2009 IEEE International Conference on Web Services

Control Flow Requirements for Automated Service Composition

Piergiorgio Bertoli*, Raman Kazhamiakin*, Massimo Paolucci†, Marco Pistore*, Heorhi Raik* and Matthias Wagner†
*FBK-Irst, via Sommarive 18, 38050, Trento, Italy
Email: [bertoli,raman,pistore,raik]@fbk.eu
†DoCoMo Euro-Labs, Landsberger Strasse 312, 80687 Munich, Germany
Email: [paolucci,wagner]@docomolab-euro.com

Abstract

Automated composition of services is a key functionality for the adoption of the service-oriented development paradigm. Solving this problem in practice requires the ability to consider asynchronous stateful services and to express complex composition requirements which may span different phases of the life-cycle of component services. In this paper we present a novel automated service composition approach which addresses these challenges by associating so-called objects *to services, and by introducing a simple yet powerful notation to express composition requirements on them. We recast this view of the problem as a specific form of planning; our experiments on a prototype implementation witness the ability of our approach to deal with realistic scenarios and requirements that cannot be tackled by other current approaches.*

1. Introduction

In recent years, significant advances in web service technology and standards have enabled a wide adoption of service-oriented applications. One of the key ideas underlying the service-oriented paradigm is that of allowing the combination of existing services, to obtain new services that satisfy some given requirements and goals. A wide range of methodologies, languages, and approaches has been developed in order to facilitate and support this activity.

In particular, this has led to significant research efforts and progress in *automated* service composition, due to its potential to significantly cut development time and costs. Most current approaches consider services as atomic, stateless, and synchronous entities [1]–[5]; as such, in essence, they produce orchestrations which simply schedule such entities appropriately. For instance, in the travel domain, where the aim is to provide a combined booking service by composing existing Hotel and a Flight reservation services, this boils down to execute them in order, routing data between them and the customer.

In most real-life scenarios, however, services are *stateful*, realizing complex protocols (e.g., a multi-phase booking procedure includes search, selection, and checkout tasks); their behavior may be *non-deterministic* (the search *may* provide no result, checkout *may* fail), and they may exchange messages *asynchronously*. This makes the composition problem significantly more complex, and requires specific ways to manage these features, which are dealt with (in some cases partially) only by few approaches [6]–[8].

A further source of complexity raises from the fact that the simple requirements adopted by current approaches (i.e., asking that certain outputs are produced or certain service states are reached) are not enough. Indeed, in many scenarios the composition goals cannot be stated only in terms of some desired final outcome: there is a need to align intermediate states of service evolution, and to define how the composed service should react to events at different execution stages. For instance, in the travel scenario, the possible delay or deletion of flights make it necessary to keep the hotel and flight reservations consistent also after the booking, e.g. allowing a user to change or delete the hotel reservation if a flight delay or cancellation is reported.

Finally, there is not a one-to-one connection between a process managing some conceptual entity and its service realization. Often, a service only manages a particular task or activity within a broader process. For example, flight management may involve three distinct services that handle flight booking, flight status notification and flight cancellation. At the same time, in many domains the same activity may be accomplished using different services, made available by different providers and adopting different representations and protocols. Given this, it is hard, if at all possible, to express composition requirements in terms of the states or outcomes of a particular service.

This indicates that there is a need for a composition framework including more sophisticated and expressive requirement notations and models that separate composition requirements from service implementation details, so to (i) handle services that express stateful, non-deterministic asynchronous behaviors, (ii) allow defining complex composition requirements going beyond mere reachability, and (iii) capture composition problems in a natural and intuitive, implementation-independent way. This paper presents a novel automated service composition approach that addresses those challenges by:

- introducing and formalizing the notion of *domain objects*, relating them to service operations, and using them to express the composition problem and the domain;
- providing a simple yet expressive notation for defining control flow requirements over the service composition, in terms of the evolution of domain objects, encompass-

978-0-7695-3709-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICWS.2009.31

