Counterexample Guided Synthesis of Monitors for Realizability Enforcement

Matthias Güdemann, Gwen Salaün, Meriem Ouederni

Counterexample Guided Synthesis of Monitors for Realizability Enforcement

Matthias Güdemann[1], Gwen Salaün[2,1], and Meriem Ouederni[3]

[1] INRIA Rhône-Alpes, Grenoble, France
[2] Grenoble INP, France
[3] LINA, University of Nantes, France

Abstract. Many of today's software systems are built using distributed services, which evolve in different organizations. In order to facilitate their integration, it is necessary to provide a *contract* that the services participating in a composition should adhere to. A contract specifies interactions among a set of services from a global point of view. One important problem in a top-down development process is figuring out whether such a contract can be implemented by a set of services, obtained by projection and communicating via message passing. It was only recently shown, that this problem, known as *realizability*, is decidable if asynchronous communication (communication via FIFO buffers) is considered. It can be verified using the *synchronizability* property. If the system is not synchronizable, the system is not realizable either. In this paper, we propose a new, automatic approach, which enforces both synchronizability and realizability by generating *local* monitors through successive equivalence checks and refinement.

1 Introduction

Many software systems are now built using independently developed services, which are mostly geographically and organizationally distributed. The specification and analysis of interactions among such distributed systems is a major concern for ensuring their correctness and reliability. In order to simplify the construction of these systems, their design often relies on a *contract*, which describes from a global point of view the admissible interaction sequences exchanged between the participants. In the area of Service Oriented Computing (SOC), this contract is called *choreography* and the participants are called *peers*. The peers correspond to a distributed implementation of this choreography, and can be derived by *projection*, *i.e.*, by projecting the choreography specification to each peer by ignoring the messages that are not sent or received by that peer. A crucial question in this context is to check whether the peers behave exactly as required in the choreography. This property is called *realizability* [10, 1] and particularly matters when the system is developed following a top-down development process.

Figure 1 presents a simple example of choreography involving three peers (identified using 1, 2, and 3), which exchange three messages in sequence (a between 1 and 2, b between 2 and 3, and c between 1 and 2) and loops. On the right hand side of Figure 1, we give the projection obtained from this choreography, where question marks correspond to receptions and exclamation marks to emissions. Realizability aims at checking whether the distributed implementation respects the ordering constraints specified in the global choreography.

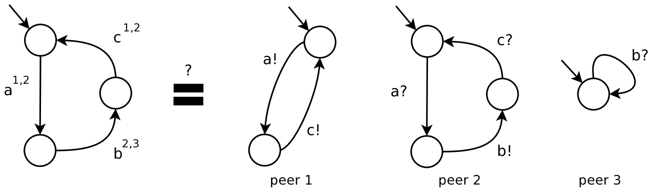

Fig. 1. Choreography, Peers, Realizability

Most distributed systems interact asynchronously where messages are sent and received through unbounded FIFO buffers. In this context, checking the realizability is a very difficult issue, because the distributed version of the system can generate infinite state spaces. This is the case of the distributed system given in Figure 1 for instance where peer 1 can infinitely send messages. Whether realizability is decidable was an open problem for several years. However, it was recently shown that it is decidable, verifying the *synchronizability* property [3]. A set of peers is synchronizable if and only if the system behavior, considering the send messages, preserves the same message sequences under synchronous and 1-bounded asynchronous communication. If a set of peers is synchronizable, one can check if it conforms to a choreography specification. If the system is not synchronizable then it is also not realizable. Both synchronizability and realizability checking involves finite state spaces and can be verified using equivalence checking techniques. The system described in Figure 1 is not synchronizable for example, because peer 1 can send a and c in sequence in the asynchronous system, whereas b occurs before c in the synchronous system as specified in the choreography.

Although this result is a significant step forward for formally analysing choreographies, there are still open issues that deserve to be studied. One of them arises when the realizability check returns *false*, due to one (or several) message exchange(s) violating the choreography ordering constraints. In this situation, there is no established solution for enforcing realizability and the designer is supposed to *patch* the choreography manually. However, correcting ordering issues may be a real burden for a designer, who just wants that the distributed implementation of his/her system behaves as specified in the choreography. This means that we need a way to control the distributed system to make it respect the global requirements. It is worth observing that, in this paper, when we refer to a *problem* in the choreography, this will always be an issue in the order of messages. Finding bugs (other than ordering issues) in choreographies can be achieved using existing verification tools.

In this paper, we propose a new approach, which identifies all problems which prevent synchronizability and realizability of a choreography, and provides a possible solution to enforce them. To do so, we generate *monitors*, which act as local controllers interacting with their peer and the rest of the system in order to make the peers respect the choreography requirements. These monitors are obtained by first generating the set of distributed peers by projection from the choreography specification. Then, we check in sequence the system synchronizability and realizability using equivalence checking. If one of these properties is violated, we exploit the generated counterexample to augment the monitors with a new synchronization message. Monitors are obtained through an iterative process, automatically refining their behaviors. The successive addition of these messages will finally enforce both synchronizability and realizability.

Our approach can be automated using any existing verification toolbox handling Labeled Transition Systems and providing an equivalence checker. We chose to encode choreographies into the value-passing process algebra LNT [6], one of the input languages of the CADP verification toolbox [12]. By doing so, we reuse existing state space exploration tools for generating peers and distributed systems, and equivalence checking techniques for verifying synchronizability and realizability. The process is fully supported (no human intervention) by calling various tools, some we reused from CADP, others we implemented ourselves, *e.g.*, for automating the iterative part of the process. We have validated our approach on hundreds of examples, some of them borrowed from real-world scenarios found in the literature.

Our monitor synthesis solution presents several advantages compared to existing results. Our approach goes beyond realizability checking by enforcing the system to respect the choreography. It is non-intrusive (peers are not modified or extended) and preserves the system parallelism by generating distributed monitors. It finds *all* problems in the choreography which prevent its realizability and suggests a distributed, implementable way to fix it. This is helpful in Service Oriented Computing or Component Based Software Engineering where black-box components are assumed. In the Web service domain, BPEL wrappers [2] can be automatically generated from our monitor models for controlling the distributed peers. In domains where the direct usage of the monitors is not an acceptable solution, the generated synchronization messages can serve to augment the choreography and provide a suggestion of how to fix it manually.

