Automated Choreography Repair

Samik Basu\(^1\) and Tevfik Bultan\(^2\)

\(^1\) Iowa State University, Ames, USA
sbasu@iastate.edu
\(^2\) University of California at Santa Barbara, Santa Barbara, USA
bultan@cs.ucsb.edu

Abstract. Choreography analysis is a crucial problem in concurrent and distributed system development. A choreography specifies the desired ordering of message exchanges among the components of a system. The realizability of a choreography amounts to determining the existence of components whose communication behavior conforms to the given choreography. The realizability problem has been shown to be decidable. In this paper, we investigate the repairability of un-realizable choreographies, where the goal is to identify a set of changes to a given un-realizable choreography that will make it realizable. We present a technique for automatically repairing un-realizable choreographies and provide formal guarantees of correctness and termination. We demonstrate the viability of our technique by applying it to several representative unrealizable choreographies from Singularity OS channel contracts and Web services.

1 Introduction

Choreography specifications are used in a variety of domains including coordination of software in service-oriented computing [18], specification of process interactions in Singularity OS [11], and specification of communication behavior among processes in distributed programs [2]. Choreographies describe desired message exchange sequences among components, programs or processes (we will refer to them as peers) of a distributed system. The choreography realizability problem is determining whether one can construct peers whose interaction behavior conforms to the given choreography. As an example, consider the choreography over two peers \(P_1\) and \(P_2\) shown in Fig. 1(a) where edges represent messages sent from one peer to another. This choreography describes a simple file transfer protocol [9] where \(P_1\) is the client asking for the file transfer and \(P_2\) is the file server. First, the client sends a message to the server to request that the server starts the transfer. When the transfer is finished, the server sends the “Transfer Finished” message and the protocol terminates. However, the client may decide to cancel the transfer before hearing back from the server by sending a “Cancel Transfer” message in which case the server responds with “Transfer Finished” message, which, again, terminates the protocol.
Figure 3(a) presents the projection of the choreography onto the participating peers resulting in the corresponding peer behaviors (send actions are denoted by “!” and receive actions are denoted by “?”). The distributed system that consists of the peer specifications shown in Fig. 3(a) can generate the message sequence:

\[ ms_{P1 \rightarrow P2}, mf_{P2 \rightarrow P1}, mc_{P1 \rightarrow P2} \] (1)

This sequence corresponds to the case where the server sends a “Transfer Finished” message (\(mf\)), but before consuming that message, the client sends the cancellation request message (\(mc\)). The sequence moves the server to an undefined (error) configuration, where the server does not know whether the file was transferred completely to the client before the client sent the cancellation request. In terms of the choreography specification shown in Fig. 1(a), the message sequence given above is not covered by the choreography, but any implementation of this choreography that uses asynchronous message passing will generate the message sequence: (1), violating the choreography specification. Hence, the choreography specification shown in Fig. 1(a) is un-realizable.

**Problem Statement.** This brings up the question: when a choreography is determined to be un-realizable, is it possible to automatically repair the choreography such that the repaired version is realizable? We will refer to this problem as the choreography repairability problem. Its importance stems from the fact that automation in repairing choreography will allow faster development of distributed systems with formal guarantees of correctness.

**Our Solution.** Our choreography repair technique analyzes and eliminates the cause of violation of the condition for choreography realizability. In [4], we have proved that choreography \(\mathcal{C}\) is realizable if and only if its behavior (i.e., the set of message sequences generated by \(\mathcal{C}\), denoted as \(\mathcal{L}(\mathcal{C})\)) is identical to the behavior of \(\mathcal{I}_1^\mathcal{C}\) (denoted as \(\mathcal{L}(\mathcal{I}_1^\mathcal{C})\)), where \(\mathcal{I}_1^\mathcal{C}\) is the asynchronous system in which each
public contract ReservationSessionContract {
  state Start: {
    request? -> Decide;
  }
  state Decide: {
    succeed! -> Success;
    failed! -> Sink;
    cancel? -> (cancelled!);
  }
  state Success: {
    cancel?;
    confirm?;
  }
  state Sink: {
    cancel?;
  }
}

Fig. 2. (a) Channel contract for ReservationSession and (b) the corresponding state machine

participating peer has at most one pending message at any point of time, and is obtained from the projection of C. We present two types of choreography repair mechanisms that eliminate the differences between L(C) and L(I_C^1):

1. Relaxation. The choreography C is changed to C’ such that L(C) ⊆ L(C’), i.e., new behavior is added to C, such that L(C’) = L(I_C’^1).
2. Restriction. The choreography C is changed to C’ such that L(C) = L(C ↓_C) = L(I_C’^1 ↓_C) ⊆ L(I_C’^1), where ↓_C denotes the behavior projected on the messages in C. This change implies that some behavior of I_C’^1 is disallowed in I_C’^1. This is achieved by adding extra synchronization messages in C’. When these extra messages are projected away, the repaired choreography C’ specifies exactly the same sequences of messages specified by the un-realizable choreography C.

