A Model-Based Synthesis Process for Choreography Realizability Enforcement

Marco Autili, Davide Di Ruscio, Amleto Di Salle, Paola Inverardi, and Massimo Tivoli*

Università degli Studi di L'Aquila, Italy
{marco.autili,davide.diruscio,amleto.disalle,paola.inverardi, massimo.tivoli}@univaq.it

Abstract. The near future in service-oriented system development envisions a ubiquitous world of available services that collaborate to fit users' needs. Modern service-oriented applications are often built by reusing and assembling distributed services. This can be done by considering a global specification of the interactions between the participant services, namely the choreography. In this paper, we propose a synthesis approach to automatically synthesize a choreography out of a specification of it and a set of services discovered as suitable participants. The synthesis is model-based in the sense that it works by assuming a finite state model of the services's protocol and a BPMN model for the choreography specification. The result of the synthesis is a set of distributed components, called coordination delegates, that coordinate the services' interaction in order to realize the specified choreography. The work advances the state-of-the-art in two directions: (i) we provide a solution to the problem of choreography realizability enforcement, and (ii) we provide a model-based tool chain to support the development of choreography-based systems.

Keywords: Service Choreographies, Model Driven Engineering, Service Oriented Architectures, Choreography Realizability Enforcement.

1 Introduction

The near future in service-oriented system development envisions a ubiquitous world of available services that collaborate to fit users' needs [7]. The trend is to build modern applications by reusing and assembling distributed services rather than realize standalone and monolithic programs. When building a service-based system, a possible Service Engineering (SE) approach is to compose together distributed services by considering a global specification, called *choreography*, of the interactions between the participant services. To this extent, the following two problems are usually considered: (i) *realizability check* - check whether the choreography can be realized by implementing/discovering each participant service so as it conforms to the played role; and (ii)

* This work is supported by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 257178 (project CHOReOS - Large Scale Choreographies for the Future Internet - www.choreos.eu).

V. Cortellessa and D. Varró (Eds.): FASE 2013, LNCS 7793, pp. 37–52, 2013.
© Springer-Verlag Berlin Heidelberg 2013

conformance check - check whether the set of services satisfies the choreography specification or not. In the literature many approaches have been proposed to address these problems (e.g., see [2,5,11] just to mention a few). However, by moving a step forward with respect to the state-of-the-art, a further problem worth considering concerns *realizability enforcement*. That is, given a choreography specification and a set of existing services discovered as suitable participants, restrict the interaction among them so to fulfill the collaboration prescribed by the choreography specification.

Contribution. In this direction, we propose a solution for realizability enforcement in the context of the CHOReOS EU project[1]. The core objective of CHOReOS is to leverage model-based methodologies and relevant SOA standards, while making *choreography* development a systematic process to the reuse and the assembling of services discovered within the Internet. CHOReOS revisits the concept of choreography-based service-oriented systems, and introduces a model-based development process and associated methods, tools, and middleware for *coordinating* services in the Internet. In particular, we describe how to automatically synthesize a choreography out of a specification of it and a set of discovered services. Since a choreography is a network of collaborating services, the notion of coordination protocol becomes crucial. In fact, it might be the case that the collaborating services, although potentially suitable in isolation, when interacting together can lead to *undesired interactions*. That is interactions that do not belong to the set of interactions modeled by the choreography specification. To prevent undesired interactions, we automatically synthesize additional software entities, called *Coordination Delegates* (CDs), and interpose them among the participant services. CDs coordinate the services' interaction in a way that the resulting collaboration realizes the specified choreography. This is done by exchanging suitable *coordination information* that is automatically generated out of the choreography specification. It is worth mentioning that, dealing with black-box services, it is not always be possible to synthesize suitable CDs. That is if all the behaviours produced by composing in parallel the participant services represent undesired interactions, then there is nothing to do in order to enforce the specified choreography.

Progress beyond state-of-the art. As already anticipated, on the one hand, we tackle the problem of realizability enforcement, which so far has been receiving little attention by the SE community. On the other hand, the definition of the CHOReOS process and its synthesis sub-process required the exploitation of state-of-the-art languages, systems, and techniques that have emerged in different contexts including SOA, model-transformations, and distributed coordination. Their integration and interoperability within the same technical space present the opportunity to harness the power and individual capabilities of different tools as part of a tool chain to support the systematic development of choreography-based systems which has thus far been largely missed.

Structure of the work. The paper is structured as follows. Section 2 describes the choreography synthesis process by means of an explanatory example, in the domain of *travel agency systems*. It gives an intuition of how CDs can be generated and used to enforce choreography realizability. In Section 3, we discuss the distributed coordination algorithm that characterizes the coordination logic performed by a synthesized CD. Furthermore, we provide details about the correctness of the algorithm with respect

[1] See at `www.choreos.eu`.

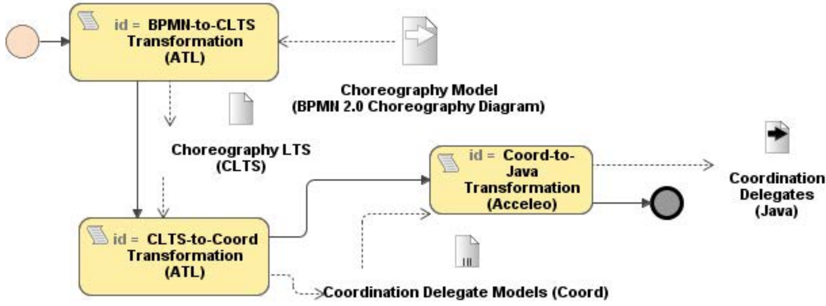

Fig. 1. The *Choreography Synthesis* process

to choreography enforcement, and we discuss the overhead due to the exchange of coordination information. Related works are discussed in Section 4. Section 5 concludes the paper and discusses future directions.

