Choreographies: using Constraints to Satisfy Service Requests

Alexander Lazovik,[1,2] Marco Aiello,[1] and Rosella Gennari[2]
1. DIT, University of Trento, Via Sommarive 14, 38100 Trento, Italy
2. ITC-irst, Via Sommarive 18, 38100 Trento, Italy
{lazovik,aiellom}@dit.unitn.it gennari@itc.it

***Abstract*— Interacting with a web service enabled marketplace in order to achieve a complex task involves sequencing a set of individual service operations, gathering information from the services, and making choices. In the context of coreographies of web services, we propose to encode the problem of issuing requests as a constraint problem. In particular, we provide a choreographic framework to handle requests, we show how the encoding of requests is performed, and we illustrate an implementation using the Choco constraint system.**

I. Introduction

Satisfying complex business requirements in service-enabled marketplaces comprises the composition of business processes, their execution and monitoring, and gathering information from services at run-time. Requesters and service providers have complex requests which express desiderata of distributed interaction and, ideally, they would want to abstract from the inner working of the marketplace. These desiderata express the achievement of complex business requests, the preference of some requests over others, and the achieving of certain requests with specific numeric values ranges. A user may desire to obtain a trip package for a given date spending a certain amount of money and preferring a certain flight carrier. A service provider might expose a business rule that forces unregistered users to pay before receiving the service.

A broad service enabled marketplace is thus a distributed system in which autonomous actors interact asynchronously according to some standardized general business process each one with its own requests and additional requirements. The interaction of the service providers and requester in such a setting is known as a *choreography*. A natural way to model choreographies is through constraints. In fact, the business process defining the marketplace can be modeled as a set of constraints; user requests are interpreted as additional constraints to be satisfied against the given business process; finally, the service providers' requirements are also modeled as constraints on how their services need to be invoked.

Methods and techniques to automatically enable choreographies of services are the subject of recent research. There are approaches based on formal logics [1], [2], [3] or other approaches based on logic programming formalisms (e.g., [4]). All these approaches work under the assumption of having available rich semantic service description and run-time information. Artificial Intelligence techniques can provide a solution to the problem of service composition. In particular, there have been several proposals using AI planning [5], while encoding planning problems as constraints is in [6].

In [7], we have proposed the XSRL request language over complex business domains. In [8], we presented the constraint model laying at the basis of the present work. In this paper, we propose a framework for the encoding of choreographies and requests as a set of constraints; finally, we propose an implementation using the Choco constraint system.

The remainder of the paper is organized as follows. A motivating example in the travel domain is introduced in Section II. In Section III, we present the framework for managing choreographies encoded as constraints. Section IV presents the rules for encoding choreographies and requests. A snapshot of our implementation in Choco is shown in Section V. Concluding remarks are presented in Section VI.

II. An example in the traveling marketplace

Consider a user requesting a trip to Nowhereland and having a number of additional requirements regarding such a trip, e.g., that the total price of the trip be lower than 300 euro, the prices of the hotel lower than 200 euro, avoid using the train, and so on. To be satisfied such a request involves the interaction with various autonomous service providers, including a travel agency, a hotel company and a flight carrier. Services reside in the same travel marketplace domain and must follow a standard business process for that domain. Such a process is exemplified in Figure 1. This process is modeled as a labelled transition graph, that is, every node is a state in which the process can be, while directed arcs, each labeled by a specific action, indicate how the process changes state. Actors involved in the process are shown at the top of the graph. The actors include the user issuing the request, a travel agency, a hotel service, an air service, a train service and a payment service.

The process is initiated by the user contacting a travel agency, hence, (1) is the initial state. The state is then changed to (2) by requesting a quote from an hotel (action a_1). The dashed arcs represent web service responses, in particular arc a_2 brings the system in the state (3). The execution continues along these lines by traversing the paths in the transition graph until we reach state (14). In this state a confirmation of an hotel and of a flight or train is given by the travel agency and the user is prompted for acceptance of the travel package (13).