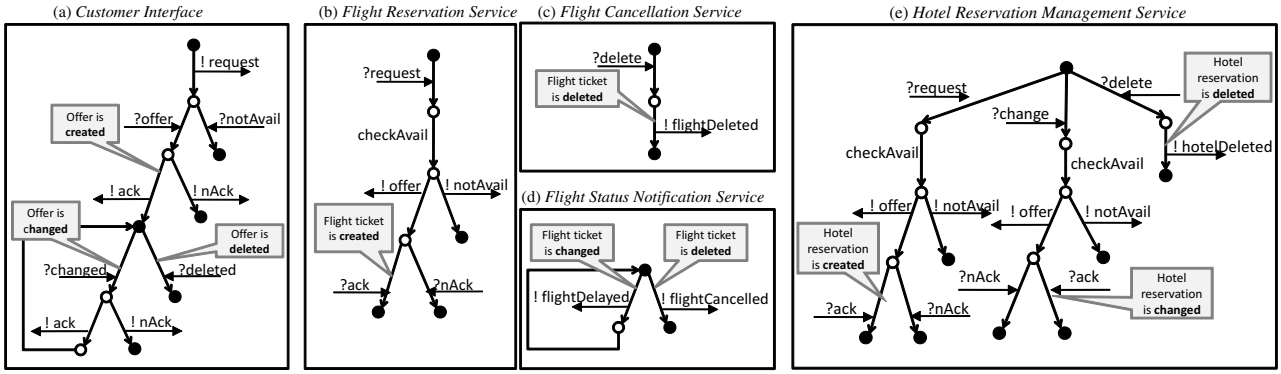


Fig. 1. Component services in the travel domain scenario

ing user preferences and going beyond reachability;

- presenting a formal framework for the automated composition of services, which allows producing executable services according to the above-mentioned object-driven requirements.

The paper is organized as follows. The next two sections present a motivating example and introduce the overall approach at a conceptual level. Section 4 presents some formal background on the underlying asynchronous planning framework, allowing us to define our modelling notations in Section 5. Then, in Section 6 we spell out the details of our automated service composition approach, also describing some experimental results. Finally, we discuss related work and provide some concluding remarks.

2. Motivating Example

In order to illustrate the need for a new approach to service composition we use a variant of a well-known travel domain scenario, which appears in various forms in the literature on service composition for demonstration purposes. In such a scenario, we aim to provide a composed service that can deliver to the users and manage travel packages consisting of flight tickets and hotel reservations.

Fig. 1 represents the five stateful services involved in our version of the scenario, whose actual implementation can be carried out in a standard service description language such as e.g. BPEL[1].

In particular, Fig. 1 (a) shows the expected customer interface, which allows the user to book a compound travel package (if available), and then to receive information on its modifications (e.g., in case of the flight delays) or cancellation. Figure 1 (b,c) represent the services for booking a flight ticket and for cancelling it respectively. The service depicted in Fig. 1 (d) shows a flight status notification service, which sends notifications about the flight delays or cancellations. Finally, Fig. 1 (e) represents a unique protocol managing in full an hotel reservation, i.e., able to create, modify, and delete it.

The problem of providing a composed service in this scenario is complicated by three main factors.

First, the involved services have a sophisticated behavior: they are stateful and non-deterministic, and message interactions are asynchronous.

Second, our aim is to cover the whole process of the travel package reservation and management. This means not only to book the hotel and flight reservation, but also to align further flight modifications with the modification of the hotel reservation and of the travel package. More precisely, we aim to build here a composed process which should satisfy the following goals:

1) It should provide a way to obtain a travel package based on the flight and hotel reservations. This should be done transactionally, i.e., the hotel shall not be booked if the flight is not available and vice versa.
2) If the flight is delayed, the composed service has to provide a way to modify the hotel reservation (and the resulting offer) as well. If this is not possible, the reservations should be cancelled.
3) If the flight is cancelled, the other reservations should be cancelled as well.

We note that such requirements cannot be expressed in terms of finally achieving some state or outcome. Requirement (1) asks reaching an intermediate state, where further events and actions may still take place, while (2,3) define reaction rules that the process should perform in order to handle specific events. Moreover, requirements (1,2) express potential alternatives that should be considered due non-determinism of the services, and a preference order among them (booking is preferred to non-booking, and modification to cancellation). Thus, an appropriate formalism is needed in order to represent and manage such requirements.

Finally, the example shows that a single service may not cover all aspects of a conceptual process. While this is the case for the hotel service, the flight services perform only particular operations of the flight reservation management.

1. In the graphical representation, input and output operations are prepended by "?" and "!" respectively, and for the sake of space we abstract away from details related to the data manipulated by the services.

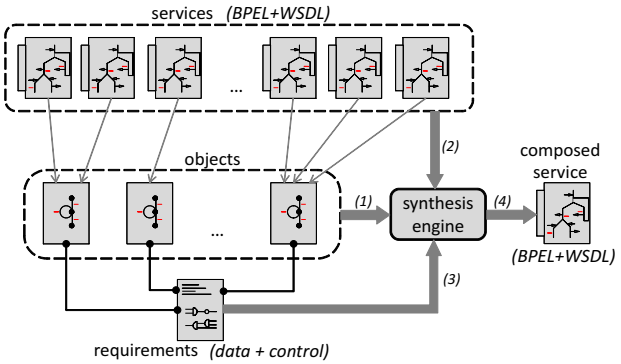


Fig. 2. Composition framework

Also, alternative hotel booking implementations could also be split over a set of services. Therefore, expressing the above composition requirements in terms of service actions and states may be difficult, and any service implementation modification would also force re-thinking the requirements.

