that should be enforced on the domain, and the composition task corresponds to synthesize a controller for the domain that realized the desired behavior. It is widely recognized that, in general, automated synthesis is a hard problem, both in theory and in practice (see, e.g. [15]). In the case of Web service composition, however, automated synthesis turns out to be feasible not only in principle, but also in practice on realistic domains of significant complexity. Indeed, in this setting the synthesized controller has "simply" to act as an orchestrator which controls and directs the execution of the existing services, but delegates to the services the other computational tasks.

One of the key aspects of Web service composition is how to derive and model composition requirements. It is clear that, in order to cope with a wide range of real composition problems, we need a way to express complex requirements on the exchanged data and on the executions of the component services. Moreover, to make the automated composition an effective and practical task, the requirements specification should be easy to write and to understand for the analyst. Surprisingly, very little effort has been devoted in the literature to address this problem.

In this paper we present a new model which addresses one of the key aspects of composition requirements, namely the data flow among the component services. We propose to specify the requirements on the data flow through a set of constraints that define the valid routings and manipulations of messages that the orchestrator can perform. These constraints can be described in a graphical way, as a data net, i.e., as a graph where the input/output ports of the existing services are modeled as nodes, the paths in the graph define the possible routes of the messages, and the arcs de-

*This work is partially funded by the MIUR-FIRB project RBNE0195K5 "KLASE", by the MIUR-PRIN 2004 project "STRAP", and by the EU-IST project FP6-016004 "SENSORIA".

fine basic manipulations of these messages performed by the orchestrator. In the paper, we develop the formal definition and the graphical notations of data nets. Moreover, we show how to integrate data flow requirements expressed as data nets within an existing automated composition framework [13, 14, 12]. Finally, we show that the proposed approach can handle realistic composition problems of significant complexity.

The rest of the document is organized as follows. In Section 2 we describe the problem of Web service composition and we illustrate the necessity to define complex data flow requirements. In Section 3 we introduce data nets and we formally define their semantics by describing the requirements that a data net defines on the possible behaviors of a Web service composition. In Section 4 we discuss how data nets can be integrated in the automated composition framework proposed in [13, 14, 12]. Finally, Section 5 concludes the paper with related work and final remarks.

2 Web Service Composition: A Scenario

By automated composition of Web services we mean the generation of a new composite service that interacts with a set of existing component services in order to satisfy given composition requirements. More specifically, we will assume that component services are described as BPEL4WS processes.[1] Given the descriptions of the component processes and the composition requirements, we automatically generate a new BPEL4WS process implementing the required composition.

In this section we illustrate on a case study the problem of the automated composition of Web services. We will focus in particular on the problem of specifying the requirements of such a composition.

Example 1 (Virtual Travel Agency) *Our reference example consists in providing a virtual travel agency service, say the VTA service, which offers holiday packages to potential customers, by combining three separate existing services: a flight booking service Flight, a hotel booking service Hotel, and a service that provides maps AllMaps. The idea is that of combining these three services so that the customer may directly interact with the composed service VTA to organize and possibly book his holiday package.*

In the following, we describe informally the three available services, whose interaction protocols are depicted in Figure 1[2]*. Hotel accepts requests for providing information on available hotels for a given date and a given location. If there are hotels available, it chooses a particular hotel and return an offer with a cost and other hotel information. This offer can be accepted or refused by the external service that has invoked the Hotel. In case of refusal, the requester can either request an offer for a different hotel, or terminate the interaction with the Hotel. The Flight service receives requests for booking flights for a given date and location. If there are available flights, it sends an offer with a cost and a flight schedule. The client can either accept or refuse the offer. If he decides to accept, the Flight will book the flight and provide additional information such as an electronic ticket. The AllMaps service receives requests with two locations and provides a digital map depicting distance information.*

Intuitively, the VTA service should try to satisfy a given customer request by providing information on available flights and hotels (e.g., holiday cost, flight schedule, hotel description and a map showing distance from the airport) and book the holiday according to customer final decision. Figure 2 describes a possible protocol that the VTA could expose to the customer. According to it, the customer sends a request for an holiday, then, if there is an available flight, it receives a flight offer. If the customer agrees on the flight schedule and cost and there is an available hotel, he receives an hotel offer consisting of the hotel cost, the distance of the hotel from the airport and other information about the hotel. The customer can either decide to accept the offer, or to ask for a different hotel offer, or to terminate the interaction with the VTA. If he decides to accept, he receives the booking confirmation with the overall holiday cost and other information about the chosen hotel and flight.

