Realizability of Choreographies using Process Algebra Encodings

Gwen Salaün, Tevfik Bultan, and Nima Roohi

***Abstract*— Service-oriented computing has emerged as a new software development paradigm that enables implementation of Web accessible software systems that are composed of distributed services which interact with each other via exchanging messages. Modeling and analysis of interactions among services is a crucial problem in this domain. Interactions among a set of services that participate in a service composition can be described from a global point of view as a *choreography*. Choreographies can be specified using specification languages such as Web Services Choreography Description Language (WS-CDL) and visualized using graphical formalisms such as collaboration diagrams. In this article, we present an encoding of collaboration diagrams into the LOTOS process algebra for choreography analysis. This encoding allows us to (i) check the temporal properties of choreographies using a LOTOS verification tool set called the Construction and Analysis of Distributed Processes (CADP) toolbox, (ii) check the realizability of choreographies for both synchronous communication and bounded asynchronous communication, and (iii) automate the peer generation process. *Realizability* indicates whether peers can be generated from a given choreography specification in such a way that the interactions of the generated peers exactly match the choreography specification. If a collaboration diagram is unrealizable, our approach extends the peer generation process by adding extra communication that guarantees that the peers behave according to the choreography specification.**

***Index Terms*— Service protocols, choreography, realizability, process algebra, asynchronous communication, verification, tools.**

I. Introduction

Specification and analysis of interactions among distributed components play an important role in service oriented computing. In order to facilitate integration of independently developed components (*i.e.*, peers) that may reside in different organizations, it is necessary to provide a global contract that the peers participating in a composite service should adhere to. Such a contract is called a *choreography*, and specifies interactions among a set of services from a global point of view. In addition to development of choreography specification languages such as the Web Services Choreography Description Language (WS-CDL), there have been efforts in formalizing semantics of choreography specifications based on various formalisms such as conversation protocols [1], collaboration diagrams [2], process algebras [3], and Petri Nets [4]. These formal models enable the use of formal verification and analysis techniques to address problems that arise in choreography analysis. One important problem in choreography analysis is figuring out if a choreography specification can be implemented by a set of peers that communicate via message passing. Given a choreography specification, it would be desirable if the local implementations, namely *peers*, could be automatically generated via projection, *i.e.*, by projecting the global choreography specification to each peer by ignoring the messages that are not sent or received by that peer. However, generation of peers that precisely implement a choreography specification is not always possible, *i.e.*, there are choreographies that are not implementable by a set of distributed peers (if no additional messages are allowed). This problem is known as *realizability*.

Recent results on choreography realizability problem [2], [5]–[8] advocate techniques to check the realizability of a choreography after the choreography specification has been written, 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 the specifications written with it. In this article we focus on analyzing choreography specifications expressed as collaboration diagrams. For this class of choreography specifications, our new contributions with respect to earlier results are the following:

- our solution generates peers for any choreography specification by extending them with additional messages if the choreography is unrealizable;
- our approach is supported by tools for (i) verification of choreographies, (ii) realizability analysis, and (iii) peer generation in a completely automated way;
- we consider both synchronous and asynchronous communication models, and present results on the effect of the queue size on realizability.

As mentioned above, in this article, we use collaboration diagrams as the choreography specification language. We propose an encoding of collaboration diagrams into the LOTOS process algebra (see Figure 1[1] for an overview of our approach). We chose LOTOS because it provides the necessary level of expressiveness to describe all the collaboration diagram interaction constraints, and is equipped with a rich tool set called the Construction and Analysis of Distributed Processes (CADP) toolbox [9] which offers state-of-the-art tools for state space exploration and verification. The LOTOS encoding allows us to generate the Labelled Transition System (LTS) corresponding to the choreography specification as well as an LTS for each service peer. We can take advantage of this encoding to verify choreography specifications using

G. Salaün is with Grenoble INP, INRIA, France.
T. Bultan is with University of California, Santa Barbara, USA.
N. Roohi is with Sharif University of Technology, Iran.

[1]Numbers on this figure will be used later on in this article.

CADP. As far as realizability is concerned, we use equivalence checking techniques to check realizability of collaboration diagrams for both synchronous communication and bounded asynchronous communication. If the collaboration diagram is not realizable, we generate peers using an alternative technique which adds new messages in order to make them respect the initial choreography. The steps of our approach are completely automated by several tools. For some steps of our approach we could have used Promela, the input language of the SPIN model checker [10], as an alternative to LOTOS, since Promela supports both synchronous and asynchronous communication whereas asynchronous communication is expressed in LOTOS by explicitly encoding queues. However, SPIN does not provide the behavioral equivalence checking functionality that we use in our approach.

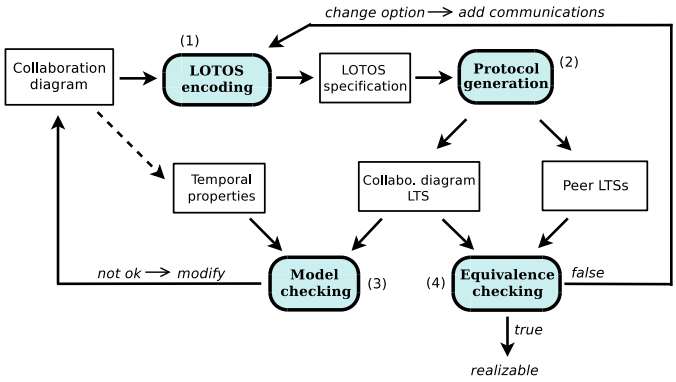


Fig. 1. Overview of our approach

A preliminary version of this work has been published in [11], and is extended here in several aspects: In this article (i) we give a formalization of the LOTOS encoding, (ii) we present a different encoding of the dependency relation among the message send events than the one used in [11] which unnecessarily restricted the possible behaviors, (iii) we discuss how our new LOTOS encoding preserves the original semantics of the choreography specifications expressed as collaboration diagrams, (iv) we show that realizability results for queue size one can be generalized to unbounded queues, (v) we present a minimization technique for the number of additional messages generated for making peers compliant to a given choreography specification, (vi) we apply our verification and analysis techniques to a larger set of examples, and (vii) we present an extended discussion comparing our approach with the related work.

The rest of this article is organized as follows. Section II introduces collaboration diagrams and the problem of their realizability. Section III presents our encoding of collaboration diagrams into LOTOS. Section IV shows how this encoding is extended to generate corresponding peers. Section V presents our realizability test for both communication models. Section VI proposes a solution to enforce the realizability of peers. Section VII sketches the tools that support our approach, and discusses some experimental results. Section VIII compares our proposal to related work, and Section IX ends the article with some concluding remarks.

II. Collaboration Diagrams

A. Syntax and Semantics

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. A message send event (which we will simply call an event) is a tuple that consists of a set of predecessor events, a (unique) label, a message, and a recurrence type. Labels (*e.g.*, `1`, `2`, `3`, ..., `A1`, `A2`, `A3`, ..., `B1`, `B2`, `B3`, ...) consist of a prefix (*e.g.*, ϵ, `A`, `B`) that organizes events into different threads and a sequence number (*e.g.*, 1, 2, 3, ...) that gives the ordering of the events in each thread. All messages in one thread share the same prefix and execute based on the numerical order defined by their sequence number. Events from different threads execute concurrently, and can be interleaved in any order that respects the dependency relation that is defined by the sets of predecessor events and sequence numbers. An event can only execute after 1) all the events in its predecessor set have been executed, and 2) the events that are in the same thread and that have smaller sequence numbers have also been executed. We assume that the event ordering relation defined by the sequence numbers and the predecessor sets do not cause any cyclic dependencies. A recurrence type is either "1" (default type) meaning that the associated event happens exactly once, "?" for a conditional event meaning that the event may occur once or it may not occur at all, or "*" for an iterative event meaning that the event may not occur at all or it may occur one or more times.

Events are written using the following syntax: The list of predecessor event labels followed by "/", followed by the event label, followed by ":", followed by the message, followed by the recurrence type. For example, `1/A1:info` is an event with the predecessor set $\{$`1`$\}$, event label `A1`, message `info` and the default recurrence type (*i.e.*, recurrence type is 1).

The semantics of collaboration diagrams is formally defined in [2] based on the above rules. We summarize the formal model below.

Definition 1 (Collaboration Diagram): Formally, we define a collaboration diagram as a tuple (P, E, M) where: P is a set of peers, E is a set of events and M is a set of messages. For each message $m \in M$, $send(m) \in P$ denotes the sending peer and $recv(m) \in P$ denotes the receiving peer. Each event $e \in E$ is a tuple $e = (B, l, m, r)$ where $B \subseteq E$ is the set of predecessor events that should execute before e, l is an event label, m is a message, and r is a recurrence type (*i.e.*, one of 1, ?, or *). Given an event e, we use $e.B$, $e.l$, $e.m$ and $e.r$ to refer to different components of the event tuple.

