Deciding Choreography Realizability

Samik Basu
Iowa State University
Email: sbasu@iastate.edu

Tevfik Bultan
University of California, Santa Barbara
Email: bultan@cs.ucsb.edu

Meriem Ouederni
University of Malaga
Email: meriem@lcc.uma.es

Abstract

Since software systems are becoming increasingly more concurrent and distributed, modeling and analysis of interactions among their components is a crucial problem. In several application domains, message-based communication is used as the interaction mechanism, and the communication contract among the components of the system is specified semantically as a state machine. In the service-oriented computing domain such communication contracts are called "choreography" specifications. A choreography specification identifies allowable ordering of message exchanges in a distributed system. A fundamental question about a choreography specification is determining its realizability, i.e., given a choreography specification, is it possible to build a distributed system that communicates exactly as the choreography specifies? Checking realizability of choreography specifications has been an open problem for several years and it was not known if this was a decidable problem. In this paper we give necessary and sufficient conditions for realizability of choreographies. We implemented the proposed realizability check and our experiments show that it can efficiently determine the realizability of 1) web service choreographies, 2) Singularity OS channel contracts, and 3) UML collaboration (communication) diagrams.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: [Formal Methods]

General Terms Verification

Keywords Message-based Interactions, Choreography, Realizability

1. Introduction

Most software systems nowadays involve concurrent or distributed behavior or both. They run concurrently on multi-core hardware, interact with each other over the network and access data and computational resources distributed over the compute cloud. An important concern in construction of concurrent and distributed software systems is the coordination of different components that form the whole system. In order to complete a task, components of a software system have to coordinate their executions by interacting with each other, and specification and analysis of such interactions is a challenging problem.

Message-based communication is a common interaction mechanism used in concurrent and distributed systems where components interact with each other by sending and receiving messages. Specification and analysis of message-based interactions has been an active research area studied in several application domains including coordination in service-oriented computing [9, 37], interactions in distributed programs [2] and process isolation at the OS level [12]. A crucial problem in all these domains is the choreography realizability problem. A choreography specification identifies the set of allowable message exchange sequences among the components (peers) of a distributed system. A choreography is realizable if there is a way to implement a set of components that conform to the choreography.

Many message-passing systems use asynchronous messaging [4, 23, 24, 28, 30] where components interact with each other by sending and receiving messages over unbounded FIFO channels. Even when the behavior of each component is modeled as a finite state machine, if asynchronous communication is used, the state space of the overall system is infinite. In fact, finite state systems communicating with unbounded FIFO communication channels are powerful enough to simulate Turing machines, and, hence, many verification and analysis problems for them are undecidable [6]. Determining realizability of choreography specifications for asynchronously communicating systems has been an open problem for several years and it was not known if it is decidable.

More precisely, the choreography realizability problem is, given a choreography specified as a finite state machine, is it possible to determine if there exists a set of asynchronously communicating (finite) state machines that generate precisely the set of message sequences specified by the choreography specification. Note that, given a set of asynchronously communicating (finite) state machines, it is not possible to automatically determine the set of message sequences generated by them. However, in this paper we show that the realizability of a choreography specification is decidable, and we give a necessary and sufficient condition for determining realizability. There have been earlier results in this area that provide sufficient conditions for choreography realizability (e.g., [14, 18, 25]). To the best of our knowledge this is the first paper that identifies a necessary and sufficient condition and demonstrates the decidability of the choreography realizability problem. We have also experimentally evaluated our approach by checking the realizability of Singularity channel contracts [12], web service choreographies [37] and collaboration diagrams [7].

Rest of the paper is organized as follows. In Section 2, to motivate our work, we discuss how the realizability problem arises in different domains. We also give a high level overview of the proposed approach. In Section 3 we formally define the realizability problem by formalizing the communication contracts as finite state conversation protocols and a distributed system as a set of finite state peers communicating via messages over FIFO message queues. In Section 4 we present our main results on realizability. In Section 5 we discuss our implementation using the CADP toolbox [17]. In Section 6 we discuss the related work and in Section 7 we conclude the paper.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

```
...
<interaction name="SendOffer"
  operation="offer"
  channelVariable="tns:R1ToR2C">

  <participate
    relationshipType="tns:Hagglers"
    fromRoleTypeDef="tns:R1"
    toRoleType="tns:R2/>

    <exchange name=... action="sendOffer">
      <send var="cdl:getVar("offer")/>
    </exchange>
...
</interaction>
```

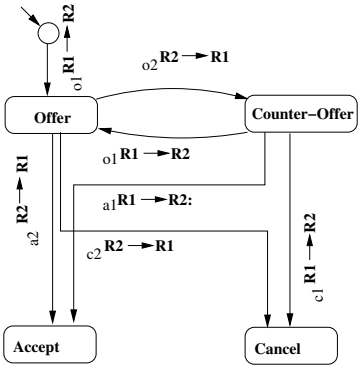

Figure 1. Part of the WS-CDL choreography specification for a bilateral negotiation protocol [22] and the corresponding state machine.

2. Motivation and Overview

In this section we motivate our work by describing four application domains where the choreography realizability problem appears and, therefore, the results from this paper are directly applicable.

2.1 Service Interactions: Choreography Specifications

Service oriented computing provides technologies that enable multiple organizations to integrate their businesses over the Internet [13, 31]. Typical execution behavior in such a distributed system involves a set of autonomous peers[1] interacting with each other through messages. For instance, consider a group of organizations that wish to integrate their online businesses. The front end that enables user interaction via web browsers may reside within one organization, however, in order to serve a user request, this front end may send and receive messages from software components that reside in other organizations. The goal of web services standards and technologies is to facilitate this type of business to business integration. Modeling and analysis of interactions is a crucial problem in this domain due to several distinguishing features of such systems.

First, organizations may not want to share the internal structures of their software components with other organizations they intend to do business with. This type of decoupling requires standardized and rich interface specifications that enable integration of software components that may be written using different languages and implementation platforms. In order to achieve such a decoupling among different components, it is necessary to specify the interactions among different components without referring to the details of their local implementations.

[1] We use "peer" to denote a process/component/program/service that interacts with other processes/components/programs/services.

Second, modeling and analyzing the global behavior of such distributed systems is challenging since no single party may know the full details of all the components in such a system. In the absence of detailed models for the distributed components that participate in such a system, the desired global behaviors have to be specified as constraints on the interactions among different components, since the messages exchanged among different components are the only observable global behavior. Moreover, for this type of distributed systems, it might be worthwhile to model the interactions among different software components before the components are written. This type of top-down design may help different organizations to better coordinate their development efforts.

Choreography specification languages target specification of this type of interactions. For example, Web Services Choreography Description Language (WS-CDL) [37] is an XML-based language for describing the interactions among the peers participating in a composite web service. A choreography specification in WS-CDL corresponds to a global ordering of the message-exchange events among the peers participating in a composite service, i.e., a choreography specification identifies the set of allowable message sequences for a composite web service. Figure 1 presents a snapshot of a WS-CDL specification for two services (behaving as `R1`, i.e., Role1 and `R2`, i.e., Role2) participating in a "haggling" process where each service continues sending offers to the other until one of them accepts the offer or cancels the process. The specification contains information regarding the roles, description of every action in terms of the sender, receiver and the message content and type, and the ordering (branching and sequencing) of actions.

2.2 Interactions Among Concurrent Processes: Singularity Channel Contracts

Singularity is a new, experimental, operating system developed by Microsoft Research to explore new approaches to OS design [21]. One of its main goals is to improve the dependability of software systems by rethinking some design decisions that have largely governed operating system architecture to date. Process isolation is a chief design principle of the Singularity operating system. To achieve this, certain constraints are enforced to ensure process independence. Among these is the rule that processes cannot share memory with each other or the kernel. All inter-process communication in Singularity, therefore, occurs via message-passing over bidirectional conduits, called channels.

Channels have two end points referred to as the client and the server. The client and the server processes use the channel to communicate with each other by sending and receiving messages. Communication through Singularity channels corresponds to asynchronous communication via FIFO queues.

In Singularity, each channel is governed by a *channel contract* [12, 33]. A channel contract is basically a state machine that specifies the allowable ordering of messages between the client and the server. Hence, channel contracts serve the same purpose that choreography specifications serve in service-oriented computing.

