Realizability of Choreographies using Process Algebra Encodings

Gwen Salaün[1] and Tevfik Bultan[2]

[1] University of Málaga, Spain
salaun@lcc.uma.es
[2] University of California, Santa Barbara, USA
bultan@cs.ucsb.edu

Abstract. Service-oriented computing has emerged as a new programming paradigm that aims at implementing software applications which can be used through a network via the exchange of messages. Interactions among a set of services involved in a new system are described from a global point of view using *choreography* specification languages such as WS-CDL or collaboration diagrams. In this paper, we present an encoding of collaboration diagrams into the LOTOS process algebra. This encoding allows to (i) check choreography specification using the LOTOS verification toolbox (CADP), (ii) check realizability of collaboration diagrams for both synchronous communication and bounded asynchronous communication, and (iii) automate service peer generation. *Realizability* indicates whether peers can be generated from a choreography such that they will behave exactly as formalized in its specification. If the collaboration diagram is unrealizable, our approach extends the peer generation process by adding some communications that make the peers respect the choreography specification.

1 Introduction

Formal methods play a key role in many open research problems that are of significant importance to the field of service-oriented applications. This is the case for problems related to *choreography*, *i.e.*, specification of interactions among a set of services from a global point of view. Several formalisms have already been proposed to specify choreographies: WS-CDL, collaboration diagrams, process calculi, BPMN, SRML, etc. Given a choreography specification, it would be desirable if the local implementations, namely *peers*, can be automatically generated via projection. However, generation of peers that precisely implement the choreography specification is not always possible: This problem is known as *realizability*.

Recent works on this topic [10, 15, 4, 2] advocate techniques to check the realizability of a choreography, or define well-formedness rules to be applied while writing the choreography specification in order to ensure its realizability. To the best of our knowledge, no solution has been proposed yet to generate peers realizing any choreography without adding rules or constraints on the choreography language or on specifications written with it. Accordingly, our contribution is twofold. First, our solution generates peers for any choreography specification by extending them with additional messages if the choreography is unrealizable. Second, our approach is supported by tools for verification of choreographies, testing realizability, and generation of peers in a completely automated way.

In this paper, we use collaboration diagrams as the choreography specification language. We propose an encoding of collaboration diagrams into the LOTOS process algebra. We chose LOTOS because it provides a good level of expressiveness to describe all the collaboration diagram interaction constraints, and is equipped with a rich toolbox (CADP) which offers state-of-the-art tools for state space exploration and verification. The LOTOS encoding allows to (i) verify choreography specification using CADP, (ii) check realizability of collaboration diagrams for both synchronous communication and bounded asynchronous communication[1], and (iii) automate service peer generation, adding new messages if the diagram is unrealizable.

The rest of this paper is organized as follows. Section 2 introduces collaboration diagrams and the problem of their realizability. Section 3 presents our encoding into LOTOS, and how this encoding is used to test realizability and generate peers. Section 4 sketches the tools that support our approach. Section 5 compares our proposal to related work, and Section 6 ends the paper with some concluding remarks.

2 Collaboration Diagrams

A collaboration diagram [2] (called communication diagram in UML 2) consists of a set of peers, a set of links between peers, and a set of message send events associated with links. An event is a tuple containing a dependency relation (facultative), a (unique) label, a message, and a recurrence type. Labels (1, 2, 3, ..., A1, A2, A3, ..., B1, B2, B3, ...) contain prefixes (ε, A, B) that organize events into different threads. All messages in one thread share the same prefix and execute based on the numerical order defined by their labels. Messages from different threads occur concurrently, and can be interleaved in any order that respects the dependency relation. A recurrence type is either "1" (default type) meaning that the associated event happens once, "?" for a conditional event (the event may occur once or it may not occur at all), or "*" for an iterative event (the event may not occur at all or it may occur multiple times).

Figure 1 presents a collaboration diagram for a train station service that we will use as a running example throughout this paper. This diagram contains four peers, respectively Customer, TrainStation, Availability, and Booking. It involves

[1] Promela (with its SPIN model-checker) is a formalism we could have used as an alternative to LOTOS, since Promela supports both synchronous and asynchronous communications whereas asynchronous communication can be expressed in LOTOS by explicitly encoding queues. However, SPIN does not provide behavioural equivalence checking we needed for the approach at hand.

three threads: 1) The main thread with prefix ε and events 1 and 2; 2) The A thread with prefix A and events A1, A2 and A3; and 3) The B thread with prefix B and events B1, B2 and B3. The collaboration diagram starts by the emission of a request to the train station (event 1). Next, the station checks ticket availability (events A1, A2, and A3), reserves tickets (events B1, B2, and B3), and sends the result to the customer (event 2).