2 The Choreography Synthesis Process

The choreography synthesis process described in this section, and shown in Figure 1, is part of the overall CHOReOS development process [6]. The CHOReOS process leverages activities that range from *requirement specification* to *service discovery*, to *choreography synthesis*, to *choreography deployment and execution*, and to *design and run-time analysis*. As mentioned in Section 1, choreography synthesis is the main contribution of the work described in this paper and it aims at automatically generating CDs that correctly coordinate the discovered services in a distributed way.

For illustrative purposes we describe the synthesis process, by means of an explanatory example that concerns the development of a choreography-based *travel agency* system. Indeed, within CHOReOS, we applied our process to a real-scale case study, namely the *passenger-friendly airport scenario*. For space reasons, in this paper, we cannot show our approach at work on this scenario. However, the interested reader can refer to a demo publically available at the CHOReOS web-site[2].

Choreography Model. We use *BPMN2 Choreography Diagrams* as notation to specify choreographies. As shown in Figure 2, BPMN2 diagrams uses rounded-corner boxes to denote choreography tasks. Each of them is labeled with the roles of the two participants involved in the task, and the name of the service operation performed by the initiating participant and provided by the other one. A role contained in a light-gray filled box denotes the initiating participant. The diagram specifies that the travel agency system can be realized by choreographing four services: a `Booking Agency` service (`ba`), two `Flight Booking` services (`fb1` and (`fb2`)), and a `Hotel Booking` service (`hb`). In particular, (i) the booking of the flight has to be performed before the booking of the hotel and (ii) only the answer from one of the two flight booking services is taken into account (see the exclusive gateway represented as a rhombus in Figure 2).

[2] See at `http://www.choreos.eu/bin/Discover/videos`. The related development code is available at `http://www.choreos.eu/bin/Download/Software`.

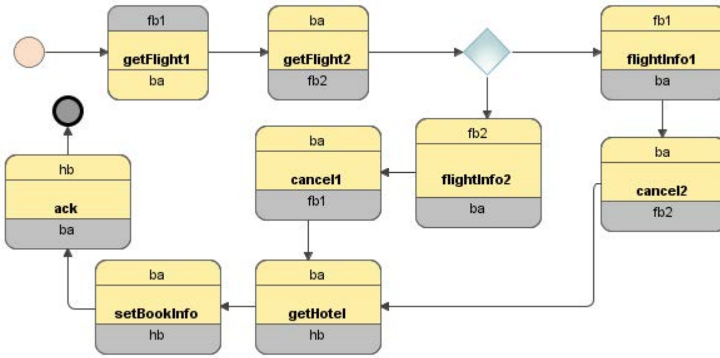

Fig. 2. BPMN2 choreography diagram for a Flight-Hotel Booking choreography

The choreography synthesis process generates the CDs required to realize a specified choreography. The generation process consists of three model transformations as discussed in the following.

BPMN-to-CLTS. By means of transformation rules implemented through the *ATLAS Transformation Language* [8] (ATL), the BPMN2 specification is transformed into an equivalent Choreography Labeled Transition System (CLTS) specification. Figure 3 shows the CLTS model for the BPMN2 choreography diagram in Figure 2. This model has been drawn by means of the developed GMF-based editor[3].

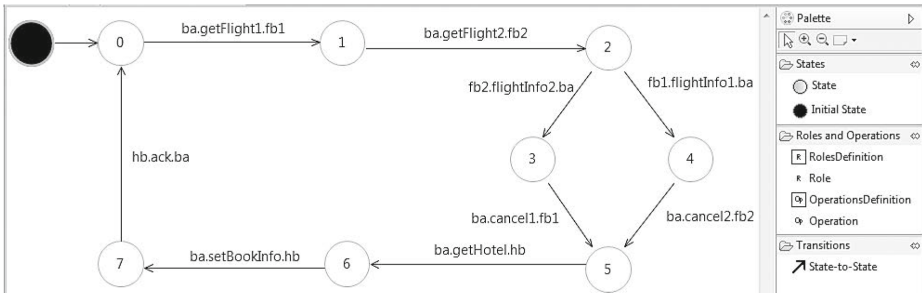

Fig. 3. CLTS model of the Flight-Hotel Booking choreography

Informally, a CLTS is a Labeled Transition System (LTS) that, for coordination purposes, is suitably extended to model choreography behavior, e.g., by considering conditional branching and multiplicities on participant instances. The transformation takes into account the main gateways found in BPMN2 Choreography Diagrams: exclusive gateways (decision, alternative paths), inclusive gateways (inclusive decision, alternative but also parallel paths), parallel gateways (creation and merging of parallel flows), and event-based gateways (choice based on events, i.e., message reception or timeout). For instance, the exclusive gateway in the BPMN2 diagram shown in Figure 2 has been

[3] See at `http://code.google.com/p/choreos-mde/`

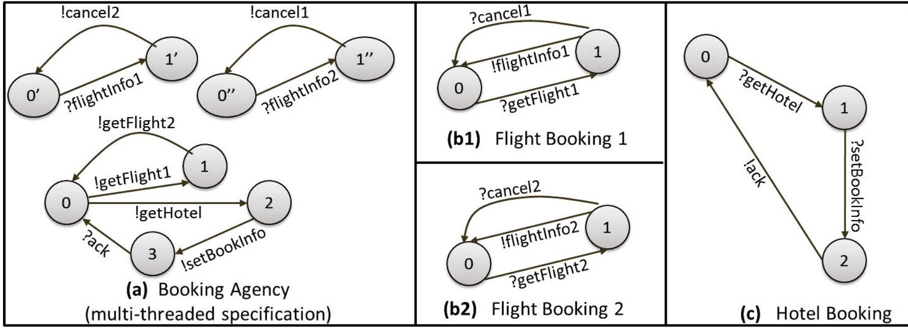

Fig. 4. LTSs for the services of the travel agency system

transformed to the exclusive branching in the CLTS diagram shown in Figure 3, hence generating two alternative paths outgoing from state 2.

