Realizability of Interactions in Collaboration Diagrams

Tevfik Bultan
Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
bultan@cs.ucsb.edu

Xiang Fu
School of Computer and Information Science
Georgia Southwestern State University
Americus, GA 31709, USA
xfu@canes.gsw.edu

Abstract

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

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 the web services domain 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 [2, 7, 11, 13]. Formalisms which focus on interactions rather than the local behaviors of individual peers are necessary for both specification and analysis of such distributed applications.

Web Services Choreography Description Language (WS-CDL) [18] is an XML-based language for describing the interactions among the peers participating to a web service. 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, characterization of interactions using a global view may lead to specification of behaviors that may not be 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 the specified interactions.

In order to study the realizability problem we give a formal model for collaborations diagrams. We model a distributed system as a set of communicating finite state machines [6]. A collaboration diagram is realizable if there exists a set of communicating finite state machines which generate exactly the set of interactions specified by the collaboration diagram. We present sufficient conditions for realizability for different classes of collaboration diagrams.

It is unlikely that all the interactions in a distributed system can be expressed using a single collaboration diagram. Hence, most of the time it is necessary to use a set of collaborations diagrams where each diagram specifies a subset of the possible interactions. We extend this idea to collaboration diagram graphs where the interactions specified by different collaboration diagrams can be combined in various ways (using union, concatenation or repetition). We show that collaboration diagram graphs are equivalent to conversation protocols [9, 10] and the realizability results

for conversation protocols can be adopted to realizability of collaboration diagram graphs.

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 interactions 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 interactions generated by such peers. In Section 4 we discuss the realizability of collaboration diagrams and give sufficient conditions for realizability of some classes of collaboration diagrams. In Section 5 we introduce the collaboration diagram graphs. We show that collaboration diagram graphs are more powerful than collaboration diagram sets. We show that collaboration diagram graphs can be translated to conversation protocols and the realizability conditions on conversation protocols can be used to determine realizability of collaboration diagram graphs. 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 specifying a set of interactions among a set of *peers*. Each peer is like an active object with its own thread of control. We will model the interactions specified by a collaboration diagram as a sequence of messages exchanged among these peers. 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 [17]) 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. 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 by listing their sequence labels (followed by the symbol "/") before the sequence label of the event. For example if an event is labeled with "B2,C3/A2" then A2 is the sequence label of the event, and the events B2, C3 and A1 must precede A2 (note that, A1 is not listed explicitly since it is implied by the implicit ordering).

The prefixes in sequence labels of collaboration diagrams enable specification of concurrent interactions where each prefix represents a *thread*. Note that, here by "thread", we do not mean a thread of execution. Rather, we are referring to a set 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 among 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 for the same message.

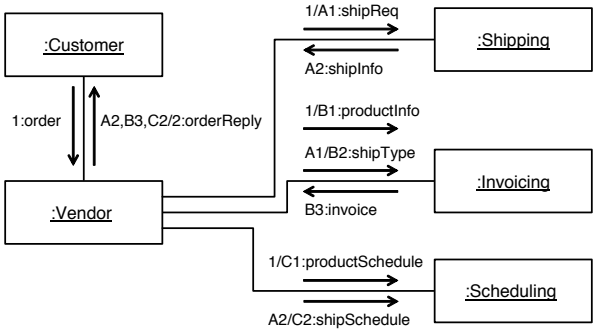

Figure 1. An example collaboration diagram for a composite web service. A vendor service processes a purchase order from a customer by delegating the tasks to three other services.

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 Ser-

vices (BPEL) 1.1 language specification [5]. In this example, a customer sends a purchase order to a vendor. The vendor calculates the price for the order including the shipping fee, arranges a shipment, and schedules the production and shipment. The vendor uses an invoicing service to calculate the price, a shipping service to arrange the shipment, 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.

2.1. A Formal Model for Collaboration Diagrams

Based on the assumptions discussed above we formalize the semantics of collaboration diagrams as follows. A *collaboration diagram* $C = (P, L, M, E, D)$ 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 tuple $e \in E$ 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 tuple in E has a unique label, i.e., for all $e, e' \in E$, $e \neq e' \Rightarrow e.l \neq e'.l$. Each tuple denotes a message send event where peer $send(m)$ sends a message m to peer $recv(m)$. 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). Note that, if there are circular dependencies among message send events, then it is not possible to find an ordering of the message send events satisfying the dependencies. Therefore, we do not allow circular dependencies. This condition can be checked in linear time before the analyses we discuss in this paper are performed.

We also assume that the dependency relation does not have any redundant dependencies. 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 $i \in [1..k-1]$, $(e_i, e_{i+1}) \in D$. A dependency $(e', e) \in D$ is redundant if there exists an $e'' \in pred(e)$ such that $e' \in pred(e'')$. Since we do not allow any redundant dependencies in D, we call e' an immediate predecessor of e if $(e', e) \in D$. We will call an event e_I with $pred(e_I) = \emptyset$ an *initial event* of $\mathcal{C}$ and an event e_F where for all $e \in E$ $e_F \notin pred(e)$ a *final event* of $\mathcal{C}$. 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).

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 $\mathcal{C} = \{P, L, M, E, D\}$ denote the formal model for

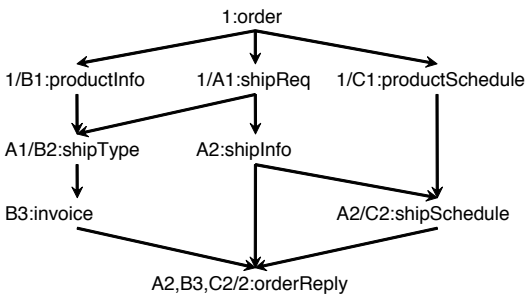

Figure 2. Dependencies among the message send events in the Purchase Order example.