Let us focus on thread A. It contains three events, the first one, 1/A1:info, means that the message info should be sent by peer TrainStation to peer Availability only after the execution of the event 1. I.e., the tuple (1, A1) is included in the dependency relation, indicating that the event A1 is executable only after the event with label 1 (namely 1:request) has been executed. The third event of thread A (A3:itinerary *) must be run after A2 (due to the sequential order within a thread), and can be executed several times (due to "*" recurrence type).

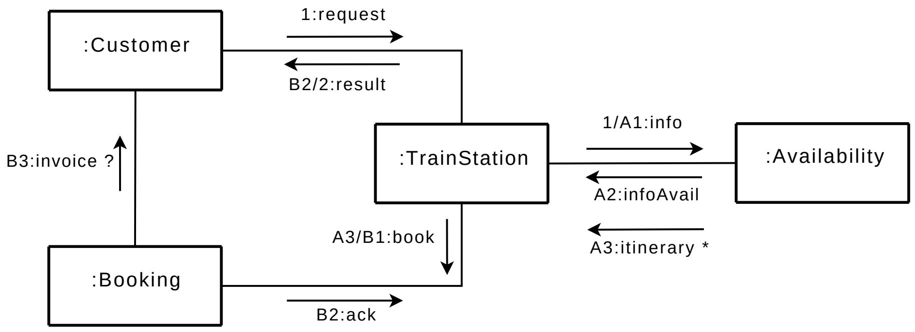

Fig. 1. Train station service collaboration diagram

Before illustrating the realizability problem for collaboration diagrams, let us introduce the *peer model.* A peer is described as a Labeled Transition System (LTS). An LTS is a tuple (A, S, I, F, T) where: A is an alphabet, that is a set of messages with direction ("!" for emission, and "?" for reception), S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ are final states, and $T \subseteq S \times A \times S$ is the transition function. Peers interact using binary communication on same message names with opposite directions. In this paper, we will consider both synchronous and asynchronous communication models.

A couple of unrealizable collaboration diagrams are presented in Figure 2. The first one (left hand side) is unrealizable because it is impossible for C to know when A sends its request message (no interaction between A and C). Hence, the peers cannot respect the execution order of messages as specified in the collaboration diagram. The second one is slightly more subtle because this diagram is realizable for synchronous communication, and unrealizable for asynchronous communication. Indeed, in case of synchronous communication, C can synchronize with A (rendez-vous) only once request is sent, so the message order is respected. This is not the case for asynchronous communication, since C sends its message to A without knowing if A has sent request or not. Therefore, the correct order between the two messages cannot be satisfied. We also give in Figure 2 the LTS generated for peer A.

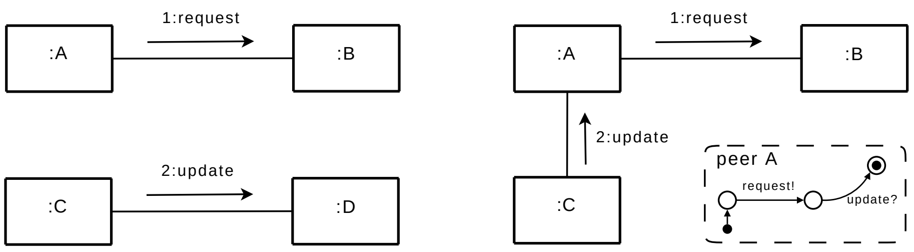

Fig. 2. Examples of unrealizable collaboration diagrams

Although realizability is easily figured out for these simple examples, it is much more complicated to say if the collaboration diagram presented in Figure 1 is realizable or not. We present in Section 3 an approach to automate the realizability test, and show that the train service collaboration diagram is unrealizable for asynchronous communication.

3 Encoding into LOTOS

The backbone of our proposal is an encoding of collaboration diagrams into the LOTOS process algebra [9]. We chose LOTOS because it relies on a rich notation that allows to specify complex concurrent systems possibly involving data types. In a second step, the LOTOS encoding allows (i) choreography verification by using model checking tools available in CADP [8], (ii) realizability test and (iii) generation of service peer implementations both for synchronous and bounded asynchronous communication models. The different steps of our approach are completely automated by different tools we present in Section 4.

3.1 Encoding Collaboration Diagrams into LOTOS

The collaboration diagram choreography is encoded using a LOTOS process. This process is split up in as many parts (referred as thread behaviour in the following) as there are threads in the diagram. Each thread behaviour encodes all the messages involved in its thread in the order in which they must be executed (achieved using the LOTOS action prefix operator). Each message is encoded using source and target peer names as prefixes to avoid name clashes. The conditional recurrence type "?" is encoded as a choice between the actual message (condition is true), and a termination (exit) meaning that the condition is false and the message not transmitted. The iterative recurrence type "*" is translated into LOTOS using an intermediate looping process whose behaviour is: message; loop_process [message] [] i ; exit.