Singularity processes are written in an extension of C# called Sing#. It provides constructs for writing channel contracts and its compiler statically checks that the processes communicating over a channel conform to its contract. Figure 2 presents a simplified version of a Sing# contract governing a channel used by Singularity for interacting with a keyboard device. This contract defines three states (`Start`, `Ready`, and `Waiting`) and the evolution among states (`->`) correspond to message exchanges. Singularity contracts are written from the perspective of the server, where send actions by the server are appended with `!` to denote communication from the server to the client and receive actions by the server are appended with `?` to denote communication from the client to the server.

```
public contract KeyboardDeviceContract {
  state Start: {
    Success! -> Ready;
  }
  state Ready: {
    GetKey? -> Waiting;
    PollKey? -> (AckKey! or NakKey!) -> Ready;
  }
  state Waiting: {
    AckKey! -> Ready;
    NakKey! -> Ready;
  }
}
```

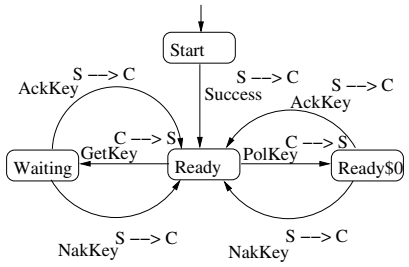

Figure 2. A simplified channel contract from the Singularity OS for keyboard interaction written in Sing# [33] and the corresponding state machine.

```
+NAME("IRC SERVER")
...
+STATE start
  logon()    => ok() & active
             |  error() & stop
+STATE active
  ls()       => files() & active
  getFile()  => fileSent() & active
             | noFileErr() & stop
...
```

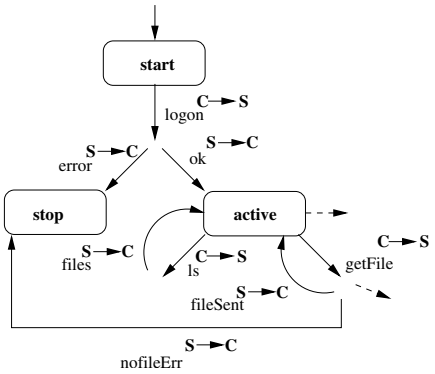

Figure 3. Part of a communication contract for a distributed Erlang program written in UBF(B) [2] and the corresponding state machine.

2.3 Interactions Among Distributed Components: UBF(B) Contracts

Erlang is a general purpose concurrent programming language that was developed initially at Ericsson for improving the dependability of telephony applications [3]. In Erlang, processes do not share memory and only interact with each other via exchanging messages asynchronously.

UBF(B) is a language for specifying *communication contracts* in distributed Erlang programs [2]. Figure 3 presents partial specification of a IRC server interface specification and describes the evolution of the server from one state (`start`, `active`, etc.) to another in response to external stimuli (`logon`, `ls`, etc.). UBF(B) contracts are based on finite state machines. Given a state (e.g., `start`), a transition from that state identifies a request-response sequence where, after receiving a message (e.g., `logon`), the process sends a response (e.g., `ok`) and moves to the destination state (e.g., `active`).

2.4 Visual Interaction Specifications: Collaboration Diagrams

Collaboration diagrams (called communication diagrams in [36]) provide a convenient visual formalism for specifying the interactions among the participants of a distributed system [7, 8]. A *collaboration diagram* is a visual representation of a set of peers, a set of communication links among them, and an ordering of the message exchanges among the peers. Unlike MSCs [29], collaboration diagrams specify the global ordering of send events rather than the local ordering of send and receive events. The semantics of collaboration diagrams can be formalized as a state machine characterizing all allowable ordering of message exchanges in a distributed system [8]. Hence, for example, collaboration diagrams can be used as a visual formalism for representing web service choreographies.

2.5 Overview of the Proposed Approach

Web service choreography specifications, Singularity channel contracts, UBF(B) contracts and collaboration diagrams are all different mechanisms for specifying ordering of messages exchanged among a set of concurrent or distributed processes. Analysis of message-based interactions is an essential problem for all these languages. And, although these languages target different application domains, the interaction analysis problem remains the same. In short (a) communication contract specifications cut across a wide range of application domains (service-oriented computing, new paradigms for OS, embedded systems); (b) communication contracts are used to specify message-based interactions among peers (services, client/servers, real-time software); and, finally, (c) one key problem for communication contracts used in each of these application domains is to check whether a given communication contract can be *realized*.

In this paper we present a necessary and sufficient condition for realizability and show how it can be applied to multiple domains. In our approach, we first translate communication contracts specified in different languages into a *conversation protocol* which is a finite state machine that specifies the allowable sequences of messages (i.e., conversations) among a set of peers. For instance, the state machine shown in Figure 1 (o_i: offer, a_i: acceptance and c_i: cancellation sent by R_i) is the conversation protocol for the corresponding WS-CDL choreography specification. We label the transitions of the conversation protocols with *Sender* $\rightarrow$ *Receiver*:*Msg*, denoting the sending of message *Msg* from the *Sender* peer to the *Receiver* peer. The state machines shown in Figures 2 and 3 are the conversation protocols for the corresponding Sing# channel contract and the UBF(B) communication contract specifications, respectively. In the following section we give a formal definition of conversation protocols.

In order to analyze realizability of conversation protocols, we also need a formal model of distributed systems that interact with messages. We model such systems as a set of peers, where the behavior of each peer is specified as a finite state machine. We assume that each peer has a FIFO message queue that stores messages sent to it by the other peers. When a peer executes a send transition, the sent message is appended to the message queue of the receiving peer. A peer can only execute a receive transition if the transition

matches the message at the head of its receive queue. When a peer executes a receive transition, the message at the head of its receive queue is removed. The behavior of the overall system is defined by interleaving the executions of peers.

A conversation protocol is said to be realizable if and only if there exists a set of peers whose interactions conform to the contract. Note that, by interactions we mean send sequences (i.e., conversations); the receive actions occur locally when a peer consumes a message from its own message-queue. This is in contrast to the earlier work on realizability of MSCs (for example in [1, 35]) where both the send and receive actions are considered. The realizability problem MSC-graphs is undecidable. It is not immediately clear if ignoring the receive actions simplifies the realizability problem, since the basic formal model we are looking at involves peers that are interacting in an asynchronous fashion, and each peer is assumed to have a receive-queue of unbounded size. As a result, the state space of the global system consisting of multiple peers may be infinite.

The **main result** we present in this paper is that choreography realizability problem is decidable which has been an open problem for several years. We provide a necessary and sufficient condition for realizability which states that a choreography specification is realizable if and only if

- its behavior is language equivalent to the 1-bounded system of asynchronously communicating peers, where each peer behavior is obtained from the projection of the choreography (on each peer) and where each peer has a message queue of size 1 (equivalence condition); and
- the 1-bounded system satisfies a specific temporal property (well-formedness condition).

Since the choreography specification and the 1-bounded system both have finite state spaces, we were able to implement an automated choreography realizability checker using existing equivalence checking and model checking tools.

3. Conversations & Realizability

We formalize choreography specifications using conversations and conversation protocols (Section 3.1), define distributed systems with asynchronous communication (Section 3.2), and present different variations of choreography realizability (Section 3.3).

3.1 Conversations

We use state machines to characterize conversation protocols, peer behaviors and distributed systems that consist of asynchronously communicating peers [14].

DEFINITION 1 (Conversation Protocol). *A conversation protocol is represented by $\mathcal{C} = (\mathcal{P}, S^C, s_0^C, L, \Delta^c)$ 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 $(s_i^C, P, m, P', s_j^C) \in \Delta^c$ represents the sending of message m from P to P' ($P, P' \in \mathcal{P}$).*

Figures 1, 2 and 3 present the communication contracts from three different domains and the corresponding conversation protocols. The start states are denoted by an incoming arrow without a source state. Each transition is labeled with a message along with the sender and the receiver of the message. We denote the transition labels as $m^{P \to P'}$, where m is the message being sent by P to P'.

3.2 Systems

DEFINITION 2 (Peer Behavior). *The behavior $\mathcal{B}$ of a peer P is a finite state machine (M, T, t_0, δ) 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 ϵ-transition of the form (t_1, ϵ, t_2). We write $t \xrightarrow{a} t'$ to denote that $(t, a, t') \in \delta$.

Figure 4 illustrates the behaviors of two communicating peers P_1 and P_2 with send and receive actions a, b and c.