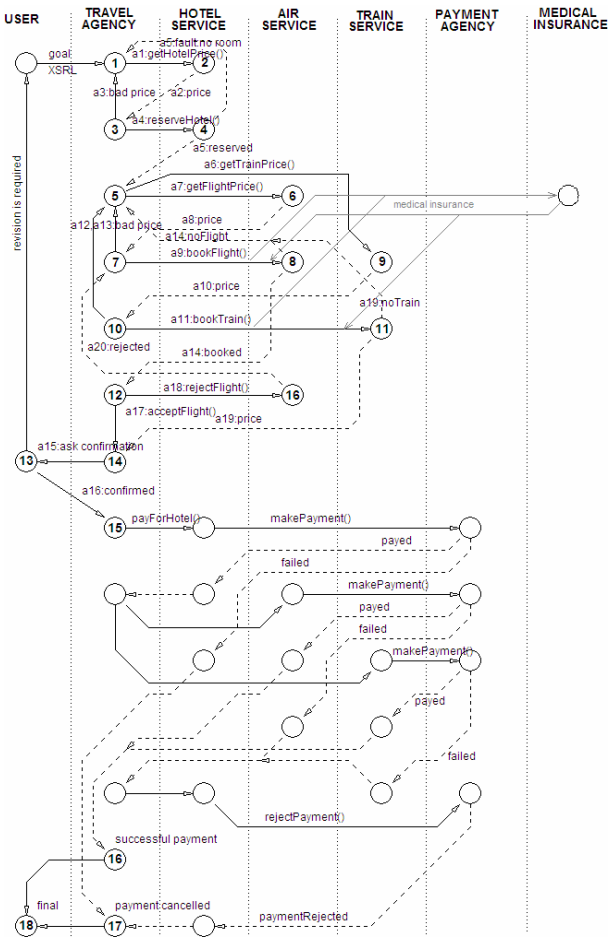

Fig. 1. A traveling business process.

Actions in the graph can be *nondeterministic*. This is illustrated, for instance, in state (4). In this state the user has accepted the hotel room price however is faced with two possible outcomes, one that a room is not available (where the system transits back to state (1)) and the other where a room reservation can be made (state (5)). The actual path will be determined only at run-time.

The lower part of the business process models the payment of the travel package just booked as an atomic action. This means the entire trip payment is atomic.

III. Web Service Execution using Constraints

Issuing a request to a marketplace generates a choreographic effect in which the various service providers are invoked and provide service following a precise order. The order is determined by the request, by the run-time conditions, by the values returned by the various providers, and by nondeterministic conditions. We view such an execution as the performing of a set of actions of the transition graph in order to achieve the issued request. Given the number of unknown elements and the nondeterministic nature of services, an initial plan will often fail, making replanning necessary. We encode the choreography as a constraint problem. In [8], we provide

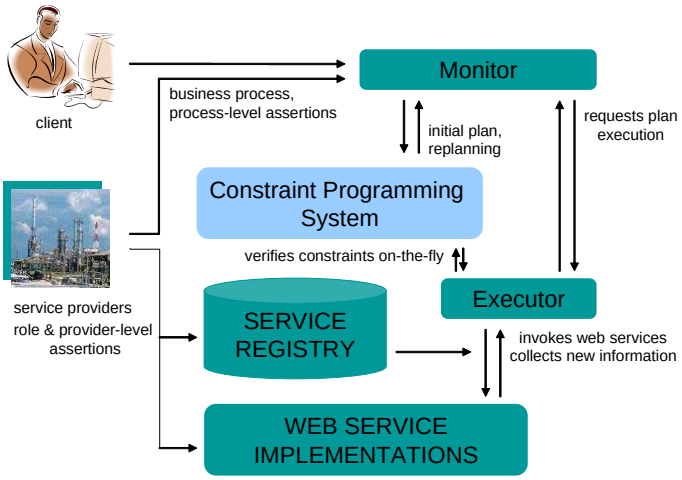

Fig. 2. Framework with a constraint programming system.

the basic algorithms for the encoding, here we introduce the framework for supporting such an encoding.

In this view of a choreography, a *business process* is a planning domain, that is, a labeled transition graph whose states represent the state of a distributed computation. Such a transition graph can be extracted from a standard choreography description. What we do is to transform this graph into a set of constraints according to the rules shown in Table I. The *user request* is also encoded as a set of constraints, as shown in Table I, to be satisfied against the constraint set of the business process. In other words, a *choreography* for satisfying a request amounts to the the execution of a plan, which in turn we view as a solution to a constraint-based problem.