For example, the choreography in Fig. 1(a) is changed to the one in Fig. 1(b) via relaxation, by adding new behavior (blue bold-edge), which makes the latter realizable. This is because the sequence that made C un-realizable (see sequence (1) above) is now included in the repaired version C’. On the other hand, Fig. 1(c) demonstrates repair via restriction, by adding synchronization messages from state s_1 to ns(0) (red dotted-edges); this repair also makes the resulting choreography realizable. In this case, the sequence in (1) is not possible in I_C’^1.

Contribution. We present a formal characterization of choreography repairability. To the best of our knowledge, this is the first time such a characterization has been presented. We present a sound and complete algorithm for choreography repair based on this characterization. We also discuss its application by demonstrating automated repair of several unrealizable choreographies. Although choreography examples we use in this paper consist of two-party choreographies, the formal model and the repair algorithm we present are general and handle multi-party choreographies.
2 Repairing Singularity OS Channel Contracts

We motivate the practical applicability of automated choreography repair using Singularity OS channel contracts. Singularity OS [16] is developed by Microsoft research with the objective of improving OS dependability by ensuring process isolation. The processes in Singularity OS communicate over FIFO channels and follow specific channel contracts (choreographies in our case); that specify allowable communication patterns between processes (client and server). The Singularity OS channel contracts correspond to choreography specifications. One problem is to determine whether one can implement a client and a server whose interaction conforms to the given channel contract, i.e., determining realizability of the given channel contract.

Figure 2(a) presents a channel contract called reservation session contract (where message declarations are omitted for brevity). The contract specifies four explicit states and the message sequences from the perspective of the server. For instance, the contract specifies that the state changes from “Start” to “Decide” when the server receives a message “request” from the client. From the state “Decide”, there are three choices: the server sends the message “succeed” to the client resulting in the state update to “Success”; the server responds to the client with message “failed” leading to the state “Sink”; the client sends “cancel” followed by the server sending “cancelled” message. Figure 2(b) presents the state machine for this contract ($C$ represents the client and $S$ represents the server).

The Singularity Design Note 5 [16] states that the client and server processes that are verified to conform to a given channel contract (i.e., that implement the projection of the channel contract correctly) are guaranteed to interact without any deadlocks. However, in [17], the authors demonstrated that this claim is incorrect since the channel contract itself can be un-realizable, in which case the processes implemented based on the projection of the contract can deadlock. One of the examples demonstrating this problem is the reservation session contract from Singularity OS shown above. Due to asynchronous communication, the client and server can move out-of-sync and deadlock. Consider the scenario where the client sends a “cancel” message and waits for the “cancelled” message from the server, while the server sends a “failed” message and consumes the “cancel” message from the client. This sequence of interactions leads to a deadlock. In fact there are no client and server processes that can conform to this contract without deadlock while interacting via FIFO channels (as required by the Singularity OS), i.e., the choreography specified by this channel contract is un-realizable.

The automated choreography repair technique we present in this paper is directly applicable to Singularity OS channel contracts. Using our technique we can repair un-realizable channel contracts, and ensure deadlock free implementation of repaired contracts. We will discuss the application of our automated choreography repair technique to the reservation contract in Sect. 5.
3 Choreography Realizability

We proceed by presenting an overview of the existing results [4] on choreography realizability, which forms the basis of our automated choreography repair strategy.

