Analyzing Conversations: Realizability, Synchronizability, and Verification

Tevfik Bultan[1], Xiang Fu[2], and Jianwen Su[1]

[1] Department of Computer Science,
University of California, Santa Barbara,
CA 93101, USA
{bultan,su}@cs.ucsb.edu
[2] School of Computer and Information Science,
Georgia Southwestern State University,
800 Wheatley Street, Americus, GA 31709, USA
xfu@canes.gsw.edu

Abstract. Conversations provide an intuitive and simple model for analyzing interactions among composite web services. A conversation is the global sequence of messages exchanged among the peers participating to a composite web service. Interactions in a composite web service can be analyzed by investigating the temporal properties of its conversations. Conversations can be specified in a top-down or bottom-up manner. In a top-down conversation specification, the set of conversations is specified first, without specifying the individual behaviors of the peers. In a bottom-up conversation specification, on the other hand, behavior of each peer is specified separately and the conversation set is defined implicitly as the set of conversations generated by these peers. For both top-down and bottom-up specification approaches we are interested in the following: 1) Automatically verifying properties of conversations, and 2) Investigating the effect of asynchronous communication on the conversation behavior. These two issues are closely related since asynchronous communication with unbounded queues increases the difficulty of automated verification significantly.

In this paper, we give an overview of our earlier results on analysis and verification of conversations. We discuss two analysis techniques for identifying bottom-up and top-down conversation specifications that can be automatically verified. Synchronizability analysis identifies bottom-up conversation specifications for which the conversation set remains the same for asynchronous and synchronous communication. Realizability analysis, on the other hand, identifies top-down conversation specifications which can be implemented by a set of finite state peers interacting with asynchronous communication. We discuss sufficient conditions for synchronizability and realizability analyses which are implemented in our Web Service Analysis Tool (WSAT). WSAT can be used for verification of LTL properties of both top-down and bottom-up conversation specifications.

1 Introduction

Web services provide a promising framework for development, integration and interoperability of distributed software applications. Wide scale adoption of the web services technology in critical business applications will depend on the feasibility of building highly dependable services. Web services technology enables interaction of software components across organizational boundaries. In such a distributed environment it is critical to eliminate errors at the design stage, before the services are deployed.

One of the important challenges in static analysis and verification of web services is dealing with asynchronous communication. Asynchronous communication makes most analysis and verification problems undecidable, even when the behaviors of web services are modeled as finite state machines. In this chapter, we give an overview of our earlier results on analysis and verification of interactions among web services in the presence of asynchronous communication.

In our formal model, we assume that a composite web service consists of a set of individual services (peers) which interact with each other using asynchronous communication. In asynchronous communication, the sender and the receiver of a message do not synchronize their send and receive actions. The sender can send a message even when the receiver is not ready to receive that message. When a message arrives, it is stored in the receiver's message buffer. Message buffers are typically implemented as FIFO queues, i.e., messages in a message buffer are processed in the order they arrive. A message will wait in the message buffer without being processed until it moves to the head of the message buffer and the receiver becomes available to consume it by executing a receive action.

Asynchronous communication is important for building robust web services [5]. Since asynchronous communication does not require the sender and the receiver to synchronize during message exchange, temporary pauses in availability of the services, and delays in the delivery of the messages can be tolerated. In practice, asynchronous messaging is supported by message delivery platforms such as Java Message Service (JMS) [26] and Microsoft Message Queuing Service (MSMQ) [32].

Although asynchronous communication improves the robustness of web services it also increases the complexity of design and verification of web service compositions as demonstrated by the two examples below.

Example 1. Consider a small portion of the example from Chapter 1, where the GPS device of the traveler automatically negotiates a purchase agreement with two existing map service providers. Figure 1(a) provides a *top-down* specification of this composition. There are three peers, the traveler (T), map provider 1 (M_1), and map provider 2 (M_2). Assume that before the composition starts, a "call for bid" message has been broadcast to both map providers. The finite state machine in Figure 1 describes the bidding process. Intuitively, the protocol specifies that the first bidder will win the contract. Figure 1(b) demonstrates a sample implementation for all peers involved in the composition. For each peer the sample implementation is generated by a *projection* operation. Given

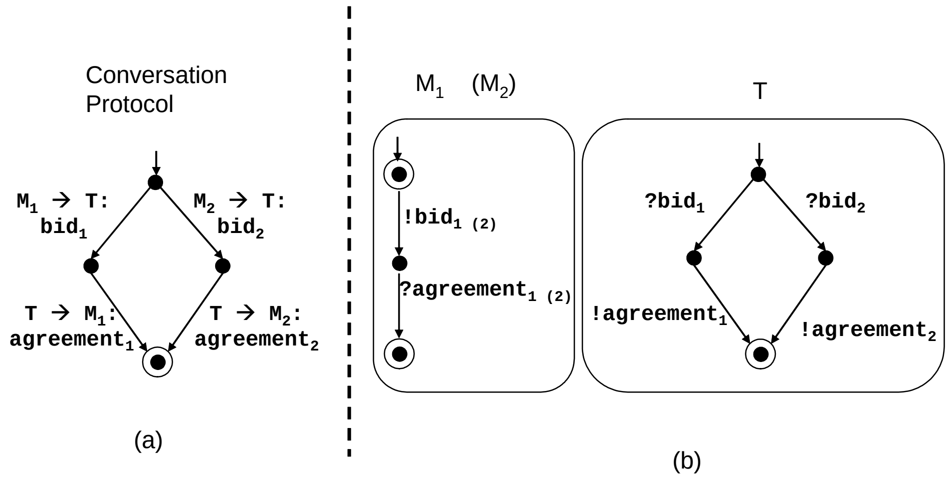

Fig. 1. An unrealizable design due to asynchronous communication.

a protocol (represented as a finite state machine) and a peer to project to, the projection operation replaces the transitions that are labeled with a message that is neither sent nor received by the given peer by ϵ edges, and then minimizes the resulting automaton.

Now, let us consider whether this protocol is *realizable*, i.e., if there are implementations for all peers, whose composition can generate exactly the same set of global behaviors as specified by the protocol automaton in Figure 1(a). If *synchronous communication* is used, the protocol can be executed without any problem. Synchronous communication is similar to communicating with telephone calls, but without answering machines. For a message exchange to occur, the sender and the receiver both have to be on the phone at the same time. With synchronous communication, the peer implementations shown in Figure 1(b) can generate exactly the conversation set as specified by Figure 1(a). Notice that according to these implementations, at the beginning stage, both map service providers call the traveler to bid. When the first bidder successfully makes the call, the traveler, according to the protocol, will not answer any other calls. Hence the call by the second bidder will not go through and the winner is decided. The second bidder will just stay in its initial state, which is also one of its final states.

