Choreographies: using Constraints to Satisfy Service Requests

Alexander Lazovik,1 Marco Aiello,1,2 and Rosella Gennari3
1. DIT, University of Trento, Via Sommarive 14, 38100 Trento, Italy
2. VITALab, TUWien, Argentinierstrasse 8, 1040 Wien, Austria
3. Bozen-Bolzano Free U., Piazza Domenicani 3, 39100 Bolzano, Italy
{lazovik,aiellom}@dit.unitn.it gennari@inf.unibz.it

Abstract

Interacting with a web service enabled marketplace 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 choreographies 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 request encoding is performed, and we illustrate an implementation using the Choco constraint system.

1 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 a choreography. Choreographies can be modeled in various ways, a novel proposal is to use constraints. In this way, user requests are interpreted as additional constraints to be satisfied against the given business process; the service providers’ requirements are then 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, 7, 8] or others based on logic programming formalisms (e.g., [6]). All mentioned approaches work under the assumption of having rich semantic service descriptions and run-time information at disposal. Artificial Intelligence techniques can provide a solution to the problem of service composition; e.g., there have been several proposals using AI planning [9], while encoding planning problems as constraints is in [2]. In [5], we proposed the XSRL request language over complex business domains. In [4], we presented the constraint model at the basis of the present work. In this paper, we propose a framework for the encoding of choreographies and requests as constraint set; finally, we propose an implementation with the Choco constraint system.

The remainder of the paper is organized as follows. A motivating example in the travel domain is introduced in Section 2. In Section 3, we present the framework for managing choreographies encoded as constraints. Section 4 presents the rules for encoding choreographies and requests. A snapshot of our implementation in Choco is shown in Section 5. Concluding remarks are presented in Section 6.

2 An example in the travel 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 do-
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). 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.

3 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 [4], we provide 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 1. The user request is also encoded as a set of constraints, as shown in Table 1, 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 constraint-based choreography framework to satisfy user requests. The framework consists of three main components: monitor, constraint programming system, and executor. 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
4 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;
- **(>)** 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 instantiation of a number of free variables. The latter process is an instance of information gathering at run time.

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

\[ \forall \xi_i \colon \Sigma_i \bowtie \text{value}, \]

- **value** is a value from the domain of the variable \( v \),
- \( \Sigma_i \) 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 \( \xi_i \) are non-controlled variables and the \( \beta_i \) are controlled variables,
- \( a_{i,k} \) is the effect of action \( a_i \) for outcome \( k \),
- \( \bowtie \) is either \(<,>,\geq,\leq\) or \( = \),
- and [ ] 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 \( \beta_i \), and non-controlled variables, denoted by \( \xi_i \). Non-controlled variables represent nondeterministic action outcomes. The underlying idea is that the constraint programming system
may not be 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 (states) which can be different; yet, once an action is invoked, we assume that its outcome is fixed. 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 \( CP = (\beta, \mathcal{N}, \xi, \mathcal{C}) \), where:

- \( \beta \) is a set of controlled boolean variables;
- \( \mathcal{N} \) is a set of controlled variables over integers;
- \( \xi \) 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 which satisfies all the constraints.

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.

4.1 Phase I

We consider the business process as a labeled-edge 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, [5]). 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 \( \mathcal{C}_e \) 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 \( \alpha_i \) is overloaded to represent not only the action, but also its effects.

The encoding is generated by 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 (see also Table 1):

(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.

(B) Single deterministic action \( a \) is encoded as \( \beta a \), where \( \beta \) 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): \( \beta_1(a_1 + \beta_2a_2) \). 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: \( \beta_1a_1 + \beta_2a_2 \), 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 \( \xi \) are introduced. The encoding (for a nondeterministic action with two possible outcomes) is the following: \( \xi_1a_1’ + \xi_2a_2’ \), 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.

4.2 Phase II

During the second phase of the encoding, we take a request (RL) and produce a set of constraints (⇒) 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 [5]. Here we give the basic definition of the language and we

<table>
<thead>
<tr>
<th>Type of action</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) No action</td>
<td>0</td>
</tr>
<tr>
<td>(B) Single deterministic action</td>
<td>( \beta a )</td>
</tr>
<tr>
<td>(C) Sequence of actions</td>
<td>( \beta_1(a_1 + \beta_2a_2) )</td>
</tr>
<tr>
<td>(D) Branching</td>
<td>( \beta_1a_1 + \beta_2a_2, \text{ with } \beta_1 + \beta_2 \leq 1 )</td>
</tr>
<tr>
<td>(E) Nondeterministic action</td>
<td>( \xi_1a_1’ + \xi_2a_2’ ), with ( \xi_1 + \xi_2 = 1 )</td>
</tr>
<tr>
<td>(F) Cycle: undirected cycle</td>
<td>state splitting, no specific encoding</td>
</tr>
<tr>
<td>(G) Cycle: directed cycle</td>
<td>( n a )</td>
</tr>
</tbody>
</table>
show the intuitions for the language constructs in Table 2.