the collaboration diagram of the Purchase Order example shown in Figure 1. The elements of the formal model are as follows (where 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 $send(o) = recv(oR) = C$, $recv(o) = send(oR) = send(sR) = recv(sI) = send(pI) = send(sT) = recv(I) = send(pS) = send(sS) = V$, $send(sI) = recv(sR) = Sh$, $recv(pI) = recv(sT) = send(I) = I$, and $recv(pS) = recv(sS) = Sc$. The set of events are $E = \{(1, o, 1), (2, oR, 1), (A1, sR, 1), (A2, sI, 1), (B1, pI, 1), (B2, sT, 1), (B3, i, 1), (C1, pS, 1), (C2, sS, 1)\}$. Finally, the dependency relation is (where we identify the events with their labels) $D = \{ (e_1, e_{A1}), (e_{A1}, e_{A2}), (e_1, e_{B1}), (e_{A1}, e_{B2}), (e_{B1}, e_{B2}), (e_{B2}, e_{B3}), (e_1, e_{C1}), (e_{A2}, e_{C2}), (e_{C1}, e_{C2}), (e_{A2}, e_2), (e_{B3}, e_2), (e_{C2}, e_2)\}$.

Given a collaboration diagram $\mathcal{C} = (P, L, M, E, D)$ we denote the *set of interactions* defined by $\mathcal{C}$ as $\mathcal{I}(\mathcal{C})$ where $\mathcal{I}(\mathcal{C}) \subseteq M^*$. An *interaction* $\sigma = m_1 m_2 \ldots m_n$ is in $\mathcal{I}(\mathcal{C})$, i.e., $\sigma \in \mathcal{I}(\mathcal{C})$, 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 interaction 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 the only events that are omitted from the event sequence γ are the conditional or iterative events. Finally, the fourth condition states that only iterative events can be repeated in the event sequence γ.

For example, a possible interaction 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 interaction which satisfy all the four conditions listed above are: $(1, o, 1)$, $(A1, sR, 1)$, $(A2, sI, 1)$, $(C1, pS, 1)$, $(B1, pI, 1)$, $(C2, sS, 1)$, $(B2, sT, 1)$, $(B3, i, 1)$, $(2, oR, 1)$.

It is common to use several collaboration diagrams to specify the interactions among a set of peers. We define a *collaboration diagram set* as $\mathcal{S} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ where n is the number of collaboration diagrams in $\mathcal{S}$ and each $\mathcal{C}_i$ is in the form $\mathcal{C}_i = (P, L, M, E_i, D_i)$, i.e., the collaboration diagrams in a collaboration diagram set only differ in their event sets and dependencies. (Note that we can always convert a set of collaboration diagrams to this form without changing their interaction sets by replacing the individual peer, link and message sets by their unions.) We define the set of interactions defined by a collaboration diagram set as $\mathcal{I}(\mathcal{S}) = \bigcup_{\mathcal{C} \in \mathcal{S}} \mathcal{I}(\mathcal{C})$.

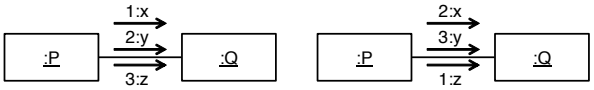

Figure 3. A collaboration diagram set which specifies a set of interactions that cannot be specified by any single collaboration diagram.

Collaboration diagram sets are strictly more powerful than the collaboration diagrams:

Theorem 1 *There exists a collaboration diagram set $\mathcal{S}$ such that there is no collaboration diagram $\mathcal{C}$ for which $\mathcal{I}(\mathcal{S}) = \mathcal{I}(\mathcal{C})$.*

Consider the collaboration diagram set shown in Figure 3. This collaboration diagram set specifies the following set of interactions $\{xyz, zyx\}$. Note that any collaboration diagram which tries to specify this set of interactions will either specify a circular dependency between the message transmissions for x and y, or it will allow more interactions which are not in this set. This can be shown by enumerating all the collaboration diagrams which contain only these message send events and these peers (there are a finite number of such collaboration diagrams).

3. Execution Model

We model the behaviors of peers that participate to a collaboration as concurrently executing finite state machines that interact via messages [9, 10]. We assume that the machines can interact with both synchronous and asynchronous messages. We assume that each finite 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 $M_i = M_i^A \cap M_i^S$ 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 $k \neq i$.

Consider the definition of the asynchronous receive action. Intuitively, the above definition says that the peer p_i executes an asynchronous receive action if there is a message at the head of its queue and a corresponding asynchronous receive transition in its transition relation from its current state. After the receive is executed, the received message is removed from the queue of p_i and the state of $\mathcal{A}_i$ is updated accordingly. The queues and states of other peers remain the same. An asynchronous send action inserts a message to the end of the message queue of the receiver and updates the state of the sender. The synchronous send and receive actions are executed simultaneously and update the states of both the sender and the receiver (the queue contents do not change during the execution of synchronous send and receive actions). In an ϵ-action a peer takes an ϵ-transition and updates its state.

Now we can define the runs of a set of peer state machines participating in a collaboration as follows: 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 *partial 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 $i \in [1..n]$.

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

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

A sequence σ is an *interaction* of a set of peer state machines $\mathcal{A}_1, \ldots, \mathcal{A}_n$, denoted as $\sigma \in \mathcal{I}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$, if there exists a complete run γ such that $\sigma = \mathcal{I}(\gamma)$, i.e., an interaction of a set of peer state machines must be an interaction generated by a complete run. The *interaction set* $\mathcal{I}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ of a set of peer state machines $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is the set of interactions 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 partial 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 $\mathcal{C}$ be a collaboration diagram. We say that the peer state machines $\mathcal{A}_1, \ldots, \mathcal{A}_n$ *realize* C if $\mathcal{I}(\mathcal{A}_1, \ldots, \mathcal{A}_n) = \mathcal{I}(\mathcal{C})$. A collaboration diagram $\mathcal{C}$ is *realizable* if there exists a set of well-behaved peer state machines which realize $\mathcal{C}$. We define the realizability for a collaboration diagram set exactly the same way.