If we continue with the telephone analogy, *asynchronous communication* is similar to communicating with answering machines where each phone call results in a message that is recorded to the answering machine of the callee. The callee retrieves the messages from the answering machine in the order they are received. If the peer implementations shown in Figure 1(b) interact with asynchronous communication, then the map service providers do not have to synchronize their send actions with the traveler's receive actions. For example, if asynchronous communication is used, at the initial state, both map service providers can send out the *bid* messages. However, in such a scenario only one of them will successfully complete the transaction, and the other will be stuck waiting for an answer and it will never reach a final state. To put it another way, if asynchronous communication is used the composition of these three peers can generate a global behavior that is not described in the protocol given in Figure 1(a). One such undesired behavior can be described using the following sequence of messages:

$$M_1 \to T : bid_1;\ M_2 \to T : bid_2;\ T \to M_1 : agreement.$$

This behavior results with the map service provider 2 being stuck because the traveler will never respond to his request. Again using the telephone analogy, in this scenario, both map providers call the traveler and leave a bid message in the traveler's answering machine. However, based on its state machine (shown on the right side of Figure 1(b)) the traveler listens to only the first bid message in its answering machine and calls back the map provider that left the first message. The other map provider never hears back from the traveler and is stuck at an intermediate state waiting for a call.

A conversation protocol specified as a finite state machine is realizable if and only if it is realized by its projections to all peers [16]. Hence, the protocol in Figure 1 is not realizable.

Figure 1 is an example of how asynchronous communication complicates the design of composite web services. In the next example given below, we discuss how asynchronous communication affects the complexity of verification. This time we consider bottom-up specification of web services.

Example 2. Assume that the GPS device of the traveler needs to invoke the service of the map service provider for a new map whenever the vehicle moves one mile away from its old position. Figure 2 presents two different sets of implementations for the GPS device and the map service provider. Note that, we are assuming that the interaction mechanism is asynchronous communication.

The map provider replies to each request message (`req`) that the client sends with a map data message (`map`); the interaction terminates when the GPS device sends an `end` message. In Figure 2(a), the GPS device does not wait for a map message from the provider after it sends a `req` message. In the resulting global behavior, the `req` and `map` messages can be interleaved arbitrarily, except that at any moment the number of `req` messages is greater than or equal to the number of `map` messages. In Figure 2(b), the GPS device waits for a `map` message before it sends the next `req` message. Now the question is: which composition is *easier* to verify?

We can show that Figure 2(b) is easier to verify because it falls into a category of compositions called *synchronizable* web service compositions. A synchronizable composition produces the same set of conversations under both synchronous and asynchronous communication semantics. When all the peers involved in a composition are finite state machines, their composition using synchronous communication semantics is also a finite state machine. Hence, the problem

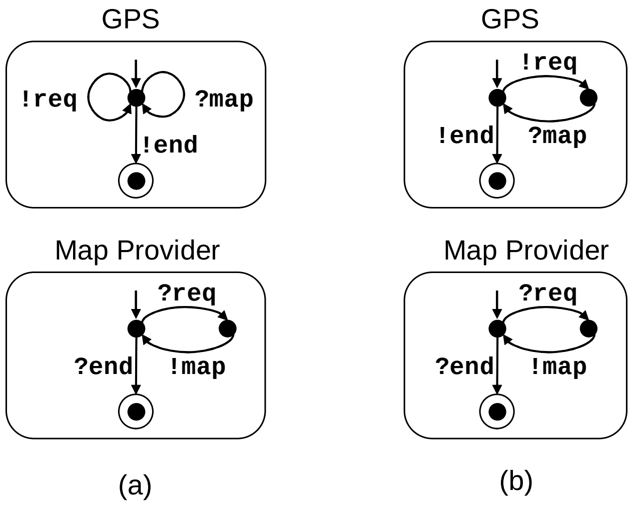

Fig. 2. An unsynchronizable (a) and a synchronizable (b) design.

becomes a finite state verification problem and can be solved using existing finite state model checking techniques and tools. On the other hand, it is impossible to characterize the conversation set of the composition in Figure 2(a) using a finite-state machine because a finite-state machine can not keep track of the number of unacknowledged `req` messages, which can be arbitrarily large.

In the rest of this chapter we will present a survey of our earlier results on realizability and synchronizability of web services that can be used for identifying realizable top-down web service specifications and synchronizable bottom-up web service specifications, respectively. The technical details and proofs of these results can be found in our earlier publications [8, 9, 14–16, 18, 20, 21]. Our goal in this chapter is to provide an overview of our earlier results and explain how they can be applied to the example discussed in Chapter 1. We will also briefly discuss how we integrated these analysis techniques into an automated verification tool for web services [19, 39].

The rest of the chapter is organized as follows. Section 2 presents our conversation model which was originally proposed in [8]. Section 3 discusses the synchronizability analysis presented in [15, 21]. Section 4 discusses the realizability analysis from [14, 16]. Section 5 discusses the extensions of the synchronizability and realizability analyses to protocols in which message contents influence the control flow [18, 20]. Section 6 briefly describes the Web Service Analysis Tool [39, 19]. Section 7 discusses the related work and Section 8 lists our conclusions.

2 A Conversation Oriented Model

In this section we present a formal model for interacting web services [8, 15, 16, 21]. We concentrate our discussion on *static* web service compositions, where the composition structure is statically determined prior to the execution of the composition and we assume that interacting web services do not dynamically create communication channels or instantiate new business processes.

We assume that a *web service composition* is a closed system where a finite set of interacting (individual) web services, called *peers*, communicate with each other via asynchronous messaging. In this section, we consider the problem of how to characterize the interactions among peers. We use the sequence of send events to characterize a global behavior generated by the composition of a set of peers. Based on this conversation model, Linear Temporal Logic (LTL) can be used to express the desired properties of the system.

We will first introduce the notion of a composition schema, which specifies the static interconnection pattern of a web service composition. Then we discuss the specification of each peer, i.e., each participant of a web service composition. Next we discuss how to characterize the interactions among the peers, and introduce the notion of a conversation. Then we present some observations on conversation sets, which motivate the synchronizability analysis presented in the next section.

2.1 Composition Architecture

There are two basic approaches for specifying a web service composition, namely the *top-down* and *bottom-up* specification approaches. In the top-down approach, the desired message exchange sequences among multiple peers are specified, e.g., the IBM Conversation Support Framework for Business Process Integration [22], the Web Service Choreography Description Language (WS-CDL) [40]. The bottom-up approach specifies the logic of individual peers and then peers are composed and their global behaviors are analyzed. Many industry standards, e.g., WSDL [41], BPEL4WS [6], use this approach. In our formalization, the bottom-up and top-down specification approaches have different expressive power. Bottom-up approach is more expressive and can be used to specify more complex interactions.

In order to explain our formal model we will use an example derived from the one discussed in Chapter 1 as our running example in this section.