We do not represent links in our formal model explicitly since they can be inferred from the set of events and messages, *i.e.*, if there exists an event that sends a message m from peer $send(m)$ to $recv(m)$, then there is a link between $send(m)$ and $recv(m)$.

Given a collaboration diagram, an event sequence of that collaboration diagram is a sequence of events that respects the predecessor set and sequence number ordering and recurrence type of each event. Given an event sequence, the sequence of messages generated by that event sequence is called a

conversation. The conversation set for a collaboration diagram is the set of conversations generated by all the event sequences of that collaboration diagram. We demonstrate these concepts on a running example below.

B. Running Example: Train Station Service

Figure 2 presents a collaboration diagram for a train station service that we will use as a running example throughout this article. This diagram contains four peers: `Customer`, `TrainStation`, `Availability`, and `Booking`. It involves 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 sending of a request message from the peer `Customer` to the peer `TrainStation` (event `1`). Next, the `TrainStation` checks ticket availability by exchanging messages with the peer `Availability` which is a component that is responsible for keeping track of the ticket availability (events `A1`, `A2`, and `A3`). After the availability check, the `TrainStation` exchanges messages with the peer `Booking` to reserve tickets (events `B1` and `B2`). If the booking is successful the `Booking` sends an invoice to the `Customer` (event `B3`). The `TrainStation` sends the final result to the `Customer` (event `2`).

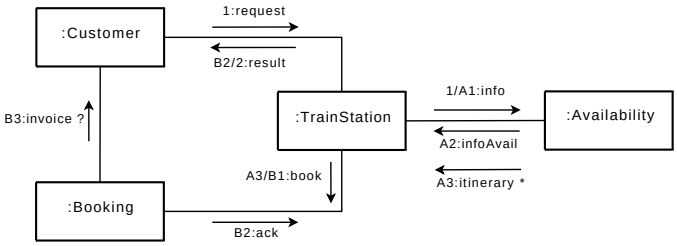


Fig. 2. Train station service collaboration diagram

Let us focus on the thread `A`. It contains three events. The first one, `1/A1:info`, indicates that the message `info` should be sent by the peer `TrainStation` to the peer `Availability` only after the execution of the event `1`. *I.e.*, `1` is in the predecessor set of the event `A1`, meaning that the event `A1` is executable only after the event with label `1` (namely `1:request`) has been executed. Formally, the event tuple for the event `1/A1:info` is $e = (\{\texttt{1}\}, \texttt{A1}, \texttt{info}, 1)$, where $e.B = \{\texttt{1}\}$, $e.l = \texttt{A1}$, $e.m = \texttt{info}$, and $e.r = 1$. The third event of thread `A` is `A3:itinerary*`. This event must be executed after the event `A2` (due to the sequential ordering of the events within a thread), and can be executed multiple times (due to the recurrence type *). The event tuple for the event `A3:itinerary*` is $(\emptyset, \texttt{A3}, \texttt{itinerary}, *)$. A possible event sequence for the diagram shown in Figure 2 is: `1`, `A1`, `A2`, `B1`, `B2`, `2`, `B3`. The conversation corresponding to this event sequence is: `request`, `info`, `infoAvail`, `book`, `ack`, `result`, `invoice`.

C. Peer Model

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 (M, S, I, F, T) where: M is the set of messages, S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ are final states, and $T \subseteq S \times M \times S$ is the transition relation. In the peer transition systems, we annotate the messages with the direction information, *i.e.*, the message m is written as $m!$ in the transition system of the peer p if $p = send(m)$ (send transition), and it is written as $m?$ if $p = recv(m)$ (receive transition). Peers interact using binary communication on same message names with opposite directions. In this article, we will consider both synchronous and asynchronous communication models. In the later case, each peer is equipped with a FIFO queue which stores the input messages received from the other peers, and from which the current peer can consume messages.

It is worth noting that taking interaction protocols (messages and their application order) into account in the peer model, and therefore in choreography specification languages, is essential. This allows one to avoid erroneous behaviours such as unexpected results or deadlock when executing a set of services together. Imagine for instance a trip planner system which is supposed to organize a trip for a client (booking flight tickets and hotel) given a set of constraints provided by the client (dates, city, price limit, etc). One ordering requirement for such a system could be that the the system must start by first interacting with the flight service, otherwise a hotel might be booked even if no flight tickets could be found for the provided dates. The role of the choreography specification and modeling is to make such ordering requirements explicit. However, as we describe below, specification of such global interaction requirements can be difficult and error prone.

D. Realizability of Collaboration Diagrams

One of the main problems in choreography specification is realizability. Two unrealizable collaboration diagrams are presented in Figure 3. The first one (left-hand side) is unrealizable because it is impossible for the peer `C` to know when the peer `A` sends its `request` message since there is no interaction between `A` and `C`. Hence, the peers cannot respect the execution order of messages as specified in the collaboration diagram. The second diagram is realizable for synchronous communication, and unrealizable for asynchronous communication. Indeed, in case of synchronous communication, the peer `C` can synchronize (rendez-vous) with the peer `A` only after the `request` message is sent, so the message order is respected. This is not the case for asynchronous communication since `A` cannot block `C` from sending the `update` message if asynchronous communication is used. Hence, `C` has to send the `update` message to `A` without knowing if `A` has sent the `request` message or not. Therefore, the correct order between the two messages cannot be satisfied. We also show in Figure 3 (right-hand side) the LTS generated for peer `A` by projection.

Although realizability can be easily determined for these simple examples, it is more difficult to determine if the collaboration diagram presented in Figure 2 is realizable or not. We present in the rest of this article an approach to automate the realizability check, and show that the train service collaboration diagram is realizable for synchronous communication

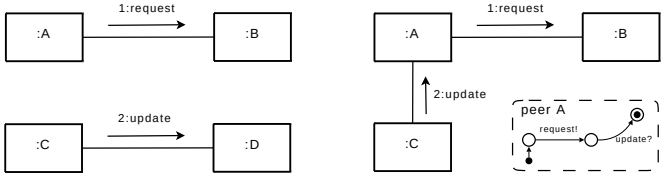


Fig. 3. Examples of unrealizable collaboration diagrams

and it is unrealizable for asynchronous communication. We also show that this collaboration diagram can be converted to a realizable choreography specification for asynchronous communication if extra messages are allowed and we show how to generate such extra messages.

III. Encoding Collaboration Diagrams in LOTOS

The backbone of our approach is an encoding of collaboration diagrams into the LOTOS process algebra [12]. We chose LOTOS because it provides a rich notation that allows specification of complex concurrent systems. Furthermore, the LOTOS encoding allows (i) choreography verification by using model checking tools available in the CADP toolbox [9], (ii) realizability analysis and (iii) generation of service peer implementations. The SVL scripting language [13] is also used to automate parts of the approach by calling the different CADP tools we use. The steps of our approach are completely automated using several tools we present in Section VII.

A. LOTOS and SVL in a Nutshell

Here we present a simplified grammar for the LOTOS notation (see [12] for a detailed presentation of the LOTOS notation). The behavior of a process B in LOTOS is specified using termination, communication, sequence (action prefix or sequential composition), choice, parallel composition, interleaving, hide, and process call (to express a looping behavior):

B	::=	`exit`	*correct termination*
	|	A`;`B	*action prefix*
	|	B_1`>>`B_2	*sequential composition*
	|	B_1`[]`B_2	*choice*
	|	B_1`|[`$a_1, \ldots, a_n$`]|`B_2	*parallel composition*
	|	B_1`|||`B_2	*interleaving*
	|	`hide` $a_1, \ldots, a_n$ `in` B	*hide*
	|	P`[`$a_1, \ldots, a_n$`]`	*process call*
A	::=	i	*internal action*
	|	a	*communication on* a

The parallel composition operator denotes that processes B_1 and B_2 evolve in parallel and synchronize on the actions $a_1, \ldots, a_n$. The interleaving operator corresponds to a concurrent evolution of B_1 and B_2 without synchronization between these two processes.

An SVL script is a sequence of statements, which describe verification operations (such as comparison modulo various equivalence relations, deadlock and livelock detection, verification of temporal logic formulas, etc.) performed on behaviors. Basic behaviors are LTSs obtained here from LOTOS descriptions. Behaviors can be combined and handled using operations such as parallel composition, label hiding, label renaming, minimization, etc. SVL has also meta-operations implementing higher-order strategies for compositional verification. In this work we only need basic operators for realizability verification purposes, namely a parallel composition and an interleaving operator:

B	::=	`"`ID`.bcg"`	*LTS*
	|	B_1`|[`$a_1, \ldots, a_n$`]|`B_2	*parallel composition*
	|	B_1`|||`B_2	*interleaving*

B. LOTOS Encoding for Collaboration Diagrams

In this section, we discuss how to encode a collaboration diagram as a LOTOS process. The LOTOS process encoding a collaboration diagram is split up in as many parts (referred as thread behavior below) as there are threads in the collaboration diagram. Each thread behavior encodes all the events in the corresponding thread in the order in which they must be executed (this ordering is achieved using the LOTOS action prefix operator). Each message is encoded using sender and receiver peer names as prefixes. The conditional recurrence type "?" is encoded as a choice between the actual execution of the send event meaning that the condition is true, and a termination (`exit`) meaning that the condition is false and the event is not executed (*i.e.*, the message is not sent). The iterative recurrence type "*" is translated into LOTOS using an intermediate looping process whose behavior is specified as: `message;loop_process[message] [] exit`.