Given the description of the component services and of the customer interface, the next step towards the definition of the automated composition domain is the formal specification of the composition requirements. As we will see from the examples presented in the rest of this section, even for simple case studies we need a way to express requirements that define complex conditions, both for what concerns the control flow and for the data exchanged among the component services.

Example 2 (Control-flow requirements) *The VTA service main goal is to "sell holiday packages". This means we want the VTA to reach a situation where the customer has accepted the offer and a flight and a hotel have been booked. However, it may be the case that there are no available flights (or no available hotels) satisfying the customer*

[1] BPEL4WS (Business Process Execution Language for Web Services) [1] is an industrial language for the specification and execution of business processes made available thorough Web services. In this paper we assume that component and composite services are expressed in BPEL4WS. However, the described approach does not depend on the specific aspects of the BPEL4WS language. The paper should be understandable also for readers who are not familiar with BPEL4WS.

[2] In the figure, labels of input transitions start with a "?", labels of output transitions start with a "!", other transitions correspond to internal operations performed by the services.

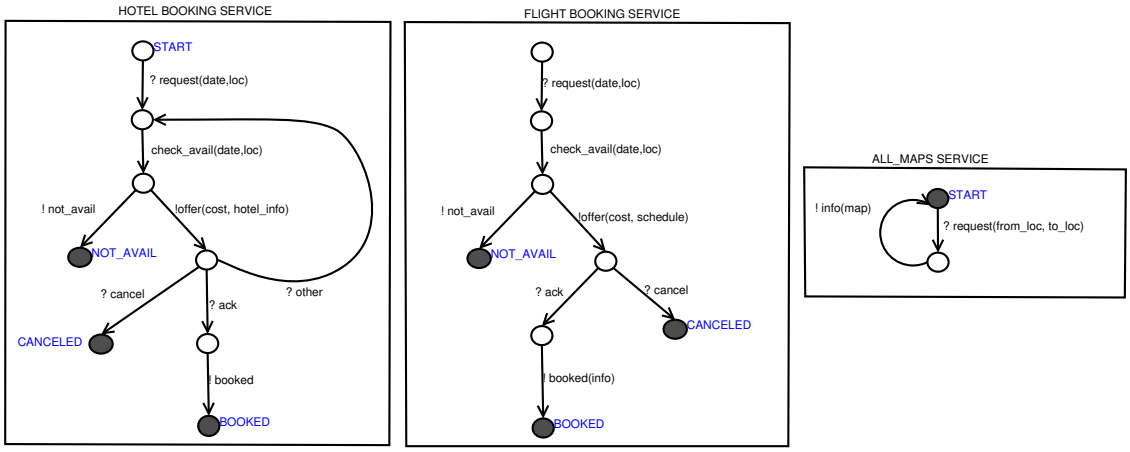


Figure 1. The Virtual Travel Agency Component Services

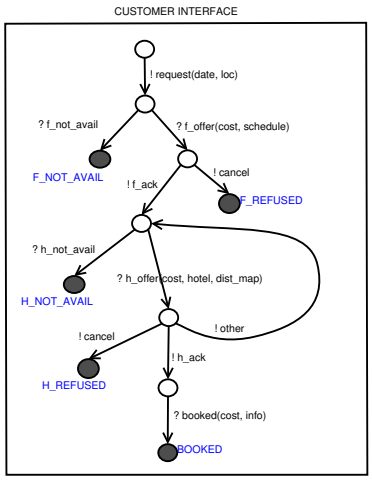


Figure 2. The VTA Customer Interface

request, or that the customer doesn't like the flight or the hotel offer and thus cancels the booking. We cannot avoid these situations, therefore we cannot ask the composite service to guarantee this requirement. In case this requirement cannot be satisfied, we do not want the VTA to book a flight (nor a hotel) without being sure that our customer accepted the offer, as well as we do not want displeased customers that have booked holidays for which there are no available flights or hotels. Thus, our global termination requirement would be something like: do whatever is possible to "sell holiday packages" and if something goes wrong guarantee that there are "no single commitments".

This termination requirement is only a partial specification of the constraints that the composition should satisfy. Indeed, we need to specify also complex requirements on the data flow.