Example 3. In this example there are three peers interacting with each other: John, Agent and Hotel. John wants to take a vacation. He has certain constraints about where he wants to vacation, so he sends a query to his Agent stating his constraints and asking for advice. The Agent responds to John's query by sending him a suggestion. If John is not happy with the Agent's suggestion he sends another query requesting another suggestion. Eventually, John makes up his mind and sends a reservation request to the hotel he picks. The hotel responds to John's reservation request with a confirmation message. Fig. 3 shows both

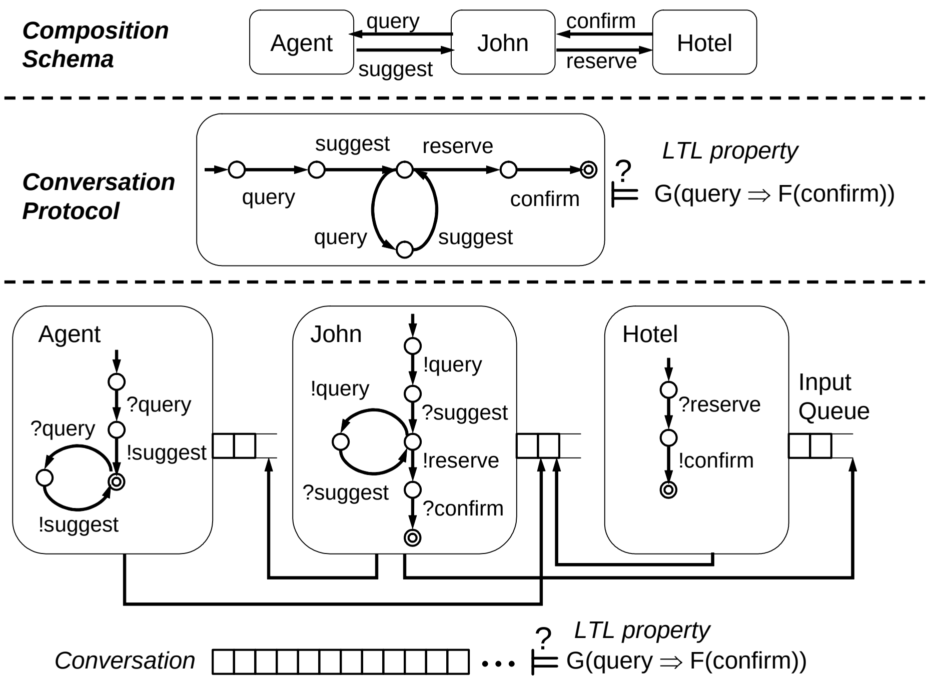

Fig. 3. An example demonstrating our model.

top-down and bottom-up specification of this example in our framework. Top part of Fig. 3 shows the set of peers participating to this composition and the messages exchanged among them. Middle part of Fig. 3 gives a top-down specification of the possible interactions among these peers. Note that, in this top-down specification the behaviors of the individual peers are not given. Bottom part of Fig. 3, on the other hand, is a bottom-up specification which gives behavioral descriptions of all the peers participating to the composition. The interaction behavior is implicitly defined as the set of interactions generated by these peers. In either approach we are interested in verifying LTL properties of interactions and we model the interactions as conversations. Below we will use this example to explain different components of our framework.

A composition schema specifies the set of peers and the set of messages exchanged among peers [8, 21].

Definition 1. *A* composition schema *is a tuple* (P, M) *where* $P = \{p_1, \ldots, p_n\}$ *is the set of* peer prototypes, *and* M *is the set of* messages. *Each peer prototype* $p_i = (M_i^{in}, M_i^{out})$ *is a pair of disjoint sets of messages* ($M_i^{in} \cap M_i^{out} = \emptyset$), *where* M_i^{in} *is the set of incoming messages,* M_i^{out} *is the set of outgoing messages, and* $M_i = M_i^{in} \cup M_i^{out}$ *is the set of messages of peer* p_i *where* $\bigcup_{i \in [1..n]} M_i^{in} =$

$\bigcup_{i\in[1..n]} M_i^{out} = M$. *We assume that each message has a unique sender and a unique receiver, and a peer cannot send a message back to itself.*

For example, top part of Fig. 3 shows a composition schema where the set of peer prototypes are $P = \{\mathsf{Agent}, \mathsf{John}, \mathsf{Hotel}\}$, and the set of messages are $M = \{\mathsf{query}, \mathsf{suggest}, \mathsf{confirm}, \mathsf{reserve}\}$. The input and output messages for peer prototypes are defined as: $M^{in}_{\mathsf{Agent}} = \{\mathsf{query}\}$, $M^{out}_{\mathsf{Agent}} = \{\mathsf{suggest}\}$, $M^{in}_{\mathsf{John}} = \{\mathsf{suggest}, \mathsf{confirm}\}$, $M^{out}_{\mathsf{John}} = \{\mathsf{query}, \mathsf{reserve}\}$, $M^{in}_{\mathsf{Hotel}} = \{\mathsf{reserve}\}$, $M^{out}_{\mathsf{Hotel}} = \{\mathsf{confirm}\}$.

2.2 Top-Down vs. Bottom-Up Specification

Conversation protocols correspond to top-down specification of interactions among web services. Middle part of Fig. 3 (labeled conversation protocol) shows a top-down specification for the interactions among a set of peers. We define a conversation protocol as a finite state machine as follows:

Definition 2. *Let $S = (P, M)$ be a composition schema. A* conversation protocol *over S is a tuple $\mathcal{R} = \langle (P, M), \mathcal{A}\rangle$ where $\mathcal{A}$ is a finite state automaton with alphabet M. We let $L(\mathcal{R}) = L(\mathcal{A})$, i.e., the language recognized by $\mathcal{A}$.*

The conversation protocol in Fig. 3 corresponds to a finite state automaton with the set of states $\{s_0, s_1, s_2, s_3, s_4, s_5\}$, the initial state s_0, the set of final states $\{s_5\}$, the alphabet $\{\mathsf{query}, \mathsf{suggest}, \mathsf{confirm}, \mathsf{reserve}\}$ and the set of transitions: $\{(s_0, \mathsf{query}, s_1)$, $(s_1, \mathsf{suggest}, s_2)$, $(s_2, \mathsf{query}, s_3)$, $(s_3, \mathsf{suggest}, s_2)$, $(s_2, \mathsf{reserve}, s_4)$, $(s_4, \mathsf{confirm}, s_5)\}$.

Note that the language recognized by the conversation protocol in Fig. 3 can be characterized by the following regular expression:
$\mathsf{query\ suggest\ (query\ suggest)^*\ reserve\ confirm}$

A bottom-up specification consists of a set of finite state peers. Bottom part of Fig. 3 shows the bottom-up specification of the same web service composition. We call a bottom-up specification a *web service composition* which is defined as follows:

Definition 3. *A* web service composition *is a tuple $\mathcal{W} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n\rangle$, where (P, M) is a composition schema, $n = |P|$, and $\mathcal{A}_i$ is the* peer implementation *for the peer prototype $p_i = (M_i^{in}, M_i^{out}) \in P$.*

We assume that each peer implementation is given as a finite state machine. Each peer implementation describes the control flow of a peer. Since peers communicate with asynchronous messages, each peer is equipped with a FIFO queue to store incoming messages. Formally, a peer implementation is defined as follows:

Definition 4. *Let $S = (P, M)$ be a composition schema and $p_i = (M_i^{in}, M_i^{out}) \in P$ be a peer prototype. A* peer implementation *$\mathcal{A}_i$ for a peer prototype p_i is a finite state machine with an input queue. Its message set is $M_i = M_i^{in} \cup M_i^{out}$. A transition between two states t_1 and t_2 in $\mathcal{A}_i$ can be one of the following three types:*

1. *a send-transition of the form* $(t_1, !m_1, t_2)$ *which sends out a message* $m_1 \in M_i^{out}$ *(i.e., inserts the message to the input queue of the receiver),*
2. *a receive-transition of the form* $(t_1, ?m_2, t_2)$ *which consumes a message* $m_2 \in M_i^{in}$ *from the input queue of* $\mathcal{A}_i$*,*
3. *an* ϵ*-transition of the form* (t_1, ϵ, t_2)*.*