Each thread behavior evolves independently, and they synchronize together to respect dependency constraints that are explicitly specified in the predecessor sets 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 executing an event, if that event has a predecessor set and, hence, depends on event executions in other threads, (ii) after executing an event, if that event appears in the predecessor set of another event 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 behavior. In both cases, if an event execution influences the execution of several other events (*i.e.*, the event is in the predecessor set of several other events), or if an event should be executed after several other events (*i.e.*, the event has more than one event in its predecessor set), we generate as many (interleaved) binary synchronizations as needed. In [11], we encoded such dependencies using a single n-ary synchronization and restricted the behavior of the LOTOS model more than necessary according to the collaboration diagram semantics [2]. The n-ary synchronization encoding enforces n threads to synchronize at the synchronization point which is not necessary according to the collaboration diagram semantics. Hence, some possible behaviors of the collaboration diagram specification were being ignored in the LOTOS encoding given in [11]. The encoding given in this article resolves this problem by using multiple binary synchronizations instead of a single n-ary synchronization.

Given a collaboration diagram $CD = (P, E, M)$, we generate the LOTOS process encoding the collaboration diagram CD, by calling $cd2l(CD)$, as follows:

```
process cd [alpha(M), SYNC] : exit :=
  cd2l_t(Th_1, SYNC)
  |[SYNC_X_1, ..., SYNC_X_m]|
  (
    cd2l_t(Th_2, SYNC)
    |[SYNC_Y_1, ..., SYNC_Y_q]|
    ...
  )
endproc
```

where $\{Th_1, \ldots, Th_n\} = sort\_by\_thread(E)$, $SYNC = compute\_sync(E)$, $\{X_1, \ldots, X_m\} = gen\_sync(Th_1, \{Th_2, \ldots, Th_n\}, SYNC)$, and $\{Y_1, \ldots, Y_q\} = gen\_sync(Th_2, \{Th_3, \ldots, Th_n\}, SYNC)$. We will explain these functions below.

Function $alpha$ transforms the message names by prefixing each message M with sender and receiver peers, respectively:

$$alpha(M) = \{send(m)\_recv(m)\_m \mid m \in M\}$$

Function $sort\_by\_thread$ traverses the set of events E of the collaboration diagram, and builds a set of event lists where each list keeps the events for only one thread. In these lists, the events are ordered with respect to their sequence numbers:

$$sort\_by\_thread(E) = \{sort\_tuples(Th_X) \mid \forall (B, l, m, r) \in E : X = pre(l) \Leftrightarrow (B, l, m, r) \in Th_X\}$$

where the function pre returns the prefix of an event label, identifying the thread for that event (ϵ, A, B, etc.). The event list Th_X contains the events that are part of the thread X. The function $sort\_tuples(Th_X)$ sorts the events in the event list Th_X by their sequence numbers.

Function $compute\_sync$ accepts as input a set of events E, and extracts "SYNC_" messages from the predecessor sets of all events:

$$compute\_sync(E) = \{l'\_l \mid (B, l, m, r) \in E \wedge l' \in B\}$$

Function $extract\_sync$ computes the "SYNC_" messages for a given thread and function gen_sync returns synchronizations between a thread and other threads by computing the intersection of "SYNC_" messages used in its behavior and in the other thread behaviors:

$$extract\_sync(Th_X, SYNC) = \{l'\_l \mid (B, l, m, r) \in Th_X \wedge l'\_l \in SYNC\} \cup \{l\_l' \mid (B, l, m, r) \in Th_X \wedge l\_l' \in SYNC\}$$

$$gen\_sync(Th_1, \{Th_2, \ldots, Th_n\}, SYNC) = extract\_sync(Th_1, SYNC) \cap (extract\_sync(Th_2, SYNC) \cup \ldots \cup extract\_sync(Th_n, SYNC))$$

Function $cd2l_t$ translates a thread into LOTOS. It takes the ordered list of thread events for a thread X as its first input: $Th_X = [(B_1, l_1, m_1, r_1), \ldots, (B_n, l_n, m_n, r_n)]$, and recursively translates one event after the other in the order they appear in the input list, that is in the order in which they must be executed (the function stops when the list is empty):

$$cd2l_t([(B_1, l_1, m_1, r_1), \ldots, (B_n, l_n, m_n, r_n)], SYNC) = add\_pre\_sync(B_1, l_1) \texttt{ >> } cd2l_m(send(m_1)\_recv(m_1)\_m_1, r_1) \texttt{ (} add\_post\_sync(l_1, SYNC) \texttt{ >> } add\_exit(n) \texttt{)} cd2l_t([(B_2, l_2, m_2, r_2), \ldots, (B_n, l_n, m_n, r_n)], SYNC)$$

Function $cd2l_m$ translates a message send event into LOTOS taking into account the recurrence type:

$$cd2l_m(m, r) = \begin{cases} m\texttt{;} & \text{if } r = 1 \\ \texttt{(}m\texttt{;exit [] exit) >>} & \text{if } r = ? \\ \texttt{loop\_process[}m\texttt{] >>} & \text{if } r = * \end{cases}$$

Function add_pre_sync and add_post_sync respectively generate additional messages used to synchronize thread behaviors in order to make them respect the dependency relation defined by the predecessor sets specified in the collaboration diagram:

$add\_pre\_sync(\{e_1, \ldots, e_n\}, l) =$
`( SYNC_`Y_1`_`l`;exit ||| ... ||| SYNC_`Y_n`_`l`;exit )`
where $Y_i = e_i.l$ and

$add\_post\_sync(l, SYNC) =$
`( SYNC_`X_1`;exit ||| ... ||| SYNC_`X_k`;exit )`
where $\{X_1, \ldots, X_k\} = \{l\_l' \mid l\_l' \in SYNC\}$.

Last, function add_exit adds a final `exit` to terminate a thread behavior:

$$add\_exit(n) = \begin{cases} \texttt{exit} & \text{if } n = 1 \\ \epsilon & \text{otherwise} \end{cases}$$

Our encoding preserves the collaboration diagram semantics formalized in [2]. We claim that, given a collaboration diagram CD, CD and its corresponding LOTOS encoding $cd2l(CD)$ are trace equivalent, *i.e.*, $[\![CD]\!]_t = [\![cd2l(CD)]\!]_t$ where $[\![CD]\!]_t$ is the set of conversations generated by the collaboration diagram CD. Recall that a conversation is the sequence of messages that are sent (recorded in the order they are sent) during an execution of the collaboration diagram that respects the ordering of events defined by the predecessor sets and the sequence numbers. The conversation set for the LOTOS encoding $[\![cd2l(CD)]\!]_t$ is defined by collecting the conversations generated by all possible executions of the LOTOS encoding. Each execution of $cd2l(CD)$ generates a conversation which is defined by recording only the message send transitions (in the order they are executed) that correspond to messages in the message set M of the collaboration diagram $CD = (P, E, M)$. *I.e.*, all transitions except message send transitions are ignored and all the send transitions corresponding to "SYNC_" messages are also ignored. For these remaining transitions, in order to make messages in both models match, we need to remove the peers names appearing as prefixes in the messages generated by our translation.

Below, we argue that based on the above definitions, the conversation sets of CD and $cd2l(CD)$ are the same, *i.e.*, $[\![CD]\!]_t = [\![cd2l(CD)]\!]_t$, and we assume that messages are not prefixed with peers names. We show this equivalence in three parts, by discussing the encoding of a thread, a message send event, and the dependency relation between events.

1) Given a set of threads $T_1, \ldots, T_n$ in a collaboration diagram CD, each thread executes its events sequentially according to the sequence numbers of the events, and events of different threads can be interleaved arbitrarily assuming that there are no dependencies among the events of different threads (*i.e.*, if all predecessor sets are empty). In this basic case, $cd2l(CD)$ generates a set of concurrent processes without any interactions where each thread of the collaboration diagram corresponds to one concurrent process $cd2l(CD) = T_1 \texttt{|||} \ldots \texttt{|||} T_n$ such that $[\![CD]\!]_t = [\![R_\tau(lts(T_1 \texttt{|||} \ldots$

$|||T_n))]\!]_t$ where R_τ corresponds to τ reductions (see Section III-D for more explanations about τ transitions and their suppression) and lts is the function compiling the LOTOS into LTS[2].