2 Background

We use conversation protocols [10] as choreography specification language in this paper. A conversation protocol is a low-level formal model, which can be computed from other existing specification formalisms such as collaboration diagrams [4], BPMN 2.0 choreographies [19], Singularity channels [22], or Message Sequence Charts (MSC) [1].

A conversation protocol is a Labeled Transition System (LTS) specifying the desired set of interactions from a global point of view. Each transition specifies an interaction between two peers $\mathcal{P}_{sender}$, $\mathcal{P}_{receiver}$ on a specific message m. A conversation protocol makes explicit the execution order of interactions. Sequence, choice, and loops are modeled using a sequence of transitions, several transitions going out from the same state and a cycle in the LTS, respectively.

Definition 1 (Conversation protocol). *A conversation protocol CP for a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *is an LTS* $CP = (S_{CP}, s^0_{CP}, L_{CP}, T_{CP})$ *where* S_{CP} *is a finite set of states and* $s^0_{CP} \in S_{CP}$ *is the initial state;* L_{CP} *is a set of labels where a label* $l \in L_{CP}$ *is a tuple* $m^{\mathcal{P}_i,\mathcal{P}_j}$ *such that* $\mathcal{P}_i$ *and* $\mathcal{P}_j$ *are the sending and receiving peers, respectively,* $\mathcal{P}_i \neq \mathcal{P}_j$*, and m is a message on which those peers interact; finally,* $T_{CP} \subseteq S_{CP} \times L_{CP} \times S_{CP}$ *is the transition relation. We require that each message has a unique sender and receiver:* $\forall m^{\mathcal{P}_i,\mathcal{P}_j}, m'^{\mathcal{P}'_i,\mathcal{P}'_j} \in L_{CP}$: $m = m' \implies \mathcal{P}_i = \mathcal{P}'_i \wedge \mathcal{P}_j = \mathcal{P}'_j$.

In the remainder of this paper, we denote a transition $t \in T_{CP}$ as $s \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_j}} s'$ where s and s' are source and target states and $m^{\mathcal{P}_i,\mathcal{P}_j}$ is the transition label.

We use LTSs for specifying the peer interaction model. This behavioral model defines the order in which the peer messages are executed. A label is a tuple (m, d) where m is the message name and d stands for the communication direction (either an emission ! or a reception ?). The set of messages in one peer LTS constitutes the peer *alphabet.*

Definition 2 (Peer). *A peer is an LTS* $\mathcal{P} = (S, s^0, \Sigma, T)$ *where S is a finite set of states,* $s^0 \in S$ *is the initial state,* $\Sigma = \Sigma^! \cup \Sigma^?$ *is a finite alphabet partitioned into a set of send and receive messages, and* $T \subseteq S \times \Sigma \times S$ *is a transition relation. We write* $m!$ *for a message* $m \in \Sigma^!$ *and* $m?$ *for* $m \in \Sigma^?$.

Peers are obtained by projection from a conversation protocol. After the projection they are determinized and minimized using standard algorithms [14], which is possible as the number of states and messages is finite.

Definition 3 (Projection). *Peer LTSs* $\mathcal{P}_i = (S_i, s^0_i, \Sigma_i, T_i)$ *are obtained by replacing in* $CP = (S_{CP}, s^0_{CP}, L_{CP}, T_{CP})$ *each label* $m^{\mathcal{P}_j,\mathcal{P}_k} \in L_{CP}$ *with* $m!$ *if* $j = i$*, with* $m?$ *if* $k = i$*, and with* τ *(internal action) otherwise; and finally removing the* τ*-transitions by applying standard minimization algorithms [14].*

The synchronous composite system corresponds to the distributed system computed over a set of peers communicating synchronously. In this context, a communication between two peers holds if and only if both agree on a synchronization label, *i.e.*, if one peer is in a state in which a message can be sent, then the other peer must be in a state in which that message can be received.

Definition 4 (Synchronous System). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *with* $\mathcal{P}_i = (S_i, s^0_i, \Sigma_i, T_i)$*, the synchronous system* $(\mathcal{P}_1 \mid \ldots \mid \mathcal{P}_n)$ *is the LTS* (S, s^0, Σ, T) *where:*

- $S = S_1 \times \ldots \times S_n$
- $s^0 \in S$ *such that* $s^0 = (s^0_1, \ldots, s^0_n)$

– $\Sigma = \cup_i \Sigma_i$
– $T \subseteq S \times \Sigma \times S$, *and for* $s = (s_1, \ldots, s_n) \in S$ *and* $s' = (s'_1, \ldots, s'_n) \in S$

(interact) $s \xrightarrow{m} s' \in T$ *if* $\exists i, j \in \{1, \ldots, n\} : m \in \Sigma^!_i \cap \Sigma^?_j$ *where* $\exists\ s_i \xrightarrow{m!} s'_i \in T_i$, *and* $s_j \xrightarrow{m?} s'_j \in T_j$ *such that* $\forall k \in \{1, \ldots, n\}, k \neq i \wedge k \neq j \Rightarrow s'_k = s_k$

In the asynchronous composite system, the peers communicate with each other asynchronously through FIFO buffers, *i.e.*, each peer $\mathcal{P}_i$ is equipped with a k-bounded message buffer Q^k_i. If k is not made explicit, noted Q_i, it means that $k = \infty$ and stands for unbounded buffers. A peer can either send a message $m \in \Sigma^!$ to the tail of the receiver buffer Q_j at any state where this send message is available, or read a message $m \in \Sigma^?$ from its buffer Q_i if the message is available at the buffer head.

