

A Tool for Choreography Analysis Using Collaboration Diagrams

Tevfik Bultan Chris Ferguson
University of California Santa Barbara
{bultan,fergy}@cs.ucsb.edu

Xiang Fu
Hofstra University
Xiang.Fu@hofstra.edu

Abstract

Analyzing interactions among peers that interact via messages is a crucial problem due to increasingly distributed nature of current software systems, especially the ones built using the service oriented computing paradigm. In service oriented computing, interactions among peers participating to a composite service involve message exchanges across organizational boundaries in a distributed computing environment. In order to build such systems in a reliable manner, it is necessary to develop techniques for analysis and verification of interactions among services. Collaboration diagrams provide a convenient visual model for modeling service interactions. In this paper, we present a tool that 1) checks the realizability of interactions specified by the given collaboration diagram, 2) verifies the LTL properties of the interactions specified by the given collaboration diagram by automatically converting it to a state machine model, and 3) synthesizes peer state machines that realize the set of interactions specified by the given collaboration diagram.

1 Introduction

Service oriented computing provides technologies that enable multiple organizations to integrate their businesses over the Internet. Typical execution behavior in such a distributed system involves a set of autonomous peers interacting with each other through messages. Choreography specification languages, such as the Web Services Choreography Description Language (WS-CDL), are used for specification of such interactions. A choreography specification identifies the global ordering of the messages exchanged among the peers participating to a composite service. We call such message sequences conversations, i.e., a choreography specification identifies the set of allowable conversations for a composite web service.

Collaboration diagrams (called communication diagrams in [20]) provide a convenient visual formalism for specifying the choreography among the services (peers) participating to a composite service [6]. Characterization of

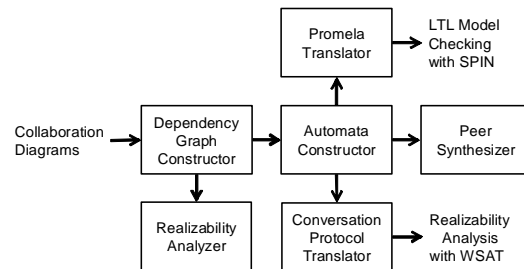


Figure 1. A tool for choreography analysis

interactions using a global view, as collaboration diagrams allow us to do, can lead to specification of choreographies that may not be implementable. Hence, using collaboration diagrams for choreography specification leads to the following realizability problem: Given a choreography specification, is it possible to find a set of distributed peers which interact exactly according to the choreography specification. If a collaboration diagram is realizable, then we can check the properties of the interactions among the peers by investigating the possible message orderings allowed by the collaboration diagram.

In this paper we present a toolset for verification and analysis of choreographies specified using collaboration diagrams. As shown in Figure 1, our tool consists of six components: The first component constructs a dependency graph for the events in the input collaboration diagram. The second component checks the realizability of the input collaboration diagram by checking a set of conditions on this dependency graph. The third component converts the collaboration diagram to a finite state automaton such that the language accepted by the automaton is equal to the set of interactions specified by the input collaboration diagram. The fourth components converts the collaboration diagram automaton to the input language of the Web Service Analysis Tool (WSAT) [11] (a tool developed for checking realizability web service choreography specifications) to check a different set of realizability conditions. The fifth component converts the collaboration diagram automaton to a Promela specification in order to check LTL properties using the Spin model checker [13]. Finally, the sixth compo-

nent synthesizes a set of state machines that generate exactly the set of interactions specified by the collaboration diagram automaton. We collected a set of collaboration diagrams from the literature and analyzed them using this toolset. Our experiments indicate that realizability analysis, LTL model checking and synthesis for collaboration diagrams is very efficient and can easily be used in practice.

Our contributions in this paper can be summarized as follows: 1) Extending the semantics for a single collaboration diagram given in [6] to collaboration diagram sets and graphs, with increasing expressive power. 2) An algorithm for converting collaboration diagrams/sets/graphs to an automaton that accepts the same set of conversations. 3) A translator for converting the collaboration diagram automaton to a Promela model, enabling LTL model checking using the Spin model checker [13]. 4) Implementing the realizability check for single collaboration diagrams from [6]. 5) A translator for converting the collaboration diagram automaton to a Conversation Protocol, enabling realizability check for collaboration diagram sets/graphs using the realizability analysis for conversation protocols implemented in Web Service Analysis Tool [11]. 6) A peer synthesis algorithm for generating state machine implementations for peers for realizable collaboration diagrams/sets/graphs by projecting the collaboration diagram automaton to each peer participating to the collaboration. 7) Experiments with several collaboration diagrams from the literature.