4. Realizability of Collaboration Diagrams

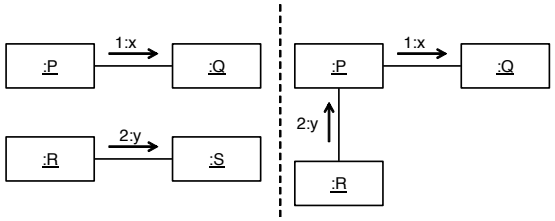

Figure 4. Two unrealizable collaboration diagrams.

Not all collaboration diagrams are realizable. Figure 4 shows two unrealizable collaboration diagrams. The interactions specified by these collaboration diagrams do not correspond to interactions of any set of peer implementations. The interaction set for both of these collaboration diagrams is $\{xy\}$. Note that for both cases, any set of peer state machines which generates the interaction xy will also generate the interaction yx since peer R cannot know if x is sent before it sends y.

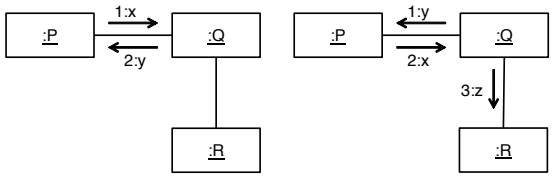

Figure 5. An unrealizable collaboration diagram set which consists of realizable collaboration diagrams.

Since each collaboration diagram corresponds to a singleton collaboration diagram set, existence of unrealizable collaboration diagrams implies existence of unrealizable collaboration diagram sets. Interestingly, a collaboration diagram set which consists of realizable collaboration diagrams may not be realizable. Figure 5 shows such a collaboration diagram set. Note that, the two individual collaboration diagrams shown in Figure 5 define realizable interactions. However, when we take the union of their interactions, the resulting interaction set is not realizable. The interaction set of the collaboration diagram set in Figure 5 is $\{xy, yxz\}$. Assume that there exists a set of peer state machines which generate the set of interactions specified by this collaboration diagram set. During an execution of these state machines, peers P and Q may behave according to the collaboration diagrams on the left and right, respectively, without realizing that the other peer is conforming to a different collaboration diagram. The following execution demonstrates this behavior: P first sends the message x, and x is stored in the message queue of Q; then Q sends the message y, and y is stored in the message queue of P. Peers P and Q consume (i.e., receive) the messages in their respective message queues, and finally Q sends the message z. Hence, the generated interaction is xyz which is not in the interaction set of the collaboration diagram set shown in Figure 5. The problem is, in the above execution, peer Q has no way of knowing 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 interaction yxz. One could consider modifying our state machine and execution models for the peers. However, allowing a conditional send action which checks the queue contents and then sends a message based on the result will not work unless the condition check and the send

are executed atomically. On the other hand, allowing such atomicity is unreasonable in modeling asynchronous communication among distributed programs (for example in the web services domain).

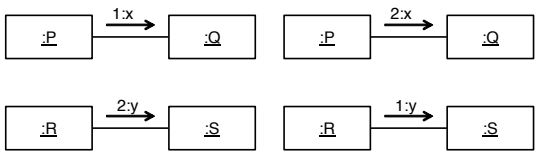

Figure 6. A realizable collaboration diagram set which consists of unrealizable collaboration diagrams.

It is also possible to have a realizable collaboration diagram set which consists of unrealizable collaboration diagrams. Figure 6 shows such an example. The interaction set specified by this collaboration diagram set is $\{xy, yx\}$ which is a superset of the interaction set specified by the collaboration diagram in Figure 4(a). Note that the set $\{xy, yx\}$ covers both the case where P sends x first and the case where Q sends y first. Since both orderings are acceptable, the ordering of x and y does not matter, and therefore, the collaboration diagram set shown in Figure 6 is realizable.

Below we first discuss conditions for realizability of individual collaboration diagrams. In the following sections we extend our discussion to realizability of collaboration diagram sets.

4.1. Separated Collaboration Diagrams

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 $\mathcal{C} = (P, L, M, E, D)$ 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. For example, the collaboration diagrams in Figure 1, Figure 4(a) and Figure 4(b) are separated whereas the collaboration diagram in Figure 7 is not separated.

Given an event $e = (l, m, r)$ in a collaboration diagram $\mathcal{C} = (P, L, M, E, D)$ let $e' = (l', m', r')$ be an immediate predecessor of e if it exists (i.e., $(e', e) \in D$). We will call the event e well-informed if 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 have the following result:

Theorem 2 *A separated collaboration diagram $\mathcal{C} = (P, L, M, E, D)$ is realizable if all the events $e \in E$ are well-informed.*

We will need the following definitions to prove this result. A word $w = e_0 e_1 ... e_k$ is said to *conform* to a dependency graph, if given any $0 \leq i < j \leq k$, e_i is not a descendant of e_j in the dependency graph. Given a collaboration diagram $\mathcal{C}$, let $S_E(\mathcal{C}) \subseteq E^*$ include all event sequences (note that, these are sequences of events not messages) which *conform* to the dependency graph of $\mathcal{C}$. Obviously, $S_E(\mathcal{C})$ is a finite set.

Given a word $w \in E^*$, its projection to a peer p_i is a word generated from w by removing all events that are not related to p_i (i.e., the events for which peer p_i is neither the sender nor the receiver) and the remaining events keep their original order in w. The projection of a set of words to a peer is obtained by applying the projection operation to each word in the set.

Since $S_E(\mathcal{C})$ is a finite set, the projection of $S_E(\mathcal{C})$ to each peer is also a finite set and can be recognized by a FSA. Let $\mathcal{A}_i^E$ be the minimal, deterministic FSA that recognizes the projection of $S_E(\mathcal{C})$ to peer p_i. Using $\mathcal{A}_i^E$s we can construct a set of peer implementations ($\mathcal{A}_i$s) that realize the interaction set $\mathcal{I}(\mathcal{C})$ as follows: Replace each transition $(s, (l, m, 1), s')$ in $\mathcal{A}_i^E$ with (s, m, s'). Replace each transition $(s, (l, m, *), s')$ in $\mathcal{A}_i^E$ with three transitions (s, ϵ, t), (t, ϵ, s'), and (t, m, t) where t is a new state. Replace each transition $(s, (l, m, ?), s')$ in $\mathcal{A}_i^E$ with two transitions (s, ϵ, s') and (s, m, s'). It is not hard to see that $\mathcal{A}_i$ recognizes the projection of the interaction set $\mathcal{I}(\mathcal{C})$ to the peer p_i. We call $\mathcal{A}_i$ the peer projection to peer p_i. Since $\mathcal{A}_i$ is generated from $\mathcal{A}_i^E$, we can annotate each transition in $\mathcal{A}_i$ with the send event that is associated with the corresponding transition in $\mathcal{A}_i^E$. During the execution of peer projections, when $\mathcal{A}_i$ takes a transition to send (receive) a message m, if the transition is with event e we say that e is being *executed* at sender (receiver).