Example 3 (Data-flow requirements) *In order to provide consistent information, the VTA service needs to exchange data with the components and its customer in an appropriate way. For instance, when invoking the Flight service, the information about the location and date of the flight must be the same ones that the VTA received in the customer request; similarly, the information sent to the customer about the distance between the proposed hotel and the airport must be those obtained from the last interaction with the AllMaps service; and such a service must receive the information on the airport and hotel location according to the last offer proposed by the Hotel service. Moreover, the cost proposed to the customer for the holiday package must be the sum of the hotel and flight cost plus some additional fee for the travel agency service; thus the cost offered to the customer must be computed by means of a function internal to the VTA service. And so on.*

The example shows that, even for apparently simple composition problems, we need a way to express complex data flow requirements: from simple data links between incoming and outgoing message parts (e.g., forwarding the information received by the customer about the location to the Flight service), to the specification of complex data manipulation (e.g., when computing the holiday package cost), to more subtle requirements concerning data (e.g., all the time the VTA invokes the AllMaps service it must send the location information of the last hotel offer received, while the same airport location information can be used more than once).

The research challenge we address in this paper is the definition of a modeling language that allows to capture data-flow composition requirements for realistic composition problems.

3 Modeling Data Flow Requirements

The aim of the data flow modeling language is to allow for the specification of complex requirements concerning data manipulation and exchange. In particular, data flow requirements specify how output messages (messages sent to component services) are obtained from input messages (messages received from component services). This includes several important aspects: whether an input message can be used several times or just once, how several input messages must be combined to obtain an output message, whether all messages received must be processed and sent, etc..

3.1 Syntax

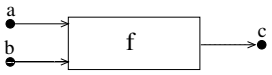
In the following we describe the basic elements of the language, show how they can be composed to obtain complex expressions and provide an intuitive semantics.

- *Connection Node*

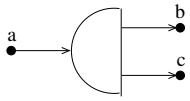
A *connection node* can be external or internal. An external connection node is associated to an output (or an input) external port. Intuitively, an external input (output) node characterizes an external source (target) of data and it is used to exchange data with the outside world.

- *Identity*
  It is connected to one connection node in input and one node in output. The requirement states that data received from the input node should be forwarded to the output node. The graphical notation for the data-flow identity element `id(a)(b)`, with input node `a` and output node `b`, is the following:

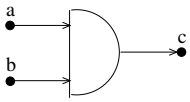
- *Operation*
  It is related to a function definition; it is connected to as many input nodes as the number of function parameters and only to one output node corresponding to the function result. The requirement states that, when data is received from all the input nodes, the result of the operation should be forwarded to the output node. The graphical notation for the data-flow operation element `oper[f](a,b)(c)` characterizing function `f`, with input nodes `a` and `b` and output node `c`, is the following:



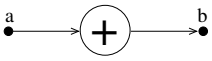
- *Fork*
  It is connected to a node in input and to as many nodes as necessary in output. It forwards data received on the input node to all the output nodes. The graphical notation for the data-flow fork element `fork(a)(b,c)`, with input node `a` and output nodes `b` and `c`, is the following:



- *Merge*
  It is connected to one node in output and as many nodes as necessary in input. It forwards data received on some input node to the output node. It preserves the temporal order of data arriving on input nodes (if it receives data on two or more input nodes at the same time, the order is nondeterministic). We represent the data-flow merge element `merge(a,b)(c)`, with input nodes `a` and `b` and output node `c` as:



- *Cloner*
  It is connected to one node in input and one node in output. It forwards, one or more times, data received from the input node to the output node. The data-flow cloner element `clone(a)(b)`, with input node `a` and output node `b` is represented as:



- *Filter*
  It is connected to one node in input and one node in output. When it receives data on the input node, it either forwards it to the output node or discards it. We represent the data-flow filter element `filt(a)(b)`, having input node `a` and output node `b` as:

- *Last*
  It is connected to one node in input and one node in output. It requires that at most one data is forwarded to the output node: the last data received on the input