DEFINITION 3 (System Behavior). *Given a set of peers $\mathcal{P} = \{P_1, \ldots, P_n\}$ with $\mathcal{B}_i = (M_i, T_i, t_{0i}, \delta_i)$ denoting the behavior of P_i and $M_i = M_i^{in} \cup M_i^{out}$ such that*

- $\forall i : M_i^{in} \cap M_i^{out} = \emptyset$,
- $\forall i, j : i \neq j \Rightarrow M_i^{in} \cap M_j^{in} = M_i^{out} \cap M_j^{out} = \emptyset$,

a system behavior or simply a system over $\mathcal{P}$ is denoted by a state machine (possibly infinite state) $\mathcal{I} = (\mathcal{P}, S, s_0, M, \Delta)$ where

1. $M = \cup_i M_i$
2. $S \subseteq \mathcal{Q}_1 \times T_1 \times \mathcal{Q}_2 \times T_2 \ldots \mathcal{Q}_n \times T_n$ *such that* $\forall i \in [1..n] : \mathcal{Q}_i \subseteq (M_i^{in})^*$
3. $s_0 \in S$ *such that* $s_0 = (\epsilon, t_{01}, \epsilon, t_{02} \ldots, \epsilon, t_{0n})$*; and*
4. $\Delta \subseteq S \times [(\mathcal{P} \times M \times \mathcal{P}) \cup \{\epsilon\}] \times S$*, and for* $s = (Q_1, t_1, Q_2, t_2, \ldots Q_n, t_n) \in S$ *and* $s' = (Q_1', t_1', Q_2', t_2', \ldots Q_n', t_n') \in S$
   (a) $s \xrightarrow{m_{P_i \to P_j}} s' \in \Delta$ *if* $\exists i, j \in [1..n] : m \in M_i^{out} \cap M_j^{in}$, *(i)* $t_i \xrightarrow{!m} t_i' \in \delta_i$, *(ii)* $Q_j' = Q_j m$, *(iii)* $\forall k \in [1..n] : k \neq j \Rightarrow Q_k = Q_k'$ *and (iv)* $\forall k \in [1..n] : k \neq i \Rightarrow t_k' = t_k$ **[send action]**
   (b) $s \xrightarrow{\epsilon} s' \in \Delta$ *if* $\exists i \in [1..n] : m \in M_i^{in}$ *(i)* $t_i \xrightarrow{?m} t_i' \in \delta_i$, *(ii)* $Q_i = mQ_i'$, *(iii)* $\forall k \in [1..n] : k \neq i \Rightarrow Q_k = Q_k'$ *and (iv)* $\forall k \in [1..n] : k \neq i \Rightarrow t_k' = t_k$ **[receive action]**
   (c) $s \xrightarrow{\epsilon} s' \in \Delta$ *if (i)* $\exists i \in [1..n] : t_i \xrightarrow{\epsilon} t_i' \in \delta_i$, *(ii)* $\forall k \in [1..n] : Q_k = Q_k'$ *and (iii)* $\forall k \in [1..n] : k \neq i \Rightarrow t_k' = t_k$ **[internal action]**

The above definition states that peers in the system communicate in an asynchronous fashion. Each peer has an unbounded message queue (Q_i) and a message sent to a peer gets inserted to the tail of the queue, while a message consumed by a peer is consumed from the head of its message queue.

Note that, send actions involve two peers, the peer sending the message and the message queue of the receiver peer; on the other hand, the receive action is local and involves only the receiver peer. The behavior of the system depends on the order the send and receive actions as well as the size of the message queues associated with each peer participating in the system. In the following, we define k-bounded systems, where each participating peer has a message queue of size k. The send actions in such a system is blocked if the receiver peer's message queue is full (i.e., contains k pending messages to be consumed).

DEFINITION 4 (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. The description of k-bounded system behavior is, therefore, realized by augmenting condition 4(a) in Definition 3 to include the condition $|Q_j| < k$, where $|Q_j|$ denotes the length of the queue for peer j.*

Figure 4 illustrates the behavior of $\mathcal{I}_1$ obtained from the two peers P_1 and P_2 with asynchronous communication. For brevity, we only show the transitions that involve send actions.

We also define synchronous behavior of a system where every send action by a peer is consumed immediately by a receiver peer, i.e., the peers interact synchronously. This can be viewed as the case where the peers do not have any message queues.

DEFINITION 5 (Synchronous Behavior). *Given a set of peers $\mathcal{P} = \{P_1, \ldots, P_n\}$ with $\mathcal{B}_i = (M_i, T_i, t_{0i}, \delta_i)$ denoting the behavior of P_i and $M_i = M_i^{in} \cup M_i^{out}$, such that*

- $\forall i : M_i^{in} \cap M_i^{out} = \emptyset$,
- $\forall i, j : i \neq j \Rightarrow M_i^{in} \cap M_j^{in} = M_i^{out} \cap M_j^{out} = \emptyset$,

the synchronous system behavior containing the peers in $\mathcal{P}$ is denoted by a state machine $\mathcal{I}_0 = (\mathcal{P}, S, s_0, M, \Delta)$ where

1. $M = \cup_i M_i$
2. $S \subseteq T_1 \times T_2 \times \ldots \times T_n$
3. $s_0 \in S$ *such that* $s_0 = (t_{01}, t_{02} \ldots, t_{0n})$*; and*
4. $\Delta \subseteq S \times [(\mathcal{P} \times M \times \mathcal{P}) \cup \{\epsilon\}] \times S$ *and for* $s = (t_1, t_2, \ldots, t_n) \in S$ *and* $s' = (t_1', t_2', \ldots, t_n') \in S$
   (a) $s \xrightarrow{m_{P_i \rightarrow P_j}} s' \in \Delta$ *if* $\exists i, j \in [1..n] : m \in M_i^{out} \cap M_j^{in}$*, (i)* $t_i \xrightarrow{!m} t_i' \in \delta_i$*, (ii)* $t_j \xrightarrow{?m} t_j' \in \delta_j$*, (iii)* $\forall k \in [1..n] : k \neq i \wedge k \neq j \Rightarrow t_k' = t_k$
   [**synchronous send-receive action**]
   (b) $s \xrightarrow{\epsilon} s' \in \Delta$ *if* $\exists i \in [1..n]$ *(i)* $t_i \xrightarrow{\epsilon} t_i' \in \delta_i$*, (ii)* $\forall k \in [1..n] : k \neq i \Rightarrow t_k' = t_k$
   [**internal action**]

The synchronous composition of behaviors $\mathcal{B}_1$ and $\mathcal{B}_2$ of peers P_1 and P_2, respectively, in Figure 4 would have a structure that is identical to $\mathcal{B}_2$, with one branch having transitions $a^{P_2 \rightarrow P_1}$, followed by $b^{P_1 \rightarrow P_2}$, followed by $c^{P_1 \rightarrow P_2}$; and the other branch having transitions $a^{P_2 \rightarrow P_1}$, followed by $c^{P_1 \rightarrow P_2}$, followed by $b^{P_1 \rightarrow P_2}$.

PROPOSITION 1. *The synchronous system behavior containing a set of peers $\mathcal{P} = \{P_1, \ldots, P_n\}$ with peer behaviors $\mathcal{B}_j$ ($1 \leq j \leq n$) where each peer behavior $\mathcal{B}_j$ ($1 \leq j \leq n$) is deterministic, is also deterministic, i.e., the labels on any pair of outgoing transitions from a state are distinct. Any peer behavior with finite state-space can be determinized and a system $\mathcal{I}$ (resp. $\mathcal{I}_k$, $k \geq 0$) obtained by determinizing the peer behaviors is denoted by* DETER($\mathcal{I}$) *(resp.* DETER($\mathcal{I}_k$)*).*

Finally, we define the concept of well-formed systems.

DEFINITION 6 (Well-formed System). *A system containing a set of peers $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ with peer behaviors $\mathcal{B}_j$ ($1 \leq j \leq n$) is said to be well-formed if and only if every message sent by any peer can be eventually consumed along some path in the system by the receiver of the message. This can be expressed precisely in temporal logic CTL [10] as*

$$AG(|Q_i| > 0 \;\Rightarrow\; EF(|Q_i| = 0)) \tag{1}$$

The property states that whenever the size of the receive queue, Q_i, of the i-th peer is greater than 0 (i.e., Q_i is non-empty), the system can eventually *move to a state where Q_i is empty.*

All synchronous systems, $\mathcal{I}_0$, are well-formed by definition. For all k-bounded asynchronous systems, it can be automatically verified (via model checking) whether the system is well-formed or not. Any k-bounded system has finite state-space; one can model check such a system against the CTL property (Equation 1).

If a system $\mathcal{I}_k$ is well-formed, we say that WF($\mathcal{I}_k$). Note that, well-formedness checking for asynchronous systems (where message queues are unbounded) is undecidable in general.

3.3 Realizability

We consider two variations of realizability. We refer to these variations as Realizability$_\forall$ and Realizability$_\exists$.

DEFINITION 7 (Realizability). *A conversation protocol $\mathcal{C}$ defined over a set of peers $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ is said to be realizable according to Realizability$_\forall$ or Realizability$_\exists$, if and only if there exists some peer behaviors $\mathcal{B}_j$ ($1 \leq j \leq n$) and a system $\mathcal{I}$ defined using these behaviors, such that*

Realizability$_\forall$: for all $i \geq 0$, $\mathcal{C}$ is equivalent to $\mathcal{I}_i$ and WF($\mathcal{I}_i$)*, or*
Realizability$_\exists$: $\mathcal{C}$ is equivalent to $\mathcal{I}$ and WF($\mathcal{I}$)

respectively.

Realizability$_\forall$ requires the existence of a system such that its behaviors *for all* possible receive queue sizes are equivalent to $\mathcal{C}$. Realizability$_\exists$ requires the existence of a system such that its behavior is equivalent to $\mathcal{C}$ when unbounded receive queues are used.

Note that, the above variations, Realizability$_\forall$ and Realizability$_\exists$, require the *equivalence* between the conversation $\mathcal{C}$, and the system(s) $\mathcal{I}_i$s and $\mathcal{I}$, respectively. We consider language equivalence, which ensures that any linear temporal logic property satisfied by the conversation protocol is also satisfied by the system that realizes the conversation. Earlier work [14, 18] on choreography realizability has focused on language equivalence and Realizability$_\exists$, where sufficient conditions for determining Realizability$_\exists$ have been provided.