Each thread behaviour evolves independently, and they synchronize together to respect dependency constraints (explicitly specified at the beginning of some events, *e.g.*, 1/A1:info) using new messages prefixed by "SYNC_". These messages are inserted in the LOTOS specification in two cases: (i) before running a message if this event depends on another message execution, (ii) after a message when this message appears in a dependency relation in the diagram. In the second case, the synchronization message should not block the thread execution, accordingly it is interleaved with the rest of the thread behaviour.

Let us give a part of the LOTOS code generated for our running example. We can distinguish the three threads, respectively for events starting by A, B, and numbers. Thread A for instance contains three messages (info, infoAvail, and itinerary) which are encoded in sequence and prefixed with peers participating in these interactions (only initials for readability reasons). The last message (a_ts_itinerary) involves an iterative recurrence type and is therefore translated using a loop_process. An example of "?" recurrence type is given at the end of thread B where the choice ([]) is used to express the execution of message invoice (b_c_invoice) or not (exit).

As regards synchronization between thread behaviours, we can see for instance that thread A synchronizes with the two others on messages SYNC_1 and SYNC_A3. SYNC_1 is used to synchronize thread A and the main thread in order to run message ts_a_info with label A1 after message c_ts_request with label 1 (execution of SYNC_1 acts as a pre-condition to the execution of ts_a_info). In thread B, the event B1 can only occur after A3 therefore message a_ts_itinerary (which is labeled by A3) is followed by a SYNC_A3 message, in order to run ts_b_book after A3. Note that the synchronization message SYNC_A3 should not block the thread execution. Accordingly it is interleaved with the rest of the thread behaviour (exit in this case, since it is the end of the thread).

```
(             (* -- thread A encoding -- *)
   SYNC_1;
      ts_a_info;
         a_ts_infoAvail;
            loop_process [a_ts_itinerary] >>
               ( SYNC_A3; exit ||| exit )
)
|[SYNC_1, SYNC_A3]|
(             (* -- thread B encoding -- *)
   (
      SYNC_A3;
         ts_b_book;
            b_ts_ack;
               (
                  SYNC_B2; exit
```

```
                     |||
                     (
                         b_c_invoice; exit
                         []
                         exit
                     ) >> exit
                  )
      )
      |[SYNC_B2]|
      ( ...  (* -- main thread's encoding -- *) )
   )
```

From this encoding, the corresponding LTS can be generated using CADP state space generation tools, and verified using the Evaluator model-checker [13]. We checked for instance the liveness property stating that each c_ts_request is eventually answered (ts_c_result). We show in Figure 3, the LTS obtained for the collaboration diagram from the LOTOS encoding. This LTS was obtained by hiding “SYNC_” messages, and by minimizing the resulting LTS (determinization, removal of τ transitions[2], and suppression of similar paths) using reduction techniques[3] available in the CADP toolbox.

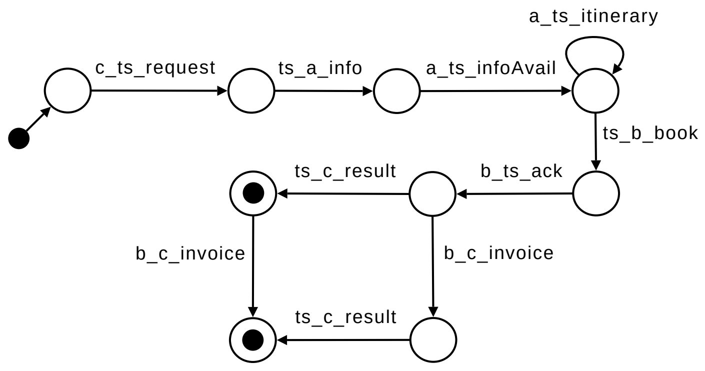

Fig. 3. Train station service: collaboration diagram LTS

3.2 Peer Generation

Peers are generated by projection from the LOTOS process encoding the collaboration diagram. This is achieved by generating a LOTOS process for each peer

[2] τ transitions stand for internal actions. These transitions are generated while compiling the LOTOS code. For example the LOTOS sequential composition “>>” inserts such a τ transition in the corresponding state space.

[3] In this paper, minimizations are achieved using weak trace, safety and strong reductions.

whose body is an instance of the collaboration diagram process, and hiding in this process all the messages in which the peer does not participate in.

Figure 4 gives a graphical view of peers generated for our running example from their LOTOS descriptions. For instance, peer Booking (Fig. 4, (b)) starts receiving a book request (ts_b_book?) from the train station, sends back an acknowledgement (b_ts_ack!), and either stops or sends an invoice to the customer (b_c_invoice!). We recall that peers interact on same message names with opposite directions, *e.g.,* c_ts_request! in the customer with c_ts_request? in the train station.