Definition 5 (Asynchronous System). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *with* $\mathcal{P}_i = (S_i, s^0_i, \Sigma_i, T_i)$, *and* Q_i *being its associated buffer, the asynchronous system* $((\mathcal{P}_1, Q_1) \mid\mid \ldots \mid\mid (\mathcal{P}_n, Q_n))$ *is the LTS* (S, s^0, Σ, T) *defined as follows:*
– $S \subseteq S_1 \times Q_1 \times \ldots \times S_n \times Q_n$ *where* $\forall i \in \{1, \ldots, n\}$, $Q_i \subseteq (\Sigma^?_i)*$
– $s^0 \in S$ *such that* $s^0 = (s^0_1, \emptyset, \ldots, s^0_n, \emptyset)$
– $\Sigma = \cup_i \Sigma_i$
– $T \subseteq S \times \Sigma \times S$,
and for $s = (s_1, Q_1, \ldots, s_n, Q_n) \in S$ *and* $s' = (s'_1, Q'_1, \ldots s'_n, Q'_n) \in S$

(send) $s \xrightarrow{m!} s' \in T$ *if* $\exists i, j \in \{1, \ldots, n\} : m \in \Sigma^!_i \cap \Sigma^?_j$, *(i)* $s_i \xrightarrow{m!} s'_i \in T_i$, *(ii)* $Q'_j = Q_j m$, *(iii)* $\forall k \in \{1, \ldots, n\} : k \neq j \Rightarrow Q'_k = Q_k$, *and (iv)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$

(read) $s \xrightarrow{m?} s' \in T$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma^?_i$, *(i)* $s_i \xrightarrow{m?} s'_i \in T_i$, *(ii)* $mQ'_i = Q_i$, *(iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow Q'_k = Q_k$, *and (iv)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$

A system is synchronizable [11, 3] when its behavior remains the same under both synchronous and asynchronous communication semantics. This is checked by bounding buffers to $k = 1$ and comparing interactions in the synchronous system with send messages in the asynchronous system.

Definition 6 (Synchronizability). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, *the synchronous system* $(\mathcal{P}_1 \mid \ldots \mid \mathcal{P}_n) = (S_s, s^0_s, L_s, T_s)$, *and the 1-bounded asynchronous system* $((\mathcal{P}_1, Q^1_1) \mid\mid \ldots \mid\mid (\mathcal{P}_n, Q^1_n)) = (S_a, s^0_a, L_a, T_a)$, *two states* $r \in S_s$ *and* $s \in S_a$ *are synchronizable if there exists a relation* R *such that* $R(r, s)$ *and:*
– *for each* $r \xrightarrow{m} r' \in T_s$, *there exists* $s \xrightarrow{m!} s' \in T_a$, *such that* $R(r', s')$;
– *for each* $s \xrightarrow{m!} s' \in T_a$, *there exists* $r \xrightarrow{m} r' \in T_s$, *such that* $R(r', s')$;
– *for each* $s \xrightarrow{m?} s' \in T_a$, $R(r, s')$.
The set of peers is synchronizable if $R(s^0_s, s^0_a)$.

The approach presented in [3] proposes a sufficient and necessary condition showing that the realizability of conversation protocols is decidable.

Definition 7 (Realizability). *A conversation protocol CP is realizable if and only if (i) the peers computed by projection from this protocol are synchronizable, (ii) the 1-bounded system resulting from the peer composition is well-formed, and (iii) the synchronous version of the distributed system $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ is equivalent to CP.*

Well-formedness states that whenever the i-th peer buffer Q_i is non-empty, the system can eventually move to a state where Q_i is empty. For every synchronizable set of peers, if the peers are deterministic, *i.e.*, for every state, the possible send messages are unique, well-formedness is implied.

Both synchronizability and realizability properties are checked automatically using equivalence checking (weak trace equivalence in [3, 17]). This check requires the modification of the asynchronous system for hiding receptions ($m? \rightsquigarrow \tau$), renaming emissions into interactions ($m! \rightsquigarrow m$), and removing τ-transitions using standard minimization techniques.

Running Example. For illustration purposes we specify the use of an application in the cloud. This system involves four peers: a client (cl), a Web interface (int), a software application (appli), and a database (db). We show first a conversation protocol (Figure 2) describing the requirements that the designer expects from the composition-to-be. The conversation protocol starts with a login interaction (connect) between the client and the interface, followed by the setup of the application triggered by the interface (setup). Then, the client can access and use the application as far as necessary (access). Finally, the client decides to logout from the interface (logout) and the application stores some information (start/end time, used resources, etc.) into a database (log).

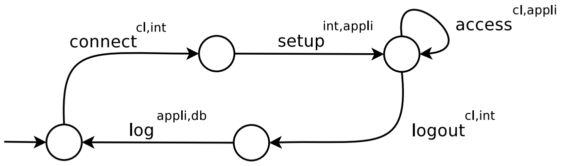

Fig. 2. Running Example: Choreography Specification

Figure 3 shows the four peers obtained by projection. This set of peers seems to respect the behavior specified in the conversation protocol, yet this is difficult to be sure using only visual analysis, even for such a simple example. In addition, as the choreography involves looping behavior, it is hard to know whether the resulting distributed system is bounded and finite, which would allow its formal analysis using existing verification techniques. Actually, this set of peers is not synchronizable (and therefore not realizable), because the trace of send messages "connect, access" is present in the 1-bounded asynchronous system, but is not present in the synchronous system. Synchronous communication enforces the sequence “connect, setup, access” as specified in the choreography, whereas in the asynchronous system peer cl can send connect! and access! in sequence.

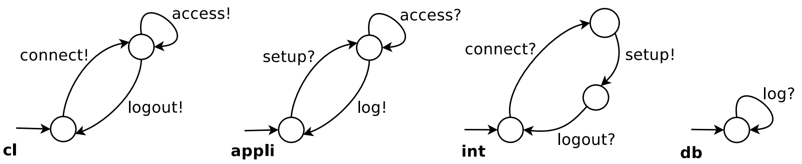

Fig. 3. Peer Projection

In the rest of this paper, we propose an automated technique to identify all problematic messages in a choreography. Our approach augments the system with new participants and interactions in order to restore the correct message sequences as specified in the global contract.

