Service Oriented Computing and Applications manuscript No.
(will be inserted by the editor)

Specification of Realizable Service Conversations Using Collaboration Diagrams

Tevfik Bultan [⋆1], **Xiang Fu**[2]

[1] Computer Science Department
University of California
Santa Barbara, CA 93106, USA
e-mail: bultan@cs.ucsb.edu

[2] School of Computer and Information Sciences
Georgia Southwestern State University
Americus, GA 31709, USA
e-mail: xfu@canes.gsw.edu

Received: 09/15/2007 / Revised version: date

Abstract Specification, modeling and analysis of interactions among peers that communicate via messages are becoming increasingly important due to the emergence of service oriented computing. Collaboration diagrams provide a convenient visual model for specifying such interactions. An interaction among a set of peers can be characterized as a conversation. A conversation is the global sequence of messages exchanged among the peers, listed in the order they are sent. A collaboration diagram can be used to specify the set of allowable conversations among the peers participating to a composite web service. Specification of interactions from a global perspective leads to the realizability problem: Is it possible to construct a set of peers that generate exactly the specified set of conversations? In this paper, we investigate the realizability of conversations specified by collaboration diagrams. We formalize the realizability problem by modeling peers as concurrently executing finite state machines, and we give sufficient realizability conditions for a class of collaboration diagrams.

1 Introduction

Collaboration diagrams are useful for modeling interactions among distributed components without exposing their internal structure. In particular, collaboration diagrams model interactions as a sequence of messages which are recorded in the order they are sent. Such an interaction model is becoming increasingly important in service oriented computing where a set of autonomous peers interact with each other using synchronous or asynchronous messages. Web services that belong to different organizations need to interact with each other through standardized interfaces and without access to each other's internal implementations. Formalisms which focus on interactions rather than the local behaviors of individual peers are necessary for both specification and analysis of such distributed applications.

⋆ This work is supported by NSF grants CCF-0614002 and CNS-0716095

The need to develop mechanisms for specifying interactions in composite services is well recognized in the web services area. For example, Web Services Choreography Description Language (WS-CDL) [1] is an XML-based language for describing the interactions among services. WS-CDL specifications describe "peer-to-peer collaborations of Web Services participants by defining, from a global viewpoint, their common and complementary observable behavior; where ordered message exchanges result in accomplishing a common business goal." Collaboration diagrams provide a suitable visual formalism for modeling such specifications.

However, characterizing interactions using a global view may lead to behavior specifications that are not implementable. In this paper, we study the problem of realizability which addresses the following question: Given an interaction specification, is it possible to find a set of distributed peers which generate exactly the specified set of interactions.

In order to study the realizability problem, we define a formal model for collaborations diagrams. We model a distributed system as a set of communicating finite state machines [2]. A collaboration diagram is realizable if there exists a set of communicating finite state machines that generate exactly the set of conversations specified by the collaboration diagram. We present sufficient conditions for realizability of a class of collaboration diagrams.

Although realizability problem for Message Sequence Charts (MSCs) has been studied extensively [3–5], to the best of our knowledge, realizability of collaboration diagrams has not been studied before. Collaboration diagrams provide a different view of interactions than the one provided by MSCs. MSCs show the *local* orderings of the message *send* and *receive* events, whereas the collaboration diagrams show the *global* ordering of the message *send* events. The ordering of the message receive events in collaboration diagrams corresponds to a "don't care" condition, i.e., receive events can be ordered in any way as long as the send events follow the specified order. Due to these differences, earlier results on realizability of MSCs are not applicable to the realizability of collaboration diagrams.

Rest of the paper is organized as follows. In Section 2 we introduce a formal model for collaboration diagrams and we define the set of conversations specified by a collaboration diagram. In Section 3 we present a formal model for a set of autonomous peers communicating via messages and we define the set of conversations generated by such peers. In Section 4 we discuss the realizability of collaboration diagrams. In Section 5 we give sufficient conditions for realizability of a class of collaboration diagrams. In Section 6 we discuss the related work and in Section 7 we conclude the paper.

2 Collaboration Diagrams

In this paper we focus on the use of collaboration diagrams for specification of interactions among a set of *peers*. We model each peer as an active object with its own thread of control. We model the interactions specified by a collaboration diagram as conversations [6, 7], i.e., sequences of messages exchanged among the peers listed in the order they are sent. This provides an appropriate model for the web services domain where a set of autonomous peers communicate with each other through messages.

A collaboration diagram (called communication diagram in [8]) consists of a set of peers, a set of links among the peers showing associations, and a set of message send events among the peers. Each message send event is shown by drawing an arrow over a link denoting the sender and the receiver of the message. Messages can be transmitted using synchronous (shown with a filled solid arrowhead) or asynchronous (shown with a stick arrowhead) communication. During a synchronous message transmission, the sender and the receiver must execute the send and receive events simultaneously.

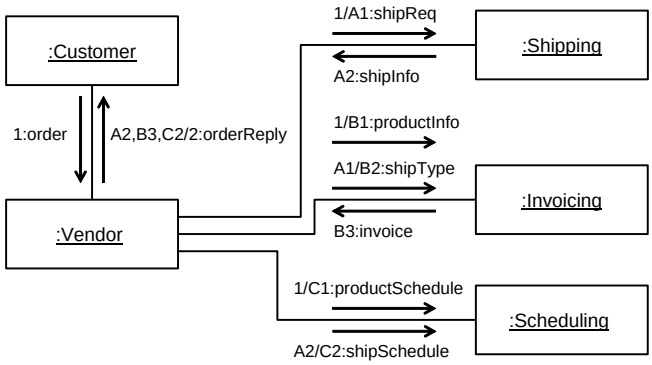

Fig. 1 An example collaboration diagram for a composite web service.

During an asynchronous message transmission, the send event appends the message to the input queue of the receiver, where it is stored until receiver consumes it with a receive event. Note that, a collaboration diagram does not show when a receive event for an asynchronous message will be executed, it just gives an ordering of the send events.

In a collaboration diagram each message send event has a unique sequence label. These sequence labels are used to declare the order the messages should be sent. Each sequence label consists of a (possibly empty) string of letters (which we call the prefix) followed by a numeric part (which we call the sequence number). The numeric ordering of the sequence numbers defines an implicit total ordering among the message send events with the same prefix. For example, event A2 can occur only after the event A1, but B1 and A2 do not have any implicit ordering. In addition to the implicit ordering defined by the sequence numbers, it is possible to explicitly state the events that should precede an event e by listing their sequence labels (followed by the symbol "/") before the sequence label of the event e. For example if an event e is marked with "B2,C3/A2" then A2 is the sequence label of the event e, and the events with sequence labels B2, C3 and A1 must precede e.

The prefixes in sequence labels of collaboration diagrams enable specification of concurrent interactions where each prefix represents a *thread*. Note that, here, "thread" does not mean a thread of execution. Rather, it refers to a set of messages that have a total ordering and that can be interleaved arbitrarily with other messages. The sequence numbers specify a total ordering of the send events in each thread. The explicitly listed dependencies, on the other hand, provide a synchronization mechanism between different threads.

In a collaboration diagram message send events can be marked to be conditional, denoted as a suffix "[*condition*]", or iterative, denoted as a suffix "*[*condition*]", where *condition* is written in some pseudocode. In our formal model we represent conditional and iterative message sends with nondeterminism where a conditional message send corresponds to either zero or one message send, and an iterative message send corresponds to either zero or one or more consecutive message sends.

2.1 An Example

As an example, consider the collaboration diagram in Figure 1 for the Purchase Order Handling service described in the Business Process Execution Language for Web Services (BPEL) 1.1 language specification [9]. In this example, a customer sends a purchase order to a vendor. The vendor arranges a shipment, calculates the price for the order including the shipping fee, and schedules the production and shipment. The vendor uses a shipping service to arrange the shipment, an invoicing

service to calculate the price, and a scheduling service to handle the scheduling. To respond to the customer in a timely manner, the vendor performs these three tasks concurrently while processing the purchase order. There are two control dependencies among these three tasks that the vendor needs to consider: The shipment type is required to complete the final price calculation, and the shipping date is required to complete the scheduling. After these tasks are completed, the vendor sends a reply to the customer.