It may appear that Realizability$_\forall$ is a stronger notion compared to Realizability$_\exists$ in the sense that if $\mathcal{C}$ is realizable according to Realizability$_\forall$ then it is also realizable according to Realizability$_\exists$. However, we show in this paper that Realizability$_\forall$ and Realizability$_\exists$ are equivalent.

3.3.1 Language Equivalence & Preorder

Language Equivalence. For a conversation protocol, the alphabet is $\Sigma = \cup\{m^{P \rightarrow P'}\}$, where P, P' are peers in the conversation and $m^{P \rightarrow P'}$ is a send action (Definition 1). We denote the language of a conversation $\mathcal{C}$ as $\mathcal{L}(\mathcal{C})$, which contains any sequence over Σ from the start state. For example, the language of the conversation in Figure 1 includes a sequence where $o_1^{R1 \rightarrow R2}$ is eventually followed by $a_2^{R2 \rightarrow R1}$ and in between there are finite number of subsequences of the form $o_2^{R2 \rightarrow R1} o_1^{R2 \rightarrow R1}$.

For a peer behavior, the alphabet is $\Sigma = M^{\text{in}} \cup M^{\text{out}}$, where M^{in} and M^{out} are receive and send actions of the peer respectively (Definition 2). We denote the language of a behavior $\mathcal{B}$ of a peer P as $\mathcal{L}(\mathcal{B})$. For example, in Figure 4, the behavior $\mathcal{B}_1$ of P_1 includes the sequence $?a!b!c$ in its language.

For a system, the alphabet is $\Sigma = \cup\{m^{P \rightarrow P'}\}$, where P, P' are peers participating in the system and $m^{P \rightarrow P'}$ is a send action in the behavior of peer P (Definition 3). We denote the language of a system $\mathcal{I}$ (resp. $\mathcal{I}_k$) as $\mathcal{L}(\mathcal{I})$ (resp. $\mathcal{L}(\mathcal{I}_k)$). For example, the language of the system $\mathcal{I}_1$ in Figure 4 includes the sequence $a^{P_2 \rightarrow P_1} b^{P_1 \rightarrow P_2} c^{P_1 \rightarrow P_2}$.

Based on the Definition 7, language realizability requires that $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{I}_i)$ for all $i \geq 0$ for Realizability$_\forall$ and $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{I})$ for Realizability$_\exists$.

Ordering. We also require the concept of an ordering between systems, conversations and peers in terms of their language. Ordering with respect to language can be easily obtained using the subset relation.

4. Deciding Realizability

In this section, we prove that determining realizability is decidable, present the necessary and sufficient condition for checking realizability of a given conversation protocol and show that this condition can be efficiently computed using existing techniques for equivalence checking and model checking. We prove that a conversation protocol $\mathcal{C}$ is realizable if and only if it is realized by a well-formed system obtained from peer projections of $\mathcal{C}$. We define the peer projection of a conversation protocol as follows.

DEFINITION 8 (Peer Projection). *The projection of a conversation protocol $\mathcal{C}$ on one of the peers P, participating in the conversation, is denoted by $\mathcal{C}_{\downarrow P}$ and is obtained from $\mathcal{C}$ by performing the following updates to the state machine describing $\mathcal{C}$.*

- *if a transition label is $m^{P \rightarrow P'}$ then replace it with $!m$,*
- *if a transition label is $m^{P' \rightarrow P}$ then replace it with $?m$,*
- *otherwise, replace transition label with ϵ.*

We denote the synchronous, k-bounded asynchronous and unbounded asynchronous systems obtained from the peer projections of $\mathcal{C}$ by $\mathcal{I}_0^{\mathcal{C}}, \mathcal{I}_k^{\mathcal{C}}$ and $\mathcal{I}^{\mathcal{C}}$, respectively. For example, Figure 5 presents a conversation protocol $\mathcal{C}$ and its projections to peers P_1 and P_2.

Next, we proceed by first considering Realizability$_\forall$ (Section 4.1) followed by Realizability$_\exists$ (Section 4.2), and finally summarize our findings in Section 4.3.

4.1 Deciding Realizability$_\forall$

In this section, by realizability, we mean Realizability$_\forall$, unless otherwise mentioned. The following outlines the steps of the proof establishing the decidability of realizability.

Outline.

1. *Behavioral Ordering.* We prove that if the conversation protocol $\mathcal{C}$ is equivalent to DETER($\mathcal{I}_0^{\mathcal{C}}$), then DETER($\mathcal{I}_0^{\mathcal{C}}$) is the "*smallest*" system with synchronously communicating peers that is language equivalent to $\mathcal{C}$. [Theorem 1].
2. *Synchronizability Checking.* The conversation protocol $\mathcal{C}$ is realizable according to Realizability$_\forall$ if and only if it is equivalent to $\mathcal{I}_i$ for all $i \geq 0$, and $\mathcal{I}_i$s are well-formed (see Definition 7). That is, $\mathcal{C}$ is equivalent to $\mathcal{I}_i$ irrespective of the size i of the receive queues. We use the concept of synchronizability of a system. A system is synchronizable if and only if the system behavior (over send actions) remains unaltered for any receive queue size. We show that $\forall i \geq 0$, $\mathcal{I}_i$s are equivalent to $\mathcal{I}$, i.e., $\mathcal{I}$ is synchronizable, if and only if $\mathcal{I}_0$ is equivalent to $\mathcal{I}_1$. [Theorem 2].
3. *Well-formedness Checking.* Realizability also requires that the system that realizes the conversation must be well-formed. We present the conditions when the synchronizable systems are well-formed; specifically, we prove that a synchronizable system $\mathcal{I}$ is well-formed if and only if $\mathcal{I}_1$ is well-formed [Theorem 3]. Finally, we show that deterministic synchronizable systems are always well-formed [Theorem 4].
4. *Realizability$_\forall$ Checking.* Finally, using the above theorems, we obtain that the conversation $\mathcal{C}$ is realizable if and only if DETER($\mathcal{I}^{\mathcal{C}}$) is synchronizable and well-formed, and $\mathcal{C}$ is equivalent to DETER($\mathcal{I}^{\mathcal{C}}$). This statement holds if and only if $\mathcal{C}$ is equivalent to DETER($\mathcal{I}_1^{\mathcal{C}}$). (Recall that DETER($\mathcal{I}^{\mathcal{C}}$) denotes the system over peer behaviors obtained by determinizing the projection of $\mathcal{C}$ over peers). As both, $\mathcal{C}$ and $\mathcal{I}_1^{\mathcal{C}}$ are finite state systems, verification of equivalence can be done effectively. [Theorem 5]

4.1.1 Language-based Ordering

For the step 1 noted above, we discuss certain ordering properties of the systems, conversations and peer behaviors, to qualify what we mean by the smallest system with synchronously communicating peers.

PROPOSITION 2.

$$\forall \mathcal{C} : \mathcal{L}(\mathcal{C}) \subseteq \mathcal{L}(\mathcal{I}_0^{\mathcal{C}}) \quad \textit{and} \quad \forall i \geq 0 : \mathcal{L}(\mathcal{I}_i) \subseteq \mathcal{L}(\mathcal{I}_{i+1})$$

Proof: The proof follows from the following observations. The peers in $\mathcal{I}_0^{\mathcal{C}}$ are obtained from projections of $\mathcal{C}$. Therefore, any path (in terms of send actions) from any state in $\mathcal{C}$ is also present in $\mathcal{I}_0^{\mathcal{C}}$. Any $\mathcal{I}_{i+1}$ can replicate the behavior (in terms of send actions) of $\mathcal{I}_i$ by avoiding the paths that occur due to the usage of message queues of length $i+1$. □

DEFINITION 9. *Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of peers. Let $\mathcal{I}_0$ be a synchronous system obtained from the peer-behaviors $\mathcal{B}_i, 1 \leq i \leq n$; and $\mathcal{I}_0'$ be a synchronous system obtained from the peer-behaviors $\mathcal{B}_i', 1 \leq i \leq n$. We say that $\mathcal{I}_0 \leq_{\mathcal{L}} \mathcal{I}_0'$ if and only if $\forall i \geq 1, i \leq n : \mathcal{L}(\mathcal{B}_i) \subseteq \mathcal{L}(\mathcal{B}_i')$.*

In the above, we define the ordering relations $\leq_{\mathcal{L}}$ between systems based on the ordering between the behaviors of the corresponding peers that constitute the respective systems.