Related Work Message Sequence Charts (MSCs) provide another visual model for specification of interactions in distributed systems. MSC model has also been used in modeling and verification of web services [8]. However, collaboration diagrams provide a global view of interactions where as MSCs provide a local view. The realizability problem for MSCs [2] have been studied before. However as we mentioned above, the type of interactions specified by collaboration diagrams and MSCs are different.

There has been work on formalizing choreography specifications using process algebras [7, 16]. Our work is complementary to work on formalizing semantics of choreography specification languages. Our focus in this paper is formal visual representations that can be used by service developers to visualize their designs.

There has been earlier work on using various UML diagrams in modeling different aspects of service compositions (for example [3, 18]). Specification and analysis of web service interactions using conversation protocols has been investigated [10, 12]. In this paper, we investigate the relationship between the collaboration diagrams and the conversation protocols using the collaboration diagram semantics from [6]. A complementary approach to the one presented here is discussed in [17], where realizability of collaboration diagrams is analyzed using process algebra encodings. However, compared to these earlier works, in this paper we

extend the collaboration diagram semantics to collaboration diagrams sets and collaboration diagram graphs which have more expressive power.

2 Formal Model

We assume that a choreography specification consists of a finite set of peers P , and a finite set of messages M . Each message $m \in M$ has a unique sender and a unique receiver denoted by $send(m) \in P$ and $recv(m) \in P$, respectively. Note that, messages can always be converted to this form by concatenating each message with tags its sender and its receiver.

A *conversation* σ is a sequence of messages exchanged among the peers that participate to a composite web service, i.e., $\sigma \in M^*$. A *choreography* \mathcal{C} is a set of conversations, i.e., $\mathcal{C} \subseteq M^*$.

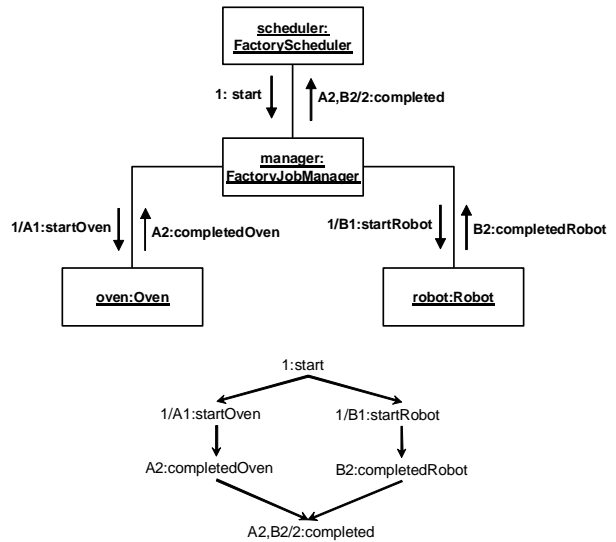


Figure 2. A collaboration diagram (top) and its dependency relation (bottom)

Figure 2 shows an example collaboration diagram from the UML 1.3 specification. The diagram consists of four peers Scheduler, Manager, Oven, Robot. The edges that connect the boxes shows the links between the peers. A link between two peers indicate that they can send each other messages. In collaboration diagrams, message send events are shown as arrows drawn over the links. The direction of the arrow indicates the sender and the receiver (the arrow points to the receiver). Each send event is marked with a sequence label. The sequence labels specify the ordering of the message send events.

Formally, 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 [6]. For each message $m \in M$, the sender and the receiver of m must be linked, i.e., $(send(m), recv(m)) \in L$.

In a collaboration diagram, each message send event has a unique sequence label. Each sequence label consists of a possibly empty prefix followed by a sequence number. The numeric ordering of the sequence numbers defines an implicit total ordering among the message send events with the same prefix. Each prefix represents a *message thread* where each message thread refers to a set of messages that have a total ordering and that can be interleaved arbitrarily with other messages. For example, event A2 can occur only after the event A1, but B1 and A2 do not have any implicit ordering. In addition to the implicit ordering defined by the sequence numbers, it is possible to explicitly state the events that should precede an event e by listing their sequence labels (followed by the symbol “/”) before the sequence label of the event e . For example if an event e is marked with “B2,C3/A2” then A2 is the sequence label of the event e , and the events with sequence labels B2, C3 and A1 must precede e . Also, 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.