The web service for this example is composed of five peers: Customer, Vendor, Shipping, Scheduling, and Invoicing. Customer orders products by sending the *order* message to the Vendor. The Vendor responds to the Customer with the *orderReply* message. The remaining peers are the ones that the Vendor uses to process the product order. The Shipping peer communicates with the *shipReq*, and *shipInfo* messages, the Scheduling peer with the *productSchedule*, and *shipSchedule* messages, and the Invoicing peer with the *productInfo, shipType*, and *invoice* messages.

Figure 1 shows the interactions among the peers in the Purchase Order Handling service using a collaboration diagram. All the messages in this example are transmitted asynchronously. Note that the collaboration diagram in Figure 1 has four threads (the main thread, which corresponds to the empty prefix, and the threads with labels A, B and C) and the interactions between the Vendor and the Shipping, Scheduling and Invoicing peers are executed concurrently. However, there are some dependencies among these concurrent interactions: *shipType* message should be sent after the *shipReq* message is sent, the *shipSchedule* message should be sent after the *shipInfo* message is sent, and the *orderReply* message should be sent after all the other messages are sent.

2.2 A Formal Model

Based on the assumptions discussed above we formalize the semantics of the collaboration diagrams as follows. A *collaboration diagram* $\mathcal{D} = (P, L, M, E, D)$ consists of

- a set of peers P,
- a set of links $L \in P \times P$,
- a set of messages M,
- a set of message send events E, and
- a dependency relation $D \subseteq E \times E$ among the message send events.

The sets P, L, M and E are all finite. To simplify our formal model, we assume that the asynchronous messages M^A and synchronous messages M^S are separate (i.e., $M = M^A \cup M^S$ and $M^A \cap M^S = \emptyset$), and that each message has a unique sender and a unique receiver denoted by $send(m) \in P$ and $recv(m) \in P$, respectively. (Note that, messages in any collaboration diagram can be converted to this form by concatenating each message with tags denoting the synchronization type and its sender and its receiver.) For each message $m \in M$, the sender and the receiver of m must be linked, i.e., $(send(m), recv(m)) \in L$.

The set of send events E is a set of tuples of the form (l, m, r) where l is the label of the event, $m \in M$ is a message, and $r \in \{1, ?, *\}$ is the recurrence type. We denote the size of the set E with $|E|$ and for each event $e \in E$ we use $e.l$, $e.m$, and $e.r$ to denote different fields of e. The labels of the events correspond to the sequence labels, and we assume that each event in E has a unique label. Each event $e \in E$ denotes a message send event where the peer $send(e.m)$ sends a message $e.m$ to the peer $recv(e.m)$. The recurrence type $r \in \{1, ?, *\}$ determines if the send event corresponds to

- a single message send event ($r = 1$),
- a conditional message send event ($r =?$), or
- an iterative message send event ($r = *$).

The dependency relation $D \subseteq E \times E$ denotes the ordering among the message send events where $(e_1, e_2) \in D$ means that e_1 has to occur before e_2. We assume that there are no circular dependencies, i.e., the dependency graph (E, D), where the send events in E form the vertices and the dependencies in D form the edges, should be a directed acyclic graph (dag).

Given a dependency relation $D \subseteq E \times E$ let $pred(e)$ denote the predecessors of the event e where $e' \in pred(e)$ if there exists a set of events $e_1, e_2, \ldots, e_k$ where $k > 1$, $e' = e_1$, $e = e_k$, and for all $i \in [1..k-1]$, $(e_i, e_{i+1}) \in D$. Similarly, we use $succ(e)$ to denote the successors of the event e where $e' \in succ(e)$ if there exists a set of events $e_1, e_2, \ldots, e_k$ where $k > 1$, $e = e_1$, $e' = e_k$, and for all $i \in [1..k-1]$, $(e_i, e_{i+1}) \in D$.

A dependency $(e', e) \in D$ is redundant if there exists an $e'' \in pred(e)$ such that $e' \in pred(e'')$. We assume that there are no redundant dependencies in D (i.e., the dependency relation D is the transitive reduction of the partial ordering of the events). Since we do not allow any redundant dependencies in D, we call e' an immediate predecessor of e if $(e', e) \in D$.

Given a collaborations diagram $\mathcal{D}$, we call an event e_I with $pred(e_I) = \emptyset$ an *initial event* of $\mathcal{D}$ and an event e_F where for all $e \in E$ $e_F \notin pred(e)$ a *final event* of $\mathcal{D}$. Note that since the dependency relation is a dag there is always at least one initial event and one final event (and there may be multiple initial events and multiple final events).

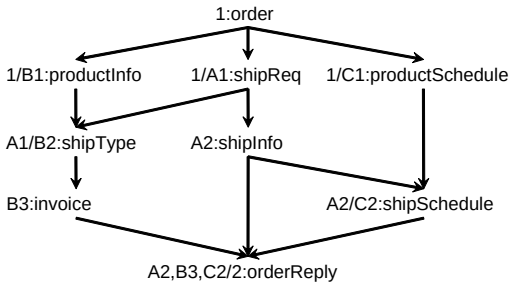

Fig. 2 Dependencies among the message send events in the Purchase Order example.

Figure 2 shows the dependency graph for the the collaboration diagram of the Purchase Order example shown in Figure 1. In this example event 1 is an initial event and event 2 is a final event. Event 2 has three immediate predecessors: A2, B3 and C2.

Let $\mathcal{D} = \{P, L, M, E, D\}$ denote the formal model for the collaboration diagram of the Purchase Order example shown in Figure 1. The elements of the formal model are as follows (we denote the peers and messages with their initials or first two letters):

- $P = \{C, V, Sh, I, Sc\}$ is the set of peers,
- $L = \{(C, V), (V, Sh), (V, I), (V, Sc)\}$ is the set of links among the peers,
- $M = \{o, oR, sR, sI, pI, sT, i, pS, sS\}$ is the set of messages, where the senders and the receivers for the messages are as follows:
  - $C = send(o) = recv(oR)$,
  - $V = recv(o) = send(oR) = send(sR) = recv(sI) = send(pI) = send(sT) = recv(i) = send(pS) = send(sS)$,
  - $Sh = send(sI) = recv(sR)$,
  - $I = recv(pI) = recv(sT) = send(i)$,
  - $Sc = recv(pS) = recv(sS)$,

Note that, we use $C = send(o) = recv(oR)$ to mean that the peer C is the sender of the message o and the receiver of the message oR.

- $E = \{(1, o, 1), (2, oR, 1), (A1, sR, 1), (A2, sI, 1), (B1, pI, 1), (B2, sT, 1), (B3, i, 1), (C1, pS, 1), (C2, sS, 1)\}$ is the set of events, and
- $D = \{ (e_1, e_{A1}), (e_{A1}, e_{A2}), (e_1, e_{B1}), (e_{A1}, e_{B2}), (e_{B1}, e_{B2}), (e_{B2}, e_{B3}), (e_1, e_{C1}), (e_{A2}, e_{C2}), (e_{C1}, e_{C2}), (e_{A2}, e_2), (e_{B3}, e_2), (e_{C2}, e_2)\}$ is the dependency relation (where we identify the events with their labels).

Given a collaboration diagram $\mathcal{D} = (P, L, M, E, D)$ with k threads, the event set E can be partitioned as $E = \bigcup_{i=1}^{k} E_i$ where E_i is the event set for thread i, and for all $i \neq j \Rightarrow E_i \cap E_j = \emptyset$. I.e., each event belongs to exactly one thread. As we mentioned above, the Purchase Order example has four threads E_1, E_2, E_3, E_4, that correspond to four sequence label prefixes: the empty prefix ($E_1 = \{e_1, e_2\}$), the prefix A ($E_2 = \{e_{A1}, e_{A2}\}$), the prefix B ($E_3 = \{e_{B1}, e_{B2}, e_{B3}\}$), and the prefix C ($E_4 = \{e_{C1}, e_{C2}\}$). Note that, although the set of events E is partially ordered, the set of events that belong to a single thread are totally ordered by definition. We have the following total orderings for each thread above: $E_1 : e_1 < e_2$, $E_2 : e_{A1} < e_{A2}$, $E_3 : e_{B1} < e_{B2} < e_{B3}$, $E_4 : e_{C1} < E_{C2}$.

Given a collaboration diagram $\mathcal{D} = (P, L, M, E, D)$ we denote the *set of conversations* defined by $\mathcal{D}$ as $\mathcal{C}(\mathcal{D})$ where $\mathcal{C}(\mathcal{D}) \subseteq M^*$.