Based on the algorithm we described above for constructing the peer projections we can infer the following property (call it P1): For any state in a peer projection, the annotation event of an out-going transition can not be an immediate predecessor of the annotation event of an in-coming transition.

Now, we prove the following property (P2): For a well-informed and separated collaboration diagram, for any

event e, any direct descendant e' (and hence all descendants) of e can not be executed before e is executed at the sender. If the above fact is true we can conclude that each interaction generated by the projected peers conforms to the dependency graph of $\mathcal{C}$ and therefore is an interaction specified by $\mathcal{C}$.

Consider an execution in which $e = (l, m, r)$ is being executed at the sender and consider any direct descendant $e' = (l', m', r')$ of e. According to the definition of well-informedness, the sender of m' is either the sender of m or the receiver of m. We discuss these two cases:

Case 1) $send(m') = send(m)$: According to P1 e' could not have been executed yet, because the execution of the peer FSA has not reached its transition. Hence P2 holds for case 1.

Case 2) $send(m') = recv(m)$: Examine the FSA of $send(m')$, since $send(m') = recv(m)$, events e and e' must be a pair of incoming and out-going transitions of some state in the projected FSA for $recv(m)$. (This is not the case if well-informedness is not enforced, e.g., Figure 4(a).)

Now the question is: Can e' be executed at its sender ($send(m')$) before e is executed at its sender ($send(m)$)? Figure 7 shows an example where this can happen if the collaboration diagram is not separated. We need the collaboration diagram to be separated and the events to be well-informed to prevent a situation we will call impersonation.

Given two send events $e_i = (l_i, m_i, r_i)$ and $e_j = (l_j, m_i, r_j)$ where e_j is a descendant of e_i in the dependency graph and the two events send the same message m_i. During the execution of all peer projections, we say that e_i *impersonates* e_j, if an m_i sent by e_i is consumed (at $recv(m_i)$) by a transition annotated with e_j. If impersonation can be prevented, for case 2 when e is executed its descendant e' cannot be executed before it (because the peer projection of $recv(m_i)$ has to execute e at receiver before executing e').

The definition of the separated property rules out the possibility of impersonation caused by a message shared by different threads. We now show that the condition 3 of the well-informedness property prevents impersonation in the same thread. We prove this by contradiction.

Let $e_1, ..., e_i, ..., e_j, ..., e_n$ be the total order of all the send events that belong to the same thread as e_i, and send the same message m_i. Before $recv(m_i)$ executes e_j (at receiver), all of $e_1, ..., e_{j-1}$ must be executed by $recv(m_i)$ at receiver. Without loss of generality, let e_j be the first impersonated event of the execution. We claim that there exists k where $i \leq k < j$ such that e_k's recurrence type is not 1. (Otherwise each send event will be counted one by one by the receiver and no confusion could be caused when executing e_j at $recv(m_i)$). Now according to the condition 3 of well-informedness, since the recurrence type of e_k is not 1, e_{k+1} must send a *different* message (let it be $m_{k+1} \neq m_i$) than e_k and has recurrence type 1. Hence $k + 1 \neq j$, since

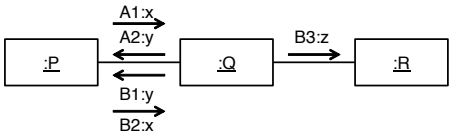

Figure 7. An unrealizable collaboration diagram which is not separated, and in which all events are well-informed.

e_{k+1} and e_j send two different messages.

Now we have $i < k + 1 < j$. Examine the receiver side. e_j is impersonated (executed at receiver $recv(m_i)$) and because e_{k+1} is a predecessor of of e_j, e_{k+1} is already executed at the receiver earlier. This implies that m_{k+1} is consumed earlier than the m_i sent by e_i (which is used to impersonate e_j). However, at the sender side (i.e., $send(m_i)$), e_i should be executed earlier than e_{k+1}, because e_i precedes e_{k+1}. This contradicts the fact that message buffer is FIFO. Hence, once condition 3 is satisfied, and $\mathcal{C}$ is separated, no impersonation can happen, which implies that property P2 also holds for case 2 which completes the proof.

Note that we can check the above realizability condition for separated collaboration diagrams in linear time. The examples given in Figure 4(a) and Figure 4(b) are both separated collaboration diagrams and they both violate the realizability condition discussed above. In both of these examples the sender for the final event (which is 2) is the peer R and R 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. On the other hand the collaboration diagram in Figure 1 is a separated collaboration diagram in which all the events are well-informed, hence, it is realizable.

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 interaction 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 interaction set will either generate the interaction $xyzxy$ or will not be well-behaved. Consider any set of peer state machines that generate this interaction set. Consider the partial 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 interactions which start with the prefix yx. Hence, peer Q can continue execution assuming that the interaction 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 interaction in the interaction set of the collaboration diagram. Therefore such a partial execution will either lead to a complete run and generate an interaction 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 3 *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 2. 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.

5. Collaboration Diagram Sets and Graphs vs. Conversation Protocols

In this section we will give sufficient conditions for realizability of collaboration diagram sets. We will do this by reducing the realizability problem for collaboration diagram sets to realizability of conversation protocols [9, 10]. We will show that this reduction allows us to handle even a larger class of interactions. We will define collaboration diagram graphs which are strictly more powerful than collaboration diagram sets and show that they are equivalent to conversation protocols in terms of the interactions they can specify.