**Peers.** The behavior $B$ of a peer $P$ is a finite state machine $(M, T, t_0, \delta)$ where $M$ is the union of input ($M^{in}$) and output ($M^{out}$) message sets, $T$ is the finite set of states, $t_0 \in T$ is the initial state, and $\delta \subseteq T \times (M \cup \{\epsilon\}) \times T$ is the transition relation. A transition $\tau \in \delta$ can be one of the following three types: (1) a send-transition of the form $(t_1, !m_1, t_2)$ which sends out a message $m_1 \in M^{out}$, (2) a receive-transition of the form $(t_1, ?m_2, t_2)$ which consumes a message $m_2 \in M^{in}$ from peer’s input queue, and (3) an $\epsilon$-transition of the form $(t_1, \epsilon, t_2)$. We write $t \xrightarrow{a} t'$ to denote that $(t, a, t') \in \delta$. Figure 3(a) illustrates the behavior of peers $P_1$ and $P_2$; states in $P_i$ are denoted by a tuple ($P_i$;“state-name”).

**System.** Given a set of peers $\mathcal{P} = \{P_1, \ldots, P_n\}$ with $B_i = (M_i, T_i, t_{0i}, \delta_i)$ denoting the behavior of $P_i$ and $M_i = M^{in}_i \cup M^{out}_i$ such that $\forall i : M^{in}_i \cap M^{out}_i = \emptyset$, and $\forall i, j : i \neq j \Rightarrow M^{in}_i \cap M^{in}_j = M^{out}_i \cap M^{out}_j = \emptyset$. A system behavior or simply a system over $\mathcal{P}$ is denoted by a (possibly infinite state) state machine $I = (\mathcal{P}, S, s_0, M, \Delta)$ where $\mathcal{P}$ is the set of peers, $S$ is the set of states in the

![Image](image-url)
system and each state $s = (Q_1, t_1, Q_2, t_2, \ldots Q_n, t_n)$ in the system is described by the local states $(t_i s)$ of the peers in $P$ along with the contents of their queues $(Q_i s)$. $s_0 \in S$ is the start state, where none of the peers have any pending messages in their queue to consume. The set $M$ contains the set of all messages that are being exchanged by the participating peers.

Finally, the transition relation $\Delta$ is described as follows. The send actions are non-blocking, i.e., when a peer $P_i$ sends a message $m$ to a peer $P_j$ (denoted by $m^{P_i \rightarrow P_j}$), the message gets appended to the tail of the queue associated to $P_j$. We refer to the queue as the receive queue of $P_j$. The receive actions are blocking, i.e., a peer can only consume a message if it is present at the head of its receive queue; on consumption of the message, it is removed from the head of the queue. Only the send actions are observable in the system as these actions involve two entities: the sender sending the message and the receive queue of the receiver. All other actions are local to one peer and, therefore, unobservable ($\epsilon$-transitions).

We will use the functions $\text{1st}(\ldots)$ and $\text{1qu}(\ldots)$ to obtain local state and queue of a peer from a state in the system, i.e., for $s = (Q_1, t_1, Q_2, t_2, \ldots Q_n, t_n) \in S$, $\text{1st}(s, P_i) = t_1$ and $\text{1qu}(s, P_i) = Q_1$.

**K-bounded System.** A $k$-bounded system (denoted by $\mathcal{I}_k$) is a system where the length of message queue for any peer is at most $k$. In any $k$-bounded system, the send actions can block if the receive queue of the receiving peer is full. Any $k$-bounded system is finite state as long as the behaviors of the participating peers are finite state. Figure 3(b) illustrates the system $\mathcal{I}_1$ obtained from the communicating peers $P_1$ and $P_2$ of Fig. 3(a). Note that initially $P_1$ is at the local state $P_1 : s_1$ with an empty receive queue denoted by $[\ ]$.

**Choreography Specification.** A choreography specification is a finite state machine $\mathcal{C} = (\mathcal{P}, S^C, s_0^C, L, \Delta^c)$ where $\mathcal{P}$ is a finite set of peers, $S^C$ is a finite set of states, $s_0^C \in C$ is the initial state, $L$ is a finite set of message labels and, finally, $\Delta^c \subseteq S^C \times \mathcal{P} \times L \times \mathcal{P} \times S^C$ is the transition relation. A transition of the form $(s^C_i, P, m, P', s^C_j) \in \Delta^c$ represents the sending of message $m$ from $P$ to $P'$ ($P, P' \in \mathcal{P}$).

**Peer Projection.** The projection of a choreography $\mathcal{C}$ on one of the peers $P_1$ is obtained from $\mathcal{C}$ by performing the following updates to the state machine specifying $\mathcal{C}$. (a) If a transition label is $m^{P \rightarrow P'}$ then replace it with $!m$; (b) if a transition label is $m^{P' \rightarrow P}$ then replace it with $?m$; (c) otherwise, replace transition label with $\epsilon$. The system obtained from the asynchronous communication of the projected peers of $\mathcal{C}$ is denoted by $\mathcal{T}^C; \mathcal{T}^C_1$ being the corresponding 1-bounded system. The language of a choreography or a system is described in terms of a set of sequences of send actions of the form $m^{P \rightarrow P'}$; the concatenation of $\epsilon$ to any sequence results in the sequence itself. The language is denoted by $\mathcal{L}(\cdot)$.