2) Messages send events are encoded differently depending on their recurrence type:

- "m 1" is encoded simply as "m"
- "m ?" produces the set of traces $\{m, \varepsilon\}$. This is encoded in LOTOS as `m;exit [] exit` and the corresponding LTS consists of three transitions $\{(s_0, m, s_1), (s_1, \surd, s_2), (s_0, \surd, s_2)\}$ where s_0 is the initial state, s_2 is the final state, and both transitions labeled with $\surd$ indicate proper termination[3]. Consequently, corresponding traces are the same as in the original collaboration diagram, namely $\{m, \varepsilon\}$.
- "m $*$" produces the set of traces $\{\varepsilon, m, mm, mmm, \ldots\}$. This is encoded in LOTOS using a looping process whose behavior is `m;loop_process[m] [] exit` and the corresponding LTS consists of two transitions $\{(s_0, m, s_0), (s_0, \surd, s_1)\}$ where s_0 is the initial state, and s_1 is the final state. Traces are therefore the same: $\{\varepsilon, m, mm, mmm, \ldots\}$.

3) Threads evolve concurrently and respect explicit ordering of messages specified in the predecessor sets. To obtain the same traces as in the input collaboration diagram we need to preserve the same dependencies between messages in the LOTOS specification. For each event "$l_1, \ldots, l_n / l : m$" in the collaboration diagram, in the LOTOS specification generated, the send event l_i with message m_i is followed by a message SYNC_l_i_l, and the send event l for message m appears only after these SYNC_l_i_l messages following the pattern:

$\ldots m_1$; SYNC_l_1_l; $\ldots$ ||| $\ldots$ ||| $\ldots m_n$; SYNC_l_n_l; $\ldots$
|[SYNC_l_1_l, $\ldots$, SYNC_l_n_l, $\ldots$]|
$\ldots$(SYNC_l_1_l;exit||| $\ldots$ |||SYNC_l_n_l;exit) >> m; $\ldots$

The corresponding LTS consists of first an interleaving of message send transitions m_i and SYNC_l_i_l (for each given i, m_i is executed before SYNC_l_i_l). When all these transitions have been executed, and in particular all the SYNC_l_i_l transitions that act as pre-condition to the execution of the event l have been executed, then the send transition for m can be executed by the LTS. Thus, the dependencies defined in the predecessor set of l ($\{l_1, \ldots, l_n\}$) are preserved in the generated LOTOS specification.

C. Running Example: The Train Station in LOTOS

Let us give the body of the LOTOS process generated by the function $cd2l$ for our running example:

```
(     (* -- thread A encoding -- *)
  SYNC_1_A1; ts_a_info; a_ts_infoAvail;
    loop_process [a_ts_itinerary] >>
      SYNC_A3_B1; exit
)
|[SYNC_1_A1, SYNC_A3_B1]|
(     (* -- thread B encoding -- *)
  (
    SYNC_A3_B1; ts_b_book; b_ts_ack;
      (
         SYNC_B2_2; exit
         |||
         ( b_c_invoice; exit [] exit ) >>
             exit
      )
  )
  |[SYNC_B2_2]|
  (   (* -- main thread's encoding -- *)
    c_ts_request;
      (
         SYNC_1_A1; exit
         |||
         SYNC_B2_2; ts_c_result; exit
      )
  )
)
```

We can distinguish the three threads, respectively for events starting by A, B, and numbers (the main thread). Thread A for instance contains three events (for messages `info`, `infoAvail`, and `itinerary`) which are encoded sequentially based on their sequence numbers and the messages are prefixed with peers participating in these interactions (only peer initials are shown). The last event (for message `a_ts_itinerary`) has 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`).

With regards to the synchronization between thread behaviors, we can see for instance that thread A synchronizes with the two other threads using messages SYNC_1_A1 and SYNC_A3_B1. The message SYNC_1_A1 is used to synchronize the thread A and the main thread in order to make sure that the event labeled A1 with the message `ts_a_info` is executed after the execution of the event labeled 1 with the message `c_ts_request`. Execution of SYNC_1_A1 acts as a pre-condition to the execution of `ts_a_info` guaranteeing the correct ordering of the events. In thread B, the event B1 can only occur after the event A3, therefore the execution of the event A3 with the message `a_ts_itinerary` is followed by a message SYNC_A3_B1 in order to enable the execution of the event B1 with the message `ts_b_book` after the execution of A3. Note that the synchronization message SYNC_A3_B1 should not block the thread execution. Accordingly it is interleaved with the rest of the thread behavior (`exit` in this case, since it is at the end of the thread behavior).

D. Compilation into LTS and Verification

After generating the LOTOS encoding of a collaboration diagram using the function $cd2l$, we can generate the corresponding LTS using the state space generation tools in the CADP toolbox, and verify temporal logic properties of the input choreography specification using the Evaluator model-checker [14]. For instance, for our running example, we checked the liveness property stating that each

[2]This can be computed by using Caesar.adt and Caesar compilers belonging to the CADP toolbox.

[3]LTSs generated from LOTOS in CADP do not have final states, therefore such $\surd$ transitions are used to distinguish proper termination from deadlocks.

`c_ts_request` is eventually answered (`ts_c_result`):

```
[true*."CUSTOMER_TRAINSTATION_REQUEST"]
  <true*."TRAINSTATION_CUSTOMER_RESULT"> true
```

We show in Figure 4 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 using reduction techniques available in the CADP toolbox. In this article, these minimizations (determinization, removal of τ transitions, and suppression of similar paths) are achieved using weak trace, safety and strong reductions. The τ transitions stand for internal actions. These transitions are generated while compiling the LOTOS code. For example, the LOTOS sequential composition operator "`>>`" inserts such a τ transition in the corresponding state space. As a consequence, they are completely removed during LTS generation and do not appear at the collaboration diagram (and peer) level.

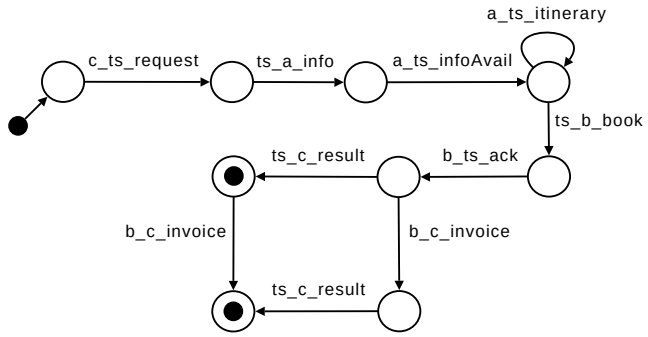


Fig. 4. Train station service: collaboration diagram LTS

IV. 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 whose body is an instance of the collaboration diagram process, and hiding in this process all the messages that the peer does not send or receive, as well as messages prefixed by "`SYNC_`" which were used only to preserve message dependencies in the encoding.

```
process cd_peer_p [alpha_peer(p,M), SYNC] : exit :=
    cd_peer_p_aux[alpha_peer(p,M), SYNC]
where
    process cd_peer_p_aux [alpha_peer'(p,M), SYNC]
        hide gen_hide(p,M), SYNC in
            cd[alpha(M), SYNC]
    endproc
endproc
```

Function gen_hide generates a subset of the collaboration diagram alphabet consisting of messages where peer p is not involved:

$gen\_hide(p, M) = \{send(m)\_recv(m)\_m \mid m \in M \wedge send(m) \neq p \wedge recv(m) \neq p\}$

Function $alpha\_peer'$ generates a subset of the collaboration diagram alphabet consisting of messages where peer p is involved:

$alpha\_peer'(p, M) = \{send(m)\_recv(m)\_m \mid m \in M \wedge (send(m) = p \vee recv(m) = p)\}$

Function $alpha\_peer$ generates the same subset of messages returned by $alpha\_peer'$, but also adds a suffix for the message direction, namely "`_SEN`" if the peer is the sender of the message and "`_REC`" if the peer is the receiver of the message. These suffixes are necessary since transitions require a direction at the peer level indicating either a receive or a send action.

$alpha\_peer(p, M) = \{send(m)\_recv(m)\_m\_\texttt{SEN} \mid m \in M \wedge send(m) = p\} \cup \{send(m)\_recv(m)\_m\_\texttt{REC} \mid m \in M \wedge recv(m) = p\}$

Once all the peer LOTOS processes are generated, corresponding LTSs are obtained automatically using CADP state space generation tools. Figure 5 gives a graphical view of peers generated for our running example from their LOTOS descriptions. For instance, peer `Booking` (Fig. 5, (b)) starts by receiving a booking 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.*, the `request` message is represented as `c_ts_request!` in the customer peer LTS and as `c_ts_request?` in the train station peer LTS.