5.1. Conversation Protocols

A *conversation protocol* [9] is a tuple $\mathcal{P} = (P, \mathcal{A})$ where P is a set of peers and $\mathcal{A} = (M, T, s, F, \delta)$ is a nondeterministic FSA where M is a set of messages such that for each $m \in M$ $recv(m) \in P$ and $send(m) \in P$, T is the finite set of states, $s \in T$ is the initial state, $F \subseteq T$ is the set of final states, and $\delta \subseteq T \times (M \cup \{\epsilon\}) \times T$ is the transition relation. Note that, a conversation protocol has two types of transitions: (1) (t_1, m, t_2) denotes a message transmission where message m is sent from peer $send(m)$ to peer $recv(m)$, and (2) (t_1, ϵ, t_2) denotes an ϵ-transition.

We define the set of interactions defined by the conversation protocol $\mathcal{P} = (P, \mathcal{A})$ as $\mathcal{I}(\mathcal{P})$ where $\mathcal{I}(\mathcal{P}) \subseteq M^*$ and $\sigma \in \mathcal{I}(C)$ if and only if $\sigma = m_1, m_2, \ldots, m_n$ where for all $1 \leq i \leq n$ $m_i \in M$ and there exists a path $t_1, t_2, \ldots, t_n, t_{n+1}$ in $\mathcal{A}$ such that $t_1 = s$, $t_{n+1} \in F$, and for all $1 \leq i \leq n$ $(t_i, m_i, t_{i+1}) \in \delta$.

Let $\mathcal{P}$ be a conversation protocol. We say that the peer state machines $\mathcal{A}_1, \ldots, \mathcal{A}_n$ *realize* $\mathcal{P}$ if $\mathcal{I}(\mathcal{A}_1, \ldots, \mathcal{A}_n) = \mathcal{I}(\mathcal{P})$. A conversation protocol $\mathcal{C}$ is *realizable* if there exists a set of well-behaved peer state machines which realize $\mathcal{P}$.

Below we will give a constructive proof for the following property:

Theorem 4 *Given a collaboration diagram set $\mathcal{S}$ there exists a conversation protocol $\mathcal{P}$ such that $\mathcal{I}(\mathcal{S}) = \mathcal{I}(\mathcal{P})$.*

Given a collaboration diagram set $\mathcal{S} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ where n is the number of collaboration diagrams in $\mathcal{S}$ and each $\mathcal{C}_i$ is in the form $\mathcal{C}_i = (P, L, M, E_i, D_i)$ we will construct a conversation protocol $\mathcal{P}_\mathcal{S} = (P, \mathcal{A}_\mathcal{S})$ such that $\mathcal{I}(\mathcal{P}_\mathcal{S}) = \mathcal{I}(\mathcal{S})$. Let $\mathcal{A}_\mathcal{S} = (M, T, s, F, \delta)$. We define the set of states of $\mathcal{A}_\mathcal{S}$ as $T = \{s\} \cup \bigcup_{\mathcal{C}_i \in \mathcal{S}} 2^{E_i}$, i.e., the set of states of $\mathcal{A}_\mathcal{S}$ consists of the start state s and the power sets of the event sets of the collaboration diagrams that are in $\mathcal{S}$. If there exists an E_i such that $E_i = \emptyset$, then $F = \{s, \emptyset\}$, otherwise $F = \{\emptyset\}$. We define the transition relation δ as follows:

- For each $i \in [1..n]$, δ contains the transition (s, ϵ, E_i).
- For each state $E \subseteq E_i$, if there exists an event $e \in E_i$ such that for all $(e', e) \in D_i$ $e' \notin E$, then
  - if $e = (l, m, 1)$ then δ contains the transition $(E, m, E - \{e\})$,
  - if $e = (l, m, ?)$ then δ contains the transitions $(E, m, E - \{e\})$ and $(E, \epsilon, E - \{e\})$,
  - if $e = (l, m, *)$ then δ contains the transitions (E, m, E) and $(E, \epsilon, E - \{e\})$.

The automaton $\mathcal{A}_\mathcal{S}$ first nondeterministically chooses one of the collaboration diagrams in the collaboration diagram set. Each state in the automaton after the start state represent a set of events that need to be executed. Given a state E, if there is an event $e \in E$ which does not have any of its predecessors in E, then we add transitions from E to $E - \{e\}$ that correspond to the send event e. If e is an iterative

event then we add a self loop to E to represent arbitrary number of sends.

Note that based on the above construction, the number of states generated for a collaboration diagram C_i with the event set E_i could be $2^{|E_i|}$ in the worst case. This worst case is realized only if C_i has $|E_i|$ threads. If C_i has a single thread then the number of states it generates will be $|E_i|$. So the number of threads generated for each collaboration diagram is exponential in the number of threads. If we determinize the automaton $\mathcal{A}_\mathcal{S}$ then the number of states will also be exponential in $|\mathcal{S}|$, i.e., the the number of collaboration diagrams in the collaboration diagram set.

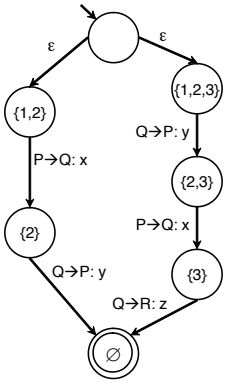

Figure 8. The conversation protocol for the collaboration diagram set shown in Figure 5.

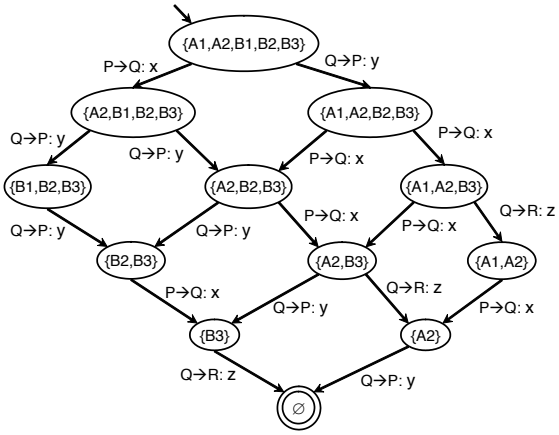

Figure 9. The conversation protocol for the collaboration diagram shown in Figure 7.