Formally, 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 $e \in E$, $e.l$, $e.m$, and $e.r$ denote the unique sequence label, the message and the recurrence type for event e , respectively. 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 . The bottom of the Figure 2 shows the dependency graph for the the collaboration diagram shown at the top. We assume that there are no circular dependencies, i.e., the dependency graph (E, D) , where the send events in E form the vertices and the dependencies in D form the edges, should be a directed acyclic graph (dag). Given a dependency relation $D \subseteq E \times E$ let $pred(e)$ denote the predecessors of the event e where $e' \in pred(e)$ if there exists a set of events e_1, e_2, \dots, e_k where $k > 1$, $e' = e_1$, $e = e_k$, and for all $i \in [1..k - 1]$, $(e_i, e_{i+1}) \in D$. We assume that there are no redundant dependencies in D (i.e., it is the transitive reduction). We call e' an immediate predecessor of e if $(e', e) \in D$. We call an event e_I with $pred(e_I) = \emptyset$ an *initial event* of D and an

event e_F where for all $e \in E$ $e_F \notin pred(e)$ a *final event* of D .

Given a collaboration diagram $\mathcal{D} = (P, L, M, E, D)$ we denote the *choreography* 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 $i \in [1..n]$ $m_i = e_i.m$ and $e_i \in E$; 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 $i \in [1..n]$ $e_i \neq e$) $\Rightarrow (e.r = * \vee e.r = ?)$; and 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 .

Collaboration Diagram Sets Without the conditional or iterative events, a single collaboration diagram with a single message thread specifies a single conversation. The conditional and iterative events and message threads introduce nondeterminism to collaboration diagrams, enabling specification of multiple conversations with a single collaboration diagram. However, the level of nondeterminism in a single collaboration diagram is still quite limited. For example, assume that we have three messages m_1, m_2 and m_3 sent from one peer to another peer and we would like to specify the following choreography $\{m_1 m_2 m_3, m_3 m_1 m_2\}$. It is not possible to specify this simple choreography using a single collaboration diagram. However, it is possible to specify each conversation in this choreography using a separate collaboration diagram. So, the choreography we want to describe is the union of the choreographies of two different collaboration diagrams.

We define a *collaboration diagram set* as $\mathcal{S} = \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_n\}$ where n is the number of collaboration diagrams in \mathcal{S} and each \mathcal{D}_i is in the form $\mathcal{D}_i = (P, L, M, E_i, D_i)$, i.e., the collaboration diagrams in a collaboration diagram set only differ in their event sets and dependencies. (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 $\mathcal{C}(\mathcal{S}) = \bigcup_{\mathcal{D} \in \mathcal{S}} \mathcal{C}(\mathcal{D})$.

Collaboration Diagram Graphs Although collaboration diagrams sets increase the expressiveness of collaboration diagrams, they still have an important limitation. It is not possible to specify looping behaviors using collaboration

diagram sets. The only looping construct in collaboration diagrams/sets is the iterative event that specifies the repetition of a single event. Assume that we have two messages m_1 and m_2 exchanged among two peers and we would like to specify the following choreography $(m_1 m_2)^*$, i.e., zero or more repetitions of the message sequence $m_1 m_2$. This could be a typical request/acknowledgement sequence for example, which can be repeated arbitrary many times. It is not possible to specify this choreography using collaboration diagram sets, however by allowing the concatenation of choreographies specified by different collaboration diagrams, we can specify such choreographies.

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 $\mathcal{C}(\mathcal{G})$. 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., $\sigma \in \mathcal{C}(\mathcal{G})$, if and only if $\sigma = \sigma_1 \sigma_2 \dots \sigma_n$ where for all $i \in [1..n]$ $\sigma_i \in M^*$ and there exists a path v_1, v_2, \dots, v_n in G such that $v_1 = v_s, v_n \in Z$, for all $i \in [1..n-1]$ $(v_i, v_{i+1}) \in O$ and for all $i \in [1..n]$ $\sigma_i \in \mathcal{C}(v_i)$.

As the two simple examples we discussed above demonstrate, collaboration diagram sets are strictly more powerful than single collaboration diagrams, and collaboration diagram graphs are strictly more powerful than collaboration diagram sets.

3 Automata Construction

Figure 3 shows an automaton automatically constructed from the collaboration diagram shown in Figure 2. The language accepted by this automaton is exactly the choreography specified by the collaboration diagram in Figure 2.

Given a collaboration diagram $\mathcal{D} = (P, L, M, E, D)$, the corresponding collaboration diagram automaton $\mathcal{A}_{\mathcal{D}} = (M, T, s, F, \delta)$ 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. A collaboration diagram automaton 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.