3 Counterexample Guided Realizability Enforcement

In our approach, we augment each peer by an accompanying monitor, which observes the behavior of the peer, and if necessary, controls the send messages according to the temporal ordering of the global specification. Adding monitors guarantees that the local behavior of the peers is not changed at all. The monitors locally receive the messages sent by their peer and relay them later after synchronization with the other monitors. They are refined by an iterative process, when it terminates the choreography is realized by the set of peers and monitors.

3.1 Monitors

A monitor interacts with its corresponding peer, with the other monitors and with receiving peers (via their buffers in the asynchronous system). The interaction with other monitors is done via synchronization messages, either incoming synchronizations of the form $m^{\leftarrow}$ for the synchronized monitor or outgoing synchronizations of the form $m^{\rightarrow}$, initiated by the synchronizing monitor. We call a message synchronized if there exists a synchronization message which delays it.

The monitor interacts with its corresponding peer over the send messages. The monitor locally receives the message from the peer. If the message needs to be synchronized, it first waits for the incoming synchronization message and then relays the message to the receiver, otherwise it relays the message directly to its receiver. If required, it will emit an outgoing synchronization message afterwards.

Definition 8 (Monitor). *A monitor is an LTS $M = (\overline{S}, \overline{s^0}, \overline{\Sigma}, \overline{T})$ where $\overline{S}$ is a finite set of states, $\overline{s^0}$ is the initial state, $\overline{\Sigma} = \overline{\Sigma^!} \cup \overline{\Sigma^?} \cup \overline{\Sigma^{\leftarrow}} \cup \overline{\Sigma^{\rightarrow}}$is a finite* *alphabet partitioned into sets of sending, locally receiving, incoming and outgoing synchronization messages and* $\overline{T} \subseteq \overline{S} \times \overline{\Sigma} \times \overline{S}$ *is a transition relation.*

The synchronous parallel composition of the peers and their monitors describes the system where all participants interact using synchronous communication.

Definition 9 (Monitored Synchronous System). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *with* $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$ *and a set of monitors* $\{M_1, \ldots, M_n\}$ *with* $M_i = (\overline{S_i}, \overline{s_i^0}, \overline{\Sigma}_i, \overline{T_i})$*, the monitored synchronous system* $((\mathcal{P}_1, M_1) \mid \ldots \mid (\mathcal{P}_n, M_n))$ *is the LTS* $\mathcal{SS}' = (S, s^0, \Sigma, T)$ *where:*

- $S = S_1 \times \overline{S_1} \times \ldots \times S_n \times \overline{S_n}$
- $s^0 \in S$ *such that* $s^0 = (s_1^0, \overline{s_1^0} \ldots, s_n^0, \overline{s_n^0})$
- $\Sigma = \cup_i \Sigma_i$
- $T \subseteq S \times \Sigma \times S$*, for* $s = (s_1, \overline{s_1} \ldots, s_n, \overline{s_n}) \in S$ *and* $s' = (s'_1, \overline{s'_n} \ldots, s'_n, \overline{s'_n}) \in S$

(send) $s \xrightarrow{\tau} s' \in T$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^! \cap \overline{\Sigma_i^?}$ *where* $\exists\ s_i \xrightarrow{m!} s'_i \in T_i$*, and* $\overline{s_i} \xrightarrow{m?} \overline{s'_i} \in \overline{T_i}$ *such that* $\forall k \in \{1, \ldots, n\}, k \neq i \Rightarrow s'_k = s_k \wedge \overline{s'_k} = \overline{s_k}$

(interact) $s \xrightarrow{m} s' \in T$ *if* $\exists i, j \in \{1, \ldots, n\} : m \in \overline{\Sigma_i^!} \cap \Sigma_j^?$ *where* $\exists\ \overline{s_i} \xrightarrow{m!} \overline{s'_i} \in \overline{T_i}$*, and* $s_j \xrightarrow{m?} s'_j \in T_j$ *such that* $\forall k \in \{1, \ldots, n\} : (k \neq j \Rightarrow s'_k = s_k) \wedge (k \neq i \Rightarrow \overline{s'_k} = \overline{s_k})$

(sync) $s \xrightarrow{\tau} s' \in T$ *if* $\exists i, j \in \{1, \ldots, n\} : m \in \overline{\Sigma_i^{\rightarrow}} \cap \overline{\Sigma_j^{\leftarrow}}$ *where* $\overline{s_i} \xrightarrow{m^{\rightarrow}} \overline{s'_i} \in \overline{T_i}$ *and* $\overline{s_j} \xrightarrow{m^{\leftarrow}} \overline{s'_j} \in \overline{T_j}$ *and* $\forall k \in \{1, \ldots, n\} : s'_k = s_k \wedge (k \neq i \wedge k \neq j \Rightarrow \overline{s'_k} = \overline{s_k})$

and finally removing the τ*-transitions.*

In the monitored asynchronous system, each pair $(\mathcal{P}_i, Q_i)$ is composed with the LTS of its monitor M_i. The asynchronous behavior of the peers and monitors corresponds to the distributed system where the sending peers communicate with their monitors, which relay the messages to the buffers of the receiving peers. This is shown in Figure 4 for two peers. The remote interactions between the monitors, local interactions between peers and their buffers or between peers and their monitors are marked with dashed lines. They are not observable from an external point of view. The visible interactions are the messages sent from one peer to the other. These are relayed by the monitor of the sending peer and are stored in the buffer of the receiving peer.