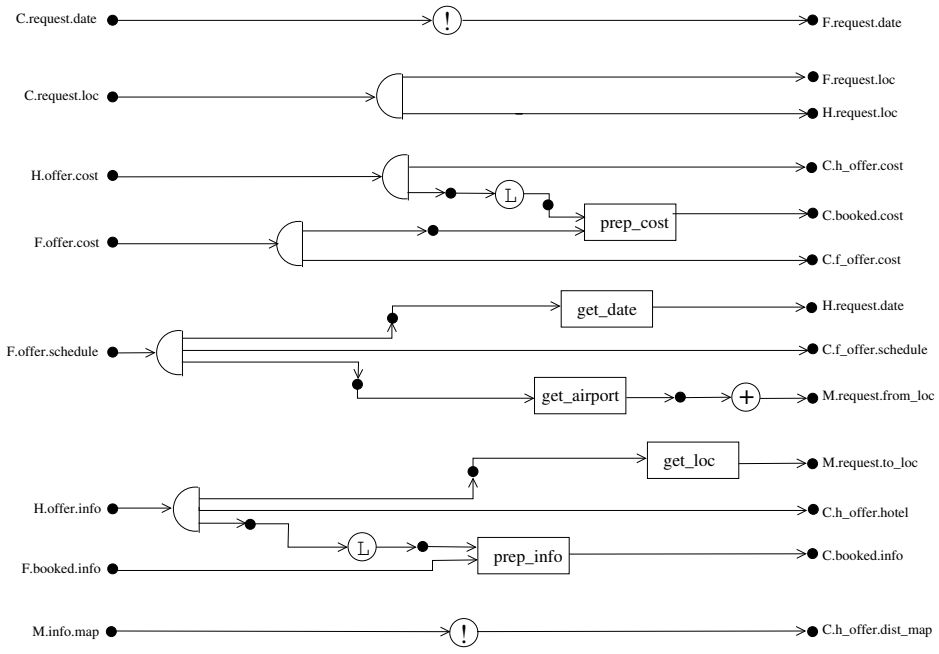
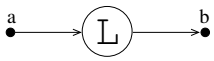


Figure 3. The data flow requirements for the Virtual Travel Agency

node. All other data that are received should be discarded. The graphical notation for the data-flow last element `last(a)(b)`, with input node `a` and output node `b`, is the following:



The diagram obtained by suitably composing data-flow elements by means of connection nodes is called *data net* (see Figure 3 for an example). A data net is characterized by a set of external connection nodes associated to input ports N^I_{ext}, a set of external connection nodes associated to output ports N^O_{ext}, a set of internal connection nodes N_{int}, a set of data-flow elements D (corresponding to the basic elements described in this section) and a set of data values V. Given a data-flow element d, we denote with $in\_nodes(d)$ the set of input connection nodes of d and with $out\_nodes(d)$ the set of output connection nodes of d.

Definition 1 (Data Net)
A data net $\mathcal{D}$ *is a tuple* $\langle N^I_{ext}, N^O_{ext}, N_{int}, D, V\rangle$ *where:*

- *for each* $n \in N^I_{ext}$ *there exists a unique data-flow element* $d \in D$ *s.t.* $n \in in\_nodes(d)$*;*
- *for each* $n \in N^O_{ext}$ *there exists a unique data-flow element* $d \in D$ *s.t.* $n \in out\_nodes(d)$*;*
- *for each* $n \in N_{int}$ *there exists a unique data-flow element* $d_1 \in D$ *s.t.* $n \in in\_nodes(d_1)$ *and there exists a unique data-flow element* $d_2 \in D$ *s.t.* $n \in out\_nodes(d_2)$*;*
- *for each* $d \in D$, $in\_nodes(d) \subseteq N^I_{ext} \cup N_{int}$ *and* $out\_nodes(d) \subseteq N^O_{ext} \cup N_{int}$.

Notice that it is possible to associate a type to each connection node in the network. Indeed, external nodes inherit the types from the corresponding BPEL4WS ports, and the types of internal nodes can be deduced from the structure of the data net. We do not consider this aspect to make the formalization more understandable; completing the model to handle typed connection nodes is straightforward.

A possible specification of the data net for the Virtual Travel Agency example is presented in Figure 3, which we will (partially) explain in the following example.

Example 4 *When the VTA receives a request from the Customer, it must forward the date information to the Flight and the loc information both to the Flight and to the Hotel.*

To obtain the cost to be sent in the offer to the Customer, the VTA must apply its internal function prep_cost on the cost received in the offer from the Flight and on the cost information in the last offer received from the Hotel.

The VTA must obtain the date information that it sends in the request to the Hotel by computing its internal function get_date on the schedule received in the offer of the Flight. The schedule received in the offer of the Flight is also forwarded to the client, as part of the f_offer message. Finally, the VTA exploits the internal function get_airport

on the flight schedule *to obtain the* from_loc *information to be sent to the* AllMaps*; the* VTA *can use this same information to send several* request*s to the* AllMaps*.*

And so on.