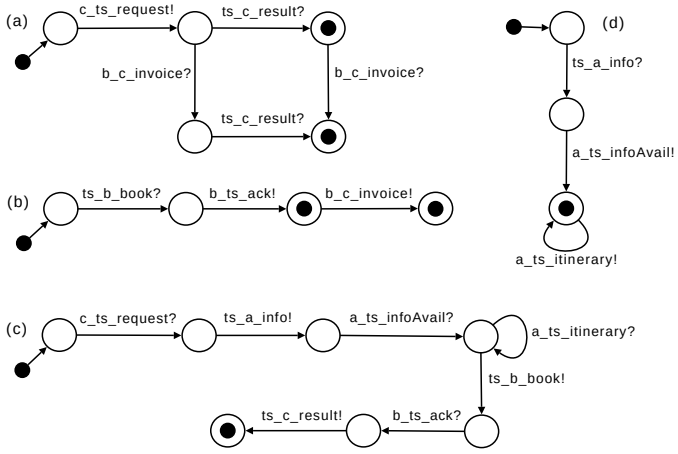


Fig. 5. Peers generated from the collaboration diagram: (a) customer, (b) booking, (c) train station, (d) availability

Once peers are generated, it is difficult to say if their execution respect the interaction constraints specified in the collaboration diagram (order of messages within a thread defined by the sequence numbers, and the inter-thread message dependencies defined by the predecessor sets). In the next subsection, we propose automated techniques for answering this question and checking realizability.

V. Realizability

In this article, we consider a *projection* realizability because this is one of the most widely used realizability definitions, see for instance [6], [7], [15]. Intuitively, a choreography

is realizable if the set of interactions specified in the collaboration diagram and those executed by the interacting peers (obtained by projection from the collaboration diagram) are the same. This realizability definition does not constrain the internal actions of peers but preserves the ordering of their interactions. Weaker realizability notions have also been investigated [6]. Another alternative would have been to say that a choreography is realizable if there exist peers which realize it and these peers do not have to be projections, see for instance [2].

We propose to compute realizability by comparing the collaboration diagram LTS with the system composed of interacting peers using behavioral equivalences and more precisely using strong equivalence or bisimulation [16]. 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.

Therefore, computing realizability is achieved in three steps: (i) generation of the collaboration diagram LTS, (ii) generation of the system composed of interacting peers, and (iii) equivalence checking between LTSs resulting from step (i) and (ii). In the following, we consider both synchronous and bounded asynchronous communication models.

Definition 2 (Projection realizability): A collaboration diagram CD with n peers and queue length $q \in \mathcal{N}^+$ is realizable iff $CDLTS$ is strongly bisimilar to the peer composition $W = (PLTS_1 || \ldots || PLTS_n)$, where $CDLTS$ is the collaboration diagram LTS obtained from CD as presented in Section III, and $PLTS_i$ are the peer LTSs obtained from CD as formalised in Section IV. In the rest of this article, this test is noted $Realizable(CD, W^q)$ where W^q stands for the composition W of peers interacting using queues of size q.

A. 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 explicitly stating the messages on which they synchronize. This system is generated using SVL [13], which is a scripting language that complements the LOTOS encoding, and automates parts of the approach by calling the different CADP tools we use. Moreover, these scripts were used to circumvent the state explosion problem (see a discussion on this issue in Section VII). `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 "?" that are shown in Figure 5 for readability reasons, have a different meaning in LOTOS (they are used for value passing). Since we do not need value passing here, we have encoded messages without any direction for the synchronous case as they appear in the synchronization sets (noted between `|[..]|`) below.

From a collaboration diagram $CD = (P, E, M)$ involving n peers, first, peer LTSs in `Bcg` format are obtained from their LOTOS encoding presented in Section IV, and then the distributed system is generated as follows:

```
"distributed_system.bcg"=
   "peer_p1_lts.bcg"
   |[ alpha_peer'(p1, M) ∩
      (alpha_peer'(p2, M) ∪ ... ∪ alpha_peer'(pn, M)) ]|
      (
         "peer_p2_lts.bcg"
         |[ alpha_peer'(p2, M) ∩
      (alpha_peer'(p3, M) ∪ ... ∪ alpha_peer'(pn, M)) ]|
         ...
      )
```

Let us go back to our running example: here is the SVL code generated for the train station service. Note that 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 III) using a strong equivalence relation [16]. 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.

B. Asynchronous Communication

This case is slightly more complicated because asynchronous communication is not directly 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 (see Fig. 6). 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 bounded queue datatype (`BQueue` below) to store received messages. This datatype is implemented using algebraic specification facilities provided by LOTOS. 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.

A queue process can either interact with other peers on messages that can be received by its own peer (m_1 in `queue_`p below), or synchronizes with its own peer if that peer wants to evolve by consuming a message available in its own queue (m_1`_REC` in `queue_`p). Note that a local communication

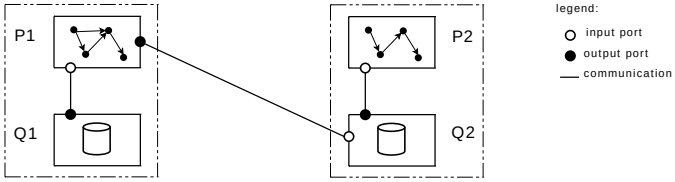


Fig. 6. System architecture: communication between peers and queues

between a peer and its queue has the suffix "_REC", whereas a communication between a peer (sender) and a queue does not have a suffix.

```
process queue_p [m1, m1_REC, ..., mn, mn_REC]
            (q:BQueue) : exit :=
  [not(bisfull(q))] -> m1;
     queue_p[m1, m1_REC, ...](binsert(m1,q))
  [] ... []
  [bishead(m1,q)] -> m1_REC;
     queue_p[m1, m1_REC, ...](bremove(q))
  [] exit
endproc
```

where $\{m_1, \ldots, m_n\} = alpha\_peer'(p, M)$.

Next, a process for each pair *(peer, queue)* is generated in LOTOS. A peer and a queue interact together on all messages (with suffix "_REC") that can be received by the peer. From an external point of view, these messages are not of interest while checking realizability (collaboration diagrams show the global ordering of message send events), and that is why they are hidden. We show below such a LOTOS process for a peer p. Notice that the process queue_p below is instantiated with a size set to B and an empty queue (nil). The queue size is an input parameter of the LOTOS encoding.

```
process peer_queue_p [m1, ..., mn, SYNC] : exit :=
    hide m1_REC, ..., mn_REC in
       (
          peer_p[m1, ..., mn, SYNC]
          |[ m1_REC, ..., mn_REC ]|
          queue_p[m1, m1_REC, ...](queue(B,nil))
       )
endproc
```

where $\{m_1, \ldots, m_n\} = alpha\_peer'(p, M)$.

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

```
"distributed_system_async.bcg"=
    "peer_queue_p1.bcg"
    |[ alpha_peer'(p1, M) ∩
      (alpha_peer'(p2, M) ∪ ... ∪ alpha_peer'(pn, M)) ]|
       (
          "peer_queue_p2.bcg"
          |[ alpha_peer'(p2, M) ∩
        (alpha_peer'(p3, M)∪...∪alpha_peer'(pn, M)) ]|
          ...
       )
```

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

As far as our running example is concerned, first the distributed system is generated as follows:

```
"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"
     )
  )
```

The equivalence check returns *false*, and indicates that the trace consisting of messages 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 a a_ts_itinerary or not because the recurrence type is "*" which means zero or one or more 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 Section VI how such unrealizable collaboration diagrams can be implemented without modifying the collaboration diagram.

C. Relating Realizability Results with Queue Sizes

The cost of the realizability check increases exponentially when queues' length increases. In [5], Fu *et al.* showed that asynchronous messaging leads to state space explosion for bounded message queues and undecidability of the model checking problem for unbounded message queues. Collaboration diagrams in combination with the projection method (introduced in Section IV) have an interesting property: it is possible to generalize results of the realizability check for queues with size one to queues with any size. More precisely, if the parallel composition of the projected peers with queue length one realizes a collaboration diagram, this parallel composition also realizes the collaboration diagram when peers have a queue length $q > 1$ or unbounded queues. Also, if the parallel composition of the projected peers with queue length one does not realize a collaboration diagram, this parallel composition does not realize the collaboration diagram when peers have queue length $q > 1$ either. Theorem 1 formalizes this property (formal proof of this theorem can be found in Appendix).

Theorem 1: Given a collaboration diagram CD, a queue length $q \in \mathcal{N}^+$, and the parallel composition of the *projected* peers W, W^q realizes CD if and only if W^1 re-

alizes CD: $\forall CD, W, q \in \mathcal{N}^+ \bullet Realizable(CD, W^q) \Leftrightarrow Realizable(CD, W^1)$

Intuitively, there are three main reasons for this property:

1) The equivalence check involves only sent messages, and received messages can be run at any moment without any control (it is not important when peers dequeue received messages or in what order these messages are dequeued). Therefore, in compositions with larger queues, while preserving realizability, peers can postpone receiving messages and send their own messages first.