**Theorem 1 (Realizability [4]).** $\mathcal{C}$ is language realizable $\iff [\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{T}^C_1)]$

This theorem states that a choreography is realizable if and only if the set of sequences of send actions of a choreography is identical to the set of sequences of send actions of the 1-bounded system where the participating peers are generated from the (determinized) projection of the choreography under consideration.
Figure 3(b) presents the behavior of the system $I_C^1$ for the choreography specification $C$ shown in Fig. 1(a), where epsilon-labeled transitions denote consumption of messages and other transitions denote sending of messages. The choreography $C$ is un-realizable because it does not include a specific send sequence that is possible in $I_C^1$ (Fig. 3(b)) (Sequence (1) discussed in Sect. 1).

4 Choreography Repair

Types of Repair. In this paper, we present two alternative techniques for repairing un-realizable choreographies. One is based on adding new behaviors (in terms of sends) to $C$, which we call relaxation. The other is based on adding constraints that do not alter allowed sequences of sends in $C$ but restrict the behavior in $I_C^1$. We call this approach restriction. The techniques will be based on the observation that from Theorem 1 and from the nature of asynchrony, it follows: $L(C) \neq L(I_C^1) \Rightarrow L(C) \subset L(I_C^1)$.

State Relationships Between $I_C^1$ and $C$. Before we describe the repair techniques, we first discuss the structure of the $I_C^1$, which is crucial for understanding our approach. If a state in $C$ is represented as $s_C$, then the corresponding state in the peer $P$ is a tuple denoted by $P:s_C$. Proceeding further, if $s$ is a state in $I_C^1$, then $s = (Q_1, t_1, \ldots, Q_n, t_n)$, where $n$ is the number of peers and $t_i$ is of the form $P_i:s_i^C$. Note that, the local states of each peer in $s$ may have been obtained from different states $s_i^C$ in $C$.

Consider for example, the second state of the system in Fig. 3(b)–$P_1$ is at a state $P_1:s_1$ obtained from the state $s_1$ in $C$ and $P_2$ is at a state $P_2:s_0$ obtained from the state $s_0$ in $C$. Using the notations introduced in Sect. 3, $1St((P_1:s_1:[\,], P_2:s_0:[ms]), P_1) = P_1:s_1; 1Qu((P_1:s_1:[\,], P_2:s_0:[ms]), P_2) = [ms]$.

4.1 Differences Between $C$ and $I_C^1$

In order to apply relaxation or restriction, it is important to identify at least one difference between $C$ and $I_C^1$ in terms of sequences of send actions. We know that for un-realizable $C$, $L(C) \subset L(I_C^1)$. Therefore, there exists at least one send sequence in $I_C^1$ which is absent in $C$.

Consider that there exists a path in $I_C^1$ in the form

$$s_1 \xrightarrow{m_{P_1-P_1'}} s_2 \xrightarrow{m_{P_2-P_2'}} s_3 \rightarrow \ldots s_i \xrightarrow{m_{P_i-P_i'}} s_{i+1} \quad (2)$$

which generates the following sequence of send actions $m_{P_1-P_1'}, m_{P_2-P_2'}, \ldots, m_{P_i-P_i'}$. Assume that, none of the paths in $C$ allow the above send sequence. However, there exists a path in $C$ which replicates the above sequence till $m_{P_{i-1}-P_{i-1}}$. Let such a path be denoted by
Case 1. Consider that the receive queue of peer \(i\) is non-empty, implying that there is one pending message to be consumed (recall that the \(I_1^C\) is 1-bounded system with each receive queue capacity being 1). In other words,
some peer (say, $R$) has sent the message (say $m$) to $P_i$ and $P_i$ has not encountered any receive action along the choreography path it has taken resulting in system path shown in (2).

This case corresponds to the situation described as independent branching (see above), when peer $P_i$ is moving along a choreography specification path $\pi$ and the other peer $R$ is moving along a different path $\pi'$ of the choreography specification, resulting in the path shown in (2). Furthermore, $R$ has sent $m$ to $P_i$ which resides un-consumed in the receive queue of $P_i$.