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

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 \( x \) associated with \( c_v \) to \( x^0 \), by which we mean that the normal execution is followed, in place of the nondeterministic failure ones.

**atomic** \( v \doteq v_0 \). This is analogous to **vital**, except that the nondeterministic variables \( x \) 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 \( x \) are universally quantified over.

Next 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 \( \{ \beta^0_{t,k,i} \ldots \} \) denotes the branch variables in step \( t_k \), that has been chosen to satisfy the request \( g \). Then \( \sum \beta^0_{t,k,i} \neq 0 \) denotes that one of the \( \beta \) 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: \( \sum \beta^0_{t,k,i} \neq 0 \wedge \sum \beta^2_{t,k,i} \neq 0 \Rightarrow \forall i : \beta^0_{t,k,i} = \beta^2_{t,k,i} \). 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: \( \sum \beta^0_{t,k,i} \neq 0 \Rightarrow \forall i : \beta^0_{t,k,i} = \beta^2_{t,k,i} \).

**before** \( g_1 \text{ then } g_2 \). 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 \( \{ \beta^0_{t,k,i} \ldots \} \).

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: \( \sum \beta^0_{t,k,i} = 0 \Rightarrow \forall i : \beta^0_{t,k,i} = \beta^2_{t,k,i} \), 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 true.

5 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 \( a_0: \text{getHotelPrice} \) 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
<table>
<thead>
<tr>
<th>Request</th>
<th>Where satisfied</th>
<th>How encoded</th>
<th>Type of request</th>
</tr>
</thead>
<tbody>
<tr>
<td>vital $p$</td>
<td>In a state where $p$ holds to which there is a path from the initial state modulo failures</td>
<td>$\xi = \xi^0: c_v \triangleright v_0$</td>
<td>reachability</td>
</tr>
<tr>
<td>atomic $p$</td>
<td>In a state where $p$ holds to which there is a path from the initial state despite failures</td>
<td>$\forall \xi: c_v \triangleright v_0$</td>
<td>reachability</td>
</tr>
<tr>
<td>vital-maint $p$</td>
<td>In a state to which there is a path from the initial state modulo failures. $p$ must hold in all states along the path</td>
<td>$\xi = \xi^0: c_v(t_i) \triangleright v_0$, for all encoding steps $t_i$</td>
<td>maintainability</td>
</tr>
<tr>
<td>atomic-maint $p$</td>
<td>In a state to which there is a path from the initial state despite failures. $p$ must hold in all states along the path</td>
<td>$\forall \xi: c_v(t_i) \triangleright v_0$, for all encoding steps $t_i$</td>
<td>maintainability</td>
</tr>
<tr>
<td>prefer $g_1$ to $g_2$</td>
<td>In states where $g_1$ is satisfied, otherwise the satisfiability of $g_2$ is checked</td>
<td>variables in $g_1$ are instantiated before those in $g_2$</td>
<td>preference</td>
</tr>
<tr>
<td>optional $g$</td>
<td>States where $g$ is satisfied are checked first, otherwise the request is ignored</td>
<td>encoded as prefer $g$ to $\top$</td>
<td>preference</td>
</tr>
<tr>
<td>before $g_1$ then $g_2$</td>
<td>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</td>
<td>for all steps $t_k$: $\sum \beta_{g_1}^{t_k,i} \neq 0 \Rightarrow \forall i : \beta_{g_1}^{t_k,i} = \beta_{g_2}^{t_k,i}$</td>
<td>sequencing</td>
</tr>
<tr>
<td>achieve-all $g_1, \ldots, g_n$</td>
<td>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</td>
<td>reachability/reachability request pairs: for all steps $t_k$, for all reachability pairs $g_r_1, g_r_2$: $\sum \beta_{g_r_1}^{t_k,i} \neq 0 \land \sum \beta_{g_r_2}^{t_k,i} \neq 0 \Rightarrow \forall i : \beta_{g_1}^{t_k,i} = \beta_{g_2}^{t_k,i}$</td>
<td>composition</td>
</tr>
</tbody>
</table>

Table 2. Request language constructs.