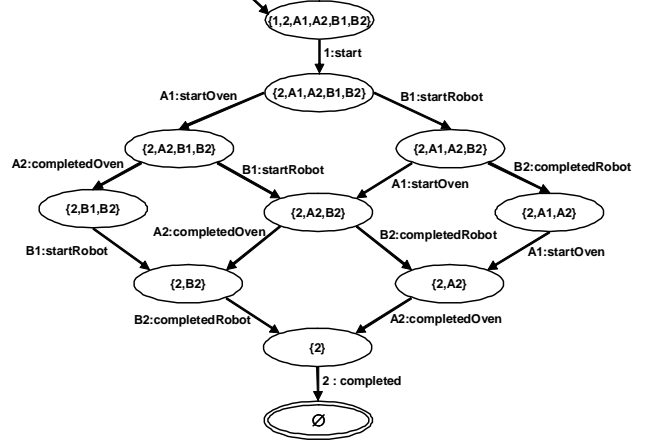


Figure 3. Automata construction

We define the choreography $\mathcal{C}(\mathcal{A})$ defined by the collaboration diagram automaton \mathcal{A} is the language accepted by \mathcal{A} , i.e., $\mathcal{C}(\mathcal{A}) \subseteq M^*$ and $\sigma \in \mathcal{C}(\mathcal{A})$ if and only if $\sigma = m_1, m_2, \dots, m_n$ where for all $i \in [1..n]$ $m_i \in M$ and there exists a path $t_1, t_2, \dots, t_n, t_{n+1}$ in \mathcal{A} such that $t_1 = s, t_{n+1} \in F$, and for all $i \in [1..n]$ $(t_i, m_i, t_{i+1}) \in \delta$.

Collaboration Diagram Automaton Construction

Given a collaboration diagram $\mathcal{D} = (P, L, M, E, D)$, we want to automatically construct a collaboration diagram automaton $\mathcal{A}_{\mathcal{D}} = (M, T, s, F, \delta)$ such that $\mathcal{C}(\mathcal{D}) = \mathcal{C}(\mathcal{A}_{\mathcal{D}})$. We define the set of states of $\mathcal{A}_{\mathcal{D}}$ as $T = 2^E$, i.e., the set of states of $\mathcal{A}_{\mathcal{D}}$ is the power sets of the event set of the collaboration diagram \mathcal{D} . The initial state is defined as $s = E$. The set of final states are defined as $F = \{\emptyset\}$. We define the transition relation δ as follows: For each state $S \subseteq E$, if there exists an event $e \in S$ such that for all $(e', e) \in D$ $e' \notin S$, then

- $e = (l, m, 1) \Rightarrow (S, m, S \setminus \{e\}) \in \delta$,
- $e = (l, m, ?) \Rightarrow \{(S, m, S \setminus \{e\}), (S, \epsilon, S \setminus \{e\})\} \subseteq \delta$,
- $e = (l, m, *) \Rightarrow \{(S, m, S), (S, \epsilon, S \setminus \{e\})\} \subseteq \delta$.

Each state in the automaton represents 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 a transition from E to $E - \{e\}$ to represent the execution of the send event e . If e is an iterative event, then we add a self loop to E to represent arbitrary number of sends. For iterative and conditional events, we also generate ϵ -transitions.

Figure 3 shows the collaboration diagram automaton automatically constructed from the collaboration diagram shown in Figure 2 based on the above construction. The initial state corresponds to the whole event set $E = \{1, 2, A1, A2, B1, B2\}$ meaning that initially all the events

have to be executed, and the final state corresponds to the empty set meaning that there are no more events to be executed. In the initial state, only event 1 is enabled since event 1 has no predecessors in the dependency graph shown in Figure 2 (i.e., it is an initial event). Hence, there is one transition from the initial state to the state $\{2, A1, A2, B1, B2\}$ labeled with the message *start*, corresponding to the execution of event 1. Note that, in state $\{2, A1, A2, B1, B2\}$ events A1 and B1 are both enabled since their only predecessor in the dependency graph is event 1 and event 1 is not in $\{2, A1, A2, B1, B2\}$, meaning that it has already been executed. Hence, there are two transitions from the $\{2, A1, A2, B1, B2\}$, one for event A1 and one for event B1.

Based on the above construction, the number of states generated for a collaboration diagram C with the event set E could be $2^{|E|}$ in the worst case. This worst case is realized only if C has $|E|$ threads, i.e., the number of states is exponential in the number of threads.