These three issues make even the most expressive current approaches inadequate to tackle this scenario, and call for a novel solution, conceptually presented in the next section.

3. Description of the Approach

Our automated service composition approach is schematically represented in Fig. 2, and revolves around the central notion of a *domain object*, which we introduce to explicitly model the key elements of the composition problem and their evolution (e.g., in our example, the hotel reservation, the flight ticket, and the travel package). Objects may have a complex life-cycle; e.g., in our scenario they may be created, modified, and deleted. The idea is that, while activities performed by component services may make objects evolve, the modeling of the objects does not depend on a particular service implementation. This makes it natural to express control-flow composition requirements in terms of the domain objects and their evolution.

Specifically, we model the evolution of an object, which includes its creation and deletion as well as reactions to specific actions (e.g., flight cancellation), with a state diagram, which defines possible object states and transitions between them. The transitions correspond to the activities that can be performed over the object (e.g., flight reservation or cancellation) and to the external events affecting the state of the object (e.g., flight delay).

To link objects to services, we allow service activities to be annotated with elements of the object diagrams, implicitly defining a mapping between the execution of service operations and the evolution of objects (e.g., for our example, we will consider the annotations appearing inside the boxes in Fig. 1). We note that in this way, it becomes easy to modify a scenario to account for different service implementations: it is enough that services are annotated appropriately, while it is not necessary to affect object models nor requirements on them.

Indeed, given objects and labeled services, control-flow composition requirements are defined on top of the object state diagrams. Namely, we use object states and events to represent both the tasks to be performed over the objects (e.g., create both flight ticket and hotel reservation), and to specify coordination requirements defining a consistent evolution of sets of related objects (e.g., delete hotel reservation in case of flight cancellation). A side-remark is in order. In general, as pointed out e.g. in [9], also data flow requirements must be considered, to define data dependencies between various activities. For the sake of space, in this paper we focus on modelling control flow requirements, and embed the approach of [9] for data-flow specifications, omitting a specific discussion on them.

Once domain objects and composition requirements are specified and component services are annotated, the specifications are converted into a formal representation which is passed to a synthesis engine. Such engine automatically identifies and generates a composite service that satisfies the composition requirements by orchestrating the component services. In particular, the engine is based on the asynchronous planning framework of [7]; therefore, the next section will be devoted to some background on such framework, prior to stepping into our novel modelling notations and formalisms, and to discussing their translation in terms of asynchronous planning.

4. Background

Our approach to service composition builds upon the composition framework presented in [7], based on planning in asynchronous domains. There, component services define a planning domain, composition requirements are formalized as a planning goal, and advanced planning algorithms are used to generate the composite service. Differently from other current approaches, such a framework assumes an asynchronous communication model, and provides the ability to deal with stateful and non-deterministic services, considering preference-based (reachability) requirements on services. As such, it provides a good basis to build upon; in the following, we discuss its key concepts.

Formally, a planning domain is defined as a state transition system, 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 *accepting 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 action τ, modelling internal computations and decisions.

Definition 1 (STS): A state transition system (STS) is a tuple $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$, where

- $\mathcal{S}$ is the set of states and $\mathcal{S}^0 \subseteq \mathcal{S}$ are the initial states;
- $\mathcal{I}$ and $\mathcal{O}$ are the input and output actions respectively;

- $\mathcal{R} \subseteq \mathcal{S} \times Bool \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{S}$ is a transition relation,
- $\mathcal{S}^F \subseteq \mathcal{S}$ is the set of accepting states.
- $\mathcal{F} : \mathcal{S} \rightarrow 2^{\mathcal{P}}$ is a labelling function that links the states with a set of propositions $\mathcal{P}$ associated to the STS.

Given a boolean expression $b \in Bool$ over propositions in $\mathcal{P}$, the labelling function defines whether the expression holds in a state:

- $s, \mathcal{F} \models \top$;
- $s, \mathcal{F} \models p$, iff $p \in \mathcal{F}(s)$;
- $s, \mathcal{F} \models \neg b$, iff $s, \mathcal{F} \not\models b$;
- $s, \mathcal{F} \models b_1 \vee b_2$, iff $s, \mathcal{F} \models b_1$ or $s, \mathcal{F} \models b_2$.

The transitions of STS are guarded: a transition $\langle s, b, a, s' \rangle$ is possible in the state s only if the guard expression b holds in that state, i.e., $s, \mathcal{F} \models b$. A run π of STS is a finite sequence of transitions $\pi = \langle s_0, b_0, a_0, s_1 \rangle, \ldots, \langle s_n, b_n, a_n, s_{n+1} \rangle$, with $a_i \in \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$, $s_i, \mathcal{F} \models b_i$, $s_0 \in \mathcal{S}^0$, and $s_{n+1} \in \mathcal{S}^F$. The set of all runs of a STS Σ is denoted with $\Pi(\Sigma)$.