Although this transformation is indispensable for the realization of the CHOReOS process, it does not represent an advance on the state-of-the-art per se. In fact, in the literature, there exist other similar attempts to transform business process models to automata-based models [3,17] (just to mention a few). For this reason, in the sequel, we do not further discuss this transformation.

Before describing the other two transformations, let us continue our example by discussing the problem underlying the notion of undesired interactions introduced in Section 1. The CLTS model in Figure 3 applies to the roles `ba`, `fb1`, `fb2`, and `hb` that, after discovery, are played by the `Booking Agency`, `Flight Booking 1`, `Flight Booking 2`, and `Hotel Booking` services, respectively. Figure 4 shows the interaction protocol of these services by using LTSs. The exclamation "!" and the question "?" marks denote required and provided operations, respectively. The `Booking Agency` service searches for a flight by exploiting two different flight booking services (see `!getFlight1` and `!getFlight2`). As soon as one of the two booking services answers by sending flight information (see `!flightInfo1` or `!flightInfo2`), the agency cancels the search on the other booking service (see `!cancel1` or `!cancel2`).

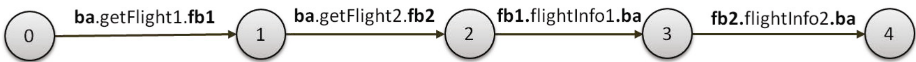

Fig. 5. A possible undesired interaction with respect to the Flight-Hotel Booking choreography

The above services have been discovered as suitable participants (i.e., each service conforms the role to be played) for the specified choreography[4]. However, this does not necessarily mean that the "uncontrolled" collaboration of the participant services is free

[4] Discovery issues and the problem of checking whether a service is a suitable participant for a choreography (conformance check) are out of the scope of this paper.

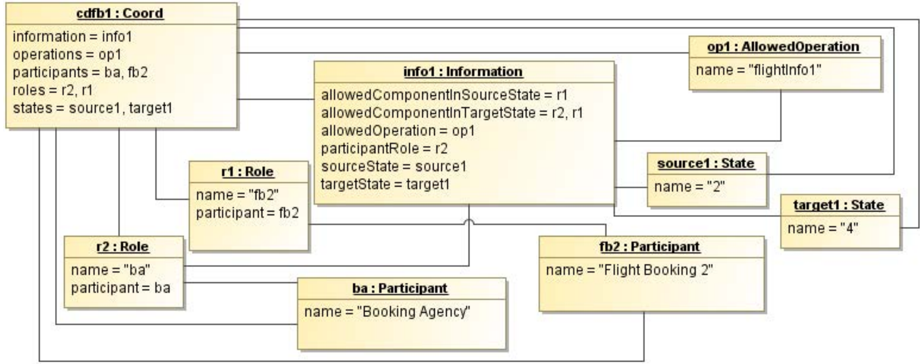

Fig. 6. Coord model for the `Flight Booking 1` service

from undesired interactions. In fact, Figure 5 shows a possible trace resulting from the parallel composition of the service protocols. This trace represents an undesired interaction, that is it is not included in the CLTS model shown in Figure 3, since both `fb1` and `fb2` proceed while only one of them should be allowed according to the exclusive branching in state 2. To prevent undesired interactions, the automatic synthesis of the CDs is carried out according to the `CLTS-to-Coord` and `Coord-to-Java` model transformations discussed below. It is worth mentioning that the kind of exclusive behaviour our running example is based on does not represent the only property that our approach is able to guarantee. In fact, in general, an undesired interaction in our approach can be represented as (the negation of) a *safety property* expressing what should not happen in the system composed by the participant services plus the CDs synthesized according to the specified CLTS.

CLTS-to-Coord. An ATL transformation is defined to automatically distribute the CLTS into a set of models, whose metamodel is denoted as `Coord` in Figure 1. A `Coord` model M_{CD_i}, for a coordination delegate CD_i, specifies the information that CD_i needs to know in order to properly cooperate with the other CDs in the system. The aim of this cooperation is to prevent undesired interactions in the global collaboration of the participant services, hence enforcing choreography realizability. Essentially, the developed ATL transformation consists of a number of rules each devoted to the managament of specific source modeling elements. For instance, given a source role name a target `Coord` instance is generated by a dedicated `Role2Cord` transformation rule. Moreover, all the source `Transition` elements give place to target `Information` elements.