PROPOSITION 3. *Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of peers. Let $\mathcal{I}_0$, $\mathcal{I}_k$, $\mathcal{I}$ denote the synchronous, k-bounded asynchronous and unbounded asynchronous systems, where the i-th peer-behavior is $\mathcal{B}_i$ for $1 \leq i \leq n$; and $\mathcal{I}_0'$, $\mathcal{I}_k'$, $\mathcal{I}'$ denote the corresponding systems, where the i-th peer-behavior is $\mathcal{B}_i'$ for $1 \leq i \leq n$.*

$$\mathcal{I}_0 \leq_{\mathcal{L}} \mathcal{I}_0' \Rightarrow [\mathcal{L}(\mathcal{I}_0) \subseteq \mathcal{L}(\mathcal{I}_0') \wedge \mathcal{L}(\mathcal{I}_k) \subseteq \mathcal{L}(\mathcal{I}_k') \wedge \mathcal{L}(\mathcal{I}) \subseteq \mathcal{L}(\mathcal{I}')]$$

Proof: The proof follows from the Definitions 3, 4, 5 and 9. $\mathcal{I}_0 \leq_{\mathcal{L}} \mathcal{I}_0'$ implies that the the language of any peer's behavior in $\mathcal{I}_0$ is a subset of the language of the corresponding peer's behavior in $\mathcal{I}_0'$ (Definition 9), i.e., any sequence of the i-th peer behavior in $\mathcal{I}_0$ is also present in the i-th peer behavior in $\mathcal{I}_0'$. As a result, from Definition 5, $\mathcal{L}(\mathcal{I}_0) \subseteq \mathcal{L}(\mathcal{I}_0')$. □

THEOREM 1. *Given a conversation protocol $\mathcal{C}$ over a set of peers $\mathcal{P}$, the following holds for all synchronous systems $\mathcal{I}_0$ defined over a set of peer behaviors for the peers in $\mathcal{P}$.*

$$\mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}_0^{\mathcal{C}})) \Rightarrow [\forall \mathcal{I}_0 : (\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{I}_0) \Rightarrow \text{DETER}(\mathcal{I}_0^{\mathcal{C}}) \leq_{\mathcal{L}} \mathcal{I}_0)]$$

Proof: Assume that there exists an $\mathcal{I}_0$ such that $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}_0^{\mathcal{C}})) = \mathcal{L}(\mathcal{I}_0)$ and $\text{DETER}(\mathcal{I}_0^{\mathcal{C}}) \not\leq_{\mathcal{L}} \mathcal{I}_0$. Let the behaviors of the i-th peer in $\text{DETER}(\mathcal{I}_0^{\mathcal{C}})$ and $\mathcal{I}_0$ be $\mathcal{B}_i$ and B_i', respectively. Note that, $\mathcal{B}_i$'s are deterministic in $\text{DETER}(\mathcal{I}_0^{\mathcal{C}})$.

Therefore, there exists an i such that $\mathcal{L}(\mathcal{B}_i) \not\subseteq \mathcal{L}(B_i')$ (Definition 9). That is, there exists at least one path in $\mathcal{B}_i$, which is not present in $\mathcal{B}_i'$. Recall that, $\mathcal{B}_i$ is $\mathcal{C}_{\downarrow P_i}$ and is determinized; each path in $\mathcal{B}_i$ corresponds to at least one path in $\mathcal{C}$ (Definition 8). Therefore, absence of a path in $\mathcal{B}_i'$ implies at least one of the paths in $\mathcal{C}$ is not realizable using $\mathcal{I}_0$. This results in contradiction. □

4.1.2 Synchronizability Checking

As mentioned in the outline (item 2), a conversation $\mathcal{C}$ is realizable according to Realizability$_\forall$ if and only if it is equivalent to systems $\mathcal{I}_i$, for all $i \geq 0$. The following theorem due to [5] establishes

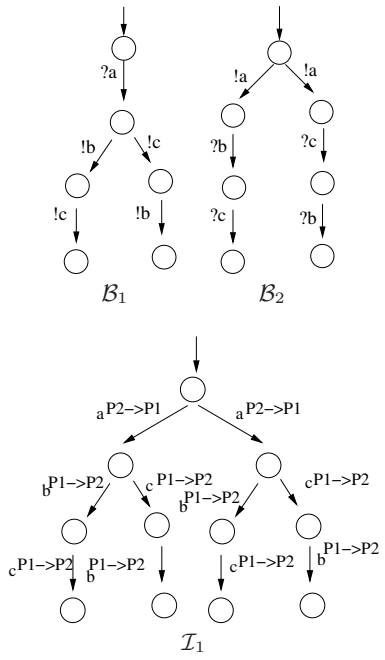

Figure 4. Two peer behaviors that are language synchronizable.

the necessary and sufficient condition under which the behaviors of $\forall i \geq 0 : \mathcal{I}_i$ are language equivalent. This is referred to as the synchronizability condition.

THEOREM 2. *$\mathcal{I}$ is language synchronizable (i.e., $\forall i \geq 0 : \mathcal{L}(\mathcal{I}_i) = \mathcal{L}(\mathcal{I}_{i+1})$) and only if $\mathcal{L}(\mathcal{I}_0) = \mathcal{L}(\mathcal{I}_1)$.*

Consider the behaviors $\mathcal{B}_1$ and $\mathcal{B}_2$ of peers P_1 and P_2, respectively, in Figure 4. The synchronous system $\mathcal{I}_0$ obtained from these peers has a behavioral structure similar to that of $\mathcal{B}_2$; there are two non-deterministic branches along which a is sent from P_2 to P_1. Along one branch, b is communicated before c and in the other branch, c is communicated before b. The 1-bounded asynchronous behavior, on the other hand, (see Figure 4) allows all possible ordering for communicating b and c along both the non-deterministic branches. The languages of $\mathcal{I}_0$ and $\mathcal{I}_1$ are identical. Therefore, the system obtained from P_1 and P_2 is language synchronizable.

4.1.3 Language Synchronizability & well-formedness

We present the condition under which a language synchronizable system is well-formed (see item 3 of outline of proof above for Realizability$_\forall$). We also use this result in the context of Realizability$_\exists$ (see Section 4.2).

THEOREM 3 (Synchronizability & well-formedness). *A synchronizable system $\mathcal{I}$ is well-formed if and only if $\mathcal{I}_1$ is well-formed.*

Proof: It is immediate that $\mathcal{I}$ is well-formed implies that $\forall i \geq 0 : \mathcal{I}_i$ is well-formed.

For proving the other direction, assume that $\mathcal{I}$ is synchronizable, WF($\mathcal{I}_1$) and $\neg$WF($\mathcal{I}$). Therefore, there exists a $k > 1$ such that $\neg$WF($\mathcal{I}_k$), and there exists a path in $\mathcal{I}_k$ over a sequence of states $t_0, t_1, \ldots, t_l, t_{l+1}$, with $0 \leq i \leq l : t_i \overset{m_i^{P_i \to P_i'}}{\Longrightarrow} t_{i+1}$, and the message m_l is never consumed by peer P_l'. Recall that, as $\mathcal{I}$ is language synchronizable, any sequence of send actions in $\mathcal{I}_k$ is also present in $\mathcal{I}_1$. From these observations, we iteratively construct a path in $\mathcal{I}_1$.

1. We denote the sequence of send actions causing violation of well-formedness in $\mathcal{I}_k$ as $\omega\, m_l^{P_l \to P_l'}$. Consider the path in the behavior of peer P_l' over the sequence of local states identical to the local states of P_l' in the sequence $t_0, t_1, \ldots, t_l$ in $\mathcal{I}_k$.
2. As $\mathcal{I}$ is synchronizable, consider a path over a sequence of states $t_0', t_1', \ldots, t_l', t_{l+1}'$ in $\mathcal{I}_1$ that results in the same sequence of send actions $\omega\, m_l^{P_l \to P_l'}$, and where all send actions are immediately consumed by the receiver peer. Such a path is possible as $\mathcal{I}_0$, $\mathcal{I}_1$ and $\mathcal{I}_k$ contain identical sequences of send actions. Note that, this path in $\mathcal{I}_1$ is well-formed.

   Consider the path in the behaviors of all peers other than P_l' over the sequence of local states that are identical to their local states in the sequence $t_0', t_1', \ldots, t_l'$.
3. Construct a path in the system by considering the local states of P_l' (item 1 above), the local states of all other peers (item 2 above), and proceed by sequentially matching the send action sequence $\omega\, m_l^{P_l \to P_l'}$. In this path, if at any point a transition results in a message sent from a sender peer P_s to a receiver peer P_r whose receive queue contains one message (to be consumed), we delay this transition by moving ahead some transition that occurs after it (shuffle) and does not involve the peers P_s and P_r.
   (a) Given the sequence $\omega\, m_l^{P_l \to P_l'}$, if the above shuffle operation can be performed till the action $m_l^{P_l \to P_l'}$, then a new path is obtained along which none of the receive queues contain more than one pending message to be consumed, i.e., a path in $\mathcal{I}_1$ is obtained. Further note that, as we have considered the path of P_l', where it does not consume the last message m_l, $\mathcal{I}_1$ is not well-formeded. That is, when $\mathcal{I}$ is synchronizable, $\neg$WF($\mathcal{I}$) $\Rightarrow$ $\neg$WF($\mathcal{I}_1$).
   (b) Consider that such shuffling (item 3 above) can be performed on the prefix ω_1 (such that $\omega_1\, \omega_2\, b\, \omega_3\, m_l^{P_l \to P_l'} = \omega\, m_l^{P_l \to P_l'}$), after which the shuffle operation of moving b before ω_2 is not possible, without allowing at least one peer's receive queue to hold more than one message. In the above, ω_1, ω_2 and ω_3 are sequences of send actions. Let the shuffling of ω_1 resulted in a new sequence ω_1'. Observe that, $\omega_1'\, b\, \omega_2\, \omega_3\, m_l^{P_l \to P_l'}$ is a sequence in $\mathcal{I}_k$ and it not well-formeded. Repeat the above steps starting from item 2 with this new sequence $\omega_1'\, b\, \omega_2\, \omega_3\, m_l^{P_l \to P_l'}$. Note that in new iteration we are still considering the same path in peer P_l' (item 1 above); but considering new paths for all peers other than P_l'. The new paths will ensure that till ω_1', all send actions are immediately received making room for at least one more send action (i.e., b) to be performed without forcing any peer's receive queue to contain more than 1 message.