Component services can be recast as STSs, and, given a set of component services $W_1, \ldots, W_n$, the planning domain Σ is defined as a synchronous product of the all the STSs of the component services: $\Sigma = \Sigma_{W_1} \| \ldots \| \Sigma_{W_n}$. The synchronous 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.

A composed service can also be represented as a state transition system Σ_c, whose aim is to control the planning domain defined by the component services. The interactions of Σ_c and Σ are modelled by the following notion of a controlled system.

Definition 2 (Controlled System): Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$ and $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{I}, \mathcal{O}, \mathcal{R}_c, \mathcal{S}_c^F, \mathcal{F}_c \rangle$ be two state transition systems. STS $\Sigma_c \triangleright \Sigma$, describing the behaviors of system Σ when controlled by Σ_c, is defined as follows:

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

where:

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

In this setting, service composition is stated as the following problem: given the services $W_1, \ldots, W_n$ and a composition goal ρ, identify a composed service Σ_c such that the controlled system $\Sigma_c \triangleright (\Sigma_{W_1} \| \ldots \| \Sigma_{W_n})$ satisfies ρ. In [10] it is shown how planning for preference-ordered goals may be applied for this purpose, considering a list $\rho = (g_1, g_2, \ldots, g_n)$ of alternative requirements where each g_i is a reachability goal. In our work, this idea will be used as a stepping stone upon which we will integrate our novel and more expressive object-driven view of requirements.

5. Modelling Service Composition

In this section we present notations for modelling a service composition problem, that is, the domain objects and their evolution, their associated services, and the corresponding control-flow composition requirements.

5.1. Representing Objects

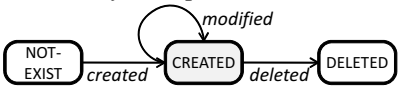
Formally, we represent objects with *object diagrams*.

Definition 3 (Object Diagram): An *object diagram* representing an object O is a tuple $\langle L, L_0, \mathcal{E}, T \rangle$, where

- L is a finite set of object configurations and $L_0 \subseteq L$ is a set of initial configurations;
- $\mathcal{E}$ is a set of possible events that reflect the evolution of the object;
- $T \subseteq L \times \mathcal{E}^+ \times L$ is a transition relation that defines the evolution of an object, based on events.

We require that there exists a predefined event $to^{\mathsf{e}}(l, o)$, with $l \in L$, to define that the object o moves to a configuration l. We also require that transitions leaving a configuration are annotated with mutually disjoint sets of events.

Example 1: The object diagram of the hotel reservation and of the flight ticket may be represented as follows:



The diagram contains three configurations, namely ***not-exist***, ***deleted***, and ***created***. The object moves to the configuration ***created*** upon the event "created", while it moves to a configuration ***deleted*** upon the event "deleted". When the reservation exists, it may be modified, which is reflected with the event "modified".

5.2. Services and Service Annotations

We assume that the description of the services associated with the considered domain objects consists of a stateful service protocol (e.g., a BPEL process), associated to a stateless service interface (e.g., a WSDL document). Each service description is related with a corresponding object and its dynamics, through special annotations. These annotations appear within the activities of the service protocol: an activity may be annotated with a set of events pertaining to the corresponding object. This implicitly defines how the evolution of the service reflects over the object.

Formally, we model services as *annotated state transition system* (ASTS), similarly to the STS defined above. The transitions of ASTS may be labelled with *object events*, thus stating that when the transition takes place, the corresponding object is changed. If, for example, the transition is annotated with an event $to^{\mathsf{e}}(l, o)$, then the object o moves to a configuration l when this transition takes place.

Definition 4 (Annotated State Transition System): An annotated STS Σ is a tuple $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{E}, \mathcal{R} \rangle$ where:

- $\mathcal{E}$ is the set of events;
- $\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{E}^* \times \mathcal{S}$ is the transition relation.

The semantics of the annotated transition is intuitively described as follows. Assume an object o with a set of events

$\mathcal{E}$, and a service transition $(s, a, \varepsilon, s') \in \mathcal{R}$, where $s, s' \in \mathcal{S}$, $a \in (\mathcal{I} \cup \mathcal{O} \cup \{\tau\})$, and $\varepsilon \subseteq \mathcal{E}$. We say that the transition is *applicable* to the object o if the object is in some configuration l, and either $\varepsilon = \emptyset$, or there exists an object transition (l, ε', l'), such that $\varepsilon' \subseteq \varepsilon$. As a result of performing this transition, in the first case the object will remain in the same configuration, while in the second case it will evolve to the configuration l'.