**Case 1a.** Let there be a transition in the behavior of peer $P_i$ at state $P_i : c_{i+1}$, where it can consume the message in its queue: $P_i : c_{i+1} \xrightarrow{m} P_i : c_i'$. That is, the choreography specification includes $c_{i+1} \xrightarrow{m_{R \rightarrow P_i}} c_i'$ along the path $\pi$. Therefore, both of the paths under consideration, $\pi$ and $\pi'$, have the send action $m_{R \rightarrow P_i}$.

In $\pi$, $m_{i_s \rightarrow P_i}$ is followed by $m_{R \rightarrow P_i}$. In $\pi'$, $m_{R \rightarrow P_i}$ is not followed by $m_{i_s \rightarrow P_i}$.

In this case, the relaxation adds $t_i \xrightarrow{m_{i_s \rightarrow P_i}} t_i'$ in the choreography specification and sets $t_i'$ to $c_i'$.

**Case 1b.** On the other hand, if there exists no transition in the behavior of peer $P_i$ starting from state $P_i : c_{i+1}$ where it can consume the message in its queue, then the following repair is done.

**Case 1b-i.** If $P_i : c_{i+1}$ belongs to a cycle then in the newly added transition $t_i \xrightarrow{m_{i_s \rightarrow P_i}} t_i'$ is set to a newly generated state, which replicates the choreography specification starting from $c_{i+1}$. Note that, the repair does not assign $t_i'$ to $c_i+1$. This is because such assignment will result in unnecessary over-relaxation of choreography specification due to the presence in $m_{R \rightarrow P_i}$ in path $\pi'$ and its possible absence in the cycle which is part of the path $\pi$. We will discuss below this scenario using the example in Fig. 4.

**Case 1b-ii.** If $P_i$ at $P_i : c_{i+1}$ cannot consume the pending message and $P_i : c_{i+1}$ does not belong to any cycle, then $t_i'$ is set to a newly generated state. The addition of the new transition removes the identified difference between the choreography and the system.

For instance, in Fig. 3(b), the path in $\mathcal{T}_i^C$ that is absent in $C$ (Fig. 1(a)) has the sequence $m_{s_1 \rightarrow P_2}, m_{f_{P_2 \rightarrow P_1}}, m_{c_{P_1 \rightarrow P_2}}$. Note that, we are considering only the send actions and the transitions are considered with zero or more occurrences of $\epsilon$ followed by a send action. The path in $C$ that replicates most of this sequence is $s_0 \xrightarrow{m_{s_1 \rightarrow P_2}} s_1 \xrightarrow{m_{f_{P_2 \rightarrow P_1}}} s_3$. Therefore, for repair by relaxation, our objective is to add a transition with send action $m_{c_{P_1 \rightarrow P_2}}$ from the choreography state $s_3$. From the system, we know that the peer $P_i$ at the state $P_i : s_2$ can consume the message $m_f$ in its receive queue and move to a state in $P_i : s_4$ (see Fig. 3). Therefore, the transition added from $s_3$ has the destination state $s_4$. The result of this repair by relaxation is the choreography specification presented in Fig. 1(b). This illustrates the Case 1a of repair by relaxation.

Figure 4 illustrates the applications of Case 1b-i and 1a. The local states of the peers participating in the system transitions are presented in bold-font.
If the check is successful, then we can infer that Case 2a. transition followed by some other sequence of transitions.

Case 2. Now consider that the receive queue \( Q_i \) of the peer \( P_i \) is empty, implying that there is no pending message to be consumed. Unlike the previous case, in this situation, the difference between \( T^C_i \) and \( C \) (represented by paths (2) and (3) in Sect. 4.1) is not necessarily due to independent branches, when two peers move along two different paths of the choreography specification.

Instead the peers may be moving along the same path of the choreography specification, and the latter has imposed an “un-realizable” ordering of send actions involving \( m_{i}^{P_i \rightarrow P_i'} \). In other words, it is not possible to “stop” \( P_i \) from sending the message \( m_i \) from its projected behavior when the choreography specification reaches \( t_i \), however \( t_i \) does not have \( m_{i}^{P_i \rightarrow P_i'} \). This corresponds to the case of independent sequences (see above).