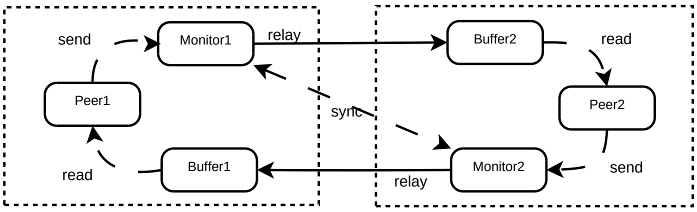

Fig. 4. Interactions in the Monitored Asynchronous System

Definition 10 (Monitored Asynchronous System). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *with* $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$*,* Q_i *its associated buffer and a set of corresponding monitors* $\{M_1, \ldots, M_n\}$ *with* $M_i = (\overline{S_i}, \overline{s_i^0}, \overline{\Sigma}_i, \overline{T_i})$*, the asynchronous system* $((\mathcal{P}_1, M_1, Q_1) \mid\mid \ldots \mid\mid (\mathcal{P}_n, M_n, Q_n))$ *is the LTS* $\mathcal{AS}' = (S, s^0, \Sigma, T)$ *where:*

- $S \subseteq S_1 \times \overline{S_1} \times Q_1 \times \ldots \times S_n \times \overline{S_n} \times Q_n$ *where* $\forall i \in \{1, \ldots, n\},\ Q_i \subseteq (\Sigma_i^?)*$
- $s^0 \in S$ *such that* $s^0 = (s_1^0, \overline{s_1^0}, \emptyset, \ldots, s_n^0, \overline{s_n^0}, \emptyset)$
- $\Sigma = \cup_i \Sigma_i$
- $T \subseteq S \times \Sigma \times S$*, and for*
  $s = (s_1, \overline{s_1}, Q_1, \ldots, s_n, \overline{s_n}, Q_n) \in S$ *and* $s' = (s_1', \overline{s_1'}, Q_1', \ldots s_n', \overline{s_n'}, Q_n') \in S$

(send) $s \xrightarrow{\tau} s' \in T$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^! \cap \overline{\Sigma_i^?}$*, (i)* $s_i \xrightarrow{m!} s_i' \in T_i$ *and* $\overline{s_i} \xrightarrow{m?} \overline{s_i'} \in \overline{T_i}$*, (ii)* $\forall k \in \{1, \ldots, n\} : Q_k' = Q_k$*, and (iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s_k' = s_k$ *and* $\overline{s_k'} = \overline{s_k}$

(relay) $s \xrightarrow{m!} s' \in T$ *if* $\exists i, j \in \{1, \ldots, n\} : m \in \Sigma_j^? \cap \overline{\Sigma_i^!}$*, (i)* $\overline{s_i} \xrightarrow{m!} \overline{s_i'} \in \overline{T_i}$*, (ii)* $Q_j' = Q_j m$*, (iii)* $\forall k \in \{1, \ldots, n\} : k \neq j \Rightarrow Q_k' = Q_k$*, (iv)* $\forall k \in \{1, \ldots, n\} : s_k' = s_k$*, and (v)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow \overline{s_k'} = \overline{s_k}$

(read) $s \xrightarrow{m?} s' \in T$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^?$*, (i)* $s_i \xrightarrow{m?} s_i' \in T_i$*, (ii)* $mQ_i' = Q_i$*, (iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow Q_k' = Q_k$*, (iv)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s_k' = s_k$*, and (v)* $\forall k \in \{1, \ldots, n\} : \overline{s_k'} = \overline{s_k}$

(sync) $s \xrightarrow{\tau} s' \in T$ *if* $\exists i, j \in \{1, \ldots, n\} : m \in \overline{\Sigma_i^{\rightarrow}} \cap \overline{\Sigma_j^{\leftarrow}}$*, (i)* $\overline{s_i} \xrightarrow{m^{\rightarrow}} \overline{s_i'} \in \overline{T_i}$ *and* $\overline{s_j} \xrightarrow{m^{\leftarrow}} \overline{s_j'} \in \overline{T_j}$*, (ii)* $\forall k \in \{1, \ldots, n\} : s_k' = s_k$*, (iii)* $\forall k \in \{1, \ldots, n\} : Q_k' = Q_k$*, and (iv)* $\forall k \in \{1, \ldots, n\} : k \neq i, j \Rightarrow \overline{s_k'} = \overline{s_k}$

and finally removing the τ*-transitions.*

Using Def. 9 and 10, synchronizability and realizability are checked as follows: For synchronizability, we check the equivalence between the monitored synchronous and monitored asynchronous system with 1-bounded buffers. For realizability we check the equivalence between the monitored synchronous system and the choreography.

3.2 Iterative Construction of the Monitors

We use an iterative approach to identify all the problematic messages in a choreography. At each iteration an equivalence check is conducted. If the check fails, its result is analyzed to decide which synchronization message must be added to the choreography. This results in the extended conversation protocol (ECP).

Definition 11 (Extended Conversation Protocol). *An extended conversation protocol ECP for a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *and corresponding set of monitors* $\{M_1, \ldots, M_n\}$ *is an LTS* $(S_{ECP}, s_{ECP}^0, L_{ECP} \cup L_{ECP}^+, T_{ECP})$ *where* S_{ECP}*,* s_{ECP}^0*,* L_{ECP} *are defined analogous to Def. 1; a synchronization label* $l \in L_{ECP}^+$ *is a tuple* sync^{M_j, M_k} *where* M_j *and* M_k *are the synchronizing and synchronized monitor* $(j \neq k)$*; finally,* $T_{ECP} \subseteq S_{ECP} \times (L_{ECP} \cup L_{ECP}^+) \times S_{ECP}$ *is the transition relation.*

The extended conversation protocol is augmented iteratively with synchronization messages until the choreography becomes realizable. This works for all non-faulty choreographies. Those which involve divergent choices are considered as faulty [22]. Realizability cannot be enforced in that case, as it is impossible to control divergent choices in a distributed system without changing the local behavior of the peers. Faulty choreographies are identified beforehand by detecting non-confluent diamonds of interactions in the conversation protocol using the executable temporal logic (XTL) [13].

The complete approach to enforce realizability of a choreography is shown as activity diagram in Figure 5. In a first step, we discard faulty choreographies. Then, we project the peers and start with the synchronizability check. At each iteration, the equivalence between the monitored synchronous and the 1-bounded monitored asynchronous system is checked. If this check fails, we analyze the counterexample, identify the problematic message, and augment the ECP with the necessary synchronization message. The synchronizability loop of the activity is executed as long as the system is not synchronizable. When the choreography is finally synchronizable, we proceed with the realizability check of the activity. Here, we check the equivalence between the monitored synchronous system and the original CP. The analysis of the counterexamples and the introduction of the synchronization messages is done as before, and we continue the activity until the realizability check succeeds.