Definition 1 *A conversation $\sigma = m_1 m_2 \ldots m_n$ is in $\mathcal{C}(\mathcal{D})$, i.e., $\sigma \in \mathcal{C}(\mathcal{D})$, if and only if $\sigma \in M^*$ and there exists a corresponding matching sequence of message send events $\gamma = e_1 e_2 \ldots e_n$ such that*

1. *for all $i \in [1..n]$ $e_i = (l_i, m_i, r_i) \in E$*
2. *for all $i, j \in [1..n]$ $(e_i, e_j) \in D \Rightarrow i < j$*
3. *for all $e \in E$ (for all $i \in [1..n]$ $e_i \neq e$) $\Rightarrow (e.r = * \vee e.r = ?)$*
4. *for all $e \in E$ if there exists $i, j \in [1..n]$ such that $i \neq j \wedge e_i = e_j$ then $e_i.r = *$.*

The first condition above ensures that each message in the conversation σ is equal to the message of the matching send event in the event sequence γ. The second condition ensures that the ordering of the events in the event sequence γ does not violate the dependencies in D. The third condition ensures that if an event does not appear in the event sequence γ then it must be either a conditional event or an iterative event. Finally, the fourth condition states that only iterative events can be repeated in the event sequence γ.

For example, a possible conversation for the collaboration diagram shown in Figure 1 is o, sR, sI, pS, pI, sS, sT, i, oR. The matching sequence of events for this conversation that satisfy all the four conditions listed above are: $(1, o, 1)$, $(A1, sR, 1)$, $(A2, sI, 1)$, $(C1, pS, 1)$, $(B1, pI, 1)$, $(C2, sS, 1)$, $(B2, sT, 1)$, $(B3, i, 1)$, $(2, oR, 1)$.

3 Execution Model

We model the behaviors of peers that participate to a collaboration as concurrently executing finite state machines that interact via messages [10,11]. We assume that the state machines can interact with both synchronous and asynchronous messages. We assume that each state machine has a single FIFO input queue for asynchronous messages. A send event for an asynchronous message appends the message to the end of the input queue of the receiver, and a receive event for an asynchronous message removes the message at the head of the input queue of the receiver. The send and receive events for synchronous messages are executed simultaneously and synchronous message

transmissions do not change the contents of the message queues. We assume reliable messaging, i.e., messages are not lost or reordered during transmission.

Formally, given a set of peers $P = \{p_1, \ldots, p_n\}$ that participate in a collaboration, the peer state machine for the peer $p_i \in P$ is a nondeterministic FSA $\mathcal{A}_i = (M_i, T_i, s_i, F_i, \delta_i)$ where $M_i = M_i^A \cup M_i^S$ is the set of messages that are either received or sent by p_i, T_i is the finite set of states, $s_i \in T$ is the initial state, $F_i \subseteq T$ is the set of final states, and $\delta_i \subseteq T_i \times (\{!, ?\} \times M_i \cup \{\epsilon\}) \times T_i$ is the transition relation. A transition $\tau \in \delta_i$ can be one of the following three types: (1) a send-transition of the form $(t_1, !m, t_2)$ which sends out a message $m \in M_i$ from peer $p_i = send(m)$ to peer $recv(m)$, (2) a receive-transition of the form $(t_1, ?m, t_2)$ which receives a message $m \in M_i$ from peer $send(m)$ to peer $p_i = recv(m)$, and (3) an ϵ-transition of the form (t_1, ϵ, t_2).

Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be the peer state machines (implementations) for a set of peers $P = \{p_1, \ldots, p_n\}$ that participate in a collaboration where $\mathcal{A}_i = (M_i, T_i, s_i, F_i, \delta_i)$ is the state machine for peer p_i. A *configuration* is a $(2n)$-tuple of the form $(Q_1, t_1, ..., Q_n, t_n)$ where for each $j \in [1..n]$, $Q_j \in (M_j^A)^*$, $t_j \in T_j$. Here t_i, Q_i denote the state and the queue contents of the peer state machine $\mathcal{A}_i$ respectively. For two configurations $c = (Q_1, t_1, ..., Q_n, t_n)$ and $c' = (Q'_1, t'_1, ..., Q'_n, t'_n)$, we say that c *derives* c', written as $c \rightarrow c'$, if one of the following three conditions hold:

- One peer executes an *asynchronous send* action (denoted as $c \xrightarrow{!m} c'$), i.e., there exist $1 \leq i, j \leq n$ and $m \in M_i^A \cap M_j^A$, such that, $p_i = send(m)$, $p_j = recv(m)$ and:
  1. $(t_i, !m, t'_i) \in \delta_i$,
  2. $Q'_j = Q_j m$,
  3. $Q_k = Q'_k$ for each $k \neq j$, and
  4. $t'_k = t_k$ for each $k \neq i$.
- One peer executes an *asynchronous receive* action (denoted as $c \xrightarrow{?m} c'$), i.e., there exists $1 \leq i \leq n$ and $m \in M_i^A$, such that, $p_i = recv(m)$ and:
  1. $(t_i, ?m, t'_i) \in \delta_i$,
  2. $Q_i = mQ'_i$,
  3. $Q_k = Q'_k$ for each $k \neq i$, and
  4. $t'_k = t_k$ for each $k \neq i$.
- Two peers execute *synchronous send* and *receive* actions (denoted as $c \xrightarrow{!?m} c'$), i.e., there exist $1 \leq i, j \leq n$ and $m \in M_i^S \cap M_j^S$, such that, $p_i = send(m)$, $p_j = recv(m)$ and:
  1. $(t_i, !m, t'_i) \in \delta_i$,
  2. $(t_j, ?m, t'_j) \in \delta_j$,
  3. $Q_k = Q'_k$ for each k, and
  4. $t'_k = t_k$ for each $k \neq i$ and $k \neq j$.
- One peer executes an ϵ-action (denoted as $c \xrightarrow{\epsilon} c'$), i.e., there exists $1 \leq i \leq n$ such that:
  1. $(t_i, \epsilon, t'_i) \in \delta_i$,
  2. $Q_k = Q'_k$ for each $k \in [1..n]$, and
  3. $t'_k = t_k$ for each $k \neq i$.

Now we can define the runs of a set of peer state machines participating in a collaboration as follows:

Definition 2 *Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be a set of peer state machines for the set of peers $P = \{p_1, \ldots, p_n\}$ participating in a collaboration, a sequence of configurations $\gamma = c_0 c_1 \ldots c_k$ is a* run *of $\mathcal{A}_1, \ldots, \mathcal{A}_n$ if it satisfies the first two of the following three conditions, and γ is a* complete run *if it satisfies all three conditions:*

1. *The configuration $c_0 = (\epsilon, s_1, \ldots, \epsilon, s_n)$ is the initial configuration where s_i is the initial state of $\mathcal{A}_i$ for each $i \in [1..n]$.*
2. *For each $j \in [0..k-1]$, $c_j \rightarrow c_{j+1}$.*
3. *The configuration $c_k = (\epsilon, t_1, \ldots, \epsilon, t_n)$ is a final configuration where t_i is a final state of $\mathcal{A}_i$ for each $i \in [1..n]$.*

Definition 3 *Given a run γ the* conversation *generated by γ, denoted by $\mathcal{C}(\gamma)$ where $\mathcal{C}(\gamma) \in M^*$, is defined inductively as follows:*

- *If $|\gamma| \leq 1$, then $\mathcal{C}(\gamma)$ is the empty sequence.*
- *If $\gamma = \gamma' c c'$, then*
  - *$\mathcal{C}(\gamma) = \mathcal{C}(\gamma' c) m$ if $c \stackrel{!m}{\rightarrow} c'$*
  - *$\mathcal{C}(\gamma) = \mathcal{C}(\gamma' c) m$ if $c \stackrel{!?m}{\rightarrow} c'$*
  - *$\mathcal{C}(\gamma) = \mathcal{C}(\gamma' c)$ otherwise.*

A sequence σ is a *conversation* of a set of peer state machines $\mathcal{A}_1, \ldots, \mathcal{A}_n$, denoted as $\sigma \in \mathcal{C}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$, if there exists a complete run γ such that $\sigma = \mathcal{C}(\gamma)$, i.e., a conversation of a set of peer state machines must be a conversation generated by a complete run. The *conversation set* $\mathcal{C}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ of a set of peer state machines $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is the set of conversations generated by all the complete runs of $\mathcal{A}_1, \ldots, \mathcal{A}_n$.