2) Peer specifications are obtained from the collaboration diagram by projection. This guarantees that being realizable or not, W^1 always strongly simulates $CDLTS$. Also, the only difference between W^q and W^1 is that W^q has larger queues. This means that W^q always strongly simulates W^1. Using these two properties, the forward direction of Theorem 1 can be proved.

3) Each collaboration diagram defines a partial order on the occurrence of its events (see Appendix for more details). As a consequence, a message can be sent if and only if sending that message does not violate the defined order for message send events. On the other hand, by definition, there is no deadlock situation in a collaboration diagram (nor in a realizable composition of peers). Thus, in a realizable composition of peers, when a message is sent, it means: i) it will be eventually received, ii) time of receiving a message does not change the possible occurrence orders of events which have not occurred yet, and thus iii) while preserving observable behaviors, a message can be received right after it has been sent. Using the last property, without changing observable behavior of W^1, it can receive (dequeue) a message right after it has been sent (enqueued). Therefore, W^1 can keep its queues empty, and this property makes peers in W^1 able to send any message that peers in W^q are capable of sending (it proves the backward direction of Theorem 1).

VI. Peer Generation, Extended

Collaboration diagrams are unrealizable because peers do not respect either (i) the application order of messages in each thread, or (ii) dependency relations of messages among threads. To make peers respect interaction constraints of unrealizable collaboration diagrams, we need to enforce peers to execute messages in the same order as specified in the diagram. To do so, we will insert additional communication among peers. The first constraint (respecting the application order of messages in each thread) is achieved by adding in the collaboration diagram encoding some explicit messages prefixed with "SEQ_" between each thread message. With regards to the second one (respecting dependency relations), we will use the "SYNC_" messages that have been used in the initial encoding to respect message dependency relations.

However, all these additional messages are not necessary to make peers realize the collaboration diagram, and adding systematically a new message for each sequence in all threads and for each dependency relation may lead to excessive communication overhead between peers. Consequently, we want to minimize the number of extra communication operations. Given two events $e = (B, l, m, r)$ and $e' = (B', l', m', r')$ to be executed in order, we state two conditions below for synchronous and asynchronous communication, respectively, in which these new messages are not needed:

$C_{sync}(e, e') = (r = 1) \wedge \big((send(m') \in \{send(m), recv(m)\}) \vee (recv(m') \in \{send(m), recv(m)\}) \big)$

$C_{async}(e, e') = \big((r = 1) \wedge (send(m') \in \{send(m), recv(m)\}) \big) \vee \big((r \neq 1) \wedge (send(m) = send(m')) \big)$

Then, we define both sets SEQ and $SYNC$ containing additional messages that must be added to realize the input choreography. SEQ is defined using function $compute\_seq$ which accepts as input the list of threads computed from the collaboration diagram using function $sort\_by\_thread$ presented in Section III. For each thread Th_i consisting of a list of events (B, l, m, r), function $compute\_seq$ keeps messages SEQ_l if condition C_x is not verified. To cover all the events in each thread, we iterate from 1 to $|Th_i| - 1$ which is the last event considered since we compare each event k with event $k + 1$ in the list.

$SEQ = compute\_seq(Th_i = [(B_i, l_i, m_i, r_i)]_{i \in 1..n}) = \{\text{SEQ\_}l \mid \forall k \in 1..|Th_i| - 1 : e = Th_i[k] \wedge e' = Th_i[k + 1] \wedge e = (B, l, m, r) \wedge e' = (B', l', m', r') \wedge \neg C_x(e, e')\}$

$SYNC$ works directly on the collaboration diagram $CD = (P, E, M)$ and takes as input the set E of events. For each event used in the collaboration diagram, we check all the prefixes ($d_1, \ldots, d_q$, below) and keep only those which do not respect condition C_x (this is checked using function $check\_C$).

$SYNC = compute\_sync(E) = \{\text{SYNC\_}l_1\text{\_}l', \ldots, \text{SYNC\_}l_k\text{\_}l' \mid \forall e' = (B', l', m', r') \in E : \{l_1, \ldots, l_k\} = check\_C(B', e', E)\}$

$check\_C(\{d_i\}_{i \in 1..q}, e', E) = \{d_i \mid \forall e = (B, d_i, m, r) \in E : \neg C_x(e, e')\}$

where x in C_x is either $sync$ or $async$ depending on the communication model.

Let us describe how the generated LOTOS and SVL code presented in Sections III, IV, and V is extended to take additional synchronizations into account. First of all, the encoding of collaboration diagram into LOTOS (Section III) should also consider new "SEQ_" messages as specified below, and all process alphabets have to be extended with messages belonging to SEQ (in functions $alpha$, $alpha\_peer$, $alpha\_peer'$).

$$cd2l_t([(B_1, l_1, m_1, r_1), \ldots, (B_n, l_n, m_n, r_n)], SYNC, \mathbf{SEQ}) =$$
$$add\_pre\_sync(B_1, l_1) \texttt{ >> } cd2l_m(send(m_1)\_recv(m_1)\_m_1, r_1)$$
$$\mathbf{add\_seq}(\mathbf{l_1}, \mathbf{SEQ})$$
$$(add\_post\_sync(l_1, SYNC) \texttt{ >> } add\_exit(n))$$
$$cd2l_t([(B_2, l_2, m_2, r_2), \ldots, (B_n, l_n, m_n, r_n)], SYNC, \mathbf{SEQ})$$

where $add\_seq(l, SEQ) = \begin{cases} \text{SEQ\_}l\texttt{;} & \text{if } l \in SEQ \\ \epsilon & \text{otherwise} \end{cases}$

While generating peers (Section IV), the main difference concerns the hide construct generated in the body of each process cd_peer_p_aux where messages "SYNC_" should not be hidden since we need them in the forthcoming peer LTSs:

```
process cd_peer_p_aux [alpha_peer'(p, M), SYNC]
  hide gen_hide(p, M), ~~SYNC~~ in
    cd[alpha(p, M), SYNC]
endproc
```

Finally, the SVL code generated in Section V is extended with synchronizations between peers on additional messages. To do so, we need a function which is able to extract messages from SEQ and $SYNC$ in which a given peer is involved. For each peer, function $proj\_peer$ checks each action in $SYNC$ (SEQ, respectively) and verifies whether the corresponding event e involves as sender or receiver the peer p passed as input to the function:

$proj\_peer(p, SYNC, SEQ, E) = \{\texttt{SYNC\_}X\texttt{\_}Y \mid \texttt{SYNC\_}X\texttt{\_}Y \in SYNC \wedge e = (B, l, m, r) \in E \wedge (X = l \vee Y = l) \wedge p \in \{send(m), recv(m)\}\} \cup \{\texttt{SEQ\_}l \mid \texttt{SEQ\_}l \in SEQ \wedge e = (B, l, m, r) \in E \wedge p \in \{send(m), recv(m)\}\}$

$proj\_peers(\{p_1, \ldots, p_n\}, SYNC, SEQ, E) = proj\_peer(p_1, SYNC, SEQ, E) \cup \ldots \cup proj\_peer(p_n, SYNC, SEQ, E)$

Now, let us illustrate how the `distributed_system_async` process is extended for the asynchronous case (the modification is similar for the synchronous communication model). Basically, synchronization sets are complemented with some of the additional actions:

```
"distributed_system_async.bcg"=
  "peer_queue_p1_lts.bcg"
|[ (alpha_peer'(p1, M) ∪ proj-peer(p1, SYNC, SEQ, E)) ∩
      (alpha_peer'(p2, M) ∪ ... ∪ alpha_peer'(pn, M)
         ∪ proj-peers({p2, ..., pn}, SYNC, SEQ, E)) ]|
      (
         "peer_queue_p2_lts.bcg"
|[ (alpha_peer'(p2, M) ∪ proj-peer(p2, SYNC, SEQ, E)) ∩
      (alpha_peer'(p3, M) ∪ ... ∪ alpha_peer'(pn, M)
         ∪ proj-peers({p3, ..., pn}, SYNC, SEQ, E)) ]|
         ...
      )
```

Let us illustrate this extension with thread A of our running example. With respect to the sequential ordering of messages within each thread, note that potential new messages `SEQ_A1` and `SEQ_A2` do not appear after messages `ts_a_info` and `a_ts_infoAvail` because they are not necessary for realizing the choreography (C_{sync} returns *true* when called with the corresponding events). As far as dependency relations are concerned, message `SYNC_A3_B1` appears at the end of the thread behavior, and this message is necessary because peers have no other way to preserve the dependency relation specified in the event labelled by `B1` (*i.e.*, `A3/B1:book`). On the other hand, message `SYNC_1_A1` is discarded (since it is not needed for realizability).

```
(     (* -- thread A encoding -- *)
  ts_a_info; a_ts_infoAvail;
   loop_process[a_ts_itinerary] >> SYNC_A3_B1;exit
)
|[SYNC_A3_B1]|
...
```

From this extended collaboration diagram encoding, peers are generated by keeping the messages in which the peer does participate in visible, and also the additional communication introduced above. Peers synchronize on all additional communication that they share in their alphabets.