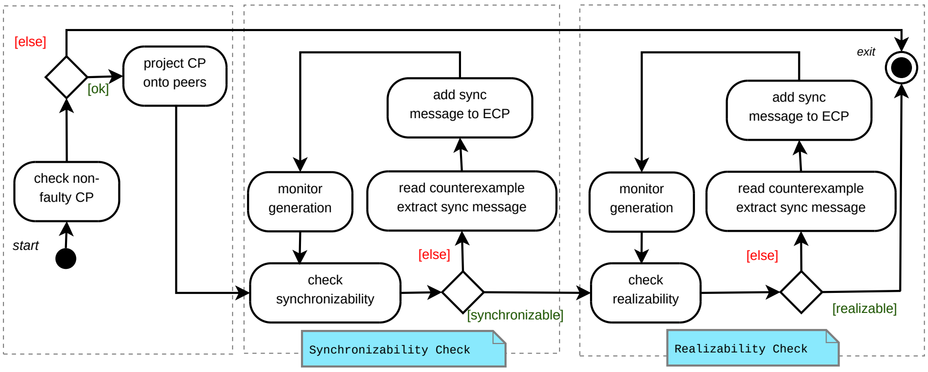

Fig. 5. Approach Overview

Now, we explain how we augment the ECP with synchronization messages for the monitors. If the equivalence check does not succeed in iteration k, a counterexample is returned. This is a finite trace, whose prefix is contained in both systems, but the sending of the last message m' is only possible in one of them. Therefore the sending of this message m' must be controlled by a monitor, in order to adhere to the specification. To do so, we introduce synchronization messages into ECP$_k$ as follows:

1. Locate in ECP_k the message m', its sending peer $\mathcal{P}_i$ and the states $s^*, s^{*'}$ for which there exists $s^* \xrightarrow{m'^{\mathcal{P}_i,\mathcal{P}}} s^{*'} \in T_{\mathrm{ECP}_k}$
2. Add a new state s_{new} to the set of states $S_{\mathrm{ECP}_{k+1}}$
3. Replace each $s^* \xrightarrow{m^{P_i,P_l}} s^{*'} \in T_{\mathrm{ECP}_k}$ with $s_{new} \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_l}} s^{*'}$ (with $s_{new} \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_l}} s_{new}$ if $s^* = s^{*'}$) in $T_{\mathrm{ECP}_{k+1}}$
4. For every incoming transition to s^*, $s \xrightarrow{m^{\mathcal{P}_j,\mathcal{P}_{j'}}} s^* \in T_{\mathrm{ECP}_k}$, add a new transition $s^* \xrightarrow{\mathrm{sync}_{m'}^{\mathcal{P}_j,\mathcal{P}_i}} s_{new}$ to $T_{\mathrm{ECP}_{k+1}}$ (for M_j to M_i, where $\mathrm{sync}_{m'}$ is a new name) and add the synchronization message $\mathrm{sync}_{m'}^{\mathcal{P}_j,\mathcal{P}_i}$ to $L^+_{\mathrm{ECP}_{k+1}}$

After each iteration, we derive the monitors from the extended conversation protocol. This can be achieved by using a process similar to the peer projection.

Definition 12 (Monitor Projection). *Monitor LTSs* $M_i = (\overline{S_i}, \overline{s_i^0}, \overline{\Sigma_i}, \overline{T_i})$ *are obtained by replacing in* $ECP = (S_{ECP}, s^0_{ECP}, L_{ECP} \cup L^+_{ECP}, T_{ECP})$ *each transition* $s \xrightarrow{m^{P_j,P_k}} s'$ *(i) with a sequence of transitions* $s \xrightarrow{m?} s^*$, $s^* \xrightarrow{m!} s'$ *if* $m \notin L^+_{ECP}$ *and* $P_j = P_i$, *(ii) with a sequence of transitions* $s \xrightarrow{m'?} s^*$, $s^* \xrightarrow{sync^{\leftarrow}_{m'}} s^{*'}$, $s^{*'} \xrightarrow{m'!} s'$ *if* $m = sync_{m'} \in L^+_{ECP}$ *and* $P_k = P_i$, *(iii) with* $s \xrightarrow{sync^{\rightarrow}_{m'}} s'$ *if* $m = sync_{m'} \in L^+_{ECP}$ *and* $P_j = P_i$, *and (iv) with* τ *otherwise; adding the new states* $s^*, s^{*'}$ *to* $\overline{S_i}$, *and finally removing the* τ*-transitions.*

Note that this projection does result in a correct monitor, but not necessarily in the most permissive one. Due to the lack of space, we do not give its formal definition here. Intuitively, we use an additional state machine composed with the monitor. This creates all possible interleavings of the monitor behavior and of the outgoing synchronization messages.

The iterative extension of the conversation protocol is guaranteed to terminate after a finite number of steps and to result in a realizable choreography. We must omit the proofs here, but the basic argument is as follows: the number of messages that may be synchronized is bounded and no message can be synchronized more than once; the equivalence checks assure that we find the right message to synchronize.

Complexity. In theory it can be necessary to synchronize every message $m \in L_{CP}$ of the conversation protocol. As the parallel composition and equivalence checks have a worst case complexity exponential in the number of peers $\#\mathcal{P}$, the worst case complexity of our approach is $O(|L_{CP}| \cdot |S_{CP}|^{\#\mathcal{P}})$. Nevertheless, our experience showed that this is unlikely in practical cases. Most often the number of additional synchronization messages is rather small and compositional verification techniques help to reduce the complexity of the parallel composition (for experimental details see Section 4).