Figure 8 shows the conversation protocol for the collaboration diagram set shown in Figure 5 and Figure 9 shows the conversation protocol for the collaboration diagram shown in Figure 4 (since there is a single collaboration diagram we omitted the extra start state).

5.2. Collaboration Diagram Graphs

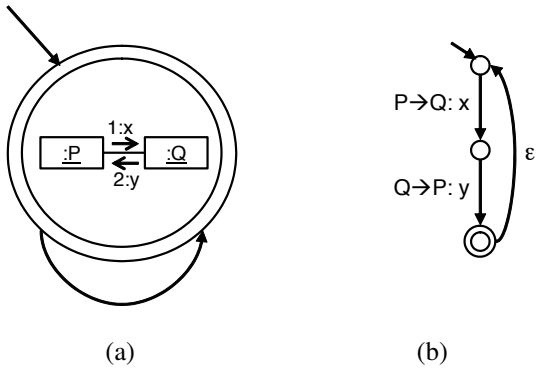

Figure 10. A collaboration diagram graph and corresponding conversation protocol.

A *collaboration diagram graph* $\mathcal{G} = (v_s, Z, V, O)$ is a directed graph which consists of a set of vertices V, a set of directed edges $O \subseteq V \times V$, an initial vertex $v_s \in V$, a set of final vertices $Z \subseteq V$, where each vertex in $v \in V$ is a collaboration diagram $v = (P, L, M, E_v, D_v)$. As with the collaboration diagram sets, to simplify our presentation, we assume that the collaboration diagrams in a collaboration diagram graph only differ in their event sets and dependency relations.

Given a collaboration diagram graph $\mathcal{G} = (v_s, Z, V, O)$ we define the set of interactions defined by G as $\mathcal{I}(\mathcal{G})$. The interactions of a collaboration diagram graph is defined as the concatenation of the interactions of its vertices on a path that starts from the initial vertex and ends at a final vertex. Formally, an interaction $\sigma \in M^*$, is in the interaction set of $\mathcal{G}$, i.e., $\sigma \in \mathcal{G}$, if and only if $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$ where for all $i \in [1..n]$ $\sigma_i \in M^*$ and there exists a path $v_1, v_2, \ldots, v_n$ in G such that $v_1 = v_s$, $v_n \in Z$, for all $1 \leq i < n$ $(v_i, v_{i+1}) \in O$ and for all $1 \leq i \leq n$ $\sigma_i \in \mathcal{I}(v_i)$.

Collaboration diagram graphs are strictly more powerful than the collaboration diagram sets:

Theorem 5 *There exists a collaboration diagram graph G such that there is no collaboration diagram $\mathcal{S}$ for which $\mathcal{I}(G) = \mathcal{I}(\mathcal{S})$.*

The set of interactions specified by the collaboration diagram graph given in Figure 10 cannot be specified by any collaboration diagram set. Note that the set of interactions specified in Figure 10 involve arbitrary number of repetitions of two consecutive events. There is no way to specify

such interactions using collaboration diagram sets since the only available repetition construct is "*" which repeats a single event.

Collaboration diagram graphs and conversation protocols have the same expressive power:

Theorem 6 *Given a collaboration diagram graph $\mathcal{G}$ there exists a conversation protocol $\mathcal{P}$ such that $\mathcal{I}(\mathcal{G}) = \mathcal{I}(\mathcal{P})$ and visa versa.*

Given a collaboration diagram graph $\mathcal{G} = (v_s, Z, V, O)$ where each $v \in V$ is a collaboration diagram $v = (P, L, M, E_v, D_v)$, we will construct a conversation protocol $\mathcal{P}_\mathcal{G} = (P, \mathcal{A}_\mathcal{G})$ where $\mathcal{A}_\mathcal{G} = (M, T, s, F, \delta)$, such that $\mathcal{I}(\mathcal{G}) = \mathcal{I}(\mathcal{P}_\mathcal{G})$. First, for each vertex $v \in V$ of G, construct a conversation protocol $\mathcal{P}_v = (P, \mathcal{A}_v)$, where $\mathcal{A}_v = (M, T_v, s_v, F_v, \delta_v)$, using the construction given above for translating collaboration diagram sets to conversation protocols (each vertex v corresponds to a singleton collaboration diagram set) such that $\mathcal{I}(v) = \mathcal{I}(\mathcal{P}_v)$. Then for $\mathcal{A}_\mathcal{G} = (M, T, s, F, \delta)$ we have $T = \bigcup_{v \in V} T_v$, i.e., the set of states of $\mathcal{A}_\mathcal{G}$ is the union of the states of the conversation protocols constructed for each vertex of $\mathcal{G}$. We define the initial state of $\mathcal{A}_\mathcal{G}$ as the initial state of the conversation protocol constructed for the initial vertex v_s, i.e., $s = s_{v_s}$. The final states of $\mathcal{A}_\mathcal{G}$ are the union of the final states of the conversation protocols constructed for vertices $v \in Z$, i.e, $F = \bigcup_{v \in Z} F_v$.

The transitions of $\mathcal{A}_\mathcal{G}$ include all the transitions of the conversation protocols constructed for all the vertices, i.e., $\delta \supseteq \bigcup_{v \in V} \delta_v$. Additionally we add some ϵ-transitions to δ as follows. For each edge $(v, v') \in O$, where $\mathcal{A}_v = (M, T_v, s_v, F_v, \delta_v)$ and $\mathcal{A}_{v'} = (M, T_{v'}, s_{v'}, F_{v'}, \delta_{v'})$ are the automata constructed for v and v', respectively, δ includes an ϵ-transition from each final state of $\mathcal{A}_v$ to the initial state of $\mathcal{A}_{v'}$, i.e., $\delta \supseteq \bigcup_{(v,v') \in O, s \in F_v} (s, \epsilon, s_{v'})$.