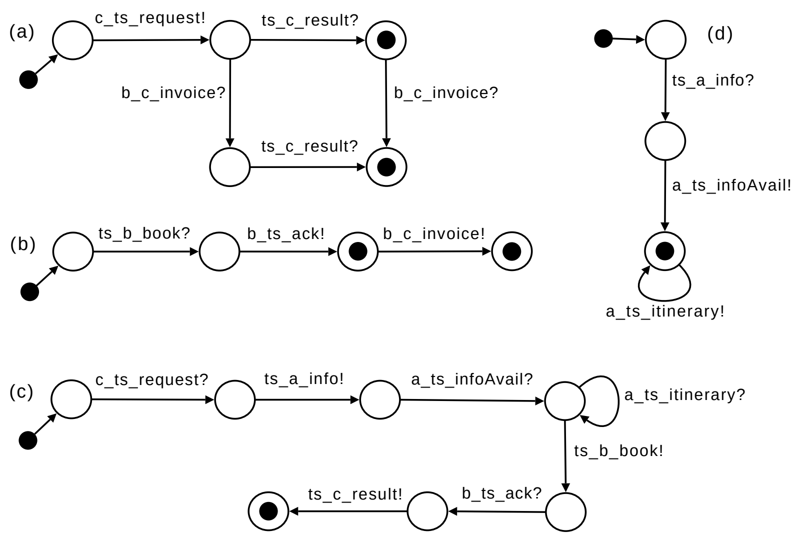

Fig. 4. Peers generated from the collaboration diagram: (a) customer, (b) booking, (c) train station, (d) availability

Once peers are generated, it is very difficult to say if their execution will respect the interaction constraints specified in the collaboration diagram (order of messages within a thread, and inter-thread message dependencies). In the next subsection, we propose automated techniques to check realizability.

3.3 Realizability

Our idea is to compute realizability by comparing the collaboration diagram LTS with the system composed of interacting peers using behavioural equivalences [14]. If these two systems are equivalent, it means that the peer generation exactly preserves the collaboration diagram constraints. If they are not, it is because peers do not generate the same interactions than those specified in the diagram, therefore it is unrealizable. Computing realizability is achieved in two steps: (i) generation of the system composed of interacting peers, and (ii) equivalence checking between the LTS resulting from step (i) and the collaboration diagram LTS. In the following, we consider both synchronous and bounded asynchronous communication models.

Synchronous communication. LOTOS relies on synchronous communication, therefore from the LOTOS code obtained previously, we generate an LTS for each peer process, and compose all peers in parallel making explicit messages on which they synchronize.

Let us now give the resulting system for our running example. This system is given in SVL [7] below. SVL is a scripting language that complements the LOTOS encoding, and automates parts of the approach by calling the different CADP tools we reuse. Moreover, these scripts were used to circumvent the state explosion problem (see a discussion on this issue in Section 4). Bcg files (delimited by double quotes and with extension bcg below) are internal state/transition representations computed (by CADP) from the LOTOS peer processes. Message directions "!" and "?" as added in Figure 4 for pretty-printing reasons, have a different semantics in LOTOS, they are used for value passing. Since, we do not need this feature here, we have encoded messages without any direction for the synchronous case as they appear in the synchronization sets (noted between |[..]|) below. If two peers do not have to synchronize, they are composed using the interleaving operator (|||).

```
"distributed_system.bcg" =
   "peer_Customer_lts.bcg"
   |[c_ts_request, ts_c_result, b_c_invoice]|
   (
     "peer_TrainStation_lts.bcg"
     |[ts_a_info, a_ts_infoAvail, a_ts_itinerary, ts_b_book, b_ts_ack]|
     (
       "peer_Availability_lts.bcg" ||| "peer_Booking_lts.bcg"
     )
   )
```

Once this system is generated and reduced, we compare it to the collaboration diagram LTS generated as explained in Section 3.1 using a strong equivalence relation [14]. This check either says that both systems are equivalent and the collaboration diagram is then realizable, or returns *false* which means that the diagram is unrealizable. As far as our running example is concerned, the equivalence test returns *true* for synchronous communication.

Asynchronous communication. This case is slightly more difficult because asynchronous communication is not supported by LOTOS. To simulate how the system evolves with an asynchronous communication model, we generate some LOTOS code to implement bounded FIFO queues. Each peer is associated with a queue (a LOTOS process) from which it can consume messages received beforehand. This also means that a peer which wants to send a message to another one, will actually interact (synchronously) with the other one's queue. A queue process needs a queue datatype (BQueue below) to store received messages.