Bottom part of Fig. 3 presents the peer implementations for the peer prototypes shown at the top. For example, the peer implementation for the peer Agent corresponds to a finite state machine with the set of states $\{q_0, q_1, q_2, q_3\}$, the initial state q_0, the set of final states $\{q_2\}$, the message set $\{\mathsf{query}, \mathsf{suggest}\}$ and the set of transitions: $\{(q_0, ?\mathsf{query}, q_1)$, $(q_1, !\mathsf{suggest}, q_2)$, $(q_2, ?\mathsf{query}, q_3)$, $(q_3, !\mathsf{suggest}, q_2)\}$. Similarly, the peer John corresponds to a finite state machine with the set of states $\{t_0, t_1, t_2, t_3, t_4, t_5\}$, the initial state t_0, the set of final states $\{t_5\}$, the message set $\{\mathsf{query}, \mathsf{suggest}, \mathsf{confirm}, \mathsf{reserve}\}$ and the set of transitions: $\{(t_0, !\mathsf{query}, t_1)$, $(t_1, ?\mathsf{suggest}, t_2)$, $(t_2, !\mathsf{query}, t_3)$, $(t_3, ?\mathsf{suggest}, t_2)$, $(t_2, !\mathsf{reserve}, t_4)$, $(t_4, ?\mathsf{confirm}, t_5)\}$. And finally, the peer Hotel corresponds to a finite state machine with the set of states $\{r_0, r_1, r_2\}$, the initial state r_0, the set of final states $\{r_2\}$, the message set $\{\mathsf{reserve}, \mathsf{confirm}\}$ and the set of transitions: $\{(r_0, ?\mathsf{reserve}, r_1)$, $(r_1, !\mathsf{confirm}, r_2)\}$. We will use these peer implementations as our running example for the rest of this section.

2.3 Conversations

A conversation is the sequence of messages exchanged among the peers during an execution, recorded in the order they are sent. In order to formalize the notion of conversations we first need to define the configurations of a composite web service and the derivation relation which specifies how the system evolves from one configuration to another [8, 16, 21].

Definition 5. *Let* $\mathcal{W} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ *be a web service composition. A configuration of* $\mathcal{W}$ *is a* $(2n)$*-tuple of the form*

$$(Q_1, t_1, ..., Q_n, t_n),$$

where for each $j \in [1..n]$, $Q_j \in (M_j^{in})^*$, $t_j \in T_j$*. Here* t_j, Q_j *denote the local state and the queue contents of* $\mathcal{A}_j$*, respectively.*

Intuitively a configuration records a snap-shot during the execution of a web service composition by recording the local state and the FIFO queue contents of each peer. For example, the initial configuration of our running example is: $(\epsilon, q_0, \epsilon, t_0, \epsilon, r_0)$ where all the peers are in their initial states and all the queues are empty. When the peer John takes the transition $(t_0, !\mathsf{query}, t_1)$, the next configuration is $(\mathsf{query}, q_0, \epsilon, t_1, \epsilon, r_0)$, i.e., in the next configuration the message query is in the input queue of the peer Agent and the peer John is in state t_1. Then, the peer Agent can receive the query message by taking the $(q_0, ?\mathsf{query}, q_1)$ transition which would lead to the following configuration: $(\epsilon, q_1, \epsilon, t_1, \epsilon, r_0)$, i.e., the message query is removed from the input queue of the peer Agent and the peer Agent is now in state q_1.

We can formalize this kind of evolution of the system from one configuration to another as a derivation relation using the transitions of the peer implementations. A derivation step is an atomic and minimal step in a global behavior generated by a web service composition. Given two configurations c, c', we say that c *derives* c', written as $c \rightarrow c'$, if it is possible to go from configuration c to configuration c' by one of the following three types of derivation steps:

1. *send action*, where one peer sends out a message m to another peer (denoted as $c \xrightarrow{!m} c'$). The send action results in the state transition for the sender, and the transmitted message is placed in the input queue of the receiver.
2. *receive action*, where one peer consumes the message m that is at the head of its input message queue (denoted as $c \xrightarrow{?m} c'$). The receive action results in the state transition of the receiver and the removal of the consumed message from the head of the receiver's input queue.
3. ϵ *action*, where one peer takes an ϵ transition (denoted as $c \xrightarrow{\epsilon} c'$). This action results in the state transition for that peer, however, it does not affect any of the message queues.

For our running example two example derivations we discussed above can be written as: $(\epsilon, q_0, \epsilon, t_0, \epsilon, r_0) \xrightarrow{!\mathsf{query}} (\mathsf{query}, q_0, \epsilon, t_1, \epsilon, r_0)$ and $(\mathsf{query}, q_0, \epsilon, t_1, \epsilon, r_0) \xrightarrow{?\mathsf{query}} (\epsilon, q_1, \epsilon, t_1, \epsilon, r_0)$.

Now we can define a run of a web service composition as follows:

Definition 6. *Let $\mathcal{W} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be a web service composition, a sequence of configurations $\gamma = c_0 c_1 \ldots c_k$ is a run of $\mathcal{W}$ if it satisfies the following 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 $i \in [1..n]$, and ϵ is the empty word.*
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 $i \in [1..n]$.*

We define the *send sequence* generated by γ, denoted by $ss(\gamma)$, as the sequence of messages containing one message for each send action (i.e., $c \xrightarrow{!m} c'$) in γ, where the messages in $ss(\gamma)$ are recorded in the order they are sent.

For example, a run of our running example would be:
$(\epsilon, q_0, \epsilon, t_0, \epsilon, r_0) \xrightarrow{!\mathsf{query}} (\mathsf{query}, q_0, \epsilon, t_1, \epsilon, r_0) \xrightarrow{?\mathsf{query}} (\epsilon, q_1, \epsilon, t_1, \epsilon, r_0) \xrightarrow{!\mathsf{suggest}}$
$(\epsilon, q_2, \mathsf{suggest}, t_1, \epsilon, r_0) \xrightarrow{?\mathsf{suggest}} (\epsilon, q_2, \epsilon, t_2, \epsilon, r_0) \xrightarrow{!\mathsf{reserve}} (\epsilon, q_2, \epsilon, t_4, \mathsf{reserve}, r_0) \xrightarrow{?\mathsf{reserve}}$
$(\epsilon, q_2, \epsilon, t_4, \epsilon, r_1) \xrightarrow{!\mathsf{confirm}} (\epsilon, q_2, \mathsf{confirm}, t_4, \epsilon, r_2) \xrightarrow{?\mathsf{confirm}} (\epsilon, q_2, \epsilon, t_5, \epsilon, r_2)$.

The send sequence generated by this run is: query suggest reserve confirm.

Finally, we define the conversations as follows:

Definition 7. *A word w over M ($w \in M^*$) is a* conversation *of web service composition $\mathcal{W}$ if there exists a run γ such that $w = ss(\gamma)$, i.e., a conversation is the send sequence generated by a run. The* conversation set *of a web service composition $\mathcal{W}$, written as $\mathcal{C}(\mathcal{W})$, is the set of all conversations for $\mathcal{W}$.*

For example, the conversation set of our running example, the web service composition at the bottom of Fig. 3, can be captured by the regular expression: $\mathsf{query\ suggest\ (query\ suggest)}^* \mathsf{\ reserve\ confirm}$.