   As $\mathcal{I}$ is synchronizable, the above iteration will always terminate in Step $3a$, proving that when $\mathcal{I}$ is synchronizable, $\neg$WF($\mathcal{I}$) $\Rightarrow$ $\neg$WF($\mathcal{I}_1$). □

THEOREM 4 (Determinism, Synchronizability & well-formedness). *A synchronizable system $\mathcal{I}$ consisting of deterministic peers is well-formed.*

Proof: A system is synchronizable and consists of deterministic peers imply that $\mathcal{L}(\mathcal{I}_0) = \mathcal{L}(\mathcal{I}_1)$ and $\mathcal{I}_1$ is deterministic. Assume that, $\mathcal{I}_1$ is not well-formed.

There exists a sequence of send actions $\omega = m_0 m_1 \ldots m \ldots$ which leads to a state t from where a message m is never consumed. Due to synchronizability, the same sequence of send actions is also present in $\mathcal{I}_0$ where each message sent is immediately consumed. A similar path is present $\mathcal{I}_1$, where every send action is immediately followed by the corresponding receive action, resulting in the same sequence of send actions ω. Recall that, all peers are deterministic. Therefore, $\mathcal{I}_1$ cannot contain two different paths with the same send sequence ω; where one path leads to a state from where m is never consumed, and in the other all sent messages are consumed. In other words, a deterministic $\mathcal{I}_1$ containing deterministic peers is well-formed. From Theorem 3, the corresponding system $\mathcal{I}$ is well-formed, and therefore, $\forall i \geq 0 : \text{WF}(\mathcal{I}_i)$ also holds. □

4.1.4 Deciding Language Realizability$_{\forall}$

Based on the above theorems and propositions, we now proceed to present the necessary and sufficient conditions for language realizability according to Realizability$_{\forall}$.

THEOREM 5. *$\mathcal{C}$ is language realizable following Realizability$_{\forall}$* $\Leftrightarrow$ $[\mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}}))]$

Proof: To prove: $\mathcal{C}$ is language realizable according to Realizability$_{\forall}$ implies $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}}))$.
Assume that $\mathcal{C}$ is language realizable and

$$\mathcal{L}(\mathcal{C}) \neq \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}}))$$
$$\text{i.e., } \mathcal{L}(\mathcal{C}) \subset \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}})) \text{ (Proposition 2)}$$

As $\mathcal{C}$ is language realizable according to Realizability$_{\forall}$, there exists a $\mathcal{I}$ such that $\forall i \geq 0 : \mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{I}_i)$. Therefore, $\mathcal{I}$ is synchronizable, i.e., $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{I}_0) \wedge \mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{I}_1)$ (From Theorem 2).

We first establish that for $\mathcal{C}$ to be realizable, $\mathcal{L}(\mathcal{C})$ must be identical to $\mathcal{L}(\text{DETER}(\mathcal{I}_0^{\mathcal{C}}))$ (or $\mathcal{L}(\mathcal{I}_0^{\mathcal{C}})$; determinizing peers does not alter the language of the system).

Assuming $\mathcal{L}(\mathcal{C}) \neq \mathcal{L}(\mathcal{I}_0^{\mathcal{C}})$, from Proposition 2, we have $\mathcal{L}(\mathcal{C}) \subset \mathcal{L}(\mathcal{I}_0^{\mathcal{C}})$. This implies that there exists a specific ordering of send actions involving two independent pairs of peers. In other words, $\mathcal{C}$ has a state from where $a^{P_1 \to P_2}$ is followed by $b^{P_3 \to P_4}$, and P_i's are distinct and the reverse order $b^{P_3 \to P_4}$ followed by $a^{P_1 \to P_2}$ is not allowed from the same state in $\mathcal{C}$. Such a conversation $\mathcal{C}$ cannot be realized by any system as the send actions a and b are independent, and any specific ordering required by $\mathcal{C}$ cannot be obtained. Therefore, if $\mathcal{C}$ is realizable then $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{I}_0^{\mathcal{C}}) = \mathcal{L}(\text{DETER}(\mathcal{I}_0^{\mathcal{C}}))$.

From Theorem 1, it follows that $\text{DETER}(\mathcal{I}_0^{\mathcal{C}}) \leq_{\mathcal{L}} \mathcal{I}_0$. This implies that $\forall i : \mathcal{L}(\mathcal{B}_i) \subseteq \mathcal{L}(B_i')$, where $\mathcal{B}_i$ and $\mathcal{B}_i'$ denote the behavior of the i-th peer in $\text{DETER}(\mathcal{I}_0^{\mathcal{C}})$ and $\mathcal{I}_0$, respectively. (Recall that in $\text{DETER}(\mathcal{I}^{\mathcal{C}})$, the peer behaviors are determinized, i.e., $\mathcal{B}_i$ is deterministic.)

Proceeding further, from Proposition 3, we have $\mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}})) \subseteq \mathcal{L}(\mathcal{I}_1)$. This leads to a contradiction as we have assumed $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{I}_1)$ and $\mathcal{L}(\mathcal{C}) \subset \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}}))$. Therefore, $\mathcal{C}$ is realizable according to Realizability$_{\forall}$ implies $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{I}_1) = \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}}))$, and note that, $\text{DETER}(\mathcal{I}_1^{\mathcal{C}})$ is well-formed (Theorem 4).

Next, $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}}))$

$\Rightarrow$ $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}_0^{\mathcal{C}})) = \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}})) \wedge \text{WF}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}}))$ (Proposition 2)

$\Rightarrow$ $\forall i \geq 0 : \mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}_i^{\mathcal{C}})) \wedge \text{WF}(\text{DETER}(\mathcal{I}_i^{\mathcal{C}}))$ (Theorems 2, 3, 4)

$\Rightarrow$ $\mathcal{C}$ is language realizable according to Realizability$_{\forall}$ (Definition 7)

□

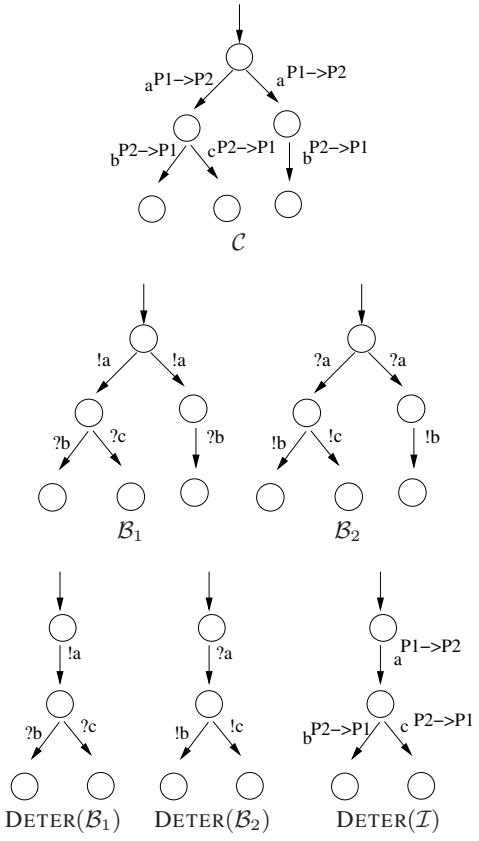

Figure 5. A conversation protocol that is language realizable according to Realizability$_{\forall}$.

Consider the conversation specification $\mathcal{C}$ in Figure 5. The system behavior $\text{DETER}(\mathcal{I})$ (irrespective of the receive queue size) based on peers determinized behaviors ($\text{DETER}(\mathcal{B}_1)$, $\text{DETER}(\mathcal{B}_2)$) is language equivalent to $\mathcal{C}$ and is $\text{WF}(\text{DETER}(\mathcal{I}))$. Therefore, $\mathcal{C}$ is language realizable according to Realizability$_{\forall}$.

4.2 Deciding Language Realizability$_{\exists}$

THEOREM 6. *$\mathcal{C}$ is language realizable following Realizability$_{\exists}$* $\Leftrightarrow$ $[\mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}}))]$.