This datatype is implemented using algebraic specification facilities provided by LOTOS. A queue process can either interact with other peers on messages that can be received by its own peer (t_s_book for the Booking queue below), or synchronizes with its own peer if this one wants to evolve by consuming a message available in its own queue (t_s_book_REC for the Booking peer). Note that a local communication between a peer and its queue is suffixed with "_REC", whereas a communication between a peer (sender) and a queue is not suffixed. The datatype encoding queues defines several operations: bisfull tests if the queue is full, binsert appends a message to the end of the queue, bishead tests if a message appears at the head of the queue, and bremove suppresses the message at the head of the queue.

```
process queue_Booking [ts_b_book, ts_b_book_REC] (q:BQueue) : exit :=
   [not(bisfull(q))] ->
      ts_b_book;
        queue_Booking [ts_b_book, ts_b_book_REC] (binsert(ts_b_book,q))
   []
   [bishead(ts_b_book,q)] ->
      ts_b_book_REC;
        queue_Booking [ts_b_book, ts_b_book_REC] (bremove(q))
   []
   exit
endproc
```

Next, a process for each couple *(peer, queue)* is generated in LOTOS. A peer and a queue interact together on all messages (suffixed with "_REC") that can be received by the peer. From an external point of view, these messages are not of interest, and that is why they are hidden. We show below the LOTOS peer_queue_Customer process body for illustration purposes. Notice that the process queue_Customer below is instantiated with a size set to 1 and no messages in the queue (nil). The queue size is an input parameter of the LOTOS encoding.

```
hide ts_c_result_REC, b_c_invoice_REC in
   (
      peer_Customer [...]
      |[ts_c_result_REC, b_c_invoice_REC]|
      queue_Customer [...] (queue (1, nil))
   )
```

Finally, the distributed system (in SVL below) is obtained by compiling all LOTOS processes encoding couples *(peer, queue)* into bcg files, and making all these couples synchronize correctly on messages exchanged among peers (that is all messages sent from peers to corresponding queues).

```
"distributed_system_async.bcg"=
   "peer_queue_Customer.bcg"
   |[c_ts_request, ts_c_result, b_c_invoice]|
   (
     "peer_queue_TrainStation.bcg"
```

```
  |[ts_a_info, a_ts_infoAvail, a_ts_itinerary, ts_b_book, b_ts_ack]|
  (
    "peer_queue_Availability.bcg" ||| "peer_queue_Booking.bcg"
  )
)
```

Once the distributed system is computed, realizability is checked similarly to the synchronous case, by comparing if the collaboration diagram LTS obtained as presented in Section 3.1 is strongly equivalent to the distributed system.

As far as our running example is concerned, the equivalence test says *false*, and indicates that the trace c_ts_request, ts_a_info, a_ts_infoAvail, ts_b_book appears in both systems, but a_ts_itinerary is then present in the distributed system (it should not be), and not in the collaboration diagram LTS. The problem here is that the train station peer has no way to know whether the availability peer will send or not a_ts_itinerary because the recurrence type is "*" which means zero or several times. So, what happens is that the train station peer sends ts_b_book to the booking peer (assuming the availability peer will never send a_ts_itinerary), and after this emission, the availability peer finally sends a_ts_itinerary, thus the dependency relation A3/B1:book is not respected. We show in the next subsection how such unrealizable collaboration diagrams can be implemented.

3.4 Peer Generation, Extended

To make peers respect interaction constraints of unrealizable collaboration diagrams, we have to insert additional communications among peers. To do so, peers have to (i) respect the application order of messages in each thread, and (ii) respect dependency relations which specify constraints on the firing of a specific message. The first constraint is achieved by adding in the collaboration diagram encoding some explicit messages prefixed with "SEQ_" between each thread message. As regards the second one, we will use the "SYNC_" messages that have been used in the initial encoding to respect message dependency relations.

Let us illustrate with thread A of our running example, where in addition to messages SYNC_1 and SYNC_A3, two new messages SEQ_A1 and SEQ_A2 appear respectively after messages ts_a_info and a_ts_infoAvail. It is not useful to insert such a message after the last message since it is the end of the thread.

```
(             (* -- thread A encoding -- *)
   SYNC_1;
      ts_a_info;
         SEQ_A1;
            a_ts_infoAvail;
               SEQ_A2;
                  loop_process [a_ts_itinerary] >>
                     ( SYNC_A3; exit ||| exit )
) ...
```

From this extended collaboration diagram encoding, peers are generated by keeping visible the messages in which the peer does participate in, and also some of the additional communications introduced above. Additional communications to be kept are figured out following two rules: (i) A peer contains in its behaviour all "SEQ_" messages of a specific thread if the peer participates in at least one interaction of this thread; (ii) a peer contains in its behaviour each "SYNC_" message for which the corresponding message (*e.g.,* for SYNC_1, the message labeled 1 that is c_ts_request) is either one of its own messages, or is used in a dependency relation of the collaboration diagram. For both rules, peers synchronize on all additional communications that they share in their alphabets.