In Figure 2, we present the choreography framework based on constraints to satisfy user requests. The framework consists of four components: monitor, constraint programming system, and supporting runtime environment. The monitor manages the overall process of interleaving planning and execution. It takes user requests, the business process, and starts interacting with the constraint programming system that synthesizes a plan and returns it to the monitor. The plan is a sequence of actions to be executed. The constraint system returns a failure if there is no plan for the user request in the given domain. Let us assume that a correct plan exists and therefore is synthesized. Then the monitor passes it to the executor. The executor is responsible for executing the plan. While executing each action of the plan, the executor may gather new information from the service registry or from the service invocations. Whenever new information is obtained, the constraint set is updated and the constraint system checks if the newly introduced constraints violate the plan under execution. The framework works iteratively until the request is satisfied or there is no satisfying execution.

For the framework shown in Figure 2 constraints come out as the natural choice. Firstly, typical web service interactions involve constraints over numeric values, and constraint programming systems provide solvers for these. Secondly, the

execution of the business process depends on the outcome of the services it consist of, new information gathered at runtime; in other words, a process result depends on the information that is available only at runtime. Interleaving planning and execution supports such iterative model of execution. Replanning is performed when new information is gathered. Due to the incremental nature of most constraint programming solvers, full replanning from scratch is not needed, in the sense that one can add new constraints to the set of already active ones; this can be seen as a refinement of the initially synthesized plan. However, when constraints are to be removed from the constraint set, for example, when changing a provider, the constraint space may have to be rebuilt—however, an extension of Choco allows for the "intelligent" removal of constraints [9].

To benefit from constraint programming we have to formulate the choreography problem in terms of constraints. In the following section we show how the service domain accompanied with requests is encoded.

IV. Expressing Choreographies and Requests

To express the execution of the requests, i.e., the run of a choreography in a specific web service context, we need three main ingredients:

- **(BP)** a representation of the Business Process or Domain;
- **(RL)** a Request Language to express requests;
- **($\bowtie$)** a mechanism to decide how to coordinate and sequence the service invocations in order to satisfy the requests.

There are a number of requirements on these ingredients. For the business process, we want to use a representation able to capture the nondeterminism typical of web service implementations and to be state based. For the request language, we want a language which is high-level and expressive. It must be possible to state preferences, to state a number of subtasks, to sequence the order in which the subtasks might be achieved. Finally, for the choreographic mechanism we want an agile framework which can be implemented, but also sound and complete.

We formalize the choreography as a constraint programming task. In this way, the requester's desires become a set of constraints over an actual business process, while the satisfaction of the request is the satisfaction of the constraints. While the constraint programming system tries to satisfy these, a number of invocations are performed which might lead to more constraints being added or to the instatiation of a number of free variables. The latter process is an instance of information gathering at run time.

More formally, we model (BP) ad (RL) as a set of constraints ($\bowtie$) over controlled and non-controlled variables. The constraints have the following form:

$$[\forall \xi_i :] \ \overline{c_v} \ \bowtie \ value, \tag{1}$$

- $value$ is a value from the domain of the variable v,
- $\overline{c_v}$ is a vector of expressions of the form $\sum \beta_i[\xi_i]a_{i,k}$ with $\beta_i, \xi_i \in \{0, 1\}$,
- the ξ_i are non-controlled variables and the β_i are controlled variables,
- $a_{i,k}$ is the effect of action a_i for outcome k,
- $\bowtie$ is either $<$,$>$,$\geq$, $\leq$ or $=$,
- and $[\cdot]$ denote that the expression is optionally present in the constraint.

Then we can define the problem of choreography as a *service constraint problem*. There are two types of Boolean variables: *controlled* variables, denoted by β_i, and *non-controlled* variables, denoted by ξ_i. Non-controlled variables represent nondeterministic action outcomes. The underlying idea is that the constraint programming system is not necessarily free to choose a specific value for a non-controlled variable, thus a solution to the problem may be such regardless of the values assigned to the non-controlled variables.

In this paper, when we talk about nondeterministic actions we refer to their outcomes (that is, states) which can be different; yet, once an action is invoked, we assume that its outcome will be always the same. That is, any of its future invocations produce the same outcome for the same provider.