Back to the example, Figure 6 shows the `Coord` model that represents the coordination information for the CD supervising `Flight Booking 1`. The `Coord` model contains the following information: when in the state `2` of the CLTS shown in Figure 3, `fb1` is allowed to perform the operation `flightInfo1` provided by `ba`, hence moving to the state `4`; when in the state `2` also `fb2` is allowed to perform an operation, namely `flightInfo2`, provided by `ba`, hence moving to the state `3`. However, since state `2` models an exclusive branching, only one of `fb1` and `fb2` must be allowed to proceed. Thus, concerning `fb1`, the CD supervising `Flight Booking 1` needs to

know that, when in the state 2, another service, i.e., `Flight Booking 2`, is allowed to take a move, and hence it must be blocked in order to solve the possible concurrency problem. Symmetrically, the CD supervising `Flight Booking 2` knows that `Flight Booking 1` must be blocked. As detailed in Section 3, the two CDs use coordination information to “perform handshaking” and “elect a winner”. This information is then exploited by the CDs to also keep track of the *global state* of the coordination protocol implied by the specified choreography, similarly to [2]. This means that each delegate can deduce the global state from the observation of the communication flow between the participant services.

Coord-to-Java. The `Coord` model specifies the logic that a CD has to perform independently from any target technology. To validate our approach in practical contexts, we chose Java as a possible target language of our Acceleo[5]-based model-to-code transformation. The Java code of a delegate CD_i exploits the information contained in its `Coord` model M_{CD_i}. Briefly, for each `Coord` model a Java class is generated by means of dedicated templates consisting of static and variable parts. The latter are fixed by means of the information retrieve from the source `Coord` model. The generated class implements an operation for each required operation of the supervised service.

Back to the example, from the `cdfb1 Coord` model, a *proxy* web service is generated as a *wrapper* for the operations required by `Flight Booking 1`. That is, the corresponding Java class implements the operation `flightInfo1`, which wraps the homonymous operation provided by `Booking Agency` and required by `Flight Booking 1`. Listing 1.1 shows an excerpt of the generated code for the `cdfb1` class. The `fb1Coord` class variable is used to store the `cdfb1 Coord` model. Such a model is used to coordinate the wrapped operations. For instance, after that the CD for `Flight Booking 1` verified that `flightInfo1` is an allowed operation with respect to the choreography global state (variable `globalState`), it establishes, through `handleRules` and `handleRule3`, whether the request of `flightInfo1` can be forwarded to `Booking Agency` (line 19) or not (line 28). The choreography global state is tracked by means of the asynchronous exchange of coordination information with the other CDs. Interestingly, `handleRules` and `handleRule3` are generic and do not depend on the information contained in `cdfb1`.

Listing 1.1. Fragment of the generated CD for `fb1`

```
1  @WebService( serviceName="cdfb1", targetNamespace="http://choreos.di.univaq.it"
       ↪, portName="fb1Port" )
2  public class cdfb1 {
3
4      private static CoordinationDelegate COORDINATION_DELEGATE = new
           ↪CoordinationDelegate("cdfb1");
5      private static final String REQUEST_FLIGHTINFO1 = "flightInfo1";
6      private static Coord fb1Coord = CoordFactory.eINSTANCE.createCoord();
7      private static ChoreographyState globalState = new ChoreographyState(
           ↪ChoreographyState.INITIAL_STATE);
8
9      public void cdfb1() {
10         ...
11     }
12
13     @WebMethod( operationName="flightInfo1" )
```

[5] `http://www.eclipse.org/acceleo/`

```
14      //@Oneway
15      public void flightInfo1() throws DiscardException {
16       CoordinationDelegateFacade facade = new CoordinationDelegateFacade();
17       CoordinationResult result = facade.handleRules(REQUEST_FLIGHTINFO1,
             ↪COORDINATION_DELEGATE, fb1coord, globalState);
18
19        if (result==CoordinationResult.FORWARD) {
20
21          //Forward message to the BookingAgency Service
22          BookingAgency_Service bookingAgencyService = new BookingAgency_Service
                ↪();
23          client.BookingAgency BookingAgencyPort = BookingAgencyService.
                ↪getBookingAgencyPort();
24          BookingAgencyPort.flightInfo1();
25
26          facade.handleRule3(REQUEST_FLIGHTINFO1, COORDINATION_DELEGATE, fb1coord
                ↪, globalState);
27        }
28        if (result==CoordinationResult.DISCARD) {
29          //Discard message
30          throw new DiscardException();
31        }
32      }
33  }
```

Once the implementation code has been generated for all the required CDs, services and CDs are composed together. Figure 7 shows the architectural configuration of the composition where ba, $fb1$, $fb2$, and hb are instances of `Booking Agency`, `Flight Booking 1`, `Flight Booking 2`, and `Hotel Booking`, respectively; `cdba`, `cdfb1`, `cdfb2`, , and `cdhb` are their respective CDs.

The required/provided interface bindings between a participant service and a CD are realized by means of synchronous connectors. A CD is connected to all the other CDs by means of asynchronous connectors (see the n-ary association shown in Figure 7 as a rhombus). The latter serve to exchange coordination information. As better explained in the next section, coordination information is exchanged only when synchronization is needed, i.e., when there is more than one component that is allowed to perform some action according to the current global state of the choreography model. For instance, in our example, this happens when both `fb1` and `fb2` can move from the state `2`. Note that, dealing with the reuse of existing (black-box) services, this is the best we can do in terms of the overhead due to the exchange of coordination information. In the next section we discuss why this is overhead is acceptable.

3 Distributed Coordination Algorithm

In this section we provide an algorithmic description of the coordination logic that a CD has to perform. The distributed coordination algorithm uses foundational notions as *happened-before* relation, *partial ordering*, *time-stamps*, and *total ordering*. The reader who is not fully familiar with such notions can refer to the work described in [9].