Recall that, the choreography specification state is \( t_i \) from where there is no matching \( m_{i}^{P_i \rightarrow P_i'} \) event. We check whether there exists a path from \( P_i : t_i \) (i.e., local state of \( P_i \) obtained from projection at \( t_i \)) to \( P_i : t_i \) in the peer \( P_i \) via a sequence of transitions such that after a sequence of \( \epsilon \)-transitions, there is a !\( m_i \) transition followed by some other sequence of transitions.

Case 2a. If the check is successful, then we can infer that \( t_i \) is part of a loop and it contains independent transitions, which cause un-realizability.

Case 2a-i. Then we identify the first intermediate state \( P_i : t \) in this loop, which has an outgoing transition over some other output action. In this case, a new transition \( t_i \xrightarrow{m_{i}^{P_i \rightarrow P_i'}} t'_i \) with \( t'_i \) set to \( t \) is added to replicate the behavior in \( T^C_i \).
Case 2a-ii. If no such intermediate state exists, then \( t_i \xrightarrow{m_{i_1} \rightarrow P_i'} t_i' \) with \( t_i' \) set to \( t_i \) (self-loop) is added.

In either case, the permutations of pairs of independent transitions that were identified as the difference between \( C \) and \( I^C_1 \) are added and nothing else.

Case 2b. On the other hand, if the check is unsuccessful, then we can infer that \( t_i \) is not part of a loop.

Case 2b-i. We find out whether \( P_c : c_{i+1} \) (local state of the sender at \( s_{i+1} \)) has a path to \( P_t : t_i \) (\( t_i \) being the choreography state that cannot replicate the behavior of the system from \( s_i \)). If such a state exists in the behavior of \( P_t \), we infer that \( P_t \) moves along a path different from \( t_1, t_2, \ldots, t_i \) (see path 3) in choreography but the path has the ability to join at \( t_i \). In this case, we add a new transition labeled with \( t_i \xrightarrow{m_{i_1} \rightarrow P_i'} c_{i+1} \) to remove the difference between the choreography and corresponding the system.

Case 2b-ii. If the condition in Case 2b-i fails, then we find out the choreography state reachable from \( c_{i+1} \) (the choreography state corresponding the senders local state at \( s_{i+1} \)) via the action \( m_{i_1} \xrightarrow{P_i} P_{i-1}' \). If such a state is \( t \), then this implies that the choreography path extending from \( c_{i+1} \) allows \( m_{i_1} \xrightarrow{P_i} P_{i-1}' \) after \( m_{i_1} \xrightarrow{P_i} P_i' \), while the choreography path along \( t_1, t_2, \ldots, t_i \) (see path 3) does not allow \( m_{i_1} \xrightarrow{P_i} P_i' \) after \( m_{i_1} \xrightarrow{P_i} P_{i-1}' \). The repair in this case is similar to Case 1a and amounts to adding \( t_i \xrightarrow{m_{i_1} \rightarrow P_i'} t \). On the other hand, if no such choreography state \( t \) exists, then a new state is generated and a transition over \( m_{i_1} \xrightarrow{P_i} P_i' \) is added from \( t_i \) to this newly generated state.

Figure 5 illustrates the application of Case 2 of relaxation.

4.3 Repair by Restriction

The objective of restriction, unlike relaxation, is to constrain the behavior of the system \( I^C_1 \). In other words, going back to paths (3) and (2) in Sect. 4.1, restriction implies disallowing the transition \( s_i \xrightarrow{m_{i_1} \rightarrow P_i'} s_{i+1} \) in \( I^C_1 \) i.e., introducing restriction to disallow the transition \( c_i \xrightarrow{m_{i_1} \rightarrow P_i'} c_i' \) in \( C \) from happening at the system state \( s_i \), where \( 1St(s_i, P_t) = P_t : c_i \) and \( 1St(s_{i+1}, P_t) = P_t : c_{i+1} \). The restriction of transition \( c_i \xrightarrow{m_{i_1} \rightarrow P_i'} c_i' \) is achieved by adding a new intermediate state between \( c_i \) and \( c_i' \).

Case 1. Let \( t_i \) have a transition to \( t \) where some peer \( P \) sends a message \( m \) to \( P' \) and \( P \) is different from \( P_i \), the sender peer of the message \( m_i \). We verify whether the transition \( c_i \xrightarrow{m_{i_1} \rightarrow P_i'} c_{i+1} \) is reachable from \( t_i \).
If the verification is successful, this corresponds to the case of unrealizability due to independent transitions. The repair, in this case, results from the addition of an intermediate state between $t_i$ and $t$ such that $t_i \xrightarrow{m_i \rightarrow P_i'} m_i \rightarrow n_s \xrightarrow{nm_i \rightarrow P_i'} t$, where $nm$ is a new message and $ns$ is a new state. Addition of such transitions will disallow the $m_i \rightarrow P_i'$ at the system state $s_i$.