Definition 1 (service constraint problem). *A* service constraint problem *is a tuple $\mathcal{CP} = \langle \beta, \mathcal{N}, \xi, \mathcal{C} \rangle$, where:*

- *β is a set of* controlled *boolean variables;*
- *$\mathcal{N}$ is a set of* controlled *variables over integers;*
- *ξ is a set of* non-controlled *boolean variables;*
- *$\mathcal{C}$ is a set of constraints, as in Equation 1, in which (i) if a non-controlled variable occurs then it is universally quantified, (ii) otherwise a value is available and substituted for the variable.*

A solution *to a service constraint problem is an assignment to controlled variables such that all constraints are satisfied.*

The encoding is performed in two phases: in phase (i) the service business process itself is encoded; in phase (ii) the request is added to the encoding.

A. Phase I

We consider the business process as a labeled transition graph with two types of actions to go from one state to another: deterministic and nondeterministic ones. This can be pictured as a graph of nodes (states) and labeled arcs (actions) with some extra information (roles, variables and effects, [7]).

During Phase I the business process is encoded. Starting from a business process as a labeled transition graph, we arrive at a set of expressions c_v as in Equation (1) plus a set of linear constraints of the form $\sum \beta_i \leq 1$. In the following, we adopt the notation of Equation (1); in addition, n varies over integers and specifies how many times a cycle is followed, while a_i is overloaded to represent not only the action, but also its effects.

The encoding is generated following an algorithm that visits the process graph, separately keeping track of cycles, and returns a set of constraints. The whole process is recursive and is divided into the following cases, summarized in Table I:

(A) Base case. If the degree of the arcs leaving the state s is 0, then there is no constraint to be returned. The case of the directed cycle, which is presented below, is also a base case.

	Type of action	Encoding
(A)	No action	0
(B)	Single deterministic action	βa
(C)	Sequence of actions	$\beta_1(a_1 + \beta_2 a_2)$
(D)	Branching	$\beta_1 a_1 + \beta_2 a_2$, with $\beta_1 + \beta_2 \leq 1$
(E)	Nondeterministic action	$\xi_1 a' + \xi_2 a''$, with $\xi_1 + \xi_2 = 1$
(F)	Cycle: undirected cycle	state splitting, no specific encoding
(G)	Cycle: directed cycle	na

TABLE I

DOMAIN ENCODING RULES.

(B) Single deterministic action a is encoded as βa, where β is a controlled boolean variable. Then $\beta = 1$ means that action a must be in the resulting plan.

(C) Sequence of actions. This rule is applied to consecutive actions as follows (for two deterministic actions): $\beta_1(a_1 + \beta_2 a_2)$. If $\beta_1 = 1$ then action a_1 is added to the plan, and if also $\beta_2 = 1$, then a_2 is added to the plan right after a_1. If $\beta_1 = 0$ then neither action a_1 nor a_2 are added.

(D) Branching. If there are several outgoing actions from the state s and the system is supposed to choose only one of them to add to the plan, then this situation (for two actions) is encoded as follows: $\beta_1 a_1 + \beta_2 a_2$, where $\beta_1 + \beta_2 \leq 1$ means that at most one action can be chosen. If some of these actions are nondeterministic, then the **(E)** rule is applied to each one.

(E) Nondeterministic action. This rule takes care of a nondeterministic action. Such an action may bring the system nondeterministically in several states. To represent the behavior, in which one has no control over the action's outcome, the non-controlled variables ξ are introduced. The encoding (for a nondeterministic action with two possible outcomes) is the following: $\xi_1 a' + \xi_2 a''$, where $\xi_1 + \xi_2 = 1$.

(F) Cycle: state splitting. This rule is applied to undirected cycles. To proceed we need to duplicate the state s already visited by creating state s' and recursively encode the duplicated state. There is no further encoding for this case.

(G) Cycle: directed cycle. This rule is applied to directed cycles. Variable n in the encoding denotes the number of times the cycle is going to be executed.

B. Phase II

During the second phase of the encoding, we take a request (RL) and produce a set of constraints ($\bowtie$) for these; then we try to satisfy the resulting constraints against the constraint set encoding the business process (BP). The request is expressed in a language derived from XSRL [7]. Here we give the basic definition of the language and we show the intuitions for the language constructs in Table II.