Let us illustrate that showing peer Booking (Fig. 5) generated with this approach. First, since peer Booking only participates in thread B, its behaviour contains messages SEQ_B1 and SEQ_B2 which means that all peers involved in thread B (namely peers Customer, Booking, and TrainStation) will synchronize using these messages so as to respect the execution order of messages in this thread. Second, two messages for dependency relations, SYNC_A3 and SYNC_B2, are used too. SYNC_A3 is necessary because message ts_b_book must be run only after the message identified by A3 (a_ts_itinerary) in the collaboration diagram. Moreover, SYNC_B2 appears in peer Booking because the message identified by B2 in the collaboration diagram (b_ts_ack) is used as dependency of another message (ts_c_result sent by the train station to the customer), thus once b_ts_ack is sent, peer Booking will interact with peers Customer and TrainStation to inform them the result can be emitted.

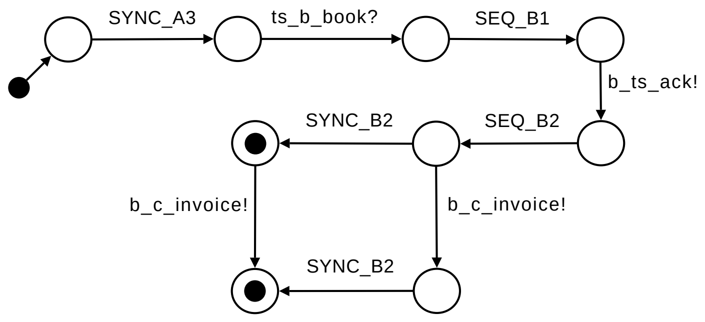

Fig. 5. Peer Booking with additional messages

Once the new peers are generated, the distributed system is built by extending the description given in Section 3.3 with additional communications and also synchronizing peers on them. We recall that all peers do not synchronize on all additional communications but only on those belonging to their alphabet and shared with the other peers. Finally, equivalence between the collaboration diagram LTS and the distributed system in which all additional communications have been hidden, confirms that the extended peers conform to the collaboration diagram.

4 Tool Support and Experiments

The different steps of our approach are completely automated by several tools (Fig. 6). We have implemented a prototype tool named cd2lotos which, given a collaboration diagram, generates the LOTOS code necessary to compute all the results we have presented before in this paper. The cd2lotos prototype also generates some SVL scripts that complement the LOTOS encoding and automate the rest of the process by calling the different CADP tools we reuse. Thus, LTS generation is achieved using Caesar.adt and Caesar LOTOS compilers, as well as reduction techniques available in Reductor. Model-checking can be performed using Evaluator. Note that model-checking is the only step which is not fully automated. Indeed, if a designer wants to go beyond basic checks (such as deadlock-freeness), (s)he has to write some formulas encoding properties to be satisfied by the choreography specification. Last, Bisimulator is used to check that the collaboration diagram LTS is equivalent to the distributed peer implementation.

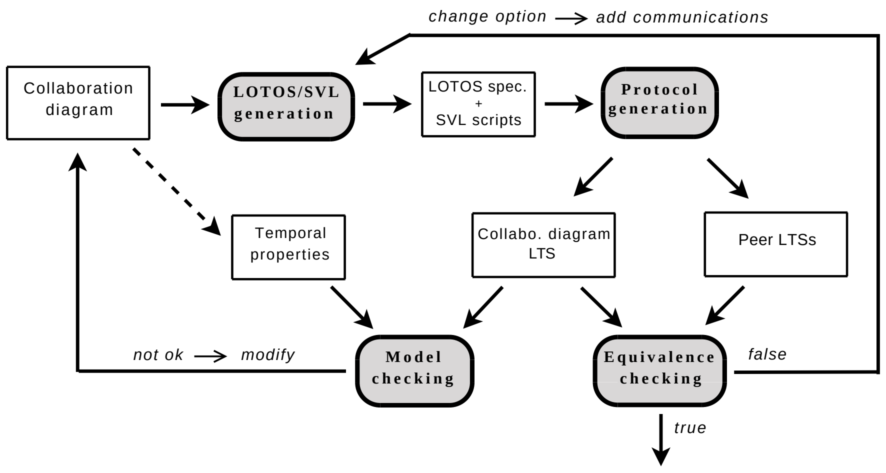

Fig. 6. Tool overview

Our approach, and especially the tool we implemented (cd2lotos), was applied and validated on about 85 collaboration diagrams (which resulted in the generation of about 49,000 lines of LOTOS and 23,000 lines of SVL). We also tested realizability on all these case studies, and all the unrealizable ones were checked equivalent once additional communications were inserted in the peer protocols.