For instance, peers generated for our running example and extended with additional messages are shown in Figure 7. `SYNC_A3_B1` is the only necessary message because message `ts_b_book` must be run only after the message identified by `A3` (`a_ts_itinerary`) in the collaboration diagram, and involved peers have no other ways to respect this dependency. Notice that three peers are involved in this interaction, namely `Booking`, `TrainStation`, and `Availability`.

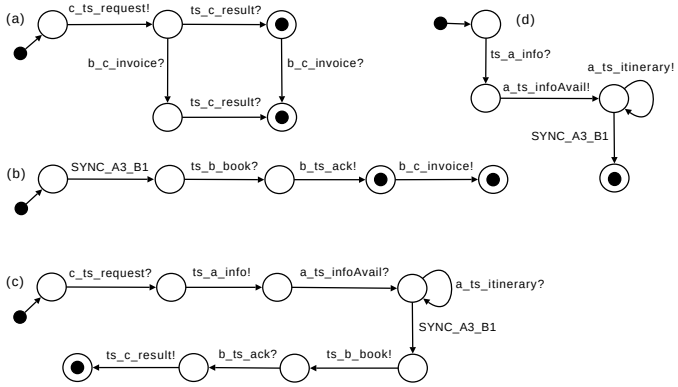


Fig. 7. Peers with additional messages: (a) customer, (b) booking, (c) train station, (d) availability

Once the new peers are generated, the distributed system is built by extending the description given in Section V with additional communication and also synchronizing peers on them. We recall that all peers do not synchronize on all additional communication 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 communication has been hidden, confirms that the extended peers realize the collaboration diagram.

VII. Tool Support and Experiments

The steps of our approach are automated by several tools. We have implemented a prototype tool named cd2lotos ((1) in Figure 1) which, given a collaboration diagram, generates the LOTOS code necessary to compute all the results we have presented in this article. 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 use. From this encoding, LTS generation is achieved using Caesar.adt and Caesar LOTOS compilers, as well as reduction techniques available in Reductor ((2) in Figure 1). Model-checking can be performed using Evaluator ((3) in Figure 1). 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 manually write some temporal properties that the choreography specification is supposed to satisfy. Last, Bisimulator is used to check that the collaboration diagram LTS is equivalent to the distributed peer implementation ((4) in Figure 1).

Our approach, and especially the tool we implemented (cd2lotos), was applied and validated on about 115 collaboration diagrams either obtained from available resources (research papers and on-line material) or written by ourselves.

Applying our code generator to these examples results in about 59,000 lines of LOTOS and 32,000 lines of SVL. It took about 61 minutes to check realizability for all the case studies of our database for both communication models: 34 examples turned out to be unrealizable in the case of synchronous communication, and 71 in the case of asynchronous communication. All the unrealizable ones were were verified to be realizable once additional communication was inserted in the peer protocols, and it took about 48 minutes to check again realizability of the whole database.

Table I 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 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 collaboration 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 running example used in this article.

It takes 2.4s for our prototype to generate LOTOS and SVL files for all the examples of our database for both communication models (synchronous and asynchronous) and both strategies (with and without additional communication). 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 collaboration diagrams (cd-059, cd-102), the computation time increases up to several minutes. It is interesting to note that examples involving more threads (cd-094) induce time consuming computations since they generate bigger intermediate state spaces due to the higher number of interleavings introduced due to the number of threads.

Table II shows results obtained for the unrealizable examples presented in Table I once additional communication is inserted. As expected, all these examples become realizable once the additional communication is added. Notice that realizability tests may take less time compared to Table I (cd-059, cd-094) because adding extra communication increases the sequentiality of the system, and therefore reduces communication interleaving.

During the experiments, we faces 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 pairs *(peer, queue)* was returning LTSs containing hundreds even thousands of states (*resp*. transitions). Consequently, we decided to build first each pair *(peer, queue)*, minimize them individually, and compose them to finally obtain the expected system. This technique (known as compositional verification in CADP) allows us to generate any step of the (distributed) system computation in a few seconds.

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

VIII. Related Work

There has been earlier work on studying and defining the realizability problem for choreography. In [17], [18], the authors define models for choreography and orchestration, and formalize a conformance relation between both models. These models are given as input whereas we focus on the generation of one from the other (generation of peers from a global specification) while ensuring conformance. In [19], the authors focus on *Let's dance* models for choreographies, and define for them an algorithm that determines if a global model is locally enforceable, and another algorithm for generating local models from global ones. In [20], the authors show through a simple example how BPEL stubs can be derived from WS-CDL choreographies, but, due to the lack of semantics of both languages, correctness of the generation cannot be ensured.

Some works define several realizability notions, and classify them in a hierarchy [6]. However, some of these notions (the weak ones) are questionable because by relaxing the message ordering constraints, the choreography specification's behavior is not preserved. Bultan and Fu [2] tackle the realizability issue in the context of asynchronous communication, and define some sufficient conditions to test realizability of choreographies specified with collaboration diagrams. In this article, we refine and extend this former work with tool support, some techniques to enforce realizability, and some new results on asynchronous communication (*wrt.* queue sizes).

Message Sequence Charts (MSCs), also know as sequence diagrams, are often compared with collaboration diagrams since they both allow the description of interactions of entities being composed. The realizability problem for MSCs has already been studied, see for instance [15], [21], [22]. Collaboration diagrams and MSCs provide a different view of interactions: MSCs specify the local orderings of the sent and received messages, whereas collaboration diagrams give the global ordering of the sent messages. In particular, received messages in collaboration diagrams can be ordered in any way as long as the sent messages respect the order specified in the diagram. Therefore, earlier results on realizability of MSCs are not applicable to the realizability of collaboration diagrams.

In terms of tools that check realizability, WSAT [23] is the only other tool we know. WSAT checks a set of realizability conditions on conversation protocols [5].

Other works [7], [8] propose well-formedness rules to enforce the choreography specification to be realizable. For example, in [8], 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. This solution is too restrictive since it may prevent the designer from specifying what (s)he wants to do. In addition, it complicates the choreography design since the designer cannot only focus on composition issues, but has to consider at the same time these well-formedness rules.

To the best of our knowledge, the only work which proposes to add messages in order to implement unrealizable choreographies is [7]. To do so, the authors extend their choreography

Collab. diagrams	Size			LOTOS (lines)	SVL (lines)	CD LTS (states/transitions)	Realizability		
	peers	messages	threads				sync.	async.	time
cd-008	5	9	4	388	148	27/46	√	√	19.56s
cd-025	4	6	3	304	130	12/15	√	√	16.20s
cd-045	5	8	3	341	130	10/13	√	×	18.69s
cd-059	10	20	3	666	238	56/175	×	×	1m12.31s
cd-094	7	13	6	495	184	96/396	×	×	1m46.14s
cd-102	16	30	4	959	346	220/748	×	×	6m31.39s

TABLE I
REALIZABILITY RESULTS FOR SOME CASE STUDIES (NO ADDITIONAL COMMUNICATION)

Collab. diagrams	Size			LOTOS (lines)	SVL (lines)	CD LTS (states/transitions)	Realizability		
	peers	messages	threads				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-094	7	13	6	501	188	96/396	√	√	1m25.25s
cd-102	16	30	4	974	350	220/748	√	√	6m51.51s

TABLE II
REALIZABILITY RESULTS FOR SOME CASE STUDIES (WITH ADDITIONAL COMMUNICATION)

language with new constructs (named dominated choice and loop). During the projection of these new operators, some additional communication is 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 explicitly state extra-constraints in the choreography specification by associating *dominant roles* to certain peers.

To sum up, most of these approaches focus on theoretical aspects (no tool support) whereas our contribution combines theoretical results (*e.g.,* relation between realizability results and message queue sizes) and tool support (the LOTOS encoding makes possible the complete automation of realizability test, choreography verification, and peer generation). Second, our approach allows implementation of any choreography specification without adding any rule or constraint on the choreography language or specifications written with it. Finally, we consider in this article both synchronous and asynchronous communication models.

