

Specification of Realizable Service Conversations Using Collaboration Diagrams

Tevfik Bultan

Computer Science Department
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 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, 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 such a global perspective leads to the realizability problem: Is it possible to construct a set of peers that generate exactly the specified 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 appli-

cations.

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) [21] 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, 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 define a formal model for collaborations diagrams. We model a distributed system as a set of communicating finite state machines [7]. A collaboration diagram is realizable if there exists a set of communicating finite state machines which 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 [2, 3, 19], 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 [8, 9], 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 [20]) 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 num-

bers, 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 non-determinism 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 [6]. 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*,

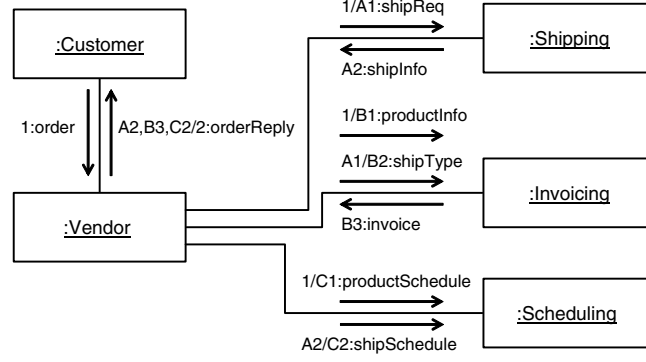


Figure 1. An example collaboration diagram for a composite web service.

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 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 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).

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, \dots, 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 $e'' \in \text{pred}(e)$ such that $e' \in \text{pred}(e'')$. We assume that there are no redundant dependencies in D . Since we do not allow any redundant dependencies in D , we call e' an immediate predecessor of e if $(e', e) \in D$.

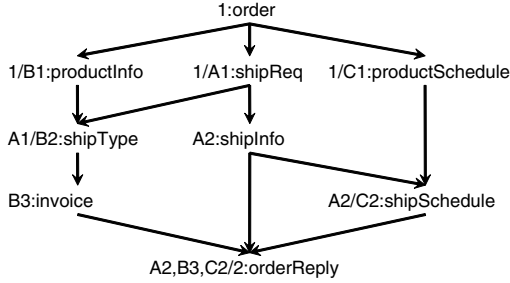


Figure 2. Dependencies among the message send events in the Purchase Order example.

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 (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
 - $C = \text{send}(o) = \text{recv}(oR)$,
 - $V = \text{recv}(o) = \text{send}(oR) = \text{send}(sR) = \text{recv}(sI) = \text{send}(pI) = \text{send}(sT) = \text{recv}(I) = \text{send}(pS) = \text{send}(sS)$,
 - $Sh = \text{send}(sI) = \text{recv}(sR)$,
 - $I = \text{recv}(pI) = \text{recv}(sT) = \text{send}(I)$,
 - $Sc = \text{recv}(pS) = \text{recv}(sS)$,
- $E = \{(1, o, 1), (2, oR, 1), (A1, sR, 1), (A2, sI, 1), (B1, pI, 1), (B2, sT, 1), (B3, i, 1), (C1, pS, 1), (C2, sS, 1)\}$ 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 $\mathcal{D} = (P, L, M, E, D)$ we denote the set of conversations defined by \mathcal{D} as $\mathcal{C}(\mathcal{D})$ where $\mathcal{C}(\mathcal{D}) \subseteq M^*$. A conversation $\sigma = m_1 m_2 \dots m_n$ 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 $\gamma = e_1 e_2 \dots e_n$ 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 [11, 13]. 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 $P = \{p_1, \dots, p_n\}$ 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 $p_i = \text{send}(m)$ to peer $\text{recv}(m)$, (2) a receive-transition of the form $(t_1, ?m, t_2)$ which receives a message $m \in M_i$ from peer $\text{send}(m)$ to peer $p_i = \text{recv}(m)$, and (3) an ϵ -transition of the form (t_1, ϵ, t_2) .