Table 1 shows experimental results[4] on some of the examples belonging to our database. For each experiment, the table gives the size of the diagram in

[4] Experiments have been carried out on a Vaio VGN-FZ11Z (Intel Core 2 Duo Processor T7300 2GHz, 2GB of RAM).

terms of number of peers, messages, and threads. Next, the table contains the number of lines of LOTOS and SVL generated by our prototype as well as the size (number of states and transitions) of the LTS generated from the diagram. Last, we give realizability results for both synchronous and asynchronous communication, and the time needed to compute both realizability checks. Example cd-045 corresponds to the case study presented in this paper.

First of all, it takes 1.9s to our prototype to generate SVL and LOTOS files for all the examples of our database for both communication models (synchronous and asynchronous) and both strategies (with and without additional communications). For medium size examples (cd-008, cd-025, cd-045), the generation of all intermediate LTSs and the realizability checks are quite fast (less than 20 seconds). For bigger diagrams (cd-059, cd-072), the computation time increases up to several minutes. It is interesting to note that examples involving more threads (cd-064) induce time consuming computations since they generate bigger intermediate state spaces due to the higher number of interleavings coming with the number of threads.

Collab.	Size			LOTOS	SVL	CD LTS	Realizability		
diagrams	peers	messages	threads	(lines)	(lines)	(st./tr.)	sync.	async.	time
cd-008	5	9	4	388	148	27/46	$\surd$	$\surd$	19.56s
cd-025	4	6	3	304	130	12/15	$\surd$	$\surd$	16.20s
cd-045	5	8	3	341	130	10/13	$\surd$	$\times$	18.69s
cd-059	10	20	3	666	238	56/175	$\times$	$\times$	1m12.31s
cd-064	7	13	6	495	184	96/396	$\times$	$\times$	1m46.14s
cd-072	16	30	4	959	346	220/748	$\times$	$\times$	6m31.39s

Table 1. Realizability results for some case studies (no additional communications)

Table 2 shows results obtained for the unrealizable examples presented in Table 1 once some additional communications are inserted. Obviously, all these examples are realized by adding these communications. Notice that realizability tests may take less time compared to Table 1 (cd-059, cd-064) because adding communications increases the sequentiality of the system, and therefore reduces communication interleaving.

During the experiments, we had to face the state explosion problem. In a first attempt, we were computing distributed systems in a single step, but, even for simple examples, the state space compilation lasted several minutes. Experiments showed that for collaboration diagrams of medium size (4/5 peers and 10/15 messages), the compilation of couples *(peer, queue)* was returning LTS containing hundreds even thousands of states (*resp.* transitions). Consequently, we decided to build first each couple *(peer, queue)*, minimize them, and compose them to finally obtain the expected system. This technique (known as compositional verification in CADP) allows to generate any step of the (distributed) system computation in less than one second.

Collab.	Size			LOTOS	SVL	CD LTS	Realizability		
diagrams	peers	messages	threads	(lines)	(lines)	(st./tr.)	sync.	async.	time
cd-045	5	8	3	343	134	10/13	√	√	17.09s
cd-059	10	20	3	674	242	56/175	√	√	44.45s
cd-064	7	13	6	501	188	96/396	√	√	1m25.25s
cd-072	16	30	4	974	350	220/748	√	√	6m51.51s

Table 2. Realizability results for some case studies (additional communications)

5 Related Work

Several works aimed at studying and defining the realizability problem for choreography [10, 3, 11, 6, 2]. In [3, 11], the authors define models for choreography and orchestration, and formalize a conformance relation between both models. These models are assumed given as input whereas we focus on the generation of one from the other (generation of peers from a global specification) while ensuring conformance. On a wider scale, all these approaches focus on theoretical aspects and most of them lack of tool support. WSAT [5] is the only tool we know which takes conversation protocols as input, and checks a set of realizability conditions on them. Our proposal is fully automated by tools. Moreover, these works only test realizability, but do not try to modify or extend peers to make them implementable as we do.

Other works [4, 15] propose well-formedness rules to enforce the specification to be realizable. For example, in [4], the authors rely on a π-calculus-like language and session types to formally describe choreographies. Then, they identify three principles for global description under which they define a sound and complete end-point projection, that is the generation of distributed processes from the choreography description. We consider this solution too restricted since it may prevent the designer from specifying what (s)he wants to. In addition, it makes the choreography design more complicated since the designer cannot only focus on composition issues, but has to consider at the same time these well-formedness rules.