We call a set of peer state machines $\mathcal{A}_1, \ldots, \mathcal{A}_n$ *well-behaved* if each run of $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is a prefix of a complete run. Note that, if a set of peer state machines are well-behaved then the peers never get stuck (i.e., each peer can always consume all the incoming messages in its input queue and reach a final state).

Let $\mathcal{D}$ be a collaboration diagram. We say that the peer state machines $\mathcal{A}_1, \ldots, \mathcal{A}_n$ *realize* $\mathcal{D}$ if $\mathcal{C}(\mathcal{A}_1, \ldots, \mathcal{A}_n) = \mathcal{C}(\mathcal{D})$. A collaboration diagram $\mathcal{D}$ is *weakly realizable* if there exists a set of peer state machines which realize $\mathcal{D}$. $\mathcal{D}$ is *(strongly) realizable* if the peer state machines that realize it are well-behaved.

4 Realizability of Collaboration Diagrams

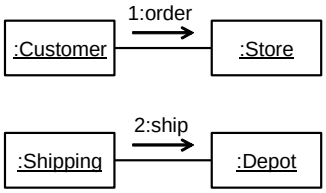

Fig. 3 Unrealizable collaboration diagram.

Not all collaboration diagrams are realizable. For example, Figure 3 shows a simple collaboration diagram that is not realizable. The conversation set specified by this collaboration diagram is $\{$*order ship*$\}$, i.e. this collaboration diagram specifies a single conversation in which, first, the Customer has to send the *order* message to the store, and then the Shipping department has to send the *ship* message to the Depot. However, this conversation set cannot be generated by any implementation of these peers. Any set of peer state machines which generates the conversation "*order ship*"

will also generate the conversation "*ship order*". The Shipping department has no way of knowing when the *order* message was sent to the Store, so it may send the *ship* message before the *order* message which will generate the conversation "*ship order*". Since the conversation "*ship order*" is not included in the conversation set of the collaboration diagram shown in Figure 3, this collaborations diagram is not realizable. To resolve this problem we have to require that, after receiving the *order* message, the Store sends a message to the Shipping department to inform it. The collaboration diagram shown in Figure 4 includes this fix and its conversation set $\{$*order orderInfo ship*$\}$ is realizable.

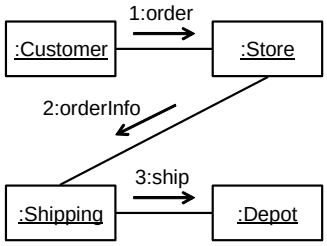

Fig. 4 Realizable collaboration diagram.

Figure 5 shows another simple collaboration diagram that is not realizable. The conversation set specified by this collaboration diagram is $\{$*order bill*$\}$, i.e. the only conversation specified by this collaboration diagram requires that the Customer sends the *order* message to the Store first and then the Accounting department sends the *bill* message to the Customer. Similar to the earlier example, this conversation set cannot be generated by any implementation of these peers. Any set of peer state machines which generates the conversation "*order bill*" will also generate the conversation "*bill order*". This time the Accounting department has no way of knowing when the *order* message was sent to the Store, so it may send the *bill* message before the *order* message which will generate the conversation "*bill order*", and since that is not included in the conversation set the collaboration diagram is not realizable. Similar to the example above, we can fix this problem if the Store sends a message to the Accounting department to inform it after it receives the *order* message as shown in Figure 6.

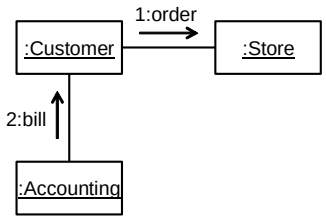

Fig. 5 Unrealizable collaboration diagram.

It is not too difficult to figure out realizability of the simple collaboration diagrams shown in Figure 3, Figure 4, Figure 5, and Figure 6. However, it is not that straightforward to figure out the realizability of the collaboration diagram shown in Figure 1. In the next section we will define a

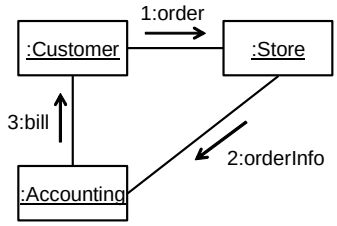

Fig. 6 Realizable collaboration diagram.

set of conditions which can be used to determine realizability of collaborations diagrams automatically. Based on these conditions we can automatically show that the collaboration diagram shown in Figure 1 is realizable.

5 Sufficient Conditions for Realizability

In this section we give sufficient conditions for realizability of a class of collaborations diagrams that are common in practice. Recall that the events in a collaboration diagram can be partitioned to a set of threads, where the set of events in each thread is totally ordered. In the class of collaboration diagrams we focus on, each message can appear only in one thread. After formally defining this class of collaboration diagrams we give sufficient conditions for their realizability.

We call a collaboration diagram *separated* if each message appears in the event set of only one thread, i.e., given a separated collaboration diagram $\mathcal{D} = (P, L, M, E, D)$ with k threads, the event set E can be partitioned as $E = \bigcup_{i=1}^{k} E_i$ where E_i is the event set for thread i, $M_i = \{e.m \mid e \in E_i\}$ is the set of messages that appear in the event set E_i and $i \neq j \Rightarrow M_i \cap M_j = \emptyset$. Recall that, the events in each E_i are totally ordered since they belong to the same thread. Note that dependencies among the events of different threads are still allowed in separated collaboration diagrams. The collaboration diagrams in Figure 1, Figure 3, Figure 4, Figure 5, and Figure 6 are separated whereas the collaboration diagram in Figure 7 is not separated. Based on our experience, requiring a collaboration diagram to be separated is not a big restriction. So far, all the collaboration diagrams we have seen in the literature have been separated collaboration diagrams.

Definition 4 *Given an event $e = (l, m, r)$ in a collaboration diagram $\mathcal{D} = (P, L, M, E, D)$, we call the event e* well-informed *if for all $e' = (l', m', r')$ such that $(e', e) \in D$ (i.e., for each event e' which is an immediate predecessor of e) one of the following conditions hold:*

1. *$e = e_I$ i.e., e is an initial event of $\mathcal{C}$, or*
2. *$r' = 1$ or $m' \in M^S$, and $send(m) \in \{recv(m'), send(m')\}$, or*
3. *$r' \neq 1$ and $m' \in M^A$ and $send(m) = send(m')$ and $recv(m) = recv(m')$ and $m \neq m'$ and $r = 1$.*

We call a collaboration diagram *well-informed* if all of its events are well-informed.

According to the above definition, there are three types of well-informed events. First, all the initial events are well-informed since they do not have any immediate predecessors. Second, if an immediate predecessor of an event e is either a synchronous message send event, or if it is not a conditional or iterative event, then for e to be well-informed, the sender of the message for e has to be either the receiver or the sender of the message for its immediate predecessor. Finally, if an immediate predecessor of an event e is either a conditional or an iterative asynchronous message

send event, then, to be well-informed, e cannot be a conditional or iterative event and it must have the same sender and the receiver but a different message than its immediate predecessor.

Next, we show that if a separated collaboration diagram is well informed (i.e., if all the events of a separated collaboration diagram are well-informed), then the collaboration diagram can be realized by a set of deterministic finite state peer implementations. The construction of such peer implementations is defined as below.

Definition 5 *Given a separated and well-informed collaboration diagram $\mathcal{D} = (P, L, M, E, D)$, for each peer $p_i \in P$, its projection $\mathcal{A}_i$ can be constructed as follows. $\mathcal{A}_i = (M_i, S_i, s_i, F_i, \delta_i)$ is a finite state machine, where M_i,$S_i = 2^{E_i}$,$s_i = E_i$, $F_i = \{\emptyset\}$, and δ_i are the alphabet, set of states, initial state, set of final states, and transition relation, respectively. Here $E_i \subseteq E$ is the set of events that are received or sent by peer p_i. For each state $s \subseteq E_i$ and for each event $e \in s$ such that for each $e' \in pred(e) \cap s$, $pred(e') \cap s = \emptyset$ and either $e'.l =?$ or $e'.l = *$:*

- *if $e.l = 1$ there exists one transition $(s, e.m, s - \{e\} - \{e' \mid (e', e) \in D \wedge (e'.l =? \vee e'.l = *)\})$.*
- *if $e.l =?$ there exists one transition $(s, e.m, s - \{e\})$.*
- *if $e.l = *$ there exists one transition $(s, e.m, s)$.*

We call an event e enabled *at a state s if there exists a transition $(s, e.m, s')$.*

Each state s of $\mathcal{A}_i$ is uniquely represented by the set of events that have not yet been completed when the automaton reaches s. Clearly, the initial state s_i is represented by E_i because initially none of the events in E_i have been executed. The set of final states $F_i = \{\emptyset\}$ is a singleton set that contains one final state, and the final state is represented by an empty set of events (i.e., all the events related to p_i have been executed). At each state s, an event e is "*enabled*" if none of its single-message predecessors appear in the state and all of its conditional and iterative immediate predecessors are enabled at s. When an event is executed, it is removed from the event set representation of the next state (except for iterative events). If there are any iterative or conditional immediate predecessors, they will also be removed from the next state. Notice that only a single message event can have conditional or iterative immediate predecessors according to the well-informedness condition.