Definition 2 (request language). *Basic requests are* ***vital*** p | ***atomic*** p | ***vital-maint*** p | ***atomic-maint*** p
where p is a constraint over the v variable. A request is a basic request or of the form ***achieve-all*** $g_1, \ldots, g_n$ | ***optional*** g | ***before*** g_1 ***then*** g_2 | ***prefer*** g_1 ***to*** g_2.

The algorithm parses the request recursively distinguishing the cases of the various operators and updating the set of constraints. Whenever a new basic request/constraint is added, a new set of controlled variables is introduced.

vital $v \bowtie v_0$. If the request is **vital** with respect to the variable v constrained by the $\bowtie$ operator on the v_0 value, we restrict the constraint c to what concerns variable v, denoting it by c_v, and we add $c_v \bowtie v_0$ to the constraints set. Since the request is **vital** we also set all variables ξ associated with c_v to ξ^0, by which we mean that the normal execution is followed, in place of the nondeterministic failure ones.

atomic $v \bowtie v_0$. This is analogous to **vital**, except that the nondeterministic variables ξ are universally quantified over.

vital-maint $v \bowtie v_0$. For maintainability requests we keep track of all the states visited during a plan execution. Thus, we apply the constraint as for **vital** for each step along the execution.

atomic-maint $v \bowtie v_0$. This is analogous to **vital-maint**, except that the nondeterministic variables ξ are universally quantified over.

Now we consider non-basic requests.

achieve-all $g_1, \ldots, g_n$. First, we recur on all sub-requests $g_1, \ldots, g_n$. Second, one considers all pairs of basic requests coming from the recursive call and all execution steps. In all these cases, if during the execution some choices have been made for the same branch point among different sub-requests, these choices have to be the same. Therefore we add, to the set of constraints, expressions forcing the same choices for the execution of any sub-requests. These expressions introduce the execution step t_k. Suppose that the set $\{\ldots \beta^g_{t_k,i} \ldots\}$ denotes the branch variables in step t_k, that has been chosen to satisfy the request g. Then $\sum \beta^{g_r}_{t_k,i} \neq 0$ denotes that one of the βs is set to 1 for the step under consideration. In order to ensure that different reachability requests are satisfied by the same sequence of actions, the following constraint is added: $\sum \beta^{g_{r_1}}_{t_k,i} \neq 0 \wedge \sum \beta^{g_{r_2}}_{t_k,i} \neq 0 \Rightarrow \forall i : \beta^{g_{r_1}}_{t_k,i} = \beta^{g_{r_2}}_{t_k,i}$. In order to guarantee that maintainability request is satisfied along the synthesized sequence of actions, one has to add implication for each pair of reachability/maintainability requests: $\sum \beta^{g_r}_{t_k,i} \neq 0 \Rightarrow \forall i : \beta^{g_r}_{t_k,i} = \beta^{g_m}_{t_k,i}$.

before g_1 **then** g_2. The principle behind the before-then operator is similar to that of the **achieve-all**, with the difference that one forces the ordering of the satisfaction of the sub-requests. Again, first we recur on the sub-requests, then we constrain the execution choice variables $\{\ldots \beta^g_{t_k,i} \ldots\}$. The second sub-request g_2 should repeat the path of the first sub-request g_1, until the first is satisfied, and only then the second expression is checked. This is ensured by expressions of the form: $\sum \beta^{g_1}_{t_k,i} \neq 0 \Rightarrow \forall i : \beta^{g_1}_{t_k,i} = \beta^{g_2}_{t_k,i}$ which are added to the set of constraints.

prefer g_1 **to** g_2. Preferences are handled not as additional constraints, but rather appropriately instantiating the variables. The first step is to recur on the two sub-requests g_1 and g_2. Then the requests g_1 and g_2 are placed in a disjunction. When constraints are checked for satisfiability, variables are assigned in preference order. Optional requests are a sub-case of **prefer-to** request, in which g_2 is simply true.