Example 2: An annotated STS of the Flight Cancellation Service (Fig. 1, c) may be defined as $\Sigma = \langle\{s_0, s_1, s_2\}, s_0\}, \{delete\}, \{flightDeleted\}, \mathcal{E}, \mathcal{R}\rangle$. The output operation "flightDeleted" is annotated with the event $d^{\mathsf{e}}(f)$ (flight is "deleted"), as the service provider confirms the flight ticket cancellation. That is, $\mathcal{E} = \{d^{\mathsf{e}}(f)\}$, and $\mathcal{R} = \{(l_0, ?delete, \emptyset, l_1), (l_1, !flightDeleted, \{d^{\mathsf{e}}(f)\}, l_2)\}$.

5.3. Composition Requirements

We now present a simple language that allows fulfilling our desiderata to model in an easy-to-specify, compositional and implementation-independent way complex requirements, namely (i) in which "stable" situations we intend to see our objects, possibly ordered according certain preferences; (ii) requirements on the evolution of objects, linking the behaviors of different objects; and (iii) reaction rules, which define how the composed service shall react to object events in different situations.

Definition 5 (Composition Requirement): A composition requirement is defined with the following generic constraint template

$$clause \Longrightarrow (clause_1 \succ \ \ldots \ \succ clause_n),$$

where $clause \equiv \top \mid s^{\mathsf{s}}(o) \mid e^{\mathsf{e}}(o) \mid cl_1 \vee cl_2 \mid cl_1 \wedge cl_2$. Here cl_1 and cl_2 are clauses, $s^{\mathsf{s}}(o)$ is used to define the fact that the object o is in the configuration s, and $e^{\mathsf{e}}(o)$ defines that the event e of the object o has taken place.

The left side of the constraint defines the "premise" of the requirement. In case it is empty (i.e., defined as $\top$), the requirement expresses the need to unconditionally reach a particular state or to achieve a particular effect, defined by the right side. Otherwise, it defines a "reaction rule": whenever the corresponding situation or events take place, the composite service should try to "recover" from it by achieving the effects/situations defined by the right side. In both cases, the right side of the constraint defines the expected results. Each of them logically groups simpler results, which may express either a certain state (require to reach a configuration) or a certain effect (require that an event happens). These results are ordered according to the order of preference denoted by the $\succ$ symbol, from the most preferred to the least preferred. The following example clarifies the usage of both unconditional and reaction-rule requirements.

Example 3: The composition requirements identified in Section 2 may be represented as follows:

- $\top \Longrightarrow (cr^{\mathsf{s}}(f) \wedge cr^{\mathsf{s}}(h) \wedge cr^{\mathsf{s}}(o)) \ \succ$
  $(d^{\mathsf{s}}(f) \vee ne^{\mathsf{s}}(f)) \wedge (d^{\mathsf{s}}(h) \vee ne^{\mathsf{s}}(h)) \wedge (d^{\mathsf{s}}(o) \vee ne^{\mathsf{s}}(o))$.
  That is, the composition should try to create the objects o (offer), h (hotel), and f (flight), i.e., bring them to the state "cr" (created). If this is not possible, it must guarantee that none of them exists, i.e., they should be in the state "d" (deleted) or "ne" (not-exist).
- $mod^{\mathsf{e}}(f) \Longrightarrow (mod^{\mathsf{e}}(h) \wedge mod^{\mathsf{e}}(o)) \ \succ$
  $(d^{\mathsf{s}}(f) \wedge d^{\mathsf{s}}(h) \wedge d^{\mathsf{s}}(o))$.
  Here as a reaction to the flight delay (event $mod^{\mathsf{e}}(f)$) we require the corresponding modification of the hotel ($mod^{\mathsf{e}}(h)$) and of an offer ($mod^{\mathsf{e}}(o)$). If this is not possible, the alternative is to cancel both reservations.
- $d^{\mathsf{e}}(f) \Longrightarrow (d^{\mathsf{e}}(h) \wedge d^{\mathsf{e}}(o))$.
  In other words, if the flight is cancelled, the composed service should cancel also the hotel reservation ($d^{\mathsf{e}}(h)$) and the offer ($mod^{\mathsf{e}}(o)$).

Given a set of requirements of this form, we aim to build a composed service that aims to satisfy all of them simultaneously, according to their above-mentioned "unconditional" and "reactive" semantics and following the preference orders specified for right side clauses.