The standard time-stamp method is used in our approach to establish, at each CD, a total order of dependent *blocking* and *unblocking* messages, hence addressing starvation problems. Acknowledging messages are used to be sure that all the blocking messages (a CD has sent) have been actually received. In order to solve concurrency problems arising when two events associated with the same time-stamp must be compared, we

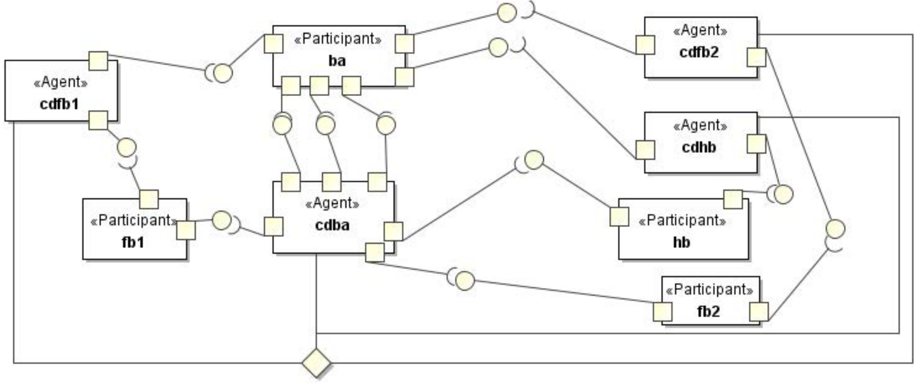

Fig. 7. Overall architecture of the choreography-based travel agency system

assume a priority order among the services to be choreographed. Since problems arising from network unreliability are out of scope, we assume that non-lossy, order-preserving and non-duplicating communications, among the CDs, are guaranteed.

In the style of [9], the distributed coordination algorithm is defined by the following rules that each delegate CD_i follows in a distributed setting, when its supervised service S_i performs a request of α, without relying on any central synchronizing entity or shared memory. These rules locally characterize the collaborative behavior of the CDs at run-time from a *one-to-many* point of view. To this end, each CD maintains its own *BLOCK queue* (i.e., the queue of blocking messages) that is unknown to the other delegates. At the beginning, each CD has its own timestamp variable set to 0 and, at each iteration of the algorithm, waits for either its supervised service to make a request or another CD to forward a request. The actions defined by each rule are assumed to form a single event (i.e., each rule has to be considered as atomic). Within the rules, we denote with TS_i the current timestamp for CD_i, and with s the current state of the CLTS model M_C of the choreography. Moreover, we denote with $Coord_i[h]$ the h-th coordination information element in the `Coord` model of CD_i; $Coord_i[h][\texttt{sourceState}]$ (resp., $Coord_i[h][\texttt{targetState}]$) is a state of M_C that is a source (resp., target) state for the transition labeled with $Coord_i[h][\texttt{allowedOperation}]$; $Coord_i[h][\texttt{allowedOperation}]$ is the operation that can be performed by S_i when M_C is in the state $Coord_i[h][\texttt{sourceState}]$; $Coord_i[h][\texttt{allowedServiceInSourceState}]$ (resp., $Coord_i[h][\texttt{allowedServiceInTargetState}]$) is the set of services (different from S_i) that, with respect to M_C, are allowed to move from $Coord_i[h][\texttt{sourceState}]$ (resp., $Coord_i[h][\texttt{targetState}]$). Detailed explanation of these rules is given below.

<u>**Rule 1:**</u> Upon receiving, from S_i, a request of α in the current state s of M_C,

1.1 if there exist h *s.t.* $Coord_i[h][\texttt{sourceState}] = s$ and $Coord_i[h][\texttt{allowedOperation}] = \alpha$ (i.e., α is allowed from s) **then**

1.1.1 CD_i updates TS_i to $TS_i + 1$;

1.1.2 for every CD_j *s.t.* $j \in Coord_i[h][\texttt{allowedServiceInSourceState}]$**:**

1.1.2.1 CD_i sends BLOCK(s,TS_i,from-CD_i,to-CD_j) to CD_j;
1.1.2.2 CD_i puts BLOCK(s,TS_i,from-CD_i,to-CD_j) on its BLOCK queue;

1.2 if there exist h *s.t.* $Coord_i[h][\texttt{sourceState}] \neq s$ and $Coord_i[h][\texttt{allowedOperation}] = \alpha$ (i.e., α is not allowed from s) **then** CD_i discards α;

1.3 if does not exist h *s.t.* $Coord_i[h][\texttt{allowedOperation}] = \alpha$ (i.e., α is not in the alphabet of M_C) **then** CD_i forwards α (hence synchronizing with S_i);

Rule 2: When a CD_j receives a BLOCK(s,TS_i,from-CD_i,to-CD_j) from some CD_i,
2.1 CD_j places BLOCK(s,TS_i,from-CD_i,to-CD_j) on its BLOCK queue;
2.2 if $(TS_j < TS_i)$ or $(TS_i = TS_j$ and $S_i \prec S_j)$ **then** CD_j updates TS_j to $TS_i + 1$; **else** CD_j updates TS_j to $TS_j + 1$;
2.3 CD_j sends ACK(s,TS_j,from-CD_j) to CD_i;

Rule 3: Once CD_i has received all the expected ACK(s,TS_j,from-CD_j) from every CD_j (see Rule 2), and it is granted the privilege (according to Rule 5) to proceed from state s,
3.1 CD_i forwards α;
3.2 CD_i updates s to s'=$Coord_i[h][\texttt{targetState}]$;
3.3 CD_i updates TS_i to $TS_i + 1$;
3.4 for every CD_j *s.t.* $j \in Coord_i[h][\texttt{allowedServiceInSourceState}]$ or $j \in Coord_i[h][\texttt{allowedServiceInTargetState}]$**:**
3.4.1 if $s == s'$ **then** CD_i removes any BLOCK(s,TS_i,from-CD_i,to-CD_j) from its own BLOCK queue; **else** CD_i empties its own BLOCK queue;
3.4.2 CD_i sends UNBLOCK(s',TS_i,from-CD_i) to CD_j;