Automaton Construction for Collaboration Diagram Sets The above construction algorithm can be extended to collaboration diagram sets as follows. Given a collaboration diagram set $\mathcal{S} = \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_n\}$ where n is the number of collaboration diagrams in \mathcal{S} and each \mathcal{D}_i is in the form $\mathcal{D}_i = (P, L, M, E_i, D_i)$ we want to construct an automaton $\mathcal{A}_{\mathcal{S}} = (M, T, s, F, \delta)$ such that $\mathcal{C}(\mathcal{A}_{\mathcal{S}}) = \mathcal{C}(\mathcal{S})$. For each $\mathcal{D}_i \in \mathcal{S}$ construct the corresponding collaboration diagram automaton $\mathcal{A}_{\mathcal{D}_i} = (M, T_i, s_i, F_i, \delta_i)$ where $\mathcal{C}(\mathcal{D}_i) = \mathcal{C}(\mathcal{A}_{\mathcal{D}_i})$ using the construction defined above. Let $\mathcal{A}_{\mathcal{S}} = (M, T, s, F, \delta)$. We define the set of states of $\mathcal{A}_{\mathcal{S}}$ as $T = \{s\} \cup \bigcup_{\mathcal{D}_i \in \mathcal{S}} T_i$, i.e., the set of states of $\mathcal{A}_{\mathcal{S}}$ consists of a start state s and the power sets of the event sets of the collaboration diagrams that are in \mathcal{S} . Each state in the automaton after the start state represent a set of events that need to be executed. 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: $\delta = (\bigcup_{\mathcal{D}_i \in \mathcal{S}} (s, \epsilon, E_i)) \cup (\bigcup_{\mathcal{D}_i \in \mathcal{S}} \delta_i)$. The automaton $\mathcal{A}_{\mathcal{S}}$ first nondeterministically chooses one of the collaboration diagrams in the collaboration diagram set and then transitions to the initial state of the corresponding collaboration diagram automaton.

Recall that, the number of states in a collaboration diagram automaton $\mathcal{A}_{\mathcal{D}_i}$ generated from a collaboration diagram \mathcal{D}_i is exponential in the number of threads in \mathcal{D}_i . If we determinize the automaton $\mathcal{A}_{\mathcal{S}}$, then the number of states will also be exponential in $|\mathcal{S}|$, i.e., the number of collaboration diagrams in the collaboration diagram set.

Automaton Construction for Collaboration Diagram Graphs Next, we show that 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 can construct an automaton where $\mathcal{A}_{\mathcal{G}} = (M, T, s, F, \delta)$, such that $\mathcal{C}(\mathcal{G}) =$

$\mathcal{C}(\mathcal{A}_{\mathcal{G}})$.

First, for each vertex $v \in V$ of \mathcal{G} , construct an automaton $\mathcal{A}_v = (M, T_v, s_v, F_v, \delta_v)$ using the construction given above for translating collaboration diagram sets to automata (each vertex v corresponds to a singleton collaboration diagram set) such that $\mathcal{C}(v) = \mathcal{C}(\mathcal{A}_v)$. 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 automata constructed for each vertex of \mathcal{G} . We define the initial state of $\mathcal{A}_{\mathcal{G}}$ as the initial state of the automaton 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 automata 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 automata 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., $\delta \supseteq \bigcup_{(v, v') \in O, s \in F_v} (s, \epsilon, s_{v'})$.

4 Synthesizing Peer Implementations

We model the behaviors of peers that participate to a composite web service as concurrently executing finite state machines that interact via messages [10, 12]. We assume that the machines interact with asynchronous messages where each finite state machine has a single FIFO input queue, and the messages are delivered reliably i.e., no message loss or reordering 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 is the set of messages that are either received or sent by p_i , T_i is the finite set of states, $s_i \in T_i$ is the initial state, $F_i \subseteq T_i$ 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 $\text{recv}(m)$ that appends the message to the end of the input queue of the receiver $\text{recv}(m)$, (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 $p_i = \text{recv}(m)$ that removes the message at the head of the input queue of the peer p_i , and (3) an ϵ -transition of the form (t_1, ϵ, t_2) .