Running Example. We illustrate the construction of the most permissive monitors for the example choreography shown in section 2, which is is not synchronizable. The message sequence "connect, access" is possible in the asynchronous system, but not in the synchronous one. The message access can only be sent from cl to appli after setup, therefore it must be deferred to be sent after that. To do so, we add a synchronization message for access to the choreography. This synchronization message is emitted by the monitor for int, who is the sender of the message setup. The left hand side of Figure 6 shows the extended conversation protocol with the first synchronization message.

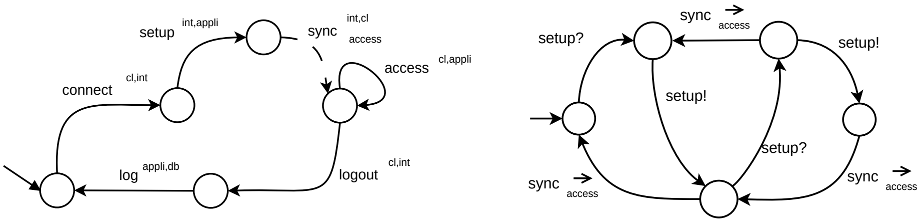

Fig. 6. After First Iteration

The right hand side of Figure 6 shows the monitor for the peer int. In the initial state it accepts the message sent by its peer (setup?). It relays this message to its receiver (setup!) and sends an outgoing synchronization message ($\mathrm{sync}^{\rightarrow}_{\mathrm{access}}$) afterwards. What may seem counter-intuitive is the possibility of the message sequence "setup?, setup!, setup?, setup!" followed by two synchronization messages. This is the result of constructing the most permissive monitor. As the peer is not blocked after it sends the first message, it may proceed to send it again. The monitor can relay both these messages. Nevertheless, after it relays the second one without an outgoing synchronization message, both must be synchronized, as synchronization messages are not buffered.

After the introduction of the first synchronization message, the choreography is synchronizable but not realizable. The equivalence check returns the counterexample "connect, setup, log", but logout must always precede log. A second synchronization message is therefore introduced right after the logout message. It is exchanged between the monitors for the peer cl (who sends logout) and for the peer appli. The left hand side of Figure 7 shows the monitor for appli after the second iteration. It accepts the local emission of the log message from its peer, waits for the incoming synchronization message, and then relays log.

Still, the choreography is not realizable in this form. The next counterexample is "connect, setup, logout, connect", *i.e.*, the peer cl starts a new connection attempt, before the log message is sent to db. A third synchronization message is introduced directly after log, between the monitor for appli and the monitor for cl. The right hand side of Figure 7 shows the monitor for appli after the third iteration. After the integration of the three synchronization messages, the choreography is finally both synchronizable and realizable.

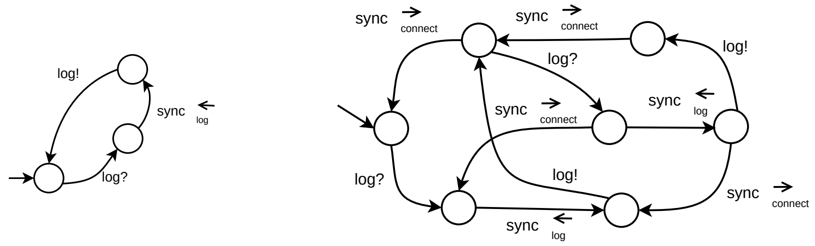

Fig. 7. Iterative Monitor Construction for Peer appli

4 Tool Support

Implementation. Our approach is tool-independent; every formal verification tool for equivalence checking of LTSs is usable. To automate the process, we chose the formal language LOTOS NT (LNT) [6]. It enables the description of concurrent processes, communicating via messages. It is fully integrated into CADP [12], which includes efficient methods for minimization under different equivalence relations, equivalence and model checking. The encoding into LNT also permits to analyze choreographies for bugs (other than message ordering issues) using CADP tools, *e.g.*, temporal properties expressed in MCL [16].

The conversation protocol, peers, monitors are encoded via the state machine pattern as LNT processes. We exploit the parallel composition operator of LNT to construct the most permissive monitors with all possible interleavings of the synchronization messages. The buffer behaviors are also encoded using LNT processes; the buffer operations are specified as LNT data types. The projection from the ECP onto the distributed peers is realized using label hiding and LTS reduction. The parallel composition of the FIFO buffers, peers and monitors, as well as of the monitored peers is done using the parallel composition and rendez-vous synchronization of LNT.

Experiments. We developed a test case generator, which we used to generate hundreds of conversation protocols with varying parameters, *e.g.*, number of peers, states and transitions. Our database of examples also includes 65 choreographies taken from the literature, as well as variants of them.

Table 1 shows the results for some of the experiments we conducted. For each example it shows the number of peers involved, the number of transitions and states in the choreography, and the number of additional synchronization messages. The fifth column shows the number of states and transitions of the largest intermediate LTS while creating the monitored asynchronous system. We use compositional verification, in particular smart parallel composition [8], where reductions are applied during the parallel composition and a composition sequence is decided heuristically. The final column shows the time for the longest iteration as well as the overall time for all computations and checks on a 3 Ghz Xeon CPU with 12 Gbyte RAM.

The number of peers has a significant influence on the state space of the intermediate LTSs, more so than the number of transitions, *e.g.*, see examples cp0031 and cp0032. The asynchronous behavior of many peers with only few messages generates many possible interleavings, while the behavior of few peers but more messages generally creates much less. This is the case, *e.g.*, in cp0153, which has a small number of peers, but a higher number of transitions, yet the intermediate state space of the LTS is rather small.