IX. CONCLUDING REMARKS

In this article, we have studied the realizability question for choreography specifications. Realizability aims at checking whether peers involved in the choreography specification produce exactly the same behavior once they are obtained by endpoint projection and interact together in a distributed fashion. In this article we focused on collaboration diagrams as a choreography specification language. In order to detect realizability issues, we have presented an encoding of collaboration diagrams into LOTOS. LOTOS is a process algebra expressive enough to specify all the interaction constraints that can be specified with collaboration diagrams. In addition, LOTOS is equipped with CADP toolbox which we used to implement the different checks required to verify the choreography realizability. Our approach can deal with both synchronous and asynchronous communication, and is completely automated with a prototype tool we implemented to generate LOTOS code, and the use of the CADP toolbox to analyze results generated from this code. If a collaboration diagram is not realizable, we have proposed an alternative projection of peers which adds some additional communication in their description, and makes peers realize the choreography. We have also proved that realizability results in the case of asynchronous communication can be checked with queue size one, and generalized to any size of queues, possibly infinite ones.

For future work, a first perspective concerns implementation issues. In this article, we focused on formal and theoretical aspects, and we have not discussed how code can be (automatically) generated from abstract descriptions of peers obtained using our approach. From these abstract specifications, new services can be implemented in any programming language, in JAVA for instance as done in [24], using Pi4SOA technologies [25], or following guidelines presented in [26] where some BPEL code generation techniques are proposed. An alternative solution is to reuse an implementation of a service that already exists. In such a case, discovery techniques [27] can be used to check whether some existing services are compatible [28] with the peer LTSs at hand. If additional messages need to be taken into account to make peers implement the choreography, some wrappers can be generated as presented in [29]. These wrappers would allow us to guide the service behavior to make it respect its specification (in particular the additional communication appearing in the peer LTS) without changing the service functional code. A second perspective aims at extending our approach considering as input a set of collaboration diagrams. Indeed, a choice construct is missing in the collaboration diagram notation, and using a set of diagrams would allow to fill this gap. Finally, we plan to keep studying the relationship between queue sizes and realizability results in the case of

asynchronous communication. In particular, we would like to see if results presented in this article (queue size one is enough to check realizability of collaboration diagrams) hold for other choreography specification languages.

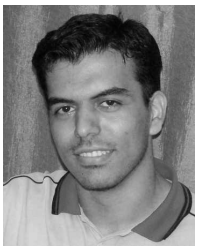
Acknowledgements. The authors thank Javier Cámara and José Antonio Martín for fruitful discussions and interesting comments on a former version of this article. This work has been partially supported by US National Science Foundation Grants CCF-0614002 and CCF-0716095.

REFERENCES

[1] X. Fu, T. Bultan, and J. Su, "Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services," *Theor. Comput. Sci.*, vol. 328, no. 1-2, pp. 19–37, 2004.
[2] T. Bultan and X. Fu, "Specification of Realizable Service Conversations using Collaboration Diagrams," *Service Oriented Computing and Applications*, vol. 2, no. 1, pp. 27–39, 2008.
[3] G. Salaün, L. Bordeaux, and M. Schaerf, "Describing and Reasoning on Web Services using Process Algebra," in *Proc. of ICWS'04*. IEEE CSP, 2004, pp. 43–51.
[4] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and H. M. W. Verbeek, "Choreography Conformance Checking: An Approach based on BPEL and Petri Nets," in *Proceedings of the Dagstuhl Seminar on The Role of Business Processes in Service Oriented Architectures*, 2006.
[5] X. Fu, T. Bultan, and J. Su, "Synchronizability of Conversations among Web Services," *IEEE Transactions on Software Engineering*, vol. 31, no. 12, pp. 1042–1055, 2005.
[6] R. Kazhamiakin and M. Pistore, "Analysis of Realizability Conditions for Web Service Choreographies," in *Proc. of FORTE'06*, ser. LNCS, vol. 4229. Springer, 2006, pp. 61–76.
[7] Z. Qiu, X. Zhao, C. Cai, and H. Yang, "Towards the Theoretical Foundation of Choreography," in *Proc. of WWW'07*. ACM Press, 2007, pp. 973–982.
[8] M. Carbone, K. Honda, and N. Yoshida, "Structured Communication-Centred Programming for Web Services," in *Proc. of ESOP'07*, ser. LNCS, vol. 4421. Springer, 2007, pp. 2–17.
[9] 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*, ser. LNCS, vol. 4590. Springer, 2007, pp. 158–163.
[10] G. J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.
[11] G. Salaün and T. Bultan, "Realizability of Choreographies using Process Algebra Encodings," in *Proc. of IFM'2009*, ser. LNCS, vol. 5423. Springer, 2009, pp. 167–182.
[12] ISO, "LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour," International Standards Organisation, Tech. Rep. 8807, 1989.
[13] H. Garavel and F. Lang, "SVL: A Scripting Language for Compositional Verification," in *Proc. of FORTE'01*. Kluwer, 2001, pp. 377–394.
[14] R. Mateescu and M. Sighireanu, "Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus," vol. 46, no. 3, pp. 255–281, 2003.
[15] R. Alur, K. Etessami, and M. Yannakakis, "Realizability and Verification of MSC Graphs," *Theoretical Computer Science*, vol. 331, no. 1, pp. 97–114, 2005.
[16] R. Milner, *Communication and Concurrency*, ser. International Series in Computer Science. Prentice Hall, 1989.
[17] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro, "Choreography and Orchestration Conformance for System Design," in *Proc. of Coordination'06*, ser. LNCS, vol. 4038. Springer, 2006, pp. 63–81.
[18] J. Li, H. Zhu, and G. Pu, "Conformance Validation between Choreography and Orchestration," in *Proc. of TASE'07*. IEEE Computer Society, 2007, pp. 473–482.
[19] J. M. Zaha, M. Dumas, A. H. M. ter Hofstede, A. P. Barros, and G. Decker, "Service Interaction Modeling: Bridging Global and Local Views," in *Proc. of EDOC'06*. IEEE Computer Society, 2006, pp. 45–55.
[20] J. Mendling and M. Hafner, "From Inter-organizational Workflows to Process Execution: Generating BPEL from WS-CDL," in *Proc. of OTM'05 Workshops*, ser. LNCS, vol. 3762. Springer, 2005, pp. 506–515.
[21] R. Alur, K. Etessami, and M. Yannakakis, "Inference of Message Sequence Charts," *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 623–633, 2003.
[22] S. Uchitel, J. Kramer, and J. Magee, "Incremental Elaboration of Scenario-based Specifications and Behavior Models using Implied Scenarios," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 13, pp. 37–85, 2004.
[23] X. Fu, T. Bultan, and J. Su, "WSAT: A Tool for Formal Analysis of Web Services," in *Proc. of CAV'04*, ser. LNCS, vol. 3114. Springer, 2004, pp. 510–514.
[24] N. Roohi, G. Salaün, and S. H. Mirian, "Analyzing Chor Specifications by Translation into FSP," in *Proc. of FOCLASA'09*, ser. ENTCS, vol. 255, 2009, pp. 159–176.
[25] "Pi4SOA Project." `www.pi4soa.org`.
[26] 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*, ser. LNCS, vol. 5364. Springer, 2008, pp. 84–99.
[27] A. Zisman, G. Spanoudakis, and J. Dooley, "A Framework for Dynamic Service Discovery," in *Proc. of ASE'08*. IEEE Computer Society, 2008, pp. 158–167.
[28] F. Duran, M. Ouederni, and G. Salaün, "Checking Protocol Compatibility using Maude," in *Proc. of FOCLASA'09*, ser. ENTCS, vol. 255, 2009, pp. 65–81.
[29] G. Salaün, "Generation of Service Wrapper Protocols from Choreography Specifications," in *Proc. of SEFM'08*. IEEE Computer Society, 2008, pp. 313–322.

Gwen Salaün received the PhD degree in Computer Science from the University of Nantes, France, in 2003. In 2003-2004, he held a post-doctoral position at the University of Rome "La Sapienza". In 2004-2006, he held a second post-doctoral position at INRIA, France. In 2006-2009, he was a research associate at the University of Malaga, Spain. He is currently an assistant professor at Ensimag (Grenoble INP) in Grenoble, France. His research interests include formal methods, specification and verification, composition of components and services.

Tevfik Bultan is a Professor in the Department of Computer Science at the University of California, Santa Barbara. He received his B.S. in electrical and electronics engineering in 1989 from the Middle East Technical University, and his M.S. in computer engineering and information science in 1992 from the Bilkent University, both in Ankara, Turkey. He received his Ph.D. in computer science in 1998 from the University of Maryland, College Park. He joined the Department of Computer Science at the University of California, Santa Barbara in 1998. His current research interests are: service oriented computing, concurrency, model checking, static analysis, and software engineering.



Nima Roohi received his MSc degree from Computer Science department of Sharif University of Technology, Iran, in 2008. He is interested in formal research areas, such as formal specification and verification, program development from formal specification, choreography and orchestration of Web services, concurrent programming.