Let $\mathcal{A}_1, \dots, \mathcal{A}_n$ be the peer state machines (implementations) for a set of peers $P = \{p_1, \dots, p_n\}$ 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 $(Q_1, t_1, \dots, Q_n, t_n)$ 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 $c = (Q_1, t_1, \dots, Q_n, t_n)$ and $c' = (Q'_1, t'_1, \dots, Q'_n, t'_n)$, 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 $m \in M_i^A \cap M_j^S$, such that, $p_i = \text{send}(m)$, $p_j = \text{recv}(m)$ 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, $p_i = \text{recv}(m)$ and:
 1. $(t_i, ?m, t'_i) \in \delta_i$,
 2. $Q_i = m Q'_i$,
 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, $p_i = \text{send}(m)$, $p_j = \text{recv}(m)$ 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: Let $\mathcal{A}_1, \dots, \mathcal{A}_n$ be a set of peer state machines for the set of peers $P = \{p_1, \dots, p_n\}$ participating in a collaboration, a sequence of configurations $\gamma = c_0 c_1 \dots c_k$ is a *partial run* of $\mathcal{A}_1, \dots, \mathcal{A}_n$ 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 $c_0 = (\epsilon, s_1, \dots, \epsilon, s_n)$ 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 $c_k = (\epsilon, t_1, \dots, \epsilon, t_n)$ 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 *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 $c \xrightarrow{!m} c'$
 - $\mathcal{C}(\gamma) = \mathcal{C}(\gamma' c) m$ if $c \xrightarrow{?m} c'$
 - $\mathcal{C}(\gamma) = \mathcal{C}(\gamma' c)$ otherwise.

A sequence σ is a *conversation* of a set of peer state machines $\mathcal{A}_1, \dots, \mathcal{A}_n$, denoted as $\sigma \in \mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$, 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* $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ of a set of peer state machines $\mathcal{A}_1, \dots, \mathcal{A}_n$ is the set of conversations generated by all the complete runs of $\mathcal{A}_1, \dots, \mathcal{A}_n$.

We call a set of peer state machines $\mathcal{A}_1, \dots, \mathcal{A}_n$ *well-behaved* if each partial run of $\mathcal{A}_1, \dots, \mathcal{A}_n$ 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 $\mathcal{A}_1, \dots, \mathcal{A}_n$ *realize* \mathcal{D} if $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n) = \mathcal{C}(\mathcal{D})$. A collaboration diagram \mathcal{D} is *realizable* if there exists a set of well-behaved peer state machines which realize \mathcal{D} .

4. Realizability of Collaboration Diagrams

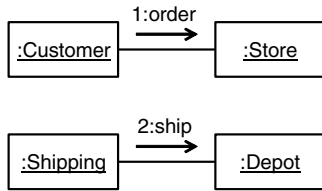


Figure 3. Unrealizable collaboration diagram.

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.

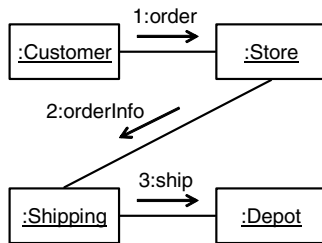


Figure 4. Realizable collaboration diagram.

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.

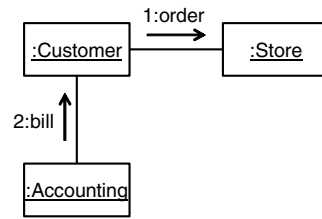


Figure 5. Unrealizable collaboration diagram.

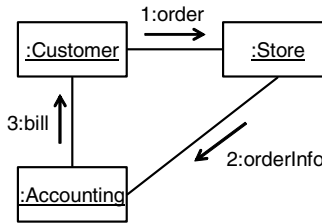


Figure 6. Realizable collaboration diagram.

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 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 will give sufficient conditions for realizability of a class of collaborations diagrams which are

common in practice. Recall that the events in a collaboration diagram consist of a set of threads, where the set of events in each thread is totally ordered. In the class of collaboration diagrams we will focus on, a message can only appear in the events that belong to one thread; i.e., we do not allow the same message to appear in the events of two different threads. 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.

Given an event $e = (l, m, r)$ in a collaboration diagram $\mathcal{D} = (P, L, M, E, D)$ let $e' = (l', m', r')$ be an immediate predecessor of e if it exists (i.e., $(e', e) \in D$). We 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 $r = 1$.

According to this definition, there are three types of well-informed events. First, all the initial events are well-informed. 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.