Lemma 1 *Each peer projection defined in Definition 5 is deterministic.*

Proof: The lemma can be proved by contradiction. Let $\mathcal{A}_i = (M_i, S_i, s_i, F_i, \delta_i)$ be a peer projection of a separated and well-informed collaboration diagram. Assume $\mathcal{A}_i$ is nondeterministic, then there must be two transitions $(s, e_1.m, s')$ and $(s, e_2.m, s'')$ such that $e_1 \neq e_2$ and $s' \neq s''$ but $e_1.m = e_2.m$. This is because according to Definition 5, for each event e and each state s there is at most one outgoing transition (from s) defined for e. Now, given $e_1.m = e_2.m$, according to the separated property, both e_1 and e_2 belong to the same thread and there exists a total order between these two events. Without loss of generality, let $e_1 < e_2$. Since both e_1 and e_2 are enabled at state s, by Definition 5, e_1 has to be an immediate predecessor of e_2 and e_1 is a conditional or an iterative event. This contradicts with condition 3 of the well-informed property that a conditional or iterative event and its immediate successor event cannot be for the same message. □

In order to prove some of the results below we augment each message that is sent with a "send event" attribute. This auxiliary attribute does not change the peer execution semantics since it does not influence any of the peer operations. It is only an auxiliary attribute that tracks the send event that corresponds to a message and enables us to reason about the correspondence between the send and receive events.

Definition 6 *Given a message m, its "send event" attribute, denoted as $m.se$ is the send event executed by $send(m)$ when m is sent out.*

Lemma 2 *Let $\mathcal{D}$ be a separated and well-informed collaboration diagram. Let γ be a run of all its peer projections. The following four propositions are true:*

- **P1 (Stuckness Freedom).** *For each configuration c_a in γ, let Q_i and s_i be the queue and local state of peer p_i at c_a. Let $Set(Q_i) = \{e \mid \text{there is a message } m \text{ in } Q_i \text{ s.t. } m.se = e\}$, then $Set(Q_i) \subseteq s_i$. (For each message in queue Q_i the corresponding receive event is in the state s_i.)*
- **P2 (Send-Receive Correspondence).** *For each receive action $c_a \xrightarrow{?m} c_{a+1}$ in γ, let $(s, e.m, s')$ be the transition taken at the receiver, then $e = m.se$. (For each message, the receive event matches the send event.)*
- **P3 (Strong Stuckness Freedom).** *For each configuration c_a in γ, for each peer p_i, if its queue has messages, let m be the head element of Q_i and s_i its local state at c_a, then $m.se$ is enabled at s_i.*
- **P4 (Dependency Relation Conformance).** *For any send event $c_t \xrightarrow{!m} c_{t+1}$ and $m.se = e$, for each predecessor $e' \in pred(e)$ where $e'.l = 1$, there must be an action $c_a \xrightarrow{!m'} c_{a+1}$ in γ where $m'.se = e'$ and $a < t$.*

Proof: We prove **P1** to **P4** by induction on the length of γ. The proofs of all the base cases are trivial because initially there all the queues are empty. The induction assumption is that for any run whose length is smaller than n, propositions **P1** to **P4** hold.

Now, let us look at a run γ with length n. Let m be the message for its *last* action, which can be either a send or a receive action (since we do not have any ϵ transitions in the peers constructed according to Definition 5). Let p_i and p_j be the sender and the receiver of m, respectively, and let $e = m.se$. We use apostrophe to represent the properties of the last configuration c_n. For example if Q_i is the queue contents of p_i at c_{n-1}, then Q_i' is that of p_i at c_n. Similarly the states of p_i are defined as s_i and s_i'.

P1 (Stuckness Freedom). Case 1: Consider the case where the last action is a send action. Clearly, we only need to consider p_i and p_j because all other peers have their states and queues intact, and hence the induction assumption applies. For sender p_i, we have three cases to consider: $e.l = 1$, $e.l =?$, and $e.l = *$. When $e.l = 1$, according to Definition 5, $s_i' = s_i - \{e\} - \{e' \mid (e', e) \in D \wedge (e'.l =? \vee e'.l = *)\}$, where e' (if there is any) is a conditional or iterative event that is an immediate predecessor of e. Note that according to Condition 3 of well-informedness, both e and e' are outgoing messages of p_i, hence they do not belong to Q_i. Therefore, given that $Set(Q_i) \subseteq s_i$ (induction assumption), it is true that $Set(Q_i) \subseteq s_i - \{e\} - \{e' \mid (e', e) \in D \wedge e'.l =? \vee e'.l = *\}$. This immediately leads to $Set(Q_i') = Set(Q_i) \subseteq s_i'$. The cases $e.l =?$ and $e.l = *$ are proved similarly.

Next consider the receiver p_j of the last send action of γ. Clearly, $Set(Q_j') = Set(Q_j) \cup \{e\}$ because the message m is appended to FIFO queue of p_j (note: $e = m.se$). The state s_j' is equal to s_j. Now, given the induction assumption $Set(Q_j) \subseteq s_j$, to prove that $Set(Q_j') \subseteq s_j'$, $e \in s_j$ is required. It can be proved by contradiction. Assume that e is not in s_j, then we also have three cases to consider:

1. If $e.l = 1$, since $e \notin s_j$, by Definition 5, peer p_j must have taken some transition $(s, e.m, t)$ that executes the receive event e. According to the induction assumption of **P2**, the receive event $?e$ has a corresponding send event $!e$ at peer p_i, which is executed before c_{n-1}. Now we know that p_i executes another $!e$ at c_{n-1} (which is the last action of γ), hence $!e$ is executed at least twice at p_i. This already contradicts with Definition 5 because a single message send event can be executed at most once (a single message send event is removed from the peer state once it is executed and cannot be enabled again, according to Definition 5).

2. If $e.l =?$, the argument is the same according to Definition 5.
3. If $e.l = *$, the only way that p_j can remove e from its local state is to execute a dependent action $?e'$ of e (s.t. $(e, e') \in D$ and $e'.l = 1$ and the sender/receiver of e' match those of e), according to Definition 5. By induction assumption of **P2**, $?e'$ must have a corresponding send action $!e'$ at p_i. Once $!e'$ is executed at p_i, according to Definition 5, e is removed from local state of p_i (and hence not enabled any more). Then there is no way that p_i can execute $!e$ at c_{n-1} (which is after $!e'$). This contradicts with the fact that e is sent at c_{n-1}.

Case 2: Consider the case where the last action is a receive action. Only receiver p_j needs to be considered because other peers' state and queue do not change. We also have three cases to examine: $e.l = 1$, $e.l =?$, and $e.l = *$. If $e.l = 1$, according to Definition 5, $!e$ will be executed at most once at its sender, hence there is no second message in Q_j with the send attribute equal to e. Given $Q_j = e.m\ Q'_j$, $Set(Q'_j) = Set(Q_j) - \{e\}$. Now given the induction assumption $Set(Q_j) \subseteq s_j$, and $Set(Q'_j) = Set(Q_j) - \{e\}$, and $s'_j = s_j - \{e\} - \{e' \mid (e', e) \in D \ \wedge\ (e'.l =? \ \vee\ e'.l = *)\}$, to prove that $Set(Q'_j) \subseteq s'_j$ we only need to prove that $\{e' \mid (e', e) \in D \ \wedge\ (e'.l =? \ \vee\ e'.l = *)\}$ is not a subset of $Set(Q_j)$. The proof is by contradiction. If there is such an event e' in $Set(Q_j)$, according to induction, there is a message m' after m in queue such that $m'.se = e'$. According to the FIFO queue property, this implies that at sender p_i, an event $!e'$ is executed after $!e$ and e' is an immediate predecessor of e. This is not possible according to Definition 5. When $e.l =?$ or $e.l = *$, the argument is similar and simpler because we do not have to consider conditional/iterative predecessors.