Proof: The proof proceeds by showing the equivalence between Realizability$_{\forall}$ and Realizability$_{\exists}$.

$\mathcal{C}$ is language realizable following Realizability$_{\forall}$
$\Leftrightarrow$ $\forall i \geq 0 : \mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{I}_i) \wedge \text{WF}(\mathcal{I}_i)$ (Definition 7)
$\Leftrightarrow$ $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}}))$ (Theorem 5)
$\Leftrightarrow$ $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}})) \wedge \text{WF}(\text{DETER}(\mathcal{I}^{\mathcal{C}}))$ (Theorem 4)
$\Leftrightarrow$ $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}^{\mathcal{C}})) \wedge \text{WF}(\text{DETER}(\mathcal{I}^{\mathcal{C}}))$ (Theorem 2)
$\Leftrightarrow$ $\exists \mathcal{I} : \mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{I}) \wedge \text{WF}(\mathcal{I})$
$\Leftrightarrow$ $\mathcal{C}$ is language realizable following Realizability$_{\exists}$

Therefore, from Theorem 5, $\mathcal{C}$ is language realizable following Realizability$_{\exists}$ $\Leftrightarrow$ $[\mathcal{L}(\mathcal{C}) = \mathcal{L}(\text{DETER}(\mathcal{I}_1^{\mathcal{C}}))]$. □

The conversation in Figure 5 is realizable as per Realizability$_{\exists}$. The peer behaviors, obtained from projection of the conversation and determinization, results in a system that is well-formed and language equivalent to the conversation.

4.3 Summary of the Results

Theorems 5 and 6 lead to a methodology for automatically checking the realizability of a conversation specification $\mathcal{C}$ for Realizability$_\forall$ and Realizability$_\exists$.

Deciding realizability was an open problem. We have proved that realizability can be verified by checking the language equivalence between $\mathcal{C}$ and 1-bounded system over a set of peers obtained from projection of $\mathcal{C}$ on peers and verifying the satisfiability of a temporal property by the 1-bounded system (well-formedness verification). Both equivalence checking and well-formedness verification can be performed automatically (using existing tools) as the 1-bounded system and the conversation representing the choreography specification both have finite state-space. Furthermore, we have proved that Realizability$_\forall$ and Realizability$_\exists$ are equivalent when language equivalence between choreography and system is considered.

The complexity for well-formedness verification is linear to the size of the 1-bounded system and the size of the CTL formula specifying the well-formed property (complexity for CTL model checking [10]). The complexity for language equivalence checking is PSPACE-complete for language equivalence.

The conversation protocols shown in Figures 1, 2 and 3 are language realizable according to Realizability$_\forall$ and Realizability$_\exists$.

5. Experimental Evaluation

We have automated our approach for checking realizability of conversation protocols leveraging the CADP toolbox [17], which provides a wide range of constructs for representing communicating finite state machines and mechanisms for checking equivalences between such state machines.

We have implemented a translator, which takes a conversation protocol specification as input (specified in the conversation protocol format of the Web Service Analysis Tool [15]) and generates two LOTOS specifications (the specification language for CADP): (a) one that corresponds to the conversation protocol itself and (b) another one that corresponds to the 1-bounded-asynchronous projection of the protocol. We have developed SVL scripts [16] that automatically construct state machine representations from the LOTOS specifications, and check language realizability by deploying CADP's Reductor tool, which reduces the state machines and checks equivalence between them in an optimized fashion.

5.1 Conversation Protocol to LOTOS Translation

In LOTOS, processes are declared as `process proc[m1, m2, ...]: exit := behavior`, where `process` is a keyword, `proc` is the process name, `m1, m2, ...` are the messages that are sent and received by this process, and `behavior` describes the process behavior. The process behavior specifies the message exchange order using the LOTOS operators. The sequential ordering is specified using the operator `;` where, *e.g.*, `m1;m2` means that message `m1` must precede message `m2`. Choices are specified using the operator `[]` where `m1[]m2` means that only one message can be executed. Looping behavior is encoded as a process with a recursive behavior. Lastly, the internal τ action and the system termination are described using the LOTOS messages `i` and `exit`, respectively.

Consider the `KeyboardDeviceContract` (Figure 2) which describes the interaction between a server and its client. Below we show a portion of the automatically generated LOTOS code for this contract. We use the suffix `_S_C` to denote the messages sent by the server to the client and we use the suffix `_C_S` to denote the messages sent by the client to the server. The process `KEYBOARDDEVICECONTRACT` starts with the message `Success_S_C` which is sent from the server to the client and then the contract makes a transition into a looping process which starts at the `Ready` state shown in Figure 2. This looping process describes all possible behaviors starting from the `Ready` state.

```
process KEYBOARDDEVICECONTRACT[Success_S_C,
 PollKey_C_S,AckKey_S_C,GetKey_C_S,NakKey_S_C]:
 exit :=
 Success_S_C ; LOOP_READY[...]
 where
 process LOOP_READY[...] : exit :=
  ((GetKey_C_S; ((AckKey_S_C ; LOOP_READY[...])
              [] (NakKey_S_C ; LOOP_READY[...]))))
  [] (PollKey_C_S; ((AckKey_S_C ; LOOP_READY[...])
              [] (NakKey_S_C ; LOOP_READY[...]))))
 endproc
endproc
```

5.2 1-Bounded Asynchronous Projection in LOTOS

The LOTOS language does not support asynchronous communication directly. In order to generate the 1-bounded asynchronous model in LOTOS we create a bounded FIFO queue process (which can store at most one message) for each message queue. Given a conversation protocol, we need to create a message queue for each process generated from its projection. For instance, the projection of the `KeyboardDeviceContract` generates two processes, namely, `SERVER` and `CLIENT`. In order to generate the asynchronous version of the server process we compose the `SERVER` and the queue process using LOTOS composition operator `|[m1, m2, ...]|` where both processes synchronize on shared messages `m1, m2, ...`. This queue is responsible for storing the (at most one) messages to be consumed by the server.

```
process ASYNC_SERVER[...] : exit := ...
    ( SERVER[...]
      |[...]|
      SERVER_queue[...](queue(1,nil)) )
endproc
```

The queue associated with the server synchronizes with the client by receiving the actions sent by the client; it also synchronizes with the server by sending the actions that are consumed immediately by the server. The synchronized actions between the queue and the server are hidden during the composition and become τ (internal) transitions in LOTOS (i.e., ϵ-transition as per our notation). Similarly we generate the asynchronous version of the client process by composing the client's receive queue and the `CLIENT` process. Finally, the overall 1-bounded asynchronous model is obtained by composing the `ASYNC_SERVER` process with the `ASYNC_CLIENT` process using the LOTOS operator `|[...]|`. These two processes synchronize on the message send events.

5.3 Equivalence Checking

After generating the LOTOS specifications for the conversation protocol and 1-bounded asynchronous projection models, we generate the two corresponding LTSs using the state space generation tools in the CADP toolbox. We check the equivalence of the two LTSs to determine realizability of the protocol. During the equivalence checking, the receive actions are hidden (as internal action τ) and the send actions are left visible.

In order to check the language realizability in an optimized way, we first reduce the resulting LTSs modulo weak trace relation which reduces the transition systems without modifying their visible traces. Then we check the equivalence of reduced LTSs for the conversation protocol and its 1-bounded-asynchronous projection using the weak trace equivalence relation. If the two LTSs are equivalent this means that send-traces for the conversation protocol and its 1-bounded-asynchronous projection are identical and, hence, the conversation protocol is language-realizable. If the two LTSs are not equivalent, then we conclude that the conversation protocol is not language-realizable.

	Conversation Protocol Size		Async. Model Size		Analysis Time (seconds)	
	$\|C\|$	$\|\Delta\|$	$\|C\|$	$\|\Delta\|$	Reduction	Equivalence
Mean	6.70	9.74	3268.83	19133.72	11.53	2.18
Min	2	1	2	1	6.50	0.48
Max	31	68	302449	1791867	39.91	3.07
STDV	4.40	9.02	30713.29	181950.70	3.12	0.27

Table 1. Protocol and model sizes and analysis time for the experiments on the realizability of conversation protocols ($|C|$: the number of states; $|\Delta|$: the number of transitions).

	Conversation Protocols	Language Realizable
Singularity Channels	86	84
Choreographies	9	8
Collaboration Diagrams	9	8

Table 2. Results of the experiments on language realizability.

5.4 Experiments

We applied our approach to 104 conversation protocols which describe web service choreographies, Singularity OS channel contracts, and UML collaboration diagrams. All these specifications were first automatically translated to conversation protocols (in the conversation protocol format of the Web Service Analysis Tool) using the translators described in [15], [34], and [7], respectively. Then we used the conversation protocol to LOTOS translator we described above to generate the LOTOS specifications for the conversation protocol and the 1-bounded asynchronous projection.