Request	Where satisfied	How encoded	Type of request
vital p	In a state where p holds to which there is a path from the initial state modulo failures	$\xi = \xi^0$: $c_v \bowtie v_0$	reachability
atomic p	In a state where p holds to which there is a path from the initial state despite failures	$\forall \xi$: $c_v \bowtie v_0$	reachability
vital-maint p	In a state to which there is a path from the initial state modulo failures. p must hold in all states along the path	$\xi = \xi^0$: $c_v(t_i) \bowtie v_0$, for all encoding steps t_i	maintainability
atomic-maint p	In a state to which there is a path from the initial state despite failures. p must hold in all states along the path	$\forall \xi$: $c_v(t_i) \bowtie v_0$, for all encoding steps t_i	maintainability
prefer g_1 **to** g_2	In states where g_1 is satisfied, otherwise the satisfiability of g_2 is checked	variables in g_1 are instantiated before those in g_2	preference
optional g	States where g is satisfied are checked first, otherwise the request is ignored	encoded as **prefer** g **to** $\top$	preference
before g_1 **then** g_2	In states, to which there is a path from the initial state, such that, states along these path where g_1 is satisfied precede those where g_2 is satisfied	for all steps t_k: $\sum \beta^{g_1}_{t_k,i} \neq 0 \Rightarrow \forall i : \beta^{g_1}_{t_k,i} = \beta^{g_2}_{t_k,i}$	sequencing
achieve-all $g_1, \ldots, g_n$	In states, to which there is a path from the initial state, such that, there are states along the path where g_i are satisfied	**reachability/reachability** pairs of requests: for all steps t_k, for all reachability pairs g_{r_1}, g_{r_2}: $\sum \beta^{g_{r_1}}_{t_k,i} \neq 0 \wedge \sum \beta^{g_{r_2}}_{t_k,i} \neq 0 \Rightarrow \forall i : \beta^{g_{r_1}}_{t_k,i} = \beta^{g_{r_2}}_{t_k,i}$ **reachability/maintainability** pairs of requests: for all steps t_k, reachability g_r, maintainability g_m: $\sum \beta^{g_r}_{t_k,i} \neq 0 \Rightarrow \forall i : \beta^{g_r}_{t_k,i} = \beta^{g_m}_{t_k,i}$	composition

TABLE II

REQUEST LANGUAGE CONSTRUCTS.

V. A RUN OF THE TRAVEL EXAMPLE

We have implemented the encoding of the service composition problem in Choco (`http://choco.sourceforge.net/`). Choco is a java library for constraint satisfaction problems (CSP), constraint programming (CP) and explanation-based constraint solving (e-CP). It is particularly well suited for adding and removing constraints while the CP system is working. This is the typical situation of a service enabled marketplace where any service interaction may result in the addition of a business rule of a provider or the gathering of new numeric information, i.e., a new constraint.

To illustrate our work and its implementation in Choco, let us introduce an example that is a snippet (shown in Figure 3) of the travel choreography definition (introduced in Figure 1). When deciding on a trip, the requester may first want to book the hotel for the final destination and then book a carrier to reach the location of the hotel. The first action a_0: `getHotelPrice` retrieves the hotel price. The next action is that of reserving a hotel (state s_1). This action may nondeterministically result in the successful booking of the room (state s_2) or in a failure (return to state s_1). Finally, there are two ways to reach the state s_3 in which a carrier to arrive at the site of the hotel is booked. One may choose to fly or to take a train. This is achieved by choosing one of the two actions `reserveTrain` or `reserveFlight`. There is one knowledge-gathering action: a_0: `getHotelPrice`. The process variables, ranging over integers, are: $hotelBooked$, $trainReserved$, $flightReserved$, which are boolean, and $hotelPrice$, $trainPrice$, $flightPrice$, $price$.

The framework works in the following way. First, the domain is encoded: $\beta_{s_0,0}(a_0 + \beta_{s_1,0}(\xi_{s_1,0} na_1^{fail} + \xi_{s_1,1}(a_1^{ok} + \beta_{s_2,0}a_2 + \beta_{s_2,1}a_3)))$; this represents the paths from state s_1 to s_3 with n being the number of times the cycle is followed, and with $\beta_{s_i,j}$ representing the j-th choice in state s_i. Additionally, the constraint on the controlled variables $\beta_{s_0,0}, \beta_{s_1,0}, \beta_{s_2,0}, \beta_{s_2,1} \in \{0,1\}$ is $\beta_{s_2,0} + \beta_{s_2,1} \leq 1$, and the constraint on the non-controlled variables $\xi_{s_1,0}, \xi_{s_1,1} \in \{0,1\}$ is $\xi_{s_1,0} + \xi_{s_1,1} = 1$, where $\xi_{s_i,j}$ represents the j-th nondeterministic outcome in the state s_j.