**Case 2.** However, if there is no transition from the state $t_i$ or the transition is labeled with a send action performed by the same peer $P_i$, then it corresponds to the case of unrealizability due to independent branches. In this case, we identify the sender peer $P_{i-1}$ for the transition from $t_{i-1}$ to $t_i$. The restriction is achieved by introducing an intermediate state between $c_i$ and $c_{i+1}$ as follows:

$c_i \xrightarrow{nm_{i-1} \rightarrow P_{i-1}} n_s \xrightarrow{m_{i-1} \rightarrow P_i} m_i \rightarrow P_i', \quad c_{i+1}$, where $nm$ and $ns$ are newly added message and newly added state, respectively.

These newly added messages and transitions in the choreography can be viewed as an extra step which forces the peer $P_i$ to come in sync with some other peer ($P'$ in Case 1a above and $P$ in Case 1b and 2 above) before sending the message $m_i$. We refer to such extra step as the synchronization step.

We will denote restriction of $C$ to generate $C'$ as $C \mid C'$. It is immediate that

$$C \mid C' = \mathcal{L}(C') \cap \mathcal{L}(C')' \subseteq \mathcal{L}(C')'$$

(4)

The operation $\mid C$ extracts the behavior with respect to actions present in $C$. The restriction does not alter the behavior of the choreography in terms of the actions in $C$ but restricts the behavior of the corresponding system in terms
Algorithm 1. Repair($C$, inputRepairMechanism)

1: Compute $I_C^1$
2: if $L(C) = L(I_C^1)$ return $C$\hspace{1cm}\triangleright\text{C is realizable}
3: Find a difference between $C$ and $I_C^1$\hspace{1cm}\triangleright\text{Sect. 4.1}
4: Apply $C$ inputRepairMechanism $C'$\hspace{1cm}\triangleright\text{Sects. 4.2, 4.3}
5: GOTO Line 1 with $C$ assigned to $C'$\hspace{1cm}\triangleright\text{Iterate}

of the actions in $C$. Figure 1(c) presents the result of applying restriction based repair of the choreography in Fig. 1(a). There exists a path in the system where it reaches the state $P_1 : s_1 : [mf], P_2 : s_3 : []$ via the send sequence $ms^{P_1\rightarrow P_2}, mf^{P_2\rightarrow P_1}$; from this state, the system is capable of producing $mc^{P_1\rightarrow P_2}$ (see Fig.3). The choreography via the same sequence of sends reaches the state $s_3$. Therefore, the restriction is achieved by following the Case 2 above resulting in a repaired choreography in Fig. 1(c).

4.4 Iterative Algorithm

It is necessary to apply the relaxation or the restriction iteratively till a realizable choreography is obtained and all differences between the choreography and the corresponding 1-bounded system behavior have been resolved. In Algorithm 1 the input parameter “inputRepairMechanism” is either set to $\nearrow$ (relaxation) or $\searrow$ (restriction). Figures 4 and 5 illustrate the application of Algorithm 1.

Theorem 2 (Correctness). The algorithm Repair is guaranteed to terminate and produce a repaired (i.e., realizable) choreography.

Proof Sketch. The algorithm iterates as long as there is a difference between the choreography $C$ and the interaction behavior of the corresponding system $I_C^1$. To address the difference, the algorithm introduces new states as part of the repair process. The number of such introduction of new states depends directly on the number of independent branches and independent transitions (that cause unrealizability of the choreography). The number of independencies are bounded by the number of branches and the maximum length of a path (with one unfolding) in the choreography, which ensures the boundedness in the introduction of new states. This, in turn, ensures that all possible causes of choreography un-realizability is removed within finite number of steps. $\square$

5 Case Studies

We have implemented Algorithm 1 and used it to repair several un-realizable choreographies that were reported earlier [7,17]. Our implementation obtains repaired versions of these un-realizable choreographies within a second.
Recall that the Singularity OS reservation contract (see Sect. 2) is unrealizable. Figure 6 presents a repaired version by adding new message exchanges. Another un-realizable contract is TpmContract (Fig. 7(a)). In Fig. 7(b), we show a repaired version that is automatically generated by our technique. The repaired version is similar to the one identified by authors in [9]; note however that [9] suggested an addition of a new state and two new transitions. Our repair mechanism achieves the same result by introducing one new transition between two existing states.