3.2 Semantics

We now formalize the semantics of the data flow modeling language. Given a data net $\mathcal{D} = \langle N^I_{ext}, N^O_{ext}, N_{int}, D, V\rangle$, we denote with N_{ext} the sets of all external connection nodes, formally $N_{ext} = N^I_{ext} \cup N^O_{ext}$. An *event* e of $\mathcal{D}$ is a couple $\langle n, v\rangle$, where $n \in N_{ext} \cup N_{int}$, and $v \in V$, which models the fact that the data value v passes through the connection node n. An *execution* ρ of $\mathcal{D}$ is a finite sequence of events $e_0, ..., e_n$. Given an execution ρ we define its projection on a set of connection nodes $N \subseteq N_{ext} \cup N_{int}$, and denote it with $\Pi_N(\rho)$, the ordered sequence $e'_0, ..e'_m$ representing the events in ρ which correspond to nodes in N.

We formally define the semantics of our language in terms of *accepted executions* of a data net $\mathcal{D}$. In the following definition, we exploit regular expressions to define the accepted execution. We use notation $\Sigma_{v\in V}$ to express alternatives that range over all the possible values $v \in V$ that can flow thorough the net.

Definition 2 (data net accepting execution)
An execution ρ is accepted by a data net $\mathcal{D} = \langle N^I_{ext}, N^O_{ext}, N_{int}, D, V\rangle$ if it satisfies all the following properties:

- *for each identity element* `id(a) (b)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v\in V} \langle a, v\rangle \cdot \langle b, v\rangle\right)^*$$

- *for each operation element* `oper[f](a,b) (c)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b,c\}}(\rho) = \left(\sum_{v,w\in V} (\langle a, v\rangle \cdot \langle b, w\rangle + \langle b, w\rangle \cdot \langle a, v\rangle) \cdot \langle c, f(v, w)\rangle\right)^*$$

- *for each fork element* `fork(a) (b,c)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b,c\}}(\rho) = \left(\sum_{v\in V} \langle a, v\rangle \cdot (\langle b, v\rangle \cdot \langle c, v\rangle + \langle c, v\rangle \cdot \langle b, v\rangle)\right)^*$$

- *for each merge element* `merge(a,b) (c)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b,c\}}(\rho) = \left(\sum_{v\in V} (\langle a, v\rangle \cdot \langle c, v\rangle + \langle b, v\rangle \cdot \langle c, v\rangle)\right)^*$$

- *for each cloner element* `clone(a) (b)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v\in V} \langle a, v\rangle \cdot \langle b, v\rangle \cdot \langle b, v\rangle^*\right)^*$$

- *for each filter element* `filt(a) (b)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v\in V} \langle a, v\rangle \cdot (\langle b, v\rangle + \epsilon)\right)^*$$

- *for each last element* `last(a) (b)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v\in V} \langle a, v\rangle\right)^* \cdot \left(\sum_{v\in V} \langle a, v\rangle \cdot \langle b, v\rangle\right) + \epsilon$$

Notice that this definition considers data net elements having at most two input/output nodes, however it can easily be extended to handle elements of the data net having more input/output nodes.

3.3 Data Net Satisfiability

A data net can be used to specify the desired behavior of a service or everything that concerns the exchange of data with its communication partners. In particular, as shown in the data net of Figure 3, external connection nodes are associated to input (or output) ports which model BPEL4WS messages, or message parts, which are used to store data received (or sent) by the process while interacting with other services.

Since the behavioral aspect we are interested in concerns the data flow among the process and its partners, we characterize an execution of a BPEL4WS process W, denoted with $exec(W)$, as the set of all possible ordered sequence of input/output messages (or message parts) received and sent by the process from its activation to its termination. Notice that each message carries both the information about the external port on which it has been sent/received and about its content (value).

Definition 3 (data net satisfiability)
Let W be a BPEL4WS *process and $\mathcal{D} = \langle N^I_{ext}, N^O_{ext}, N_{int}, D, V\rangle$ a data net. We say that W* satisfies *$\mathcal{D}$ if for each process execution $\rho_W \in exec(W)$ there exists an accepting execution ρ of $\mathcal{D}$ such that $\Pi_{N_{ext}}(\rho) = \rho_W$.*

4 Automated Composition of Web Services

In this section we show how data nets can be integrated in a general framework for the automated composition of Web services.

4.1 An Automated Composition Framework

The work in [13] (see also [14, 12]) presents a formal framework for the automated composition of Web services which is based on planning techniques: component services define the planning domain, composition requirements are formalized as a planning goal, and planning algorithms are used to generate the composite service. The framework of [13] differs from other planning frameworks since it assumes an asynchronous, message-based interaction between the domain (encoding the component services) and the plan (encoding the composite service). We now recall the most relevant features of the framework defined in [13].