A run of a set of peers is a sequence of transitions executed by the peers. A complete run is one such that at the end of the run each peer is in a final state and each FIFO queue is empty. The corresponding sequence of messages induced from the *send transitions* of a complete run is called a conversation (see [12] for the detailed formal definition). The *choreography* $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ of a set of peer state ma-

chines $\mathcal{A}_1, \dots, \mathcal{A}_n$ 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 is a prefix of a complete run. 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 choreography. We say that the peer state machines $\mathcal{A}_1, \dots, \mathcal{A}_n$ *realize* \mathcal{C} if $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n) = \mathcal{C}$ and $\mathcal{A}_1, \dots, \mathcal{A}_n$ are well-behaved.

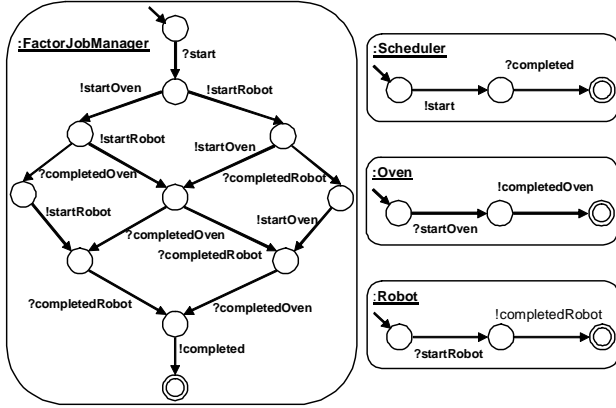


Figure 4. Peer synthesis

Given a choreography specification in the form of a collaboration diagram, it would be helpful to synthesize peer implementations that realize the interactions defined by the choreography specification. Since we already showed that collaboration diagrams can be converted to automata, we can use the collaboration diagram automaton to synthesize the peer state machines. In fact, one can obtain the peer state machines by projecting the transitions of the collaboration diagram automata to the peers. Consider a transition in collaboration diagram automaton for a message send event from peer p_i to peer p_j . This transition should be projected to the peer state machine of peer p_i as a send transition and it should be projected to the peer state machine of peer p_j as a receive transition. Given a peer p_k that is different than peers p_i and p_j , the same transition should be projected to the peer state machine of peer p_k as an ϵ transition. We formalize this projection operation below.

Given a collaboration diagram automaton $\mathcal{A} = (M, T, s, F, \delta)$ we denote the projection of \mathcal{A} 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 4 shows the determinized peer projection of the collaboration diagram automaton shown in Figure 3 to the peers Manager, Scheduler, Oven and Robot. The set of conversations generated by the peer state machines shown in Figure 4 is exactly the choreography specified by the collaboration diagram automaton in Figure 3 and the collaboration diagram in Figure 2. In the next section we show that this is not the case for some collaboration diagrams.

5 Realizability

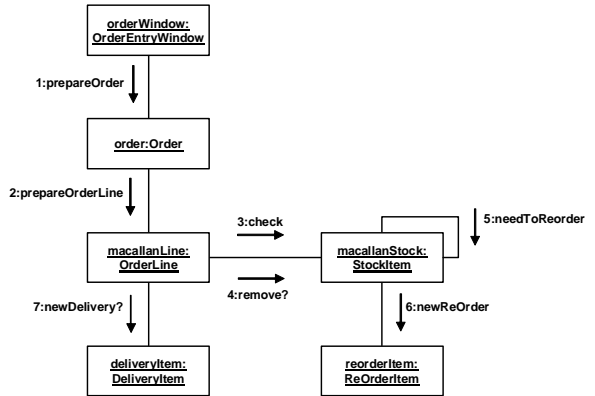


Figure 5. An unrealizable example

Figure 5 shows a collaboration diagram taken from a book on UML [9]. This collaboration diagram is not realizable since it is not possible to guarantee that *newDelivery* message will be sent after the *newReorder* message as required by this collaboration diagram. Based on the ordering of the send events in this collaboration diagram there is no way for OrderLine process to know that StockItem process has already sent the *newReorder* message. Hence, in any implementation of this collaboration diagram, *newDelivery* message may be sent before the *newReorder* message. The realizability analysis techniques we implement in our

toolset will identify that this collaboration diagram is not realizable. It is possible to fix this collaboration diagram by adding an extra message from StockItem to Orderline and changing the event labels so that this new message is sent after the *newReorder* message and before the *newDelivery* message. After this modification, our tool identifies the modified collaboration diagram to be realizable.

We formalize the realizability problem as follows. Let \mathcal{D} be a collaboration diagram. We say that a set of peer state machines $\mathcal{A}_1, \dots, \mathcal{A}_n$ *realize* \mathcal{D} if the set of conversations generated by the peer state machines $\mathcal{A}_1, \dots, \mathcal{A}_n$ is the same as the choreography defined by \mathcal{D} , i.e., $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n) = \mathcal{C}(\mathcal{D})$. A collaboration diagram \mathcal{D} is *realizable* if there exists a set of well-behaved peer state machines which realize \mathcal{D} .