P2 (Send-Receive Correspondence). We only need to consider the receive action case, because send actions are not related. Now consider the last action $c_{n-1} \xrightarrow{?m} c_n$, peer p_j is taking a transition $(s_j, e'.m, s'_j)$. We need to prove that $e' = e$ where e is defined as $m.se$. Now according to induction assumption of **P3**, e is enabled at s_j. Then by Lemma 1, for each state, there is only one outgoing transition labeled for $e.m$ and it is the transition to consume e. Hence, we have proved that $e' = e$.

P3 (Strong Stuckness Freedom). We prove this via contradiction. Assume that at c_n (the last configuration of γ), letting $m = Q_j[0]$ and $e = m.se$, e is not enabled at s_j of peer p_j. Note that by induction assumption of **P1**, e is contained in s_j. Then according to Definition 5, there are two cases left to discuss:

1. Case 1: There is a $e' \in pred(e) \cap s_j$ and $e'.l = 1$. Note that $!e$ has been executed at $send(e.m)$. According to induction assumption of **P4**, all of e's single message predecessors have been executed. Hence $!e'$ has been executed (for exactly once according to Definition 5), and it is executed before $!e$. Let $m' = e'.m$. Clearly m' arrives at p_j earlier than m. Since m is the head of queue, and queue is FIFO, at peer p_j, m' must have been consumed. Now by induction assumption of **P2**, the transition taken at p_j to consume m' must be for receiving event $e' = m'.se$. By Definition 5, such transition removes e' from the local state of p_j forever (since $e'.l = 1$), and e' cannot be contained in the local state of p_j at c_n, which contradicts with the assumption.
2. Case 2: There is a $e' \in pred(e) \cap s_j$, and $e'.l =?$ or $e'.l = *$, and $pred(e') \cap s_j \neq \emptyset$. According to Condition 3 of well-informedness, for each immediate predecessor $e'' : (e'', e') \in D$, $e''.l = 1$. Clearly $e'' \in pred(e)$, we simply have to apply argument in Case 1 for the contradiction.

P4 (Dependency Relation Conformance). First, consider all the immediate predecessors of e. Let $e.m$ be a message from p_i to p_j. According to the well-informedness property, all immediate predecessors of e either have p_i as sender or receiver. So they must appear in the initial state of $\mathcal{A}_i$. According to the construction algorithm of $\mathcal{A}_i$, all immediate predecessors of e have to be removed from local state p_i before executing $!e$. Given an immediate predecessor e' of e, if $e'.l = 1$, the only way to remove it from p_i's state is to execute it. There are two cases to consider. Case 1: e' is a send

event. Now by induction assumption of **P4** itself, all of the single message send events of e' have been sent. Case 2: e' is a receive event. According to the induction assumption of **P2**, there exists a corresponding send event for e' at its sender. Then by applying induction assumption again, we have all single message send predecessors of e' already executed. Now the last situation to consider is $e'.l =?$ or $e'.l = *$. According to well-informedness requirement, for any predecessor e'' of e', $e''.l = 1$. We apply similar arguments to e'' and can prove that all single message predecessors of e'' have been executed. Now we have exhausted all possible situations for e, and by induction we have that all single message send events of e have been executed. □

Theorem 1 *A separated collaboration diagram $\mathcal{D} = (P, L, M, E, D)$ is weakly realizable if all the events $e \in E$ are well-informed.*

Proof: Construct peer implementations according to Definition 5, and let them be $\mathcal{A}_1, \cdots, \mathcal{A}_n$. First we prove that every conversation defined by $\mathcal{D}$ can be generated by the composition of $\mathcal{A}_1, \cdots, \mathcal{A}_n$. This proof is straight-forward because given any send event sequence $\gamma = e_1 e_2 \cdots e_k$, we can always make the peer implementations simulate a conversation generated by the collaboration diagram by following the send actions step by step and executing a receive action immediately after the corresponding send action (and making receiver to receive a message simultaneously with the sender if the message is synchronous).

Next we prove that every conversation generated by peer implementation belongs to the conversation set defined for $\mathcal{D}$, i.e., satisfies the four conditions in the conversation definition. The proof of the first condition "for all $i \in [1..n]$ $e_i \in E$" is trivial according to Definition 5. The proof of condition 3 is also trivial. By Definition 5, for each single message event e, it is removed from the local state of its sender only if it is executed. Since at the end of complete run, the final state of all peers is an empty set of events, thus all single message events have been executed which implies condition 3. Proof of condition 4 is also based on Definition 5 – once a single or conditional message event is executed, it is removed from the local state of its sender hence it will not be executed a second time. This directly implies condition 4.

Finally, we prove Condition 2 by contradiction. Assume that there are two send events $!e_1$ and $!e_2$ such that $e_1 \in pred(e_2)$, however, during the run $!e_2$ is executed before $!e_1$. If $e_1.l = 1$, this directly contradicts with **P4** of Lemma 2. If $e_1.l = *$ *or* $?$, study the configuration where e_2 is sent. If e_1 is the immediate predecessor of e_2, then e_1 and e_2 are sent by the same peer. According to the construction algorithm in Definition 5, e_2 cannot be sent before e_1. Now the only case to consider is that $e_1 \in pred(e_2)$ however $(e_1, e_2) \notin D$. According to well-informedness, there must be another event e' s.t. $(e_1, e') \in D$ and $e' \in pred(e_2)$. Note that $e'.l = 1$. By **P4** of Lemma 2, $!e'$ must be executed before $!e$. When peer $send(e'.m)$ sends e', it also removes e_1 from its local state, according to Definition 5, and it disables e_1 at sender forever. Thus, e_1 cannot be sent by $send(e_1.m)$ after e_2 is sent (which is after $!e'$). Up to now, all cases have been examined and contradiction is established for each case. Hence, the assumption that e_1 is sent after e_2 cannot be true. Condition 2 is proved. □

Next we show that each conversation generated by peer projections can be expanded into a complete conversation (hence no deadlock and no stuckness). This allows us to prove the strong realizability.

Lemma 3 *Let $\mathcal{D}$ be a separated and well-informed collaboration diagram. Let $\mathcal{A}_1, \cdots, \mathcal{A}_n$ be the peer implementations in Definition 5. Let γ be a run of $\mathcal{A}_1, \cdots, \mathcal{A}_n$, and let c_n be the last configuration of γ. The following is true for the local state s_i of each peer p_i at c_n:*

$$s_i = E_i - \pi_i(E(\gamma)) - \pi_i(pred(E(\gamma))) \\ +\{e \mid e \in \pi_i(E(\gamma)) \ \wedge \ e.l = * \ \wedge \ succ(e) \cap E(\gamma) = \emptyset\} \tag{1}$$

where $E(\gamma)$ is set of send/receive events executed during γ and $\pi_i(E(\gamma))$ is the projection of $E(\gamma)$ to the event set E_i of peer p_i such that each event e in $\pi_i(E(\gamma))$ is executed by p_i (notice that if a peer p_a sends a message m to p_j, the send event $!m.se$ is not projected to peer p_j). For convenience, we call the items on the right hand side of the equation components 1,2,3,4. For example, $\pi_i(pred((E(\gamma)))$ is component 3.

Proof: We prove the lemma by induction on the length of γ. Assume that for all runs whose length is smaller than or equal to $n-1$, the lemma holds. Now consider the last action, we need to consider two cases: send and receive.

Case 1: the last action $c_{n-1} \stackrel{!m}{\rightarrow} c_n$ is a send event and let $e = m.se$. We do not have to consider peers other than $send(m)$ because the state of other peers are not changed. Consider $send(m)$, let s_i and s_i' be its local state at c_{n-1} and c_n respectively.