Rule 4: When a CD_j receives an UNBLOCK(s',TS_i,from-CD_i) from some CD_i,
4.1 CD_j updates s to s';
4.2 if $(TS_j < TS_i)$ or $(TS_i = TS_j$ and $S_i \prec S_j)$ **then** CD_j updates TS_j to $TS_i + 1$; **else** CD_j updates TS_j to $TS_j + 1$;
4.3 if $s == s'$ **then** CD_j removes any BLOCK(s,TS_i,from-CD_i,to-CD_j) from its BLOCK queue; **else** CD_j empties its own BLOCK queue;
4.4 CD_j retries Rule 1 from the (updated) state s;

Rule 5: CD_i is granted the privilege to proceed from the current state s of M_C when, ranging over j, **for every** pair of messages BLOCK(s,TS_i,from-CD_i,to-CD_j) and BLOCK(s,TS_j,from-CD_j,to-CD_i) on its BLOCK queue**:** either (i) $TS_i < TS_j$ or (ii) $TS_i = TS_j$ and $S_i \prec S_j$;

If the conditions on Rule 1.2 hold (i.e., the conditions on Rules 1.1 and 1.3 fail), it means that S_i is trying to perform an operation that is in the alphabet of M_C but is not allowed from the current state of M_C. In this case, CD_i prevents S_i to perform that operation by discarding it. Indeed, one cannot always assume that the actual code of a (black-box) service has been developed in a way that it is possible to discard a service operation by the external environment. Actually, it can be done only if the developer had preemptively foreseen it and, for instance, an exception handling logic was aptly

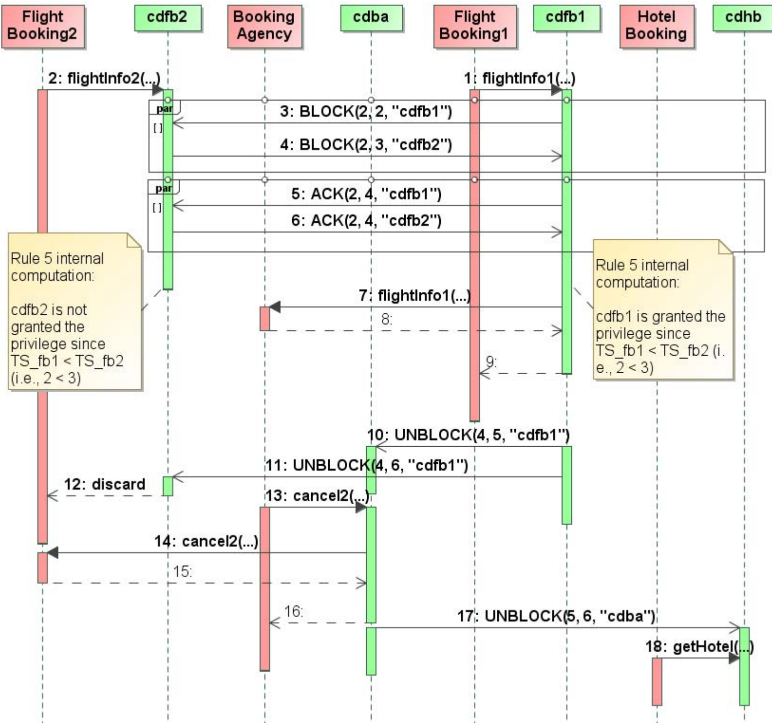

Fig. 8. An excerpt of a possible execution of the distributed coordination algorithm

coded for such an operation. However, a service client can easily detect those operations that are *controllable* by the external environment, e.g., through the declaration of *thrown* exceptions on interface operations, or of *fault messages* on WSDL operations, or simply of *error return values* for class methods. Since in this paper we focus on the automatic distribution of the choreography-based coordination logic, we avoid to address *controllability* issues and we assume that all service operations are controllable. However, the extension to account for controllability issues is straightforward.

Rule 1.3, allows CDs to be *permissive* on the operations that do not belong to the alphabet of M_C (i.e., operations "outside the scope" of the choreography). Note that one could instead choose to be *restrictive* on those operations by disabling Rule 1.3 hence preventing the service to perform those operations by discarding them (as in the case of a service trying to perform an operation that is in the alphabet of M_C but is not allowed from the current state).

Rule 4.4 resumes the execution of an unblocked CD by "restarting" from Rule 1. If this CD is still trying to handle a request α that is *pending* from the previous iteration of the algorithm (see the operation `flightInfo2` in the coordination scenario shown in Figure 8), retrying Rule 1 means to directly re-check the conditions of Rules 1.1, 1.2, and 1.3 with the new updated state and the pending α. Otherwise, it means that the CD retries Rule 1 from an updated choreography global state.

It is worthwhile to observe that conditions (i) and (ii) of Rule 5 are tested locally by a CD.