In [6] a sufficient condition for realizability of collaboration diagrams was given. This realizability condition can be checked on the dependency relation of the collaboration diagram. We implemented this realizability condition in our toolset. However, the realizability condition in [6] can only be used in determining realizability of a single collaboration diagram and results on realizability of collaboration diagrams are not directly applicable to collaboration diagrams. A collaboration diagram set that consists of realizable collaboration diagrams may not be realizable, and, it is also possible to have a realizable collaboration diagram set which consists of unrealizable collaboration diagrams [5].

Hence, determining realizability of a single collaboration diagram is not sufficient for checking realizability of a collaboration diagram set. However, our results in this paper show that the realizability of collaboration diagram sets can be reduced to realizability of *conversation protocols* [10]. A conversation protocol is a finite state automaton that specifies a choreography. In fact, the collaboration diagram automata we discussed in Section 3 are conversation protocols. For example, the collaboration diagram automaton shown in Figure 3 is a conversation protocol. Hence, the collaboration diagram to finite state automata translation we presented in Section 3 is equivalent to a translation from a collaboration diagram to a conversation protocol. Furthermore, as we discussed in Section 3, the translation can be extended to collaboration diagram sets and graphs.

In [10, 12] sufficient conditions for realizability of conversation protocols were presented. Given a collaboration diagram set \mathcal{S} , let $\mathcal{A}_{\mathcal{S}}$ be the conversation protocol with the same choreography set. If $\mathcal{A}_{\mathcal{S}}$ satisfies the realizability conditions presented in [10, 12], then we conclude that \mathcal{S} is realizable. Moreover, if the realizability condition holds, \mathcal{S} will be realized by the determinized projections of its collaboration diagram automaton $\mathcal{A}_{\mathcal{S}}$ [10, 12] which means that the peers synthesized based on the algorithm given in Section 4 will realize \mathcal{S} . These results also apply to collaboration diagram graphs.

6 Implementation and Experiments

We implemented the techniques described above in our collaboration diagram analysis and verification tool. We chose the Sparx Systems Enterprise Architect UML Editor [19] as the front end to our tool because of its comprehensive support for UML diagrams and ability to add custom modules. The Add-In we built translates Collaboration Diagrams defined by the user into our implementation of a Collaboration Diagram consisting of Peers, Links, Messages, and Events, based on the formal model defined in Section 2. From there, we are able to construct the dependency graph based on the event orderings defined in each event label as defined in Section 2. Using the dependency graph, we create the collaboration diagram automaton based on the construction given in Section 3. Using the collaboration diagram automaton we generate the peer state machines using the peer synthesis algorithm described in Section 4.

We implement two types of realizability checks. The first one is an implementation of the realizability condition described in [6]. This realizability check is implemented by checking a set of condition on the dependency graph. However, this realizability check cannot be used for checking realizability of collaboration diagram sets and graphs. So we also implemented a translator that converts collaboration diagrams/set/graphs to conversation protocols and uses the Web Service Analysis Tool (WSAT) [11] to check the realizability condition from [10, 12].

Finally, we convert the collaboration diagram automaton to Promela and use the model checker Spin [13] to check LTL properties of the choreography defined by a given collaboration diagram, collaboration diagram set or a collaboration diagram graph. In addition, the Add-In creates visual representations of the dependency graphs, collaboration diagram automaton, and the peer state machines.

Using our collaboration diagram analysis and verification tool we experimented with several examples we found in the literature on collaboration diagrams. For each example, we checked the realizability first. If the example was not realizable we manually added new events to make them realizable. We then used our tool to generate a Promela specification and wrote temporal logic properties for each example collaboration diagram. These specifications were then verified using the Spin model checker.

In Table 1, we summarize each example and our experimental results. All of the examples in Table 1 are single collaboration diagrams, so we able to use the realizability condition from [6] for all of them. In Table 1, R1 corresponds to the realizability condition from [6]. and R2 corresponds to the realizability condition from [10, 12]. Note that both of these conditions are sufficient conditions, so the fact that they are not satisfied does not mean that the collaboration diagram is not realizable. However if they are satisfied, we are sure that the collaboration diagram is realizable. Two

Problem Instance	Source	R1	R2	States
Factory Manager	[20]	YES	NO	383
Order Item	[9]	NO	NO	42 (after fix)
Purchase Order	[4]	YES	NO	246
Company Store	[1]	YES	YES	22
Information Exchange	[14]	YES	YES	50
Voting Booth	[15]	NO	NO	59 (after fix)
Causality Model	[20]	YES	NO	116

Table 1. Realizability analysis and verification results

of the collaboration diagrams we analyzed (Order Item and Voting Booth) violated both of the realizability conditions and after manual inspection we concluded that they were not realizable. Order Item example is shown in Figure 5.