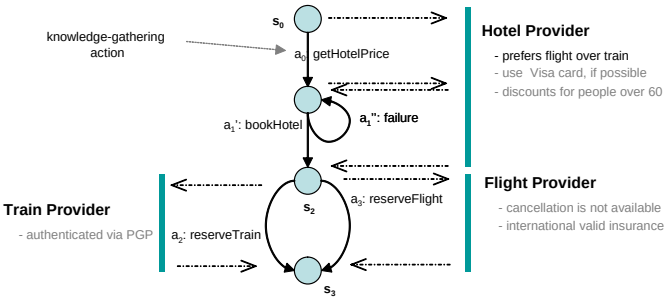

Fig. 3. A component of a travel business process.

Suppose the user provides the following request:

achieve-all
vital $hotelBooked \wedge$ **vital** $trainReserved$
vital-maint $price \leq 300$

The request is encoded as follows. Every basic request creates its own subset of controlled variables (Table II). The variables coming from the encoding of the request are shown next. As a point of notation, we use `courier` to present the output of our implementation.

```
xi_s1_0 in {0, 1}
beta_s2_0 in {0, 1}
```

```
beta_s2_1 in {0, 1}
beta_s2_0 + beta_s2_1 <= 1
xi_s1_1 in {0, 1}
xi_s1_0 + xi_s1_1 <= 1
beta_s1_0 in {0, 1}
```

For each of the **vital** request we additionally have `xi_s1_0=0`, `xi_s1_1=1`, representing the execution without failure. The first sub-request to be parsed is **vital** $hotelBooked$. Since the $hotelBooked$ variable is influenced only by the a_1^{ok} outcome, then the encoding is $\beta_{s_1,0}\xi_{s_1,1} = 1$ and, since $\xi_{s_1,1} = 1$, then $\beta_{s_1,0} = 1$. By the same reasoning, one has the encoding for the train: $\beta_{s_1,0}\beta_{s_2,0} = 1$:

```
0.0+beta_s1_0*(xi_s1_1*(1.0))=1.0
0.0+beta_s1_0*(xi_s1_1*(beta_s2_0*(1.0)))=1.0
```

The atomic request **vital-maint** $price < 300$ is slightly different as $price < 300$ has to be checked for each state. We assume that $a_1^{fail} = 0$, that is, no fee is paid for non-successful reservation. Recalling that $a_0 = 0$ and $a_4 = 0$ since getting price information is free, only three actions affect the price (`bookHotel`, `reserveFlight`, and `reserveTrain`). These simply add the corresponding cost to the overall price.

```
0.0 <= 300.0
0.0 + beta_s1_0 * (xi_s1_1 * (100.0)) <= 300.0
0.0 + beta_s1_0 * (xi_s1_1 * (100.0 +
  beta_s2_0 * (200.0) + beta_s2_1 * (400.0)))<=300.0
```

For each nested **vital** request pair within an **achieve-all**, there may be common branching points. If this is the case, the choice made has to be the same for all requests. Therefore, **achieve-all** adds the following constraints for **vital** requests (`vital_1` is a prefix for **vital** $hotelBooked$ and `vital_2` for **vital** $trainReserved$). Note that constraints are added only in states where there are at least two deterministic actions:

```
if (vital_1_beta_s2_0 + vital_1_beta_s2_1 = 1) ∧
   (vital_2_beta_s2_0 + vital_2_beta_s2_1 = 1)
then (vital_1_beta_s2_0 = vital_2_beta_s2_0) ∧
     (vital_1_beta_s2_1 = vital_2_beta_s2_1)
```