The composition domain is modeled as a *state transition system* (STS from now on) which describes a dynamic system that can be in one of its possible *states* (some of which are marked as *initial states* and/or as *final states*) and can evolve to new states as a result of performing some *actions*. Actions are distinguished in *input actions*, which represent the reception of messages, *output actions*, which represent messages sent to external services, and *internal actions*, which represent internal evolutions that are not visible to external services, i.e., data computation that the system performs without interacting with external services. A *transition relation* describes how the state can evolve on the basis of inputs, outputs, or internal actions.

Definition 4 (state transition system (STS))
A state transition system Σ *is a tuple* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \mathcal{R}, \mathcal{F} \rangle$ *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;*
- $\mathcal{A}$ *is the finite set of internal actions;*
- $\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \mathcal{A}) \times \mathcal{S}$ *is the transition relation;*
- $\mathcal{F} \subseteq \mathcal{S}$ *is the set of final states.*

We have defined a translation that associates a state transition system to each component service, starting from its BPEL4WS specification. We omit the formal definition of the translation, which can be found at `http://astroproject.org`. Intuitively, input actions of the STS represent messages received from the component services, output actions are messages sent to the component services, internal actions model assignments and other operations which do not involve communications, and the transition relation models the evolution of the service. Examples of the STS representation of BPEL4WS component services can be found in Figure 1.

The automated synthesis problem consists in generating a state transition system Σ_c that, once connected to Σ, satisfies the composition requirements. We now recall the definition of the state transition system describing the behavior of Σ when connected to Σ_c.

Definition 5 (Controlled system)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \mathcal{R}, \mathcal{F} \rangle$ *and* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{I}_c, \mathcal{O}_c, \mathcal{A}, \mathcal{R}_c, \mathcal{F}_c \rangle$ *be two state transition systems such that* $\mathcal{I} = \mathcal{O}_c$ *and* $\mathcal{O} = \mathcal{I}_c$. *The state transition system* $\Sigma_c \triangleright \Sigma$, *describing the behaviors of system* Σ *when controlled by* Σ_c, *is defined as follows:*

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

where:

$$\langle (s_c, s), a, (s_c', s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R}), \text{ if}$$
$$\langle s_c, a, s_c' \rangle \in \mathcal{R}_c \text{ and } \langle s, a, s' \rangle \in \mathcal{R}$$

In an automated synthesis problem, the composition requirements are formalized as a specification ρ, and the composition task consists in generating a Σ_c that guarantees that the controlled system $\Sigma_c \triangleright \Sigma$ satisfies the requirement ρ, written $\Sigma_c \triangleright \Sigma \models \rho$. In [13], ρ is formalized using EAGLE, a requirement language which allows to specify conditions of different strengths (like "try" and "do"), and preferences among different (e.g., primary and secondary) requirements. EAGLE operators are similar to CTL operators, but their semantics, formally defined in [6], takes into account the notion of preference and the handling of failure when subgoals cannot be achieved.

Example 5 *The* EAGLE *formalization of the control-flow requirements in Example 2 is the following.*

TryReach
C.BOOKED $\wedge$ *F.BOOKED* $\wedge$ *H.BOOKED* $\wedge$ *M.SUCC*
Fail DoReach
(*C.F_NOT_AVAIL* $\vee$ *C.F_REFUSED* $\vee$
C.H_NOT_AVAIL $\vee$ *C.H_REFUSED*) $\wedge$
(*H.NOT_AVAIL* $\vee$ *H.CANCELED* $\vee$ *H.START*) $\wedge$
(*F.NOT_AVAIL* $\vee$ *F.CANCELED*) $\wedge$
(*M.START*)

*The goal is of the form "**TryReach** c **Fail DoReach** d". **TryReach** c requires a service that tries to reach condition*

c, in our case the condition "sell holiday packages". During the execution of the service, a state may be reached from which it is not possible to reach c, e.g., since the product is not available. When such a state is reached, the requirement ***TryReach*** *c fails and the recovery condition* ***DoReach*** *d, in our case "no single commitments" is considered.*

The definition of whether ρ is satisfied is given in terms of the executions that $\Sigma_c \triangleright \Sigma$ can perform. Given this, we can characterize formally an automated synthesis problem.