The realizability condition from [6] identified remaining five collaboration diagrams as realizable. Three of these five violated the realizability condition from [10, 12]. All the three examples that violate the the realizability condition from [10, 12] have multiple message threads and violate this realizability condition due to nondeterminism between message send and receive events. Our results show that it is beneficial to use the realizability condition from [6] whenever it is applicable rather than using the more general realizability condition from [10, 12].

Finally, the verification of LTL properties of these examples with the Spin model checker took less than 15 milliseconds each and used 2.5 MBytes of memory. In Table 1 we show the number of states visited during verification. Note that, as expected, the three examples with larger state spaces are the ones with multiple message threads. Spin is able to handle much larger state spaces than any of these examples, so it is safe to say that verification of collaboration diagrams with a model checker is feasible.

The unrealizable examples we discussed above are unrealizable under the concurrent execution semantics we defined in Section 4. We believe that in some of these cases the intention of the developers were to specify a sequential execution rather than a concurrent execution and under the concurrent execution semantics these specifications become unrealizable. Even for such specifications the realizability analysis we implement in our tool is useful since it can help in identifying specifications for which concurrent execution can create problems.

7 Conclusions

In this paper we discussed choreography specification with collaboration diagrams. We defined three classes of collaboration diagrams with increasing expressive power: single collaboration diagrams, collaboration diagram sets and collaboration diagram graphs. We presented techniques for realizability, synthesis and verification and we implemented these techniques in a toolset. Our experimental results indicate that realizability analysis, synthesis and verification of choreographers specified using collaboration diagrams can be done efficiently.

References

- [1] A. Abdurazik and A. J. Offutt. Using uml collaboration diagrams for static checking and test generation. In *Proc. 3rd Int. Conf. on the Unified Modeling Language (UML'00)*, pages 383–395, 2000.
- [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] 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.
- [4] Business process execution language for web services (BPEL), version 1.1. <http://www.ibm.com/developerworks/library/ws-bpel>.
- [5] T. Bultan and X. Fu. Realizability of interactions in collaboration diagrams. Technical Report 2006-11, Computer Science Department, University of California, Santa Barbara, September 2006.
- [6] T. Bultan and X. Fu. Specification of realizable service conversations using collaboration diagrams. In *Proc. IEEE Int. Conf. on Service-Oriented Computing and Applications (SOCA'07)*, pages 122–132, 2007.
- [7] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A theoretical basis of communication-centred concurrent programming. WCD-Working Note, 2006.
- [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*, pages 152–163, 2003.
- [9] M. Fowler. *UML Distilled*. Addison Wesley, 2004.
- [10] 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.
- [11] X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of web services. In *Proc. 16th Int. Conf. on Computer Aided Verification (CAV'04)*, pages 510–514, 2004.
- [12] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, December 2005.
- [13] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, 2003.
- [14] J. Pu, Z. Zhang, Y. Xu, and H. Yang. Reusing legacy cobol code with uml collaboration diagrams via a wide spectrum language. In *Proceedings of the 2005 IEEE International Conference on Information Reuse and Integration (IRI'05)*, pages 78–83, 2005.
- [15] H. C. Purchase, L. Colpoys, M. McGill, and D. A. Carrington. Uml collaboration diagram syntax: An empirical study of comprehension. In *Proc. 1st Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*, pages 13–22, 2002.
- [16] Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *Proceedings of WWW 2007*, 2007.
- [17] G. Salaün and T. Bultan. Realizability of choreographies using process algebra encodings. In *Proc. 7th Int. Conf. on Integrated Formal Methods (IFM'09)*, pages 167–182, 2009.
- [18] D. Skogan, R. Gronmo, and I. Solheim. Web Service Composition in UML. In *Proc. of 8th Int. IEEE Enterprise Distributed Object Computing Conference*, 2004.
- [19] Sparx systems enterprise architect UML editor. <https://www.sparxsystems.com.au/>.
- [20] OMG unified modeling language superstructure, version 2.1.2. <http://www.uml.org/>, October 2007.