Linear Temporal Logic (LTL) can be used to characterize the properties of conversation sets in order to specify the desired system properties. The semantics of LTL formulas can be adapted to conversations by defining the set of atomic propositions as the power set of messages. For example, the composition in Fig. 3 satisfies the LTL property: $\mathbf{G}(\mathsf{query} \Rightarrow \mathbf{F}(\mathsf{confirm}))$, where $\mathbf{G}$ and $\mathbf{F}$ are temporal operators which mean "globally" and "eventually", respectively.

Standard LTL semantics is defined on infinite sequences [11] whereas in our definitions above we used finite conversations. It is possible to extend the definitions above to infinite conversations and then use the standard LTL semantics as in [14, 16]. We can also adapt the standard LTL semantics to finite conversations by extending each conversation to an infinite string by adding an infinite suffix which is the repetition of a special termination symbol.

Unfortunately, due to the asynchronous communication of web services, LTL verification of conversations of web service compositions is undecidable [16]:

Theorem 1. *Given a web service composition $\mathcal{W}$ and an LTL property ϕ, determining if all the conversations of $\mathcal{W}$ satisfy ϕ is undecidable.*

The proof is based on an earlier result on Communicating Finite State Machines (CFSMs) [7]. We can show that a web service composition is essentially a system of CFSMs. It is known that CFSMs can simulate Turing Machines [7]. Similarly, one can show that, given a Turing Machine TM it is possible to construct a web service composition $\mathcal{W}$ that simulates TM and exchanges a special message (say m_t) once TM terminates. Thus TM terminates if and only if the conversations of $\mathcal{W}$ satisfy the LTL formula $\mathbf{F}(m_t)$, which means that "eventually message m_t will be sent". Hence, undecidability of the halting problem implies that verification of LTL properties of conversations of a web service composition is an undecidable problem.

3 Synchronizability

Asynchronous communication among web services leads to the undecidability of the LTL verification problem. If synchronous communication is used instead of asynchronous communication, the set of configurations of a web service composition would be a finite set, and it is well-known that LTL model checking is decidable for finite state systems. In this section we discuss the *synchronizability analysis* [15, 21] which identifies bottom-up web service specifications which generate the same conversation set with synchronous and asynchronous communication semantics. We call such web service compositions *synchronizable*. We can verify synchronizable web service compositions using the synchronous communication semantics, and the verification results we obtain are guaranteed to hold for the asynchronous communication semantics.

3.1 Synchronous Communication

To define synchronizability we first have to define synchronous communication. Intuitively, synchronous communication requires that the sender and receiver of a message should take the send and receive actions simultaneously to complete the message transmission. In other words, the send and receive actions of a message transmission form an atomic and non-interruptible step. In the following, we define the synchronous global configuration and synchronous communication semantics.

Given a web service composition $\mathcal{W} = \langle (P, M), \mathcal{A}_1, ..., \mathcal{A}_n \rangle$ where each automaton $\mathcal{A}_i$ describes the behavior of a peer, the configuration of a web service composition 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 $\mathcal{A}_j$. Notice that in a syn-configuration only the local automata state of each peer is recorded—peers do not need message buffers to store the incoming messages due to the synchronous communication semantics.

For two syn-configurations c, c', we say that c *synchronously derives* c', written as $c \rightarrow_{syn} c'$, if c' is the result of simultaneous execution of the send and receive actions for the same message by two peers, or the execution of an ϵ action by a single peer.

The definition of the derivation relation between two syn-configurations is different than the asynchronous case. In the synchronous case a send action can only be executed concurrently with a matching receive action, i.e., sending and receiving of a message occur synchronously. We call this semantics the *synchronous semantics* for a web service composition and the semantics defined in Section 2 is called the *asynchronous semantics*.

The definitions of a run, a send sequence and a conversation for synchronous semantics is similar to those of the asynchronous semantics given in Section 2 (we will use "syn" as a prefix to distinguish between the synchronous and asynchronous versions of these definitions when it is not clear from the context). Given a web service composition $\mathcal{W}$, let $\mathcal{C}_{\text{syn}}(\mathcal{W})$ denote the conversation set under the synchronous semantics. Then synchronizability is defined as follows:

Definition 8. *A web service composition $\mathcal{W}$ is* synchronizable *if its conversation set remains the same when the synchronous semantics is used instead of the asynchronous semantics, i.e., $\mathcal{C}(\mathcal{W}) = \mathcal{C}_{\text{syn}}(\mathcal{W})$.*

Clearly, if a web service composition is synchronizable, then we can verify its interaction behavior using synchronous semantics (without any input queues) and the results of the verification will hold for the behaviors of the web service composition in the presence of asynchronous communication with unbounded queues.

Given a web service composition $\mathcal{W}$, its conversation set with respect to synchronous semantics is always a subset of its conversation set with respect to asynchronous semantics, i.e., $\mathcal{C}_{\text{syn}}(\mathcal{W}) \subseteq \mathcal{C}(\mathcal{W})$ [21]. In some cases the containment relationship can be strict, i.e., there are web service compositions that are not synchronizable. The following is an example.

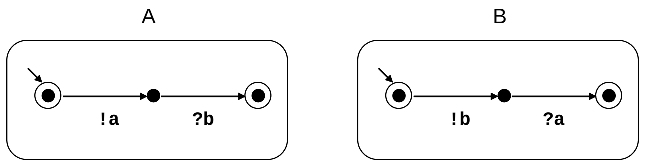

Fig. 4. An example specification that is not synchronizable.

Example 4. Consider a web service composition $\mathcal{W}$ in Figure 4. Two peers A and B can exchange two messages a (from A to B) and b (from B to A). The peer implementation of A sends out a and then waits for and consumes message b from its input queue. Peer b sends out b first then receives a. Obviously, if asynchronous semantics is used there exists a run which generates the conversation ab. However, note that, when synchronous semantics is used there is no run which generates the same conversation, because at the initial state both peers are trying to send out a message and neither of them can get the co-operation of the other peer to complete the send operation. Based on the definitions of the conversation sets we have $\mathcal{C}(\mathcal{W}) = \{ab, ba\}$ and $\mathcal{C}_{\mathrm{syn}}(\mathcal{W}) = \emptyset$. Hence, $\mathcal{W}$ is not synchronizable.

3.2 Synchronizability Analysis

We now present two conditions for identifying synchronizable web service compositions. Together these conditions guarantee synchronizability, i.e., they form a sufficient condition for synchronizability.

Synchronous compatible condition: If we construct the synchronous composition of a set of peers, the synchronous compatible condition requires that for each syn-configuration c that is reachable from the initial configuration, if there is a peer which has a send transition for a message m from its local state in c, then the receiver of m should have a receive transition for m either from its local state in c or from a configuration reachable from c via ϵ-actions.

Note that, the composition of A and B in Figure 4 does not satisfy the synchronous compatible condition. The initial syn-configuration c_0 of the composition can be represented as a tuple (s_1^A, s_1^B) where s_1^A and s_1^B are the local initial states of A and B respectively. Obviously, at c_0 peer A can send out a however it is not able to, because B is not in a state where it can receive the message.