Maintainability requests in **achieve-all** are treated differently: if a choice is made for any **vital** requests, then the same choice must be made for **vital-maint**. Thus, for each pair of **vital** and **vital-maint** inside the **achieve-all** request we have the following constraints (where `vitalm_1` is a prefix for request **vital-maint** $price \leq 300$):

```
if (vital_1_beta_s0_0 = 1) then
    vital_1_beta_s0_0 = vitalm_1_beta_s0_0
if (vital_1_beta_s1_0 = 1) then
    vital_1_beta_s1_0 = vitalm_1_beta_s1_0
if (vital_1_beta_s2_0 + vital_1_beta_s2_1 = 1)
then (vital_1_beta_s2_0 = vitalm_1_beta_s2_0) ∧
     (vital_1_beta_s2_1 = vitalm_1_beta_s2_1)

if (vital_2_beta_s0_0 = 1) then
    vital_2_beta_s0_0 = vitalm_1_beta_s0_0
if (vital_2_beta_s1_0 = 1) then
    vital_2_beta_s1_0 = vitalm_1_beta_s1_0
if (vital_2_beta_s2_0 + vital_2_beta_s2_1 = 1)
then (vital_2_beta_s2_0 = vitalm_1_beta_s2_0) ∧
     (vital_2_beta_s2_1 = vitalm_1_beta_s2_1)
```

There are two solutions for the above constraint, and these are provided by our implementation. One of them is:

```
 *** vital-maint (price <= 300)
vitalm_1_beta_s1_0 = 1
vitalm_1_beta_s2_1 = 0
vitalm_1_beta_s2_0 = 1
 *** vital (trainBooked = true)
vital_2_beta_s1_0 = 1
vital_2_beta_s2_1 = 0
vital_2_beta_s2_0 = 1
 *** vital (hotelReserved = true)
vital_1_beta_s1_0 = 1
vital_1_beta_s2_1 = 0
vital_1_beta_s2_0 = 0
```

Summarizing, the solution means that the plan involves first invoking the `reserveHotel` and then `reserveTrain`.

The framework implementation includes algorithms for the interleaving of planning and execution (monitor, executor, and interaction with a constraint programming system) as well as for Phase I of the encoding. As for Phase II, all the request encodings are implemented except for **atomic** and **prefer-to**, the implementation of which is underway.

VI. Concluding Remarks

The choreography of independent services to achieve complex business requests is a labeled transition graph in which the transition from one state to another is governed by constraints coming from different actors. The requester, the service providers and the rules of the marketplace constrain the possible interactions. We proposed a framework to handle choreographies in which the problem of satisfying a request is encoded as a constraint-based problem. We have also provided an implementation of our framework to show the feasibility of the approach.

References

[1] L. Chen, N. Shadbolt, C. Goble, F. Tao, S. Cox, C. Puleston, and P. Smart, "Towards a knowledge-based approach to semantic service composition," in *2nd Int. Semantic Web Conf. (ISWC2003)*, ser. LNCS 2870, G. Goos, J. Hartmanis, and J. van Leeuwen, Eds. Springer-Verlag, 2003, pp. 319–334.

[2] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara, "Semantic matching of web services capabilities," in *Int. Semantic Web Conf. (ISWC2002)*, ser. Lecture Notes in Computer Science 2342, I. Horrocks and J. Hendler, Eds. Springer, 2002, pp. 333–347.

[3] E. Sirin, J. Hendler, and B. Parsia, "Semi-automatic composition of web services using semantic descriptions," in *Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, 2003. [Online]. Available: http://www.mindswap.org/papers/composition.pdf

[4] S. McIlraith and T. C. Son, "Adapting Golog for composition of semantic web-services," in *Conf. on principles of Knowledge Representation (KR)*, D. Fensel, F. Giunchiglia, D. McGuinness, and M. Williams, Eds., 2002.

[5] B. Srivastava and J. Koehler, "Web Service Composition - Current Solutions and Open Problems," in *Workshop on Planning for Web Services – ICAPS'03*, 2003.

[6] M. B. Do and S. Kambhampati, "Planning as constraint satisfaction: solving the planning graph by compiling it into csp," *Artificial Intelligence*, vol. 132, pp. 151–182, 2001.

[7] A. Lazovik, M. Aiello, and M. Papazoglou, "Planning and monitoring the execution of web service requests," *Journal on Digital Libraries*, 2005, to appear.

[8] A. Lazovik, M. Aiello, and R. Gennari, "Encoding requests to web service compositions as constraints," in *Principles and Practice of Constraint Programming (CP-05)*, 2005.

[9] N. Jussien, "e-constraints: explanation-based constraint programming," in *CP01 Workshop on User-Interaction in Constraint Satisfaction*, 2001.