The set of interactions specified by the collaboration diagram graph given in Figure 10(b) is the conversation protocol constructed for the collaboration diagram graph in Figure 10(a) based on the above construction.

It is easy to show that translation in the other direction is also possible, i.e., given a conversation protocol it is possible to construct a collaboration diagram graph with the same interaction set. Note that, given a conversation protocol, we can generate a collaboration diagram with a single send event for each transition of the conversation protocol. Then, we can combine these collaboration diagrams in a collaboration diagram graph based on the transition relation of the conversation protocol. We can also define the initial and final states of the collaboration diagram based on the initial and final states of the conversation protocol. Then, the resulting collaboration diagram graph would specify exactly the same set of interactions defined by the conversation protocol.

5.3. Realizability Revisited

Since we showed that we can translate collaboration diagrams to conversation protocols, we can use the earlier realizability results on conversation protocols to identify realizable collaboration diagram graphs. In [9, 10], three realizability conditions for conversation protocols are defined: *synchronous compatibility*, *autonomy*, and *lossless join*. Given a collaboration diagram graph $\mathcal{G}$, let $\mathcal{P}_\mathcal{G}$ be a conversation protocol with the same interaction set. If $\mathcal{P}$ satisfies the synchronous compatibility, autonomy, and lossless join conditions then $\mathcal{G}$ is realizable.

Before presenting the three realizability conditions, we need to define an alternative *synchronous* semantics for the execution of a set of peers [10]. Intuitively, the synchronous semantics dictates that the sending and receiving peers take the send and receive actions concurrently even for asynchronous messages (i.e., asynchronous messages are treated exactly like the synchronous messages). During the execution based on the synchronous semantics there is no need to have the input message queues.

Assume that we are given a set of peers $\mathcal{A}_1, ..., \mathcal{A}_n$ where each automaton $\mathcal{A}_i$ is a finite state automaton defined as in Section 3. The global configuration with respect to the *synchronous semantics*, called the *syn-configuration*, is a tuple $(t_1, ..., t_n)$, where for each $j \in [1..n]$, $t_j \in T_j$ is the local state of peer $\mathcal{A}_i$. For two syn-configurations $c = (t_1, ..., t_n)$ and $c' = (t'_1, ..., t'_n)$, we say that c *derives* c', written as $c \rightarrow c'$, if the following condition hold: Two peers execute a *send* action (denoted as $c \xrightarrow{m} c'$), i.e., there exist $1 \leq i, j \leq n$ and $m \in M_i^{\text{out}} \cap M_j^{\text{in}}$ such that:

1. $(t_i, !m, t'_i) \in \delta_i$,
2. $(t_j, ?m, t'_j) \in \delta_j$,
3. $t'_k = t_k$ for each $k \neq i$ and $k \neq j$.

Notice that the message m in the above definition can either be an asynchronous or a synchronous message. We call this new semantics described above the *synchronous semantics* and the original semantics defined in Section 3 the *asynchronous semantics*. The definitions of a run, a partial run, and an interaction generated by a run for synchronous semantics is similar to those of the asynchronous semantics given in Section 3.

Given a conversation protocol $\mathcal{P} = (P, \mathcal{A})$ we denote the projection of $\mathcal{A} = (M, T, s, F, \delta)$ to peer $p_i \in P$ as $\pi_i(\mathcal{A})$ which is defined as follows: $\pi_i(\mathcal{A}) = (M_i, T, s, F, \delta_i)$ where $M_i \subseteq M$ contains all the messages m such that $send(m) = p_i$ or $recv(m) = p_i$. The set of states, the initial state and the final states of $\mathcal{A}$ and $\pi_i(\mathcal{A})$ are the same. We define δ_i as follows:

- For each $m \in M$ such that $m \notin M_i$, for each transition $(t_1, m, t_2) \in \delta$, or $(t_1, m, t_2) \in \delta$ we add the transition (t_1, ϵ, t_2) to δ_i.

- For each $m \in M_i$ such that $send(m) = p_i$, for each transition $(t_1\ ,\ m\ ,\ t_2) \in \delta$, we add the transition $(t_1,\ !m,\ t_2)$ to δ_i.
- For each $m \in M_i$ such that $recv(m) = p_i$, for each transition $(t_1\ ,\ m,\ t_2) \in \delta$, we add the transition $(t_1,\ ?m,\ t_2)$ to δ_i.
- For each transition $(t_1, \epsilon, t_2) \in \delta$ we add the transition (t_1, ϵ, t_2) to δ_i.

Using the standard automata algorithms, we can remove ϵ-transitions in a projection using determinization and then minimize it. We call the resulting automaton the determinized peer projection to p_i. Figure 11 shows the determinized peer projection of the conversation protocol in Figure 8 to the peers P, Q and R.

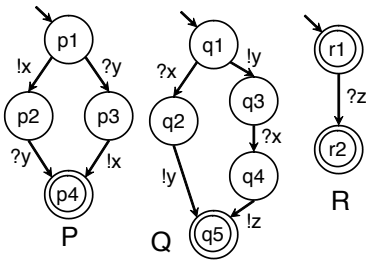

Figure 11. Projection of the conversation protocol in Figure 8 to peers P, Q and R (after they are determinized and minimized).

We now describe the three realizability conditions presented in [9, 10]:

Synchronous compatible condition: This condition requires that if we project the conversation protocol to each peer, then for each syn-configuration that is reachable from the initial state, if there is a peer which has a send transition for a message from its local state in that configuration, then in the same configuration there should be a corresponding peer with a receive transition from its local state for the same message.

Autonomous condition: A conversation protocol is autonomous if each of its determinized peer projections, at any moment, can do only one of the following 1) terminate, 2) send a message, or 3) receive a message. Notice that autonomous condition still allows a certain level of nondeterminism, e.g., a peer can have a choice of sending one of many messages. However, autonomous condition does not permit a choice between send and receive actions. To check the autonomous condition we obtain each determinized peer projection and check that out-going transitions for each non-final state are either all send transitions or all receive transitions. We also check that each final state has no out-going transitions.