An algorithm for checking the synchronous compatible condition is given in [21]. The basic idea in the algorithm is to construct a finite state machine that is the product (i.e., the synchronous composition) of all peers. Each state (i.e., syn-configuration) of the product machine is a vector of local states of all peers. During the construction, if we find a peer ready to send a message but the corresponding receiver is not ready to receive it (either immediately or after executing several ϵ-actions), the composition is identified as not synchronous compatible. If all states of the product machine are examined without finding a violation of the synchronous compatible condition, then the algorithm returns true. The worst case complexity of the algorithm is quadratic on the size of the product and the size of the product is exponential in the number of peers.

Autonomous condition: A web service composition is autonomous if each peer, at any moment, can do only one of the following: 1) terminate, 2) send a message, or 3) receive a message.

To check the autonomous condition we determinize each peer implementation and check that out-going transitions for each non-final state are either all send transitions or all receive transitions [21]. We also check that final states have no out-going transitions. The complexity of the algorithm can be exponential in the size of the peers in the worst case due to the determinization step.

In Figure 1(b), neither of the peer implementations of the map service providers (M_1 and M_2) are autonomous because there is a transition originating from the initial state which is also a final state. However, the implementation of traveler (T) is autonomous.

In Figure 2(a) the implementation of GPS is not autonomous, because at the initial state the peer can send message `req` and receive message `map`.

We now present the key result concerning the synchronizability analysis. The proof for the following results can be found in [21].

Theorem 2. *Let $\mathcal{W} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be a web service composition. If $\mathcal{W}$ is synchronous compatible and autonomous, then for any conversation generated by $\mathcal{W}$ there exists a run which generates the same conversation in which every send action is immediately followed by the corresponding receive action.*

When the synchronous compatibility and autonomy conditions are satisfied by a web service composition, then for each conversation generated by that composition, there is always a run which generates the same conversation where each send action is immediately followed by the corresponding receive action. By collapsing the pairs of send/receive actions for the same message, we get a synchronous run which generates the same conversation. Then based on Theorem 2 we get the following result:

Theorem 3. *Let $\mathcal{W} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be a web service composition. If $\mathcal{W}$ is synchronous compatible and autonomous, then $\mathcal{W}$ is synchronizable.*

Theorem 3 implies that web service compositions that satisfy the two synchronizability conditions can be analyzed using the synchronous communication semantics and the verification results hold for asynchronous semantics.

Notice that, synchronizability does not imply deadlock freedom. Think about the following composition of two peers A and B, which exchange messages m_1 (from A to B) and m_2 (from B to A). If A accepts one word $?m_2$, and B accepts one word $?m_1$, it is not hard to verify that the composition of A and B is synchronizable, however, they are involved in a deadlock right at the initial state

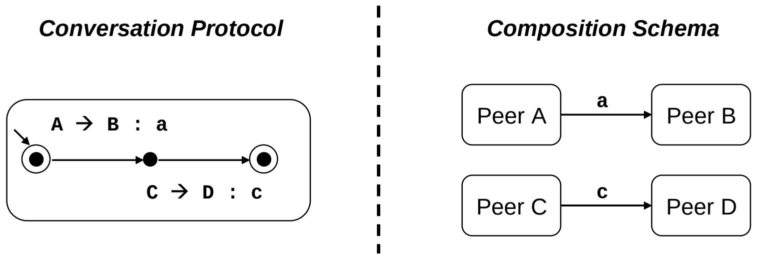

Fig. 5. A non-realizable protocol in both synchronous and asynchronous semantics.

since both peers are waiting for each other. Hence, before the LTL verification of a web service composition, designers may have to check the composition for deadlocks. However, for synchronizable web service compositions the deadlock check can be done using the synchronous semantics (instead of the asynchronous semantics), since it is possible to show that [13] a synchronizable web service composition has a run (with asynchronous semantics) that leads to a deadlock if and only if it has a syn-run (with synchronous semantics) that leads to a deadlock.

4 Realizability of Conversation Protocols

In this section we discuss the realizability problem for top-down web-service specifications, i.e., conversation protocols [8, 16]. We also discuss the relationship between synchronizability and realizability analyses.

Intuitively, realizability means that given a conversation protocol it can be realized by some web service composition, i.e., the conversation set generated by the web service composition is exactly the same as the language accepted by the conversation protocol.

Definition 9. *Let $S = (P, M)$ be a composition schema, and let the conversation protocol $\mathcal{R}$ and the web service composition $\mathcal{W}$ both share the same schema S. We say that $\mathcal{W}$* realizes *$\mathcal{R}$ if $\mathcal{C}(\mathcal{W}) = L(\mathcal{R})$. A conversation protocol $\mathcal{R}$ is* realizable *if there exists a web service composition that realizes $\mathcal{R}$.*

Let us first consider the following question: Are all conversation protocols realizable? The answer is negative as we show below.

Example 5. Figure 5 shows a conversation protocol over four peers A, B, C, and D. The message alphabet consists of two messages: a (from A to B) and c (from C to D). The protocol specifies a conversation set which consists of one conversation only ($\{ac\}$). It is not hard to see that any peer implementation which can generate the conversation ac can generate ca too, because there is no way for peers A and C to coordinate their actions. Hence, the conversation protocol shown in Figure 5 is not realizable.

Notice that the problem of realizability is also an issue for synchronous communication semantics. For example, the protocol in Figure 5 is not realizable using synchronous semantics either. However, the asynchronous semantics does introduce new complexities into this problem as discussed in [16, 21].

Below we will argue that realizability of conversation protocols can be solved by extending the synchronizability analysis. First we need to introduce notions of projection and join for peer implementations and conversation protocols.

For a composition schema (P, M) the projection of a word w to the alphabet M_i of the peer prototype p_i, denoted by $\pi_i(w)$, is a subsequence of w obtained by removing all the messages which are not in M_i. When the projection operation is applied to a set of words the result is the set of words generated by application of the projection operator to each word in the set.

For composition schema (P, M) let $n = |P|$ and let $L_1 \subseteq M_1^*, \ldots, L_n \subseteq M_n^*$, the join operator is defined as follows:

$$\textsc{join}(L_1, \ldots, L_n) = \{w \mid w \in M^*, \forall i \in [1..n] : \pi_i(w) \in L_i\}.$$

Let $L = \{ac\}$ be the conversation set specified by the conversation protocol in Figure 5. $\pi_A(L) = \{a\}$, $\pi_B(L) = \{a\}$, $\pi_C(L) = \{c\}$, and $\pi_D(L) = \{c\}$. The join of all these peer projections will produce a larger conversation set:

$$\textsc{join}(\pi_A(L), \pi_B(L), \pi_C(L), \pi_D(L)) = \{ac, ca\}$$

We now introduce a third condition used in the realizability analysis.

Lossless join condition: A conversation protocol $\mathcal{R}$ is *lossless join* if $L(\mathcal{R}) = \textsc{join}(\pi_1(L(\mathcal{R})), \ldots, \pi_n(L(\mathcal{R})))$, where n is the number of peers involved in the protocol.

The lossless join condition requires that a conversation protocol should include all words in the join of its projections to all peers. An algorithm for checking the lossless join property is given in [21]. Intuitively, the lossless join property requires that the protocol should be realizable under synchronous communication semantics. The algorithm simply projects the conversation protocol to each peer prototype, and then constructs the product of all projections. If the resulting product is equivalent to the protocol, then the algorithm reports that the lossless join property is satisfied. The algorithm can be exponential in the size of the conversation protocol in the worst case due to the equivalence check on two nondeterministic finite state machines.