1. if $e.l = 1$: According to Definition 5, $s_i' = s_i - \{e\} - \{e' \mid (e', e) \in D \wedge (e'.l =? \vee e'.l = *)\}$. Clearly, $E(\gamma')$ = $E(\gamma\ e) = E(\gamma) \cup \{e\}$, and each e' (if there is any) is included in $pred(E(\gamma'))$. Now given induction assumption that Equation 1 holds at c_{n-1}, let us observe the changes on both sides of the equation for c_n. The left side has e and all its immediate conditional/iterative predecessors removed. On the right side, component 2 has e added, component 4 does not change. Component 3 has all predecessors of e included; however, notice that only the immediate conditional/iterative predecessors of e and e itself could be new members of the union of component 2 and 3, because according to **P4** of Lemma 2 all single-message event predecessors of e have already been contained in component 2 (which already includes immediate predecessors of each e'). Hence, we have shown that both sides of equation has e and its immediate conditional/iterative predecessors removed. Equation 1 still holds at c_n.
2. if $e.l =?$: the proof is similar to the above except that we do not have to consider the immediate/iterative conditional predecessor of e. Well-informedness property guarantees that the recurrence type of each immediate predecessor of e is 1.
3. if $e.l = *$: the proof is similar except that e is still included in s_i' due to component 4 in Equation 1. Note that the proof also relies on **P4** of Lemma 2 for arguing that predecessors of e are already contained in $\pi_i(E(\gamma))$ at c_{n-1}.

Now consider the receiver side. For $recv(m)$ its local state does not change (on the left hand side of Equation 1). None of the components on the right hand side is changed either. Notice that the $!e$ is not contained in $\pi_j(E(\gamma))$ because $recv(m)$ did not execute $!e$ (only $?e$ will be projected).

Case 2: the last action $c_{n-1} \stackrel{?m}{\rightarrow} c_n$ is a receive event and let $e = m.se$. Only $recv(m)$ needs to be considered because its state is affected. We also have three cases to discuss. Let s_j and s_j' be the local state of $recv(m)$ at c_{n-1} and c_n respectively. When $e.l = 1$, $s_j' = s_j - \{e\} - \{e' \mid (e', e) \in D \wedge e'.l = ?\ or\ *\}$. Clearly, the receive event $?e$ will be contained in component 2 in the new equation, and e'(s) will be contained in component 3. By induction assumption, both sides of Equation 1 have e and e' removed, so the equation still holds at s_j'. When $e.l =?$ and $e.l = *$, the proof is similar. □

Theorem 2 *Let $\mathcal{D}$ be a separated and well-informed collaboration diagram, and $\mathcal{A}_1, \cdots, \mathcal{A}_n$ be the peer implementations in Definition 5. Let γ be a run of $\mathcal{A}_1, \cdots, \mathcal{A}_n$, if γ is not a complete run it can always be expanded into a complete run.*

Proof: Let γ be an incomplete run, according to **P3** of Lemma 2: at any moment of a run each queue head is always enabled at its receiver, we can append a series of receive actions to γ until all messages in queues of every peer are consumed. Let the expanded run be γ'. Clearly the following argument (let it be **A1**) is true: in γ' each receive event has a corresponding send event (according to **P2** of Lemma 2), and each send event has a corresponding receive event (because all messages

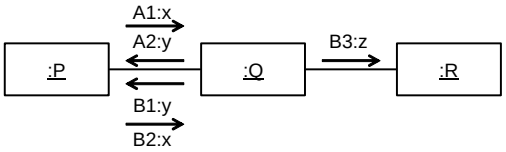

Fig. 7 Unrealizable collaboration diagram with well-informed events.

in queues have been consumed). Now let us study the status of each peer at the end of γ'. Let $E' = \pi_{send}(E(\gamma)) + \pi_{send}(pred(E(\gamma))) - \{e \mid e \in \pi_{send}(E(\gamma)) \ \wedge \ e.l = * \ \wedge \ succ(e) \cap \pi_{send}(E(\gamma)) = \emptyset\}$. Clearly E' represents the events which have been completely executed by γ'. We can construct a new collaboration diagram (let it be $\mathcal{D}'$) with its dependency graph created from that of $\mathcal{D}$ by removing E' from E and corresponding dependency links. Also a new peer projection $\mathcal{A}'_i$ can be generated for each peer p_i, by making the state of p_i at the end of γ' the initial state. Study the initial state of $\mathcal{A}'_i$. Clearly if we project $E - E'$ to a peer p_i, the set $\pi_i(E - E')$ is equal to $E_i - \pi_i(E')$ which is

$$E_i - \pi_i(E(\gamma)) - \pi_i(pred(E(\gamma))) + \{e \mid e \in \pi_i(E(\gamma)) \ \wedge \ e.l = * \ \wedge succ(e) \cap E(\gamma) = \emptyset\}$$

Note that in the above formula, $send$ and $recv$ do not appear in the subscripts of π_i due to argument **A1**. The above formula is equal to the formula given in Lemma 3. This implies that when γ' is completed, all states of peers are exactly the initial states for the remaining collaboration diagram $\mathcal{D}'$. Then by applying Theorem 1, immediately we have that for $\mathcal{D}'$ there is a complete run (let it be γ'') generated by the modified peer projections $\mathcal{A}'_i (i \in [1..n])$. Obviously, concatenation of γ' and γ'' is the complete run for γ. □

Then by Theorems 1 and 2, we have the following.

Theorem 3 *A separated collaboration diagram $\mathcal{D} = (P, L, M, E, D)$ is (strongly) realizable if all the events $e \in E$ are well-informed.*

The realizability condition for separated collaboration diagrams given above can be checked in linear time. As mentioned above, the collaboration diagrams shown in Figure 3, Figure 4, Figure 5, and Figure 6 are all separated collaboration diagrams. However, the collaboration diagrams shown in Figure 3 and Figure 5 violate the realizability condition given above and they are not realizable, whereas the the collaboration diagrams shown in Figure 4 and Figure 6 satisfy the realizability condition given above and hence they are realizable.

Note that, in Figure 3 the sender for the final event (i.e., the event labeled 2) is the peer Shipping and this peer is not the receiver or the sender of the message for event 1 which is the immediate predecessor of event 2. Hence event 2 is not well-informed. However, in Figure 4 the sender for the final event (i.e., the event labeled 3) is the receiver of the message for event 2 which is the immediate predecessor of event 3. Hence, in Figure 4, the final event is well-informed. In fact all the events in Figure 4 are well-informed and therefore it is realizable.

Similarly, in Figure 3 the sender for the event 2 is Accounting and Accounting is not the receiver or the sender of the message for event 1 which is the immediate predecessor of event 2. Hence event 2 is not well-informed. However, in Figure 6 the sender for the event 3 is the receiver of the message for event 2 which is the immediate predecessor of event 3. In Figure 6 all events are well-informed and it is realizable.

Finally, the collaboration diagram shown in Figure 1 is realizable since all the events shown in Figure 1 are well-informed.

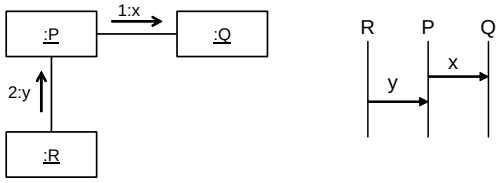

Fig. 8 A collaboration diagram with no corresponding Message Sequence Charts.

Now, we will give an example to show that well-informedness of the events alone does not guarantee realizability of a collaboration diagram which is not separated. Consider the collaboration diagram given in Figure 7. This collaboration diagram has two threads (A and B) and it is not separated since both threads have send events for messages x and y. Note that all the events in this collaboration diagram are well-informed. The conversation set specified by this collaboration diagram consists of all interleavings of the sequences xy and yxz which is the set $\{xyyxz, xyxyz, xyxzy, yxzxy, yxxzy, yxxyz, yxyxz\}$. However any set of peer state machines that generate this conversation set will either generate the conversation $xyzxy$ or will not be well-behaved. Consider any set of peer state machines that generate this conversation set. Consider the incomplete run in which first peer P sends x and then the peer Q sends y. From the peer Q's perspective there is no way to tell if y was sent first or if x was sent first. If we require peer Q to receive the message x before sending y (hence, ensuring that x is sent before y) then we cannot generate the conversations which start with the prefix yx. Hence, peer Q can continue execution assuming that the conversation being generated is $yxzxy$ and send the message z before peer P sends another message. Such a partial execution will generate the sequence xyz which is not the prefix of any conversation in the conversation set of the collaboration diagram. Therefore such a partial execution will either lead to a complete run and generate a conversation that is not allowed or it will not lead to any complete run, either of which violate the realizability condition.