Correctness. The above algorithm satisfies three crucial conditions [9] for *correct* distributed coordination: (1) a coordination delegate which has been granted the privilege to proceed must proceed and unblock the other competing delegates before the privilege to proceed can be granted to another delegate; (2) different block messages for granting the privilege to proceed must be privileged in the order in which they are made, excluding the ones "associated" to discarded operations; (3) if every coordination delegate which is granted the privilege to proceed eventually proceeds and unblocks the other competing delegates, then every block message for granting the privilege to proceed is eventually privileged, excluding the ones "associated" to discarded operations. In fact, condition (i) of Rule 5, together with the assumption that the messages concerning coordination information are received in order, guarantees that CD_i knows all operation requests which preceded its current operation request. Since Rules 3 and 4 are the only ones which remove messages from the BLOCK queue, condition (1) trivially holds. Condition (2) follows from the fact that the total ordering $\prec$ (happened-before relation plus component priority) extends the partial ordering $\rightarrow$ (happened-before relation). Rule 2 guarantees that after CD_i requests the privilege to proceed (by sending BLOCK messages), condition (i) of Rule 5 will eventually hold. Rules 3 and 4 imply that if each coordination delegate which is granted the privilege to proceed eventually proceeds and unblocks the other competing delegates, then condition (ii) of Rule 5 will eventually hold, thus ensuring condition (3).

Analysis of the overhead due to the exchange of coordination information. The overhead due to the exchange of coordination information among the coordination delegates is acceptable. First of all, note that BLOCK messages are exchanged only when non-determinism occurs from the current state s of M_C. In the worst case[6], the non-determinism degree is asymptotically bounded by the number n of components, i.e., it is $O(n)$. For each received BLOCK message an ACK message is exchanged. UNBLOCK messages are instead exchanged at each state of M_C and for a maximum number that is $O(n)$. Thus, if m is the number of states of M_C then the maximum number of coordination information messages (BLOCK, UNBLOCK, ACK) that are exchanged is $O(3 * m * n)$, i.e., $O(m * n)$. However, very often, in the practice, $n \leq m$ holds ($m \leq n$ is less frequent). This means that the maximum number of exchanged coordination information messages can be considered as $O(m^2)$. We can, then, conclude that the introduced overhead is polynomial in the number of states of M_C and, hence, acceptable further considering that the size of coordination information messages is insignificant. Indeed, as also shown by the work described in [9], this is the minimum that one can do to ensure correct distributed coordination.

By continuing the explanatory example introduced in Section 2, we better show how CDs use, at run-time, the information in their `Coord` models to correctly and distributively interact with each other, hence enforcing the realizability of the choreography

[6] Note that, in the practice, the worst case is unusual.

specified by M_C. By referring to Figure 3, we focus on the fact that only the answer from one of the two flight booking services is taken into account. Following the rules of the distributed coordination algorithm, Figure 8 shows how `Flight Booking 2` is blocked whenever `Flight Booking 1` is faster in collecting the information to be provided to `Booking Agency`.

The shown scenario concerns an excerpt of a possible execution of the distributed coordination algorithm. It starts when the two allowed operations `flightInfo1` and `flightInfo2`, required by `Flight Booking 1` and `Flight Booking 2` respectively, concurrently occur while in the current state `2` of the CLTS model of the choreography. At state `2`, the timestamps for `Flight Booking 1` and `Flight Booking 2` are 1 and 2, respectively. Furthermore, `Flight Booking 1` $\prec$ `Flight Booking 2`.

4 Related Work

The approach to the automatic generation of CDs presented in this paper is related to a number of other approaches that have been considered in the literature.

Many approaches have been proposed in the literature aiming at automatically composing services by means of BPEL, WSCI, or WS-CDL choreographers [4,5,10,13,16]. The common idea underlying these approaches is to assume a high-level specification of the requirements that the choreography has to fulfill and a behavioral specification of the services participating in the choreography. From these two assumptions, by applying data- and control-flow analysis, the BPEL, WSCI or WS-CDL description of a centralized choreographer specification is automatically derived. This description is derived in order to satisfy the specified choreography requirements.

In particular, in [16], the authors propose an approach to automatically derive service implementations from a choreography specification. In [13], the author strives towards the same goal, however assuming that some services are reused. The proposed approach exploits wrappers to make the reused services match the choreography.

Most of the previous approaches concern orchestration that is the most common approach to service composition. Conversely, our approach is one of the few in the literature that consider choreography as a means for composing services. Despite the fact that the works described in [13,16] focus on choreography, they consider the problem of automatically checking whether a choreography can be realized by a set of interacting services, each of them synthesized by simply projecting the choreography specification on the role to be played. This problem is known as *choreography realizability check*. Note that it is a fundamentally different problem with respect to the one considered in this paper, i.e., *choreography realizability enforcement*. In fact, our approach is reuse-oriented and aims at restricting, by means of the automatically synthesized CDs, the interaction behavior of the discovered (third-party) services in order to realize the specified choreography. Differently, the approaches described in [13,16] are focused on verifying whether the set of services, required to realize a given choreography, can be easily implemented by simply considering the role-based local views of the specified choreography. That is, this verification does not aim at synthesizing the coordination logic, which is needed whenever the collaboration among the services leads to global interactions that violate the choreography behavior.

In [12] a game theoretic strategy is used for checking whether incompatible component interfaces can be made compatible by inserting a converter between them. This approach is able to automatically synthesize the converter. Contrarily to what we have presented in this paper, the synthesized converter can be seen as a centralized CD.

In our previous work [1] a preliminary version of the coordination algorithm presented in Section 3 has been applied in a component-based setting, namely EJB components for J2EE component-based systems, to support automated composition and coordination of software components. In this paper, it has been completely revised to deal with service-oriented systems and solve some open issues. For instance, differently from what is done in [1], we are able to prevent undesired interactions without producing a centralized model of the coordinator.