We have also analyzed the “Meta Conversation” protocol developed by IBM [12] and discussed in [7]. Two peers $P_1$ and $P_2$ race to decide the initiator of the interaction. The protocol is illustrated in Fig. 8(a). It is un-realizable because the peers can both send the start messages ($a\text{Start}_{cp}$ and $b\text{Start}_{cp}$) which is not allowed in the protocol. The restriction based solution (Fig. 8(b)) only allows peer $P_1$ to start the interaction.
Note that the repair only considers the transitions and their labels, and not their semantics. For instance, in Fig. 6, the added bold blue edges (relaxation) do not follow the semantics of the messages being exchanged. Consider the new path in the interaction, where “cancel” from client to server can be followed by “succeed” from the server to client. This is present in the repair in order to allow any ordering between “succeed” and “cancel” messages (as “succeed” followed by “cancel” is allowed in the original contract), which may not make sense in the context of the contract. Therefore, it is sometimes necessary to obtain certain application-domain specific information from the user such that relaxations can be guided appropriately. If the user had provided additional information that “cancel” can never be followed by “succeed”, then relaxation would have been impossible and the only choice for removing difference between the un-realizable choreography and the corresponding 1-bounded system will be restriction. We allow users to provide such domain knowledge in our implementation. We have also allowed user-interaction to decide on whether relaxation or restriction is preferred for repair. The user-interaction essentially involves examination of the difference (as presented by our tool) and deciding on the choice between relaxation and restriction. Figure 9 presents an alternative solution for repairing the contract in Fig. 6 generated by our tool. Observe that in this solution, a combination of relaxation and restriction has been applied.

6 Related Work

Realizability of choreographies has been studied before. The authors in [7, 9] use state machine based specifications while the authors in [6, 10] use session types; both present sufficient conditions for realizability. In [4], we have proved the decidability of choreography realizability in terms of send sequences by presenting a necessary and sufficient condition for realizability.

1 Note that, the realizability problem for the MSC-graphs, which considers both send and receive actions for realizability, is undecidable [1].
In [15], the realizability of choreography requires the developer to specify a “dominant” process for each branch and loop construct, which allows the projection mechanism to synthesize necessary synchronization messages between the dominant process and others. Similarly, techniques proposed in [3,8,14,19] rely on introducing new processes, monitors and central controllers to ensure realizability. These may not be viable options if one is using a distributed computing paradigm. Moreover these techniques can be conservative in the sense that unnecessary synchronization messages can be added to even realizable choreographies. Furthermore, the focus of these works is technically different from that of our— for instance, the technique in [3] coordinates the activities of the peers in a distributed fashion such that their coordinated behavior conforms to the given choreography. The repair technique developed by authors in [13] focuses on process algebraic description of choreographies and repair by restriction in the context of independent sequences (referred to as connected choreography by the authors); additionally, the description does not take into consideration iterations, which makes the technique inapplicable to choreographies with cycles.

In contrast, our work (which includes both relaxation and restriction mechanisms) does not require introduction of new processes, does not require a central controller, and does not require use of synchronous communication between any entities/peers. As our technique is based on finite state machines and their language equivalence, it is applicable to choreographies and interactions which are specified at different levels of abstractions, such as session-types [10] and collaboration diagrams [5], as long as these specifications are translated to state-machine based representation described in [4] and used in this paper.
7 Conclusion

We present techniques for automatically repairing un-realizable choreographies based on two strategies: (1) relaxation, where new behaviors are added to the choreography as part of the repair and (2) restriction, where un-desired (excluded by the choreography) behaviors in the system obtained by projecting the choreography are removed as part of the repair. We prove that our repair algorithm always terminates with a realizable choreography. To the best of our knowledge, our method is the first to consider automatically repairing choreographies and to provide formal guarantees of correctness.

References


Fundamental Approaches to Software Engineering
19th International Conference, FASE 2016, Held as Part
of the European Joint Conferences on Theory and
Practice of Software, ETAPS 2016, Eindhoven, The
Netherlands, April 2-8, 2016, Proceedings
Stevens, P.; Wąsowski, A. (Eds.)
2016, XIII, 436 p. 127 illus., Softcover
ISBN: 978-3-662-49664-0