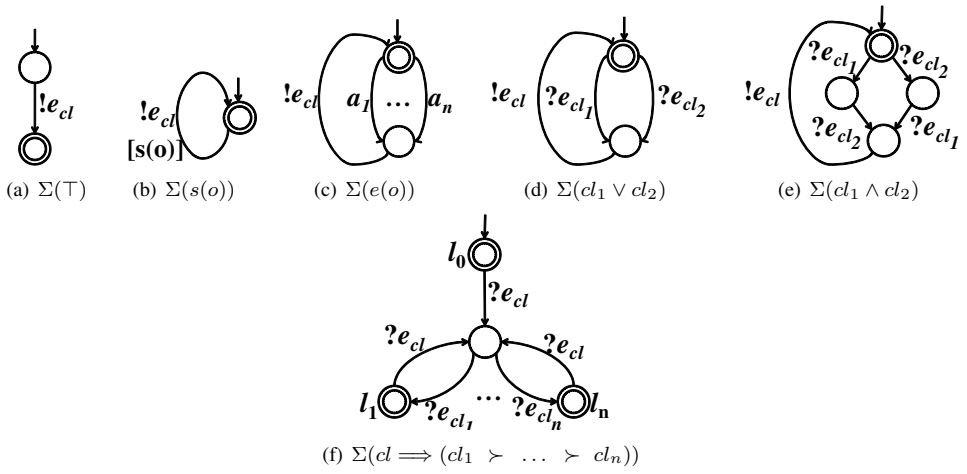
We remark that the requirements defined in this way are completely detached from the way services are defined and realized, providing a degree of flexibility which is crucial for the design and maintenance of requirements, especially in a dynamic settings where services may not be known a priori, or evolve in time.

6. Automated Service Composition

Our requirements impose constraints on the evolution of objects, which must be achieved by executing services associated to those objects. In order to recast this in terms of planning, we therefore have to transform our objects and the ASTSs of the component services into state transition systems, and we must express our composition requirements in terms of the states of such STSs. In the following, we describe in turn the transformation of services, objects and composition requirements.

Transformation of component services. The transformation of the component services (i.e., ASTSs) is performed as follows. Given an ASTS $\langle\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{E}, \mathcal{R}\rangle$, we define a corresponding STS as $\langle\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}', \mathcal{S}^F, \mathcal{F}\rangle$, where for each transition $(s, a, \varepsilon, s') \in \mathcal{R}$ we define a corresponding (non-guarded) transition $(s, \top, a, s') \in \mathcal{R}'$, and the states are not labeled (for each $s \in \mathcal{S}$ $\mathcal{F}(s) = \emptyset$). In order to require that the service protocols are either unused or fully completed, all the terminating and initial states of the ASTS are marked as accepting.

Transformation of object diagrams. The transformation of the object diagram into STS is more complex. First, to capture the states of the objects in the requirements, we define a set of atomic propositions $\mathcal{P} \equiv \{s_j{}^{\mathsf{s}}(o_i)\}$, which specify that an object o_i is in state s_j for all objects and their



(a) $\Sigma(\top)$ (b) $\Sigma(s(o))$ (c) $\Sigma(e(o))$ (d) $\Sigma(cl_1 \vee cl_2)$ (e) $\Sigma(cl_1 \wedge cl_2)$

(f) $\Sigma(cl \Longrightarrow (cl_1 \succ \ldots \succ cl_n))$

Fig. 3. STS diagrams of composition requirements

states. Then, given an object diagram $\langle L, L_0, \mathcal{E}, T\rangle$ we define an STS $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F}\rangle$, where $\mathcal{S} = L$, $\mathcal{S}^0 = L_0$, each state is labelled with the corresponding proposition (i.e., $\forall s \in \mathcal{S} : \mathcal{F}(s) = \{s^{\mathsf{s}}(o)\}$), and all object configurations are accepting (i.e., $\mathcal{S}^F = \mathcal{S}$). We define the transition relation so to capture the effects of the evolution of component services on the object: for each $(l, \varepsilon', l') \in T$ and for any transition (s, a, ε, s') of some ASTS such that $\varepsilon' \subseteq \varepsilon$ we define a (non-guarded) transition $(l, \top, a, l') \in \mathcal{R}$.

Transformation of requirements. The composition requirements speak of both the object states and of occurrences of object events. In order to capture this information, for every requirement we define a corresponding STS that reflects the satisfiability of the requirement. Given a clause cl, we define a corresponding STS that contains a single output action e_{cl} representing the completion of the clause. The diagrams corresponding to the different clauses, to their combinations, and to the representing diagram itself are represented in Fig. 3. Intuitively, they have the following meaning.

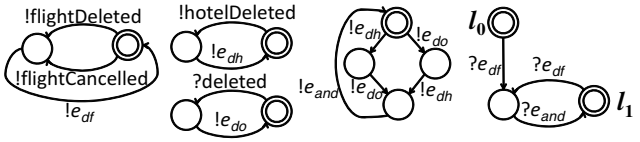
- The STS for the $\top$ clause (Fig. 3(a)) is completed immediately.
- The STS for $s^{\mathsf{s}}(o)$ (Fig. 3(b)) is blocked until the object is not in the required state: the transition is guarded with the corresponding proposition.
- The STS for $e^{\mathsf{e}}(o)$ (Fig. 3(c)) waits for any of the service actions that contain the corresponding event in its effects: for any transition (s, a, ε, s') of some ASTS such that $e^{\mathsf{e}}(o) \in \varepsilon$ a corresponding transition is defined. When it happens, a completion is reported.
- The STS for $cl_1 \vee cl_2$ (Fig. 3(d)) waits for any of the sub-clauses to complete, while the STS of the $cl_1 \wedge cl_2$ (Fig. 3(e)) waits for both of them to be completed.