Although well-informedness property is not a necessary condition for realizability of separated collaboration diagrams. It is a necessary condition for a more restricted class of collaboration diagrams. We call a collaboration diagram $\mathcal{C} = (P, L, M, E, D)$ *simple* if for all $e \in E$ $e.r = 1$. Then we have the following result:

Theorem 4 *A simple separated collaboration diagram $\mathcal{C} = (P, L, M, E, D)$ is realizable if and only if all the events $e \in E$ are well-informed.*

The "if" direction follows from Theorem 1. For the "only if" direction assume that there exists an event $e \in E$ which is not well-informed. Then there must be an immediate predecessor of event $e = (l, m, r)$, say event $e' = (l', m', r')$, such that $send(m) \notin \{recv(m'), send(m')\}$. Then we have $m \neq m'$ and for any implementation of the peers, sender of message m has no way of knowing if message m' has been sent. So it is always possible to get an interaction where message m is sent before message m', violating the dependency relation.

6 Related Work

Message Sequence Charts (MSCs) [12] provide another visual model for specification of interactions in distributed systems. MSC model has also been used in modeling and verification of web services [13]. As opposed to the collaboration diagrams which only specify the ordering of send

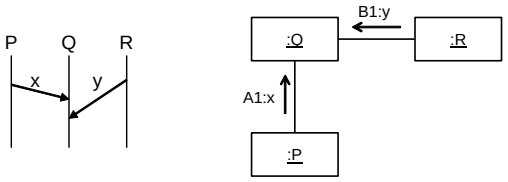

Fig. 9 A Message Sequence Chart with no corresponding collaboration diagram.

events, in the MSC model ordering of both send and receive events are captured. Another difference between the collaboration diagram model and the MSC model is the fact that MSC model gives a *local* ordering of the send and receive events whereas a collaboration diagram gives a *global* ordering of the send events. It is possible to show that there are collaboration diagrams which specify interactions that cannot be specified using MSCs and there are MSCs which specify interactions that cannot be specified using collaboration diagrams.

The examples in Figure 8 and Figure 9 demonstrate the differences between the MSC and collaboration diagram models. Consider the collaboration diagram shown in Figure 8 which states that the peer P should send the message x before peer R sends the message y. There is no way to express this ordering using a MSC since the senders of messages y and x are different. Even if peer P makes sure that it sends message x before it receives message y (as shown in Figure 8), this does not guarantee that message y is sent after message x is sent (note that these are asynchronous messages).

Figure 9, on the other hand, shows an MSC which specifies and ordering of send and receive events which cannot be specified using a collaboration diagram. The MSC in Figure 9 states that the peer Q should receive message x before it receives message y, however, it does not specify any ordering between the send events for messages x and y. The collaboration diagram in Figure 9 also leaves the ordering of send events for messages x and y unspecified, however, there is no way of restricting the ordering of the receive events in collaboration diagrams.

The realizability problem for MSCs [3] and its extensions such as high-level MSC (hMSC) [5] and MSC Graphs [4] have been studied before. However as we discussed above, the type of interactions specified by collaboration diagrams and MSCs are different.

There has been earlier work on using various UML diagrams in modeling different aspects of service compositions (for example [14–16]). However, we are not aware of any work that focuses on realizability of interactions specified as collaboration diagrams.

In [17], interactions among agents are represented using various UML diagrams, including collaboration diagrams, however, the realizability problem is not investigated. In [18, 19], Dooley graphs are used to model conversations. In [20] collaboration diagrams are used to represent Dooley graphs, and a formal coordination modeling approach for supply-chain management is proposed by using collaboration diagrams in conjunction with other UML diagrams, such as state diagrams. Some of the conditions on Dooley graphs presented in these earlier papers are similar to the realizability conditions presented in this paper. However, these earlier results do not address the realizability problem discussed in this paper. In fact, in [20], resolving technical issues in Dooley graph representation of conversations is left as future work. Moreover, the computational model we present in this paper is different and involves both synchronous and asynchronous communication, and the interaction model we use has both conditional and iterative send events.

In our earlier work we have studied the realizability of conversations specified using automata, called *conversation protocols* [6, 10, 21, 11, 7]. Conversations protocols provide a different model for specifying conversations. Unlike collaboration diagrams, conversation protocols allow specification

of arbitrary cycles. In fact, conversation protocols can be used for specification of any conversation set that is regular (i.e., that can be recognized by a finite state automaton). For example, given two messages x and y, the conversation set specified by the regular expression $(xy)^*$ can be easily specified using a conversation protocol. However, this conversation set cannot be specified using collaboration diagrams since the only loop construct in collaboration diagrams allows repetition of a single send event. In [22] we show that conversation protocols are more expressive than collaboration diagrams. The fact that collaboration diagrams provide a more restricted language for specification of interactions can also mean that one can find more efficient techniques for checking their realizability. In fact, the realizability condition for the collaboration diagrams given in this paper can be checked more efficiently than the realizability conditions for conversation protocols given in [10].

7 Conclusions

Analysis of interactions specified by collaborations diagrams is becoming increasingly important in the web services domain where autonomous peers interact with each other through messages to achieve a common goal. Since such interactions can cross organizational boundaries, it is necessary to focus on specification of interactions rather then the internal structure of individual peers. In this paper we argued that collaboration diagrams are a useful visual formalism for specification of interactions among web services. However, specification of interactions from a global perspective inevitably leads to the realizability problem. In this paper, we formalized the realizability problem for collaboration diagrams and gave sufficient conditions for realizability.

References

1. W3C: Web Service Choreography Description Language (WS-CDL). `http://www.w3.org/TR/ws-cdl-10/` (2005)
2. Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of the ACM **30**(2) (1983) 323–342
3. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. In: Proc. 22nd Int. Conf. on Software Engineering. (2000) 304–313
4. Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. In: Proc. 28th Int. Colloq. on Automata, Languages, and Programming. (2001) 797–808
5. Uchitel, S., Kramer, J., Magee, J.: Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. ACM Transactions on Software Engineering and Methodology **13**(1) (2004) 37–85
6. Bultan, T., Fu, X., Hull, R., Su, J.: Conversation specification: A new approach to design and analysis of e-service composition. In: Proc. 12th Int. World Wide Web Conf. (2003) 403–410
7. Bultan, T., Fu, X., Su, J.: Analyzing conversations of web services. IEEE Internet Computing **10**(1) (2006) 18–25
8. OMG: UML 2.0 superstructure specification. `http://ww.uml.org/` (2004)
9. OASIS: Web services business process execution language version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html` (2006)
10. Fu, X., Bultan, T., Su, J.: Conversation protocols: A formalism for specification and analysis of reactive electronic services. Theoretical Computer Science **328**(1-2) (2004) 19–37
11. Fu, X., Bultan, T., Su, J.: Synchronizability of conversations among web services. IEEE Transactions on Software Engineering **31**(12) (2005) 1042–1055
12. ITU-T: Message Sequence Chart (MSC). Geneva Recommendation Z.120 (1994)
13. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based verification of web service compositions. In: Proc. 18th IEEE Int. Conf. on Automated Software Engineering Conference. (2003) 152–163

14. Benatallah, B., Sheng, Q.Z., Dumas, M.: The self-serv environment for web services composition. IEEE Internet Computing **7**(1) (2003) 40–48
15. Skogan, D., Gronmo, R., Solheim, I.: Web Service Composition in UML. In: Proc. of 8th International IEEE Enterprise Distributed Object Computing Conference. (2004)
16. Blake, M.B.: A lightweight software design process for web services workflows. In: Proc. of the 2006 IEEE International Conference on Web Services. (2006) 411–418
17. Odell, J.J., Parunak, H.V.D., Bauer, B.: Representing Agent Interaction Protocols in UML. In: Proc. of First International Workshop on Agent-Oriented Software Engineering. (1999)
18. Parunak, H.V.D.: Visualizing agent conversations: Using enhanced Dooley graphs for agent design and analysis. In: Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS'96). (1996)
19. Singh, M.P.: Synthesizing coordination requirements for heterogeneous autonomous agents. Autonomous Agents and Multi-Agent Systems **3**(2) (2000) 107–132
20. Huhns, M.N., Stephens, L.M., Ivezic, N.: Automating supply-chain management. In: Proceedings of the First International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2002). (2002) 1017–1024
21. Fu, X., Bultan, T., Su, J.: Realizability of conversation protocols with message contents. International Journal of Web Services Research (JWSR) **2**(4) (2005) 68 – 93
22. Bultan, T., Fu, X.: Realizability of interactions in collaboration diagrams. Technical Report 2006-11, Computer Science Department, University of California, Santa Barbara (2006)