Table 1. Experimental Results

example	|peers|	$\|T\|/\|S\|$	|sync|	parallel composition	time max / total
cp0121	3	12 / 8	0	355 / 931	- / 54s
cp0016	3	4 / 3	1	121 / 337	46s / 1m 31s
cp0063	4	5 / 4	3	337 / 988	58s / 3m 54s
cp0153	3	29 / 16	5	15,182 / 59,033	53s / 7m 03s
cp0031	7	11 / 11	6	158,741 / 853,559	5m 47s / 19m 31s
cp0032	9	11 / 12	5	105,598 / 856,617	25m 53s / 1h 25m 10s

5 Related Work

There exists much work on the verification of realizability, *e.g.*, [10, 1, 4, 21, 15, 3], but none provides a solution if the choreography is not realizable. Let us focus on related approaches, which propose solutions for ensuring realizability of a choreography. In [5], the authors identify three principles for global descriptions under which they define a sound and complete end-point projection, *i.e.*, the generation of distributed processes from the choreography description. If these rules are respected, the distributed system obtained by projection will behave exactly as specified in the choreography. The same approach is chosen for BPMN 2.0 choreographies [18]. In [20], the authors propose to modify their choreography language to include new constructs (dominated choice and loop). During projection of these new operators, some communication is added to make the peers respect the choreography specification. However, these solutions prevent the designer from specifying what (s)he wants to, and complicates the design by obliging the designer to make explicit extra-constraints in the specification, *e.g.*, by associating *dominant roles* to certain peers. In [9], the authors propose a Petri Net-based formalism for choreographies and algorithms to check realizability and local enforceability. A choreography is locally enforceable if interacting peers are able to satisfy a subset of the requirements of the choreography. To ensure this, some message exchanges in the distributed system are disabled. In [21], the authors propose automated techniques to check the realizability of collaboration diagrams for different communication models. In case of non-realizability messages are added directly to the peers to enforce realizability. Collaboration diagrams are much less expressive than conversation protocols, as choices and loops cannot be specified, except for repetition of the same interaction.

Beyond advocating a solution for enforcing realizability, our contribution differs from these related works as follows. We focus on asynchronous communication and choreographies involving loops that may result in infinite state spaces. Our approach is non-intrusive; we do not add any constraints on the choreography language or specification, and the designer neither has to modify the original choreography specification, nor the peer models. Instead, we generate local monitors that preserve the system parallelism and control the peer behaviors to make them respect the choreography requirements.

The technique we rely on here shares some similarities with counterexample-guided abstraction refinement (CEGAR) [7]. In CEGAR, an abstract system is analyzed for temporal logic properties. If a property holds, the abstraction mechanism guarantees that the property also holds in the concrete design. If the property does not hold, the reason may be a too coarse approximation by the abstraction. In this case, the counterexample generated by the model checker, is used to refine the system to a finer abstraction and the process is iterated.

To the best of our knowledge, our approach is the first application of equivalence checking for a technique inspired from CEGAR. Moreover, our contribution goes beyond CEGAR related approaches, because we do not only automatically find problems in the model, but also offer a fix for (all of) them. Our approach allows to solve a problem, namely automatically fixing message ordering issues in a distributed system modeled using global contracts, for which no solution has been yet suggested.

6 Conclusion

In this paper, we have presented a new solution to identify all necessary changes to choreographies and synthesize distributed, local monitors which enforce realizability. Our approach is directly applicable to all notations which are transformable into conversation protocols. This is the case for most existing languages such as BPMN 2.0, collaboration diagrams, WS-CDL, Singularity channels, and MSC. We generate the monitors in successive iterations by checking both the synchronizability and realizability properties on the distributed system obtained by projection from the choreography specification. If one of these two properties is not satisfied, we use the counterexample resulting from this check to extend the monitors with additional synchronization messages. When both properties are finally ensured, we know that the system is bounded, synchronizable, and realizable. This assures the correct behavior of the distributed system according to the choreography, without making any change in the services themselves. Our main perspective aims at working with models closer to implementations that consider not only message passing communications, but also data exchanged between peers. This impacts choreography semantics and raises new issues such as dead code detection.

Acknowledgements. The authors would like to thank Samik Basu, Tevfik Bultan, Frédéric Lang, and Radu Mateescu for interesting discussions on the topics of this paper.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Realizability and Verification of MSC Graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.
2. T. Andrews et al. *Business Process Execution Language for Web Services (WS-BPEL)*. BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems, 2005.
3. S. Basu, T. Bultan, and M. Ouederni. Deciding Choreography Realizability. In *Proc. of POPL'12*. ACM Press, 2012.
4. T. Bultan and X. Fu. Specification of Realizable Service Conversations using Collaboration Diagrams. *Service Oriented Computing and Applications*, 2(1):27–39, 2008.
5. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *Proc. of ESOP'07*, LNCS. Springer, 2007.
6. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4). INRIA/VASY, 149 pages, 2011.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of CAV'00*, volume 1855. Springer, 2000.
8. P. Crouzen and F. Lang. Smart Reduction. In *Proc. of FASE'11*, volume 6603 of *LNCS*. Springer, 2011.
9. G. Decker and M. Weske. Local Enforceability in Interaction Petri Nets. In *Proc. of BPM'07*, volume 4714 of *LNCS*. Springer, 2007.
10. X. Fu, T. Bultan, and J. Su. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. *Theoretical Computer Science*, 328(1-2):19–37, 2004.
11. X. Fu, T. Bultan, and J. Su. Synchronizability of Conversations among Web Services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, 2005.
12. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of TACAS'11*, volume 6605 of *LNCS*. Springer, 2011.
13. H. Garavel and R. Mateescu. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In *Proc. STTT'98*, 1998.
14. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
15. N. Lohmann and K. Wolf. Realizability Is Controllability. In *Proc. of WS-FM'09*, volume 6194 of *LNCS*. Springer, 2010.
16. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*. Springer, 2008.
17. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
18. OMG. *Business Process Model and Notation (BPMN) – Version 2.0*. 2011.
19. P. Poizat and G. Salaün. Checking the Realizability of BPMN 2.0 Choreographies. In *Proc. of SAC'12*. ACM Press, 2012.
20. Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the Theoretical Foundation of Choreography. In *Proc. of WWW'07*. ACM Press, 2007.
21. G. Salaün and T. Bultan. Realizability of Choreographies using Process Algebra Encodings. In *Proc. of IFM'09*, volume 5423 of *LNCS*. Springer, 2009.
22. Z. Stengel and T. Bultan. Analyzing Singularity Channel Contracts. In *Proc. of ISSTA'09*. ACM, 2009.