Definition 6 (Automated Synthesis)
Let Σ be a state transition system, and let ρ be an EAGLE *formula defining a composition requirement. The automated synthesis problem for Σ and ρ is the problem of finding a state transition system Σ_c such that*

$$\Sigma_c \triangleright \Sigma \models \rho.$$

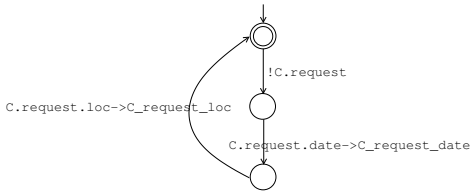
The work in [13] shows how to adapt to this task the "Planning as Model Checking" approach, which is able to deal with large nondeterministic domains and with requirements expressed in EAGLE. It exploit powerful BDD-based techniques developed for Symbolic Model Checking to efficiently explore domain Σ during the construction of Σ_c.

4.2 Data Requirements as STSs

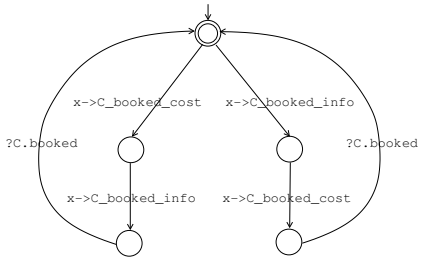
As we have seen in previous sections, a data net $\mathcal{D}$ of a particular composition problem specifies how messages received from the component services can be used by the new composite process to generate outgoing messages. Therefore, it is possible to represent $\mathcal{D}$ as a STS $\Sigma_{\mathcal{D}}$, which models the allowed data flow actions. In particular, input actions in $\Sigma_{\mathcal{D}}$ represent messages received by the component services, output actions represent messages sent by the component services and internal actions represent assignments that the composite process performs on its internal variables.

We assume that, in the BPEL4WS specification of the composite service, a variable will exist for each connection node in $\mathcal{D}$; variables associated to external connection nodes are those used by the new composite process to store received messages and to prepare the messages to be sent, while variables associated to internal connection nodes are those used to manipulate messages by means of internal functions and assignments. Then $\Sigma_{\mathcal{D}}$ defines constraints on the possible operations that the composite process can perform on these variables. A nice feature of our approach is that this can be done compositionally, i.e., a "small" automaton can be associated to each element of the data net, and STS $\Sigma_{\mathcal{D}}$ is obtained as the product of all these small automata.

More precisely, for each output operation of a component service, which is associated to some external input port in the data net, we define a STS which represents the sending of the message (as an output action) and the storing of all message parts (as internal actions). As an example, considering the VTA composition problem, for the output operation C.request with message parts date and loc we define the following STS:[3]

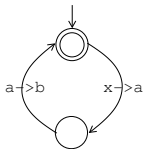


Similarly, for each input operation of a component service, which is associated to some external output port in the data net, we define a STS which represents the storing of all message parts (as internal actions) and the reception of the message (as an input action). As an example, for the input operation C.booked with message parts info and cost we define the following STS:



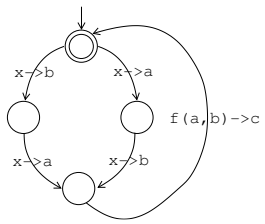
Finally, we define a STS for each data-flow element of the data net. These STSs have no input/ouput actions since they model manipulation of variables through assignments. In particular:

- for each identity element `id(a)(b)` in the data net we define the following STS:

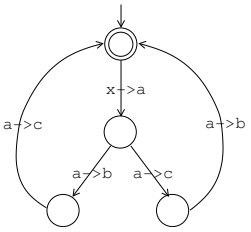


- for each operation element `oper[f](a,b)(c)` in the data net we define the following STS:

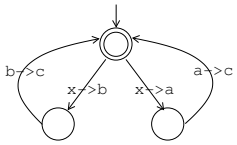
[3] In the STS that we use to model data net requirements, we represent input operations by a `?` followed by the operation name, output actions by a `!` followed by the operation name, while internal actions, denoted with `a->b`, model an internal operation that copies the value of `a` in variable `b`. We use `x` as a place-holder for arbitrary nodes/expression: so for instance `x->a` denotes all internal actions copying any variable/expression to variable `a`, and similarly for `a->x`. Final states are marked with an internal circle.