The lossless join property is a necessary condition for the realizability of conversation protocols. If synchronous semantics is used, it is the necessary and sufficient condition. The following result connects the synchronizability analysis and the realizability analysis:

Theorem 4. *Given a conversation protocol $\mathcal{R} = \langle (P, M), \mathcal{A} \rangle$ where $n = |P|$, let $\mathcal{W} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be a web service composition s.t. for each $i \in [1..n]$, $\mathcal{A}_i$ is the minimal deterministic FSA such that $L(\mathcal{A}_i) = \pi_i(L(\mathcal{R}))$. If $\mathcal{W}$ is synchronizable, and $\mathcal{R}$ is lossless join, then $\mathcal{R}$ is realized by $\mathcal{W}$.*

The proof of this property follows directly from Theorem 3 and the fact that the synchronous composition of a set of peers accepts the join of their languages. Theorem 4 demonstrates an interesting relationship between the synchronizability analysis introduced in [21] and the realizability analysis introduced in [16].

5 Message Contents

In the previous sections, we assumed that the contents of the messages were abstracted away, i.e., in our formal model messages did not have any content. This type of abstraction would be fine as long as the contents of the messages do not influence the control flow of the peers. In practice, this assumption may be too restrictive, i.e., contents of a message received by a peer may influence the control flow of that peer. One natural question is: Is it possible to extend the analyses introduced in the earlier sections to an extended web service model where messages have contents?

To facilitate the technical discussions, let us extend the web service specification framework as follows: Assume that each peer in a web service composition is a *guarded automaton* instead of a standard finite state automaton. In the guarded automata model, messages have contents. A message class defines the structure of a message and a message is an instance of a message class. Each transition is labeled with a message class and a guard. A guard is a relational expression which evaluates to a boolean value. The building elements of a guard are the attributes of messages. Only when the guard evaluates to true, can the transition be fired (if the automaton is in its source state).

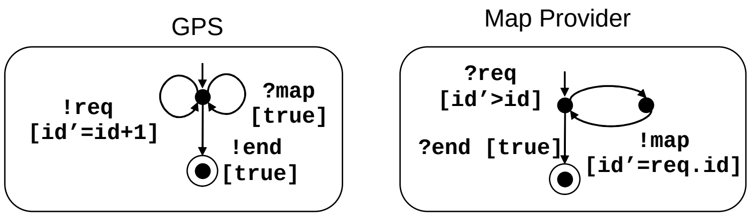

Fig. 6. An example with message contents.

Example 6. Figure 6 presents a modified version of the example given in Figure 3 by extending the messages with contents and the transitions with guards. In Figure 6 message classes req and map have an integer attribute id. The guard of each transition is a boolean expression enclosed in a pair of square brackets. For example, the send transition !req has a guard "$\mathsf{id}' = \mathsf{id} + 1$". This means that whenever a new req message is sent, its id is attribute is incremented by 1. Note that here the primed-variable id′ represents the "next value" of the attribute id. The receive transition ?req in the map provider service requires that the ids of the incoming req messages must monotonically increase. Obviously the implementation of GPS satisfies this requirement. Similarly, the guard of the send transition !map guarantees that the id attribute of a map message must match that of the most recent req message.

We call a web service composition a "guarded composition" if its peers are specified using guarded automata. Similarly we define the "guarded peer", "guarded protocol" and so on. Given a guarded automaton, if we remove the contents of the messages and the guards of the transitions we get a standard finite state automaton. We call this resulting automaton the *skeleton automaton.* Similarly we use the name "skeleton peer", "skeleton composition", and "skeleton protocol" to refer the skeleton of a guarded peer, guarded composition, and guarded protocol, respectively.

One natural conjecture is the following: Does the synchronizability of a skeleton composition imply the synchronizability of the corresponding guarded composition? The answer is negative as demonstrated by the following example.

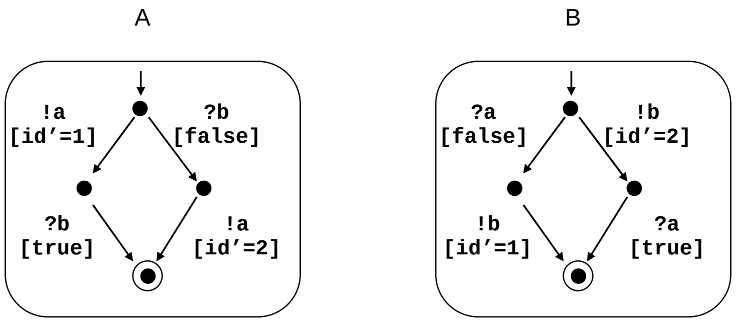

Fig. 7. A counter-example for the conjecture on skeleton synchronizability.

Example 7. Figure 7 presents an example guarded composition that shows that the above conjecture is false. The composition consists of two peers A and B. Peer A can send a message a to B, and B can send a message b to A. Both messages a and b have an integer attribute id which varies between 1 and 2. In the following we use the notation a(1) to represent a message a whose attribute id is 1. The composition produces two conversations $a(1)b(2)$ and $b(2)a(1)$. In addition, to produce these two conversations, asynchronous semantics has to be used. For example, to produce $a(1)b(2)$, the message $a(1)$ has to stay in the input queue of peer B when b is sent out. Such a conversation cannot be generated by synchronous composition of these two peers.

On the other hand, if we drop the message contents and guards of the guarded automata in Figure 7 we get two standard finite state automata, which accept conversations $\{\mathsf{!a?b}, \mathsf{?b!a}\}$ and $\{\mathsf{!b?a}, \mathsf{?a!b}\}$, respectively. The composition of these two finite state automata peers are synchronizable.

Example 7 demonstrates that the synchronizability of the skeleton composition does not imply the synchronizability of the guarded composition. Interestingly, if the skeleton composition is not synchronizable, it does not imply that the guarded composition is not synchronizable either. Similar observations hold for conversation protocols. It is not possible to tell if a guarded conversation protocol is realizable or not based on the realizability of its skeleton protocol. Examples and arguments for the above conclusions can be found in [13, 18, 20].

Skeleton of a guarded composition, however, can still be used for synchronizability analysis. The following theorem forms the basis of a skeleton analysis for synchronizability of guarded compositions.

Theorem 5. *A guarded web service composition is synchronizable if its skeleton satisfies the autonomous and synchronous compatible conditions.*

Theorem 5 implies that if the skeletons of a guarded composition satisfies the two sufficient synchronizability conditions, then the guarded composition is guaranteed to be synchronizable. The proof of Theorem 5 is based on the following observation. For any run of a guarded composition, we can find a corresponding run of its skeleton composition, which traverses through the same path (states and transitions) and has the same input queue contents (disregarding message contents) at each peer. Since the skeleton composition satisfies autonomous and synchronous compatible conditions, there exists an equivalent execution of the skeleton composition in which each message is consumed immediately after it is sent. From this execution of the skeleton composition we can construct an execution for the guarded composition in which each message is consumed immediately after it is sent. This leads to the synchronizability of the guarded composition as shown in [13].