We report the cumulative results of our analysis in Table 1 showing the sizes of the conversation protocol specifications, asynchronous models and the execution times for the reduction and the equivalence checking steps. The conversation protocol specifications are not very large, the biggest one has 31 states and 68 transitions. 1-bounded asynchronous model can be very large in some cases, however the reduction techniques we use reduces the sizes of the models significantly. The reductions take about 11 seconds on average and the equivalence check takes about 2 seconds on average. So realizability of a conversation protocol can be determined in about 13 seconds on average.

According to the results of our analysis (see Table 2) all conversation protocols in our base are language realizable except four. Two of the four that fail the realizability check are Singularity channel contracts which were confirmed by the Singularity developers to be faulty [34].

6. Related Work

The language-based choreography realizability problem for conversation protocols was first proposed in [14] where sufficient conditions for realizability were given. The work on session types [19, 20] is also related to the realizability of conversation protocols and has been used as a formal basis for modeling choreography languages [9]. The restrictions used in session types to guarantee that local implementations follow the global interaction protocol are similar to the sufficient conditions for realizability given in [14] and they are not necessary conditions, i.e., there are realizable choreography specifications which fail the conditions given in these earlier results. In particular, both of these earlier approaches do not allow a protocol containing a state with an *arbitrary initiator* [18], i.e., a state where more than one peer could send the next message and the protocol works fine for either case. Protocols which are of this type and are realizable appear in practice (for example, protocols where one of the peers can cancel the interaction at an arbitrary point) and cannot be shown to be realizable with these earlier approaches.

Choreography realizability problem for several different communication models have been investigated in [25], however, the presented techniques can only show realizability if the asynchronous projection of the protocol has a finite state space (which would not be the case if the protocol has a single self loop for example). More recently, [27] presented an approach that only allows specification of realizable choreographies but, like the approaches discussed above, this approach does not allow specification of realizable choreographies that have arbitrary initiator states. Finally, [18] proposed a new realizability check that correctly identifies the realizability of many arbitrary initiator protocols, however, like all the earlier results, it still provides a sufficient condition for realizability, and decidability of the realizability problem has remained open. In this paper we give a necessary and sufficient condition for realizability and show that it is a decidable problem. Interestingly, the similar realizability problem for the MSC-graphs (which is an extension of MSCs) is undecidable [1].

Realizability of collaboration diagrams has been studied [8] and it has been showed that language realizability for collaboration diagrams can be checked by checking the equivalence of the choreography model with the 1-bounded asynchronous model [32]. However, the collaboration diagram model used in [32] is not as expressive as the conversation protocols, and cannot model the Singularity contracts and the web service choreographies we analyzed in our experiments.

Realizability of Singularity channel contracts have been first studied in [34] using the realizability conditions from [14]. The realizability check we present in this paper can identify some Singularity channel contracts as realizable for which the realizability check used in [34] gives false positives.

Message patterns expressed with Petri nets using synchronous communication are "de-synchronized" in [11]—that is, one is interested in finding a specification that produces the same pattern of messages when communications become asynchronous. This work, however, already assumes that a conversation is realizable and does not provide realizability conditions.

The work presented in [26] checks choreography realizability using the controllability concept. Given a choreography description, a monitor service is computed from that choreography, and is used as a centralized orchestrator of the interaction to compute the distributed peers. The choreography is said to be realizable if the monitor service is controllable, that is, there exists a set of peers such that the composition of the monitor service and those peers is deadlock-free. Our approach is different since the distributed peers are computed without the centralized orchestrator, i.e., the realizability notion we study in this paper does not require a monitor service, and our techniques for checking realizability rely on equivalence checking rather than controllability checking.

7. Conclusion

In this paper, we prove that the choreography realizability problem is decidable for systems communicating with asynchronous messages using unbounded FIFO message queues. We provide a necessary and sufficient condition for realizability which states that, a choreography specification is realizable for systems with asynchronously communicating peers over unbounded message queues, if and only if, the choreography specification behavior is equivalent to the behavior of a well-formed 1-bounded system where each peer behavior is obtained from determinizing the projection of the choreography (on each peer) and where each peer has a message queue of size 1. As the choreography specification and the 1-bounded system both exhibit finite state-space, checking language realizability of the choreography specification can be automatically performed using existing equivalence checking and model checking tools. We have also implemented our technique for realizability checking in a prototype tool using CADP toolbox and verified the realizability of a wide range of choreography specifications that describe Web service interactions, Singularity OS contracts and UML collaboration diagrams.

Acknowledgment

The authors thank Gwen Salaün for fruitful discussions on the CADP implementation. This work has been partially supported by the US National Science Foundation grants CCF1117708, CCF1116836, CCF0702758, and project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science and FEDER.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *Proc. 28th Int. Colloq. on Automata, Languages, and Programming*, pages 797–808, 2001.

[2] J. Armstrong. Getting Erlang to Talk to the Outside World. In *Proc. ACM SIGPLAN Work. on Erlang*, pages 64–72, 2002.

[3] J. Armstrong. *Programming Erlang, Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] G. Banavar, T. Deepak Chandra, R. E. Strom, and D. C. Sturman. A Case for Message Oriented Middleware. In *Proceedings of the 13th Int. Symp. Distributed Computing*, pages 1–18, 1999.

[5] S. Basu and T. Bultan. Choreography Conformance via Synchronizability. In *Proc. 20th Int. World Wide Web Conf.*, 2011.

[6] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[7] T. Bultan, C. Ferguson, and X. Fu. A tool for choreography analysis using collaboration diagrams. In *Proc. 7th IEEE Int. Conf. Web Services*, 2009.

[8] T. Bultan and X. Fu. Specification of Realizable service conversations using collaboration diagrams. *Service Oriented Computing and Applications*, 2(1):27–39, 2008.

[9] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A Theoretical Basis of Communication-Centred Concurrent Programming, W3C Note, October 2006. http://www.w3.org/2002/-ws/chor/edcopies/theory/note.pdf.

[10] E.M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[11] G. Decker, A. P. Barros, F. M. Kraft, and N. Lohmann. Non-desynchronizable Service Choreographies. In *Proc. ICSOC*, pages 331–346, 2008.

[12] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. In *Proc. 2006 EuroSys Conf.*, pages 177–190, 2006.

[13] C. Ferris and J. Farrell. What are web services? *Comm. of the ACM*, 46(6):31–31, June 2003.

[14] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and analysis of reactive electronic services. *Theoretical Computer Science*, 328(1-2):19–37, November 2004.

[15] X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of web services. In *Proc. 16th Int. Conf. on Computer Aided Verification*, pages 510–514, 2004.

[16] H. Garavel and Frédéric Lang. SVL: A Scripting Language for Compositional Verification. In *Proc. FORTE*, pages 377–394, 2001.

[17] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. 18th Int. Conf. Computer Aided Verification*, 2006.

[18] S. Hallé and T. Bultan. Realizability Analysis for Message-Based Interactions using Shared-State Projections. In *Proc. 18th ACM SIGSOFT Int. Sym. Foundations of Software Engineering*, pages 27–36, 2010.

[19] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. 7th European Symp. on Programming Languages and Systems*, pages 122–138, 1998.

[20] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *Proc. 35th ACM SIGPLAN-SIGACT Sym. Principles of Programming Languages*, pages 273–284, 2008.

[21] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.

[22] Conversation Support for agents, e-business, and component integration. http://www.research.ibm.com/convsupport.

[23] Java API for XML messaging (JAXM). http://java.sun.com/developer/earlyAccess/xml/jaxm/.

[24] Java Message Service. http://java.sun.com/products/jms/.

[25] R. Kazhamiakin and M. Pistore. Analysis of Realizability Conditions for Web Service Choreographies. In *Proc. FORTE*, pages 61–76, 2006.

[26] N. Lohmann and K. Wolf. Realizability is Controllability. In *Proc. 1st Central-European Work. on Services and Their Composition*, pages 61–67, 2009.

[27] A. McNeile. Protocol contracts with application to choreographed multiparty collaborations. *Service Oriented Computing and Applications*, 4:109–136, 2010.

[28] D. A. Menascé. Mom vs. rpc: Communication models for distributed applications. *IEEE Internet Computing*, 9(2):90–93, 2005.

[29] Message Sequence Chart (MSC). ITU-T, Geneva Recommendation Z.120, 1996.

[30] Microsoft Message Queuing Service. http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx.

[31] E. Newcomer. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Springer, 2004.

[32] G. Salaün and T. Bultan. Realizability of Choreographies Using Process Algebra Encodings. In *Proceedings of the 7th International Conference on Integrated Formal Methods*, pages 167–182, 2009.

[33] Singularity design note 5 : Channel contracts. singularity rdk documentation (v1.1). `http://www.codeplex.com/singularity`, 2004.

[34] Z. Stengel and T. Bultan. Analyzing Singularity Channel Contracts. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 13–24, 2009.

[35] 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*, 13(1):37–85, 2004.

[36] OMG unified modeling language superstructure, version 2.1.2. `http://ww.uml.org/`, October 2007.

[37] Web Service Choreography Description Language (WS-CDL). http://www.w3.org/TR/ws-cdl-10/, 2005.