The STS that represents the evolution of a composition requirement is represented in Fig. 3(f). The STS is initially in an accepting location (l_0). If the premise takes place (e_{cl} is reported), then it moves to a non-accepting state, from which it may be satisfied by completing one of the clauses $e_{cl_1}, \ldots, e_{cl_n}$ (moving to locations $l_1, \ldots, l_n$ respectively). The corresponding goal with preferences will have the following form:

$$\rho_c = (l_0, l_1, \ldots, l_n). \tag{1}$$

That is, we require that whenever the premise take place, the composition tries to move the STS to one of the accepting states, respecting the ordering of preferences.

Example 4: The requirement $d^{\mathsf{e}}(f) \Longrightarrow (d^{\mathsf{e}}(h) \wedge d^{\mathsf{e}}(o))$ is modelled with the following STSs:



6.1. Generating a Composite Service

To integrate our approach into the automated composition framework, we essentially need to include the STS-encoded object diagrams and composition requirements within the composition domain, prior to applying the approach described in [7]. In particular, given n composite services $W_1, \ldots, W_n$, m objects $O_1, \ldots, O_m$ and k (event) composition requirements $C_1, \ldots, C_k$ we encode each component service W_i as the corresponding state transition systems Σ_{W_i} (Fig. 2, step 2); each object diagram O_i as Σ_{O_i} (Fig. 2, step 1); each composition requirement C_i as Σ_{C_i} (Fig. 2, step 3). The translation is defined according to the rules presented above.

Then, we build a planning domain and goal. Namely, the planning domain Σ is defined as a synchronous product of the all the STSs of the component services, objects, and requirements, and the composition goal is constructed from the requirements defined according to the formula (1):

$$\Sigma = \Sigma_{W_1} \| \ldots \| \Sigma_{W_n} \parallel \Sigma_{O_1} \| \ldots \| \Sigma_{O_m} \parallel \Sigma_{C_1} \| \ldots \| \Sigma_{C_k}$$

$$\rho = \bigwedge_c \rho_c.$$

Finally, given the domain Σ and the planning goal ρ, we apply the approach presented in [7] 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 executable BPEL process to obtain the new process which implements the required composition (Fig. 2, step 4). 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.

Correctness of the approach. In order to prove that the proposed approach is correct, we have to show that all the executions of the composed services (controller Σ_c) satisfy the control flow requirements expressed as constraints. For the sake of simplicity, we omit the formal proof, and simply sketch the key points. It is easy to see that each execution π of the composed service is also a run of the domain, i.e., if $\pi \in \Pi(\Sigma_c)$ then $\pi \in (\Sigma)$. Under the requirement that all the executions of requirement STSs terminate in accepting states, we have that the executions of the domain satisfy the composition requirements. As a consequence the following theorem holds.

Theorem 1 (Correctness of the approach):
Let $\Sigma_{C_1}, \ldots, \Sigma_{C_k}$ be the STS encoding of the composition requirements $C_1, \ldots C_k$, and $\Sigma_{O_1}, \ldots, \Sigma_{O_m}$ be the STS encoding of the domain objects $O_1, \ldots, O_m$. Let Σ_c be the controller for a particular composition problem $\Sigma = \Sigma_{W_1} \| \ldots \| \Sigma_{W_n} \parallel \Sigma_{O_1} \| \ldots \| \Sigma_{O_m} \parallel \Sigma_{C_1} \| \ldots \| \Sigma_{C_k}$. Then the executions $\Pi(\Sigma_c)$ satisfy the requirements $C_1, \ldots, C_k$.

6.2. Experimental Results

To evaluate the feasibility of our approach, we implemented a prototype of the composition framework and tested it on the reference scenario, using the control flow requirements given in Example 3. The scenario model includes also the data flow requirements to appropriately route data. These are defined with the data-net approach of [9].

As a result, our prototype generated an executable composed process that orchestrates the five services in the scenario to realize the above requirements. The composed process tries to perform the flight booking and hotel reservation, correctly taking into account possible non-deterministic outcomes of the component services, and creates a travel offer upon successful reservations. Then the process continuously handles the delays or cancellations from the flight status notification service. In the first case, it tries to modify the hotel reservation: if the hotel agrees and the user accepts the new offer, the process is ready to handle new modifications. Otherwise, the process deletes the reservations and terminates. In the second case, the process cancels the hotel reservation and informs the user.