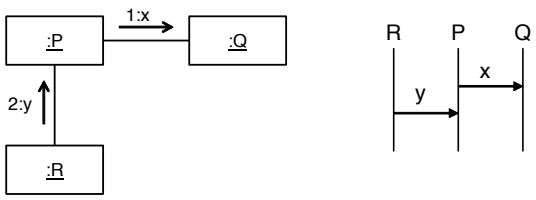

Figure 12. A collaboration diagram which specifies a set of interactions that cannot be specified by Message Sequence Charts.

Lossless Join Condition: A conversation protocol is lossless join if the product automaton constructed from its determinized projections is equivalent to itself. This condition is a necessary condition for realizability.

Algorithms for checking the synchronous compatible, autonomous and lossless join conditions are given in [10].

We now present the following realizability result:

Theorem 5.1 Given a collaboration diagram graph $\mathcal{G}$, let $\mathcal{P}$ be a conversation protocol where $\mathcal{I}(\mathcal{P}) = \mathcal{I}(\mathcal{G})$. Let $\mathcal{A}_{p_1}, ..., \mathcal{A}_{p_n}$ be the determinized projections of $\mathcal{P}$. If $\mathcal{P}$ satisfies the synchronous compatible, autonomous and lossless join conditions, then $\mathcal{I}(\mathcal{A}_{p_1}, ..., \mathcal{A}_{p_n}) = \mathcal{I}(\mathcal{P}) = \mathcal{I}(\mathcal{G})$. Moreover, $\mathcal{A}_{p_1}, ..., \mathcal{A}_{p_n}$ are well-behaved and for each interaction generated by $\mathcal{A}_{p_1}, ..., \mathcal{A}_{p_n}$ (based on the asynchronous semantics) there exists a run which generates the same interaction in which each asynchronous send is immediately followed by the corresponding receive.

This result follows directly from the realizability results on conversation protocols presented in [9, 10]. Theorem 5.1 implies that if we convert a collaboration diagram graph to an equivalent conversation protocol and examine the three sufficient realizability conditions, then the collaboration diagram graph is guaranteed to be realizable if the equivalent conversation protocol satisfies the three realizability conditions. Moreover, the collaboration diagram graph will be realized by the determinized projections of the equivalent conversation protocol. Note that, in addition to providing a way to determine realizability, this gives us a way to construct the peers which realize the set of interactions specified by the collaboration diagram graph.

6. Related Work

Message Sequence Charts (MSC) [12] provide another visual model for specification of interactions in distributed

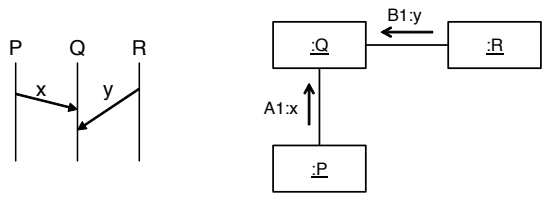

Figure 13. The ordering of the send and receive events described by the Message Sequence Chart on the left cannot be specified by any collaboration diagram.

systems. MSC model has also been used in modeling and verification of web services [8]. As opposed to the collaboration diagrams which only specify the ordering of send 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. A detailed comparison of MSCs, MSC graphs and conversation protocols is given in [10].

The examples in Figures 12 and 13 demonstrate the differences between the MSC and collaboration diagram models. Consider the collaboration diagram shown in Figure 12 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 12), this does not guarantee that message y is sent after message x is sent (note that these are asynchronous messages).

Figure 13, on the other hand, shows a MSC which specifies and ordering of send and receive events which cannot be specified using a collaboration diagram. The MSC in Figure 13 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 13 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) [16] 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. Our generalization of the collaboration diagrams to collaboration diagram graphs is similar to (and in fact inspired by) the generalization of MSCs to MSC graphs [4].

The notion of realizability has been studied for several models of concurrent and distributed systems [1, 14, 15]. In [1, 14, 15], realizability problem is defined as whether a peer has a strategy to cope with the environment no matter how the environment decides to move. In our model, on the other hand, the environment of an individual peer consists of other peers which behave according to the given collaboration diagram. Also our definition of realizability requires that implementation should generate all (instead of a subset of) behaviors as specified by a collaboration diagram.

7. Conclusions

In this paper we formalized the realizability problem for collaboration diagrams. We gave sufficient conditions for realizability of some classes of collaboration diagrams. We generalized the collaboration diagrams to collaboration diagram sets and collaboration diagram graphs and showed that these models have increasing expressive power. We showed that collaboration diagram graphs are equivalent to conversation protocols. Hence, realizability conditions on conversation protocols can be used to determine realizability of conversation diagram graphs.

We believe that analysis of type 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. However, specification of interactions from a global perspective inevitably leads to the realizability problem. Our results in this paper address the realizability problem when such interactions are specified using collaboration diagrams.

References

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In *Proc. of 16th Int. Colloq. on Automata, Languages and Programming*, volume 372 of *LNCS*, pages 1–17. Springer Verlag, 1989.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services Concepts, Architectures and Applications Series: Data-Centric Systems and Applications*. Addison Wesley Professional, 2002.

[3] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proc. 22nd Int. Conf. on Software Engineering*, pages 304–313, 2000.

[4] 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.

[5] Business process execution language for web services (BPEL), version 1.1. http://www.ibm.com/developerworks/library/ws-bpel.

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

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

[8] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proc. 18th IEEE Int. Conf. on Automated Software Engineering Conference*, pages 152–163, 2003.

[9] 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.

[10] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, December 2005.

[11] S. Kleijnen and S. Raju. An open web services architecture. *ACM Queue*, 1(1):39–46, March 2003.

[12] Message Sequence Chart (MSC). ITU-T, Geneva Recommendation Z.120, 1994.

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

[14] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Principles of Programming Languages*, pages 179–190, 1989.

[15] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloq. on Automata, Languages, and Programs*, volume 372 of *LNCS*, pages 652–671, 1989.

[16] 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.

[17] UML 2.0 superstructure specification. http://ww.uml.org/, October 2004.

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