In [14], the authors show how to monitor safety properties locally specified (to each component). They observe the system behavior simply raising a *warning message* when a violation of the specified property is detected. Our approach goes beyond simply detecting properties (e.g., a choreography specification) by also allowing their enforcement. In [14] the best thing that they can do is to reason about the global state that each component *is aware of*. Note that, differently from what is done in our approach, such a global state might not be the actual current one and, hence, the property could be considered guaranteed in an *"expired"* state. Another work in the area of the synthesis of runtime monitors from automata is described in [15]. Note that runtime monitoring is mostly focused on the detection of undesired behaviours, while runtime enforcement focuses on their prevention/solution.

5 Conclusions and Future Work

In this paper we presented a model-based synthesis process for automatically enforcing choreography realizability. The main contributions of the presented work with respect to the *choreography generation* research area are: (i) an automated solution to the problem of choreography realizability enforcement, in contrast with the fundamentally different problem of choreography realizability check; (ii) the formalization of a distributed algorithm for choreography-based coordination; (iii) the definition of model transformations capable to produce both the model and the actual implementation of a *choreographer* distributed into a set of cooperating CDs - this is done without generating any centralized model, hence addressing state-explosion problems and scalability issues; and (iv) the full automation and applicability of the approach to practical contexts, e.g., SOAP Web-Services.

In Section 4, we related our approach to existing centralized solutions. Summing up, the most relevant advantage of our approach with respect to these solutions is that the degree of parallelism of the system is maintained despite the introduction of the coordinator. Often, centralized approaches do not permit parallelism since the coordinator is usually implemented as a centralized single-threaded component and the communication with it is synchronous. For space reasons, we cannot further detail this discussion and the reader who is interested on it can refer to the CHOReOS web site.

The approach is viable and the automatically generated code allows for the correct enforcement of the specified choreography. The proposed approach has

already been applied to a large-scale realistic case study, namely the passenger-friendly airport scenario and a public demo is available at the CHOReOS web-site `http://www.choreos.eu/bin/Download/Software`. Currently, we are applying the process at two other industrial case studies of CHOReOS in the domains of marketing and sales, and Internet of things. The results will also be publicly available by the CHOReOS web site. The current implementation of the whole approach supports the generation of Java code for coordinating SOAP-based Web-services. Considering the general-purpose nature of the approach, other languages and application domains are eligible, and other forms of wrapping can be easily realized.

An interesting future direction is the investigation of non-functional properties of the choreography, e.g., by extending the choreography specification with performance or reliability attributes and accounting for them in the CDs synthesis process.

As discussed in Section 3, our approach allows supervised services to perform an operation that is outside the scope of the specified choreography. In this sense our approach is permissive. However, it can be parameterized to be either permissive or restrictive with respect to these operations. However, simply enabling or disabling the execution of operations outside the scope of the choreography is a trivial strategy. In the future we plan to investigate, and embed into the approach implementation, more accurate strategies to suitably deal with these operations.

This paper has been mainly focused on describing the model-based and automatic synthesis of CDs at work, within a choreographic static scenario. Thus, as further future work, dynamic scenarios should be considered and our process should be revised accordingly. For instance, such scenarios are related to contexts in which services may change their behaviour according to the "global state" of the choreography.

The correctness of our coordination algorithm with respect to choreography enforcement has been informally discussed in Section 3. For a rigorous assessment of our method, as another future work, a formal proof of the algorithm is needed.

References

1. Autili, M., Mostarda, L., Navarra, A., Tivoli, M.: Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems. Journal of Systems and Software 81(12), 2210–2236 (2008)
2. Basu, S., Bultan, T.: Choreography conformance via synchronizability. In: Proceedings of WWW 2011, pp. 795–804 (2011)
3. Bisztray, D., Heckel, R.: Rule-Level Verification of Business Process Transformations using CSP. In: Proceedings of GT-VMT 2007 (2007)
4. Brogi, A., Popescu, R.: Automated Generation of BPEL Adapters. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 27–39. Springer, Heidelberg (2006)
5. Calvanese, D., Giacomo, G.D., Lenzerini, M., Mecella, M., Patrizi, F.: Automatic service composition and synthesis: the roman model. IEEE Data Eng. Bull. 31(3), 18–22 (2008)
6. CHOReOS Consortium. CHOReOS dynamic development model definition - Public Project deliverable D2.1 (September 2011)
7. ERCIM News. Special Theme: Future Internet Technology. Number 77 (April 2009)
8. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)

9. Lamport, L.: Time clocks, and the ordering of events in a distributed system. Commun. ACM 21, 558–565 (1978)
10. Marconi, A., Pistore, M., Traverso, P.: Automated Composition of Web Services: the ASTRO Approach. IEEE Data Eng. Bull. 31(3), 23–26 (2008)
11. Poizat, P., Salaün, G.: Checking the Realizability of BPMN 2.0 Choreographies. In: Proceedings of SAC 2012, pp. 1927–1934 (2012)
12. Passerone, R., De Alfaro, L., Henzinger, T.A., Sangiovanni-Vincentelli, A.L.: Convertibility Verification and Converter Synthesis: Two Faces of the Same Coin. In: ICCAD (2002)
13. Salaün, G.: Generation of service wrapper protocols from choreography specifications. In: Proceedings of SEFM (2008)
14. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Proceedings of ICSE 2004 (2004)
15. Simmonds, J., Gan, Y., Chechik, M., Nejati, S., O'Farrell, B., Litani, E., Waterhouse, J.: Runtime monitoring of web service conversations. IEEE T. Services Computing 2(3) (2009)
16. Su, J., Bultan, T., Fu, X., Zhao, X.: Towards a Theory of Web Service Choreographies. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 1–16. Springer, Heidelberg (2008)
17. Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases 14(1) (2003)