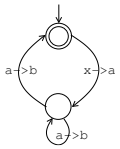
- for each fork element `fork(a)(b,c)` in the data net we define the following STS:



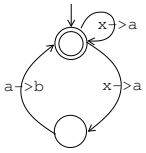
- for each merge element `merge(a,b)(c)` in the data net we define the following STS:



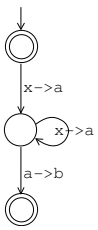
- for each cloner element `clone(a)(b)` in the data net we define the following STS:



- for each filter element `filt(a)(b)` in the data net we define the following STS:



- for each last element `last(a)(b)` in the data net we define the following STS:



The STS $\Sigma_{\mathcal{D}}$ modeling the data net $\mathcal{D}$ is the synchronized product of all the STSs corresponding to external connection nodes and to data-flow elements of $\mathcal{D}$. The synchronized product $\Sigma_1||\Sigma_2$ models the fact that the systems Σ_1 and Σ_2 evolve simultaneously on common actions and independently on actions belonging to a single system.

4.3 Generating the Composite Process

We are ready to show how we can integrate the proposed composition approach within the automated composition framework presented in Section 4.1. Given n component services $W_1, ..., W_n$ and a data net $\mathcal{D}$ modeling the data-flow composition requirements, we encode each component service W_i as a STS Σ_{W_i} and the data net $\mathcal{D}$ as a STS $\Sigma_{\mathcal{D}}$. The composition domain Σ for the automated composition problem is the synchronized product of all these STSs. Formally, $\Sigma = \Sigma_{\mathcal{D}} \parallel \Sigma_{W_1} \parallel .. \parallel \Sigma_{W_n}$. The planning goal ρ is the EAGLE formalization of the composition termination requirements, enriched with the requirements that all the data flow STS need to terminate in a final state.

Given the domain Σ and the planning goal ρ we can apply the approach presented in [13] to generate a controller Σ_c, which is such that $\Sigma_c \triangleright \Sigma \models \rho$. Once the state transition system Σ_c has been generated, it is translated into BPEL4WS to obtain the new process which implements the required composition. The translation is conceptually simple; intuitively, input actions in Σ_c model the receiving of a message from a component service, output actions in Σ_c model the sending of a message to a component service, internal actions model manipulation of data by means expressions and assignments.

5 Conclusions and Related Work

In the paper we have described a new model for defining data flow requirements for the automated composition of Web services. Its interesting features are the possibility to exploit easy to understand graphical notations and to model the requirements in a data net, as well as the possibility to integrate the requirements in a general framework for the automated composition of Web services. Future work will include the implementation of a graphical tool for drawing the requirements and its inclusion in the ASTRO toolset (`http://www.astroproject.org`) and its experimental evaluation.

Several methodologies have been proposed to model different aspects of requirements for service oriented applications, from goal-oriented approaches, see, e.g., [8, 5], to UML-based object-oriented methodologies, see, e.g., [17]. In these works, high-level requirements are used to guide and construct by hand the composition. The problem of the automated synthesis of compositions is not addressed.

These methodologies can be integrated with our approach and used in a pre-analysis step to guide the analyst to the definition of the data net that we use to generate automatically the composition.

Most of the works that address the problem of the automated synthesis of process-level compositions do not take into account data flow specifications. This is the case of the work on synthesis based on automata theory that is proposed in [7, 2, 3], and of work within the semantic web community, see, e.g., [10]. Some other approaches, see, e.g., [16], are limited to simple composition problems, where component services are either atomic and/or deterministic.

The work closest to ours is the one described in [4], which proposes an approach to service aggregation that takes into account data flow requirements. The main difference is that data flow requirements in [4] are much simpler and at a lower level than in our framework, since they express direct identity routings of data among processes, and do not allow for manipulations of data. The examples reported in this paper clearly show the need for expressing manipulations in data-flow requirements and higher level requirements.

References

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

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

[3] D. Berardi, D. Calvanese, G. D. Giacomo, and M. Mecella. Composition of Services with Nondeterministic Observable Behaviour. In *Proc. ICSOC'05*, 2005.

[4] A. Brogi and R. Popescu. Towards Semi-automated Workflow-Based Aggregation of Web Services. In *Proc. ICSOC'05*, 2005.

[5] E. Colombo, J. Mylopolous, and P. Spoletini. Modeling and Analyzing Context-Aware Compositions of Services. In *Proc. ICSOC'05*, 2005.

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

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

[8] D. Lau and J. Mylopoulos. Designing Web Services with Tropos. In *Proc. ICWS'04*, 2004.

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

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

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

[12] M. Pistore, A. Marconi, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI'05*, 2005.

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

[14] M. Pistore, P. Traverso, P. Bertoli, and A.Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *Proc. ICWS'05*, 2005.

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

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

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