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: $\beta_{s_0,0}(a_0 + \beta_{s_1,0}((\xi_{s_1,0} a_1^{\text{fail}} + \xi_{s_1,1}(a_1^{\text{ok}} + \beta_{s_2,0} a_2 + \beta_{s_2,1} a_3)))$; this represents the paths from state $s_1$ to $s_3$ with $n$ being the number of times the cycle is followed, and with $\beta$, representing the $j$-th choice in state $s_j$. 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 $\xi_{s_1,j}$ represents the $j$-th nondeterministic outcome in the state $s_j$. Suppose the user provides the following request:

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

The request is encoded as follows. Every basic request creates its own subset of controlled variables (Table 2). 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.

```plaintext
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_2 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 $a_1^{\text{fail}}$ 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: $\beta_{s_1,0} \beta_{s_2,0} = 1$:

```plaintext
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 $a_1^{\text{fail}} = 0$, 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.

\[
0.0 \leq 300.0 \\
0.0 + \beta_1 s_1 0 \times (x_1 s_1 1 \times (100.0)) \leq 300.0 \\
0.0 + \beta_1 s_1 0 \times (x_1 s_1 1 \times (100.0) + \\
\beta_2 s_2 0 \times (200.0) + \beta_2 s_2 1 \times (400.0)) \leq 300.0
\]

For each nested \textbf{vital} request pair within an \textbf{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, \textbf{achieve-all} adds the following constraints for \textbf{vital} requests (vital\_1 is a prefix for \textbf{vital hotelBooked} and vital\_2 for \textbf{vital trainReserved}). Note that constraints are added only in states where there are at least two deterministic actions:

\[
\text{if } (\text{vital}_1 \text{beta}_s_2 0 + \text{vital}_1 \text{beta}_s_2 1 = 1) \land \\
(\text{vital}_2 \text{beta}_s_2 0 + \text{vital}_2 \text{beta}_s_2 1 = 1) \\
\text{then } (\text{vital}_1 \text{beta}_s_2 0 = \text{vital}_2 \text{beta}_s_2 0) \land \\
(\text{vital}_1 \text{beta}_s_2 1 = \text{vital}_2 \text{beta}_s_2 1)
\]

Maintainability requests in \textbf{achieve-all} are treated differently: if a choice is made for any \textbf{vital} requests, then the same choice must be made for \textbf{vital-maint}. Thus, for each pair of \textbf{vital} and \textbf{vital-maint} inside the \textbf{achieve-all} request we have the following constraints (where vitalm\_1 is a prefix for request \textbf{vital-maint price} \( \leq 300 \)):

\[
\text{if } (\text{vital}_1 \text{beta}_s_0 0 = 1) \text{ then } \\
\text{vital}_1 \text{beta}_s_0 0 = \text{vitalm}_1 \text{beta}_s_0 0 \\
\text{if } (\text{vital}_1 \text{beta}_s_1 0 = 1) \text{ then } \\
\text{vital}_1 \text{beta}_s_1 0 = \text{vitalm}_1 \text{beta}_s_1 0 \\
\text{if } (\text{vital}_1 \text{beta}_s_2 0 + \text{vital}_1 \text{beta}_s_2 1 = 1) \text{ then } \\
(\text{vital}_1 \text{beta}_s_2 0 = \text{vital}_1 \text{beta}_s_2 0) \land \\
(\text{vital}_1 \text{beta}_s_2 1 = \text{vital}_1 \text{beta}_s_2 1)
\]

\[
\text{if } (\text{vital}_2 \text{beta}_s_0 0 = 1) \text{ then } \\
\text{vital}_2 \text{beta}_s_0 0 = \text{vital}_2 \text{beta}_s_0 0 \\
\text{if } (\text{vital}_2 \text{beta}_s_1 0 = 1) \text{ then } \\
\text{vital}_2 \text{beta}_s_1 0 = \text{vital}_2 \text{beta}_s_1 0 \\
\text{if } (\text{vital}_2 \text{beta}_s_2 0 + \text{vital}_2 \text{beta}_s_2 1 = 1) \text{ then } \\
(\text{vital}_2 \text{beta}_s_2 0 = \text{vital}_2 \text{beta}_s_2 0) \land \\
(\text{vital}_2 \text{beta}_s_2 1 = \text{vital}_2 \text{beta}_s_2 1)
\]

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

\[
\text{*** vital-maint (price} \leq 300) \\
\text{vitalm}_1 \text{beta}_s_0 0 = 1 \\
\text{vitalm}_1 \text{beta}_s_2 1 = 0 \\
\text{vitalm}_1 \text{beta}_s_2 0 = 1 \\
\text{*** vital (trainBooked} = \text{true}) \\
\text{vital}_2 \text{beta}_s_1 0 = 1 \\
\text{vital}_2 \text{beta}_s_2 0 = 0 \\
\text{vital}_2 \text{beta}_s_2 1 = 1 \\
\text{*** vital (hotelReserved} = \text{true}) \\
\text{vital}_1 \text{beta}_s_0 0 = 1 \\
\text{vital}_1 \text{beta}_m 2 1 = 0 \\
\text{vital}_1 \text{beta}_m 2 0 = 0
\]

Summarizing, the solution means that the plan involves first invoking the \textbf{reserveHotel} and then \textbf{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 \textbf{atomic} and \textbf{prefer-to}, the implementation of which is underway.

\section{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. Concrete test cases will be considered in our future work together with detailed comparisons with other approaches for handling choreographies.

\section{References}