Such orchestration would be far from trivial to design and develop even for a skilled analyst. Its BPEL representation, which we do not report for lack of space, involves more than 50 activities (e.g., receive, invoke, assignment), and features a fairly complex structure, which includes several decision points, on-message clauses, and two different loops. Such protocol is way more complex than the complex of starting components, and its manual encoding into BPEL would be a time-demanding and error-prone task.

The composed service was generated in about 35 seconds on a 2.6GHz, 4Gb Dual Core machine running Linux. Given the complexity of the composition task, we take this experiment as a first important witness of the practical applicability of our approach.

7. Related Work and Conclusions

In this paper we have presented a novel notation for modelling expressive control flow requirements for service compositions, based on the notion of domain object, and an automated composition framework that is able to support those requirements. Our approach allows designers to easily express requirements that go beyond simple goals of achieving certain state or effects, but capture more sophisticated constraints, like, e.g., event reaction rules. Furthermore, by detaching the requirements from the specific service implementations, we significantly improve on current approaches in terms of the ability to express compositions in a flexible and maintenable way. We also reported preliminary experimental results, showing the feasibility of the approach. In this respect we remark that, while in our current setting the analyst is fully in charge of the design of objects and annotations, both such elements feature simple structures, and are amenable to intuitive representations, in terms of e.g. graphical languages. As such, developing a graphical design support tool for objects and requirements, possibly making use of specific forms of semantic linking and allowing the adoption of libraries of reusable business objects, is high in our agenda.

Our composition approach is significantly more expressive than the vast set of approaches where services are viewed as atomic entities [1]–[5]. It also significantly extends recent approaches which tackle stateful, non-deterministic services [6]–[10], by providing a more expressive requirement language, detached from implementation details, and giving an effective planning problem encoding. In particular, we remark that, concerning data-flow requirements, our framework is agnostic, and indeed in our implementation we adopt the approach of [9]. Detaching data-flow requirements from service implementations is a relevant research topic, high in our research agenda.

The idea of representing service implementations and requirements by more abstract "object" entities can be related to languages such as WS-conversation [11], and to approaches to model workflows based on formal languages such as Petri nets [12] and graphical notations such as state-charts and object diagrams [13], [14]. However, to the best of our knowledge, this is the first attempt to adopt this view for service composition purposes, and none of the approaches

above confront with the issue of automatically deriving stateful orchestrations from given specifications. Similarly, the general idea of having expressive, automata-based composition requirements is also used in the fields of program synthesis and planning, often behind the adoption of logic languages, see e.g. [15]–[17]. However, so far the idea has not been exploited for service composition, and languages proposed in the planning area do not appear as adequate tools for the design and engineering of composition requirements by service analysts.

Acknowledgment

The research leading to these results has received funding from the European Communitys Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube) and from DOCOMO EuroLabs under research agreement RA088 (YourWay!).

References

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

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

[3] M. Sheshagiri, M. des Jardins, and T. Finin, "A Planner for Composing Services Described in DAML-S," in *Proc. AAMAS'03*, 2003.

[4] R. Aggarwal, K. Verma, J. A. Miller, and W. Milnor, "Constraint Driven Web Service Composition in METEOR-S," in *Proc. of SCC'04*, 2004, pp. 23–30.

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

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

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

[8] R. Hull, "Web Services Composition: A Story of Models, Automata, and Logics," in *Proc. of ICWS'05*, 2005.

[9] A. Marconi, M. Pistore, and P. Traverso, "Specifying Data-Flow Requirements for the Automated Composition of Web Services," in *Proc. SEFM'06*, 2006.

[10] D. Shaparau, M. Pistore, and P. Traverso, "Contingent Planning with Goal Preferences," in *Proc. AAAI'06*, 2006.

[11] A. Banerji, C. Bartolini, and D. Beringer, "Web Services Conversation Language (WSCL) 1.0," http://www.w3.org/TR/wscl10/, 2002.

[12] W. M. P. van der Aalst, "Inheritance of Interorganizational Workflows to Enable Business-to-Business," *Electronic Commerce Research*, vol. 2, no. 3, pp. 195–231, 2002.

[13] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[14] D. Dori, *Object-Process Methodology - A Holistic Systems Paradigm*. Springer Verlag, 2002.

[15] F. Bacchus and F. Kabanza, "Planning for Temporally Extended Goals," *Ann. Math. Artif. Intell*, vol. 1-2, no. 22, pp. 5–27, 1998.

[16] J. Baier, F. Bacchus, and S. McIlraith, "A Heuristic Search Approach to Planning with Temporally Extended Preferences," in *Proc. IJCAI'07*, 2007.

[17] J. Baier, C. Fritz, and S. McIlraith, "Exploiting procedural domain control knowledge in state-of-the-art planners," in *Proc. of ICAPS'07*, 2007.