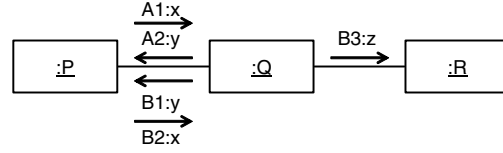


Figure 7. Unrealizable collaboration diagram with well-informed events.

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

The proof of this property is given in [1]. 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.

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

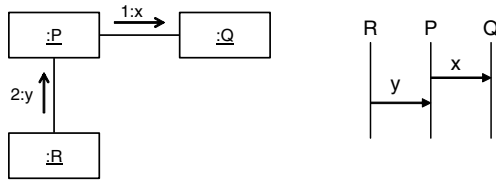


Figure 8. A collaboration diagram with no corresponding Message Sequence Charts.

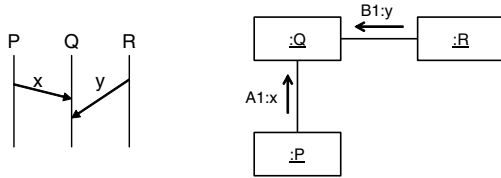


Figure 9. A Message Sequence Chart with no corresponding collaboration diagram.

by this collaboration diagram consists of all interleavings of the sequences xy and yxz which is the set $\{xyyxz, xyxyz, xyxzy, yxxzy, yxxzy, yxxzy, yxxzy, yxxzy\}$. 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 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 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.

6. Related Work

Message Sequence Charts (MSCs) [14] provide another visual model for specification of interactions in distributed systems. MSC model has also been used in modeling and

verification of web services [10]. 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. 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 [2] and its extensions such as high-level MSC (hMSC) [19] and MSC Graphs [3] 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 [4, 18, 5]). However, we are not aware of any work that focuses on realizability of interactions specified as collaboration diagrams.

In [15], interactions among agents are represented using various UML diagrams, including collaboration diagrams, however, the realizability problem is not investigated. In [16, 17], Dooley graphs are used to model conversations. Although some of the conditions on Dooley graphs presented in these earlier works are similar to the realizability conditions presented in this paper, they do not address the realizability problem discussed in this paper. Moreover, the computational model we present in this paper is different and involves both synchronous and asynchronous commu-

nication, 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* [8, 11, 12, 13, 9]. 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 [1] 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 [11].

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 than 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] Realizability of interactions in collaboration diagrams. Technical Report 2006-11, Computer Science Department, University of California, Santa Barbara, September 2006.
- [2] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proc. 22nd Int. Conf. on Software Engineering*, pages 304–313, 2000.
- [3] 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.
- [4] B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, Jan 2003.
- [5] M. B. Blake. A lightweight software design process for web services workflows. In *Proc. of the 2006 IEEE International Conference on Web Services*, pages 411–418, 2006.
- [6] Business process execution language for web services (BPEL), version 1.1. <http://www.ibm.com/developerworks/library/ws-bpel>.
- [7] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.
- [8] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proc. 12th Int. World Wide Web Conf.*, pages 403–410, May 2003.
- [9] T. Bultan, X. Fu, and J. Su. Analyzing conversations of web services. *IEEE Internet Computing*, 10(1):18–25, 2006.
- [10] 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.
- [11] 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.
- [12] X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. *International Journal of Web Services Research (JWSR)*, 2(4):68 – 93, 2005.
- [13] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, December 2005.
- [14] Message Sequence Chart (MSC). ITU-T, Geneva Recommendation Z.120, 1994.
- [15] J. J. Odell, H. V. D. Parunak, and B. Bauer. Representing Agent Interaction Protocols in UML. In *Proc. of First International Workshop on Agent-Oriented Software Engineering*, 1999.
- [16] H. V. D. Parunak. 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.
- [17] M. P. Singh. Synthesizing coordination requirements for heterogeneous autonomous agents. *Autonomous Agents and Multi-Agent Systems*, 3(2):107–132, June 2000.
- [18] D. Skogan, R. Gronmo, and I. Solheim. Web Service Composition in UML. In *Proc. of 8th International IEEE Enterprise Distributed Object Computing Conference*, 2004.
- [19] 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.
- [20] UML 2.0 superstructure specification. <http://www.uml.org/>, October 2004.
- [21] Web Service Choreography Description Language (WS-CDL). <http://www.w3.org/TR/ws-cdl-10/>, 2005.