Last, to the best of our knowledge, the only work which proposes to add messages in order to implement unrealizable choreographies is [15]. To do so, the authors modify their choreography language to take new constructs (named dominated choice and loop) into account. During the projection of these new operators, some communications are added in order to make peers respect the choreography specification. This solution complicates the design because these new constructs are more restricting than the original ones, and they oblige the designer to explicit extra-constraints in the choreography specification by associating *dominant roles* to certain peers.

With respect to all these works, ours allows to implement any choreography specification (here written with collaboration diagrams) without adding any rule or constraint on the choreography language or specifications written with it.

Furthermore, the LOTOS encoding makes possible the complete automation of realizability test, choreography verification, and peer generation (possibly with additional messages). Last but not least, we consider in this paper both synchronous and asynchronous communication models.

6 Concluding Remarks

In this paper, we have presented an encoding of collaboration diagrams into LOTOS in order to detect realizability issues, and if necessary solve them while generating peers by adding some communications. Our proposal deals with synchronous communication but also bounded asynchronous communication, and allows a completely automated and smooth process thanks to a prototype tool we implemented to generate LOTOS code, and the use of the CADP toolbox to analyze results generated from this code.

We have not discussed implementation issues here because it was out of scope. However, from peers generated using our approach either new services can be implemented, or some wrappers can be generated if an implementation of a service already exists [16]. In the latter case, the wrapper aims at constraining the functionality of the existing service to make it respect the application order of operations as specified in the generated peer behaviour. Implementation of executable services (Java, BPEL) from abstract descriptions can be achieved using Pi4SOA technologies [1], or following guidelines presented in [12].

As regards future works, a first perspective aims at extending our approach by considering as input to our problem a set of collaboration diagrams. Indeed, choice is a missing construct in the collaboration diagram notation, and using a set of diagrams allows to fill this gap. Second, realizability results for asynchronous communication were computed with various queue sizes. During these experiments, we noticed that results for queues of size one can be generalised to any size (*i.e.*, if a collaboration diagram is realizable for peers with queues of size one, it will be realizable too for queues of size k). Intuitively, this is because the equivalence check involves only sent messages, and received messages can be run at any time without any control. However, although this conjecture was experimentally validated, we would like to go forward and formally prove that realizability results for queues of size one hold for queues of size k and unbounded queues. Last, additional communications inserted in peer protocols to make them respect the collaboration diagram choreography can be minimized. In this paper, we systematically added a new message for each sequence of two actions in every thread, as well as for each dependency relation. However, all these messages are not always useful, and removing some of them for certain collaboration diagrams does not invalid their realizability. We would like to propose automatic techniques which figure out the minimal number of necessary additional messages to implement a given collaboration diagram.

Acknowledgements. The authors thank Javier Cámara and José Antonio Martín for fruitful discussions and interesting comments on a former version of this paper. This work has been partially supported by project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science, project P06-TIC2250 funded by the Andalusian local Government, and US National Science Foundation Grants CCF-0614002 and CCF-0716095.

References

1. Pi4SOA Project. `www.pi4soa.org`.
2. T. Bultan and X. Fu. Specification of Realizable Service Conversations using Collaboration Diagrams. *Service Oriented Computing and Applications*, 2(1):27–39, 2008.
3. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and Orchestration Conformance for System Design. In *Proc. of Coordination'06*, volume 4038 of *LNCS*, pages 63–81. Springer, 2006.
4. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
5. X. Fu, T. Bultan, and J. Su. WSAT: A Tool for Formal Analysis of Web Services. In *Proc. of CAV'04*, volume 3114 of *LNCS*, pages 510–514. Springer, 2004.
6. X. Fu, T. Bultan, and J. Su. Synchronizability of Conversations among Web Services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, 2005.
7. H. Garavel and F. Lang. SVL: A Scripting Language for Compositional Verification. In *Proc. of FORTE'01*, pages 377–394. Kluwer, 2001.
8. H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pages 158–163. Springer, 2007.
9. ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, International Standards Organisation, 1989.
10. R. Kazhamiakin and M. Pistore. Analysis of Realizability Conditions for Web Service Choreographies. In *Proc. of FORTE'06*, volume 4229 of *LNCS*, pages 61–76. Springer, 2006.
11. J. Li, H. Zhu, and G. Pu. Conformance Validation between Choreography and Orchestration. In *Proc. of TASE'07*, pages 473–482. IEEE Computer Society, 2007.
12. R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. In *Proc. of ICSOC'08.* Springer, 2008.
13. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, 2003.
14. R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.
15. Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the Theoretical Foundation of Choreography. In *Proc. of WWW'07*, pages 973–982. ACM Press, 2007.
16. G. Salaün. Generation of Service Wrapper Protocols from Choreography Specifications. In *Proc. of SEFM'08*, pages 313–322. IEEE Computer Society, 2008.