A similar skeleton analysis can be developed for guarded conversation protocols. A guarded conversation protocol is realizable if its skeleton satisfies the autonomous, synchronous compatible, lossless join conditions and a fourth condition called "deterministic guards condition". Intuitively, the deterministic guards condition requires that for each peer, according to the guarded conversation protocol, when it is about to send out a message, the guard that is used to compute the contents of the message is uniquely decided by the sequence of message classes (note, not messages) exchanged by the peer in the past. The details of this analysis can be found in [20].

Skeleton analysis sometimes can be inaccurate. Below we will discuss this inaccuracy and techniques that can be use the refine the skeleton analysis.

Example 8. Consider the modified composition of GPS and Map Provider in Figure 8. The composition is actually synchronizable. In GPS implementation, the guard $\mathsf{id} = \mathsf{map.id}$ in transition $\mathsf{!req}$ enforces that the sending of next req message must wait for the last req message being matched by a corresponding map message. Thus the interaction of two services runs in lock-step fashion, where the id attribute of req messages alternates between 0 and 1. However, the skeleton analysis cannot reach the conclusion that the guarded composition is synchronous, because the skeleton of GPS does not satisfy the autonomous condition.

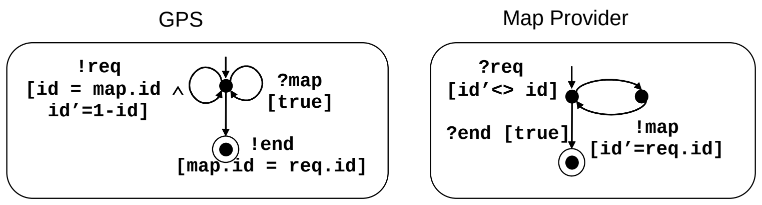

Fig. 8. An example on inaccuracy of skeleton analysis.

The inaccuracy of skeleton analysis can be fixed by a refined symbolic analysis of guarded compositions. The basic idea is to symbolically explore the configuration space of a guarded automaton, and split its states and remove redundant transitions if necessary. The result is another guarded automaton which generates the same set of conversations, but has more states.

Example 9. For example, after applying the iterative symbolic analysis on the GPS service in Figure 8, we obtain the refined guarded automaton in Figure 9. The refined automaton splits the initial state to 4 different states. If we examine the 4 non-final states (starting from the initial state and walking anti-clockwise), these states represent 4 different system configurations where the id attributes of the latest copies of req and map messages are $(0, 0)$, $(1, 0)$, $(1, 1)$, and $(0, 1)$, respectively. The refined automaton is equivalent to the original GPS implementation in Figure 8. If we apply the skeleton analysis on Figure 9, we can now reach the conclusion that the composition is synchronizable.

The algorithm for the iterative symbolic analysis can be found in [20].

6 Web Service Analysis Tool

The synchronizability and realizability analyses are implemented and integrated to the Web Service Analysis Tool (WSAT) [19, 39]. WSAT accepts web service specifications in popular web service description languages (such as WSDL and BPEL4WS), system properties specified in LTL, and verifies if the conversations generated conform to the LTL property.

Fig. 10 shows the architecture of WSAT. WSAT uses Guarded Automata (GA) as an intermediate representation. A GA is a finite state machine which sends and receives XML messages and has a finite number of XML variables. The types of XML messages and variables are defined using XML Schema. In the GA

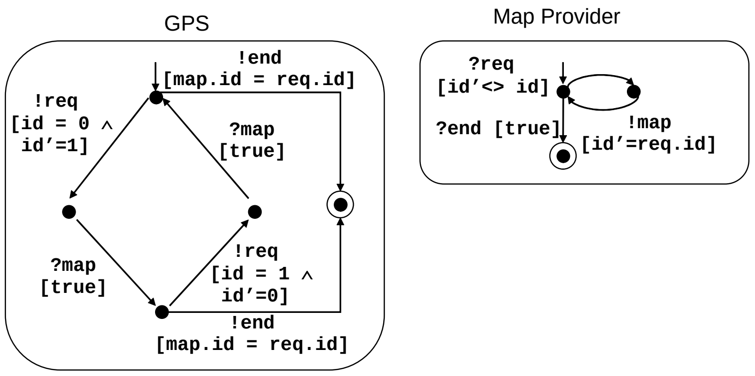

Fig. 9. A refined version of the guarded composition in Figure 8.

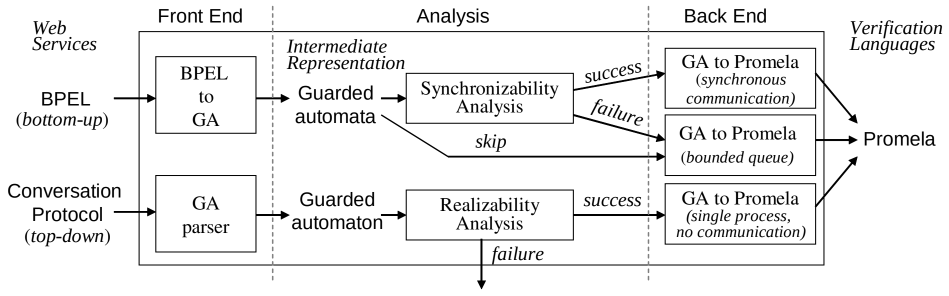
Front End
Analysis
Back End
Web Services
Verification Languages
BPEL (bottom-up)
Conversation Protocol (top-down)
BPEL to GA
GA parser
Intermediate Representation
Guarded automata
Guarded automaton
Synchronizability Analysis
Realizability Analysis
success
failure
skip
success
failure
GA to Promela (synchronous communication)
GA to Promela (bounded queue)
GA to Promela (single process, no communication)
Promela

Fig. 10. WSAT architecture.

representation used by WSAT all the variable and message types are bounded. Each send transition can have a guard, which is essentially an assignment that determines the contents of the message being sent. Each receive transition can also have a guard—if the message being received does not satisfy the guard, the receive action is blocked. The GA representation is capable of capturing both the control flow and data manipulation semantics of web services. WSAT includes a translator from BPEL to GA that supports bottom-up specification of web service compositions. It also includes a translator from top-down conversation protocol specifications to GA. Support for other languages can be added to WSAT by integrating new translators to its front end without changing the analysis and the verification modules.

Synchronizability and realizability analyses are implemented in WSAT. When the analysis succeeds, LTL verification can be performed using the synchronous communication semantics instead of asynchronous communication semantics. When the analysis is not successful on the web service input, asynchronous semantics is used and a partial verification is conducted for bounded communication channels. WSAT also implements extensions to the synchronizability and realizability analyses to handle the guards of the transitions in the GA model [18]. Algorithms for translating XPath expressions to Promela code are presented in [17] where model checker SPIN [24] is used at the back-end of WSAT to check LTL properties.

We applied WSAT to a range of examples, including six conversation protocols converted from the IBM Conversation Support Project [25], five BPEL services from BPEL standard and Collaxa.com, and the SAS example from [17]. We applied the synchronizability or the realizability analysis to each example, depending on whether the specification is bottom-up or top-down. As reported in [21], only 2 of the 12 examples violate the conditions discussed in this paper (both violate the autonomous condition). This demonstrates that the sufficient conditions used in the synchronizability and realizability analyses are not too restrictive and that they are able to show the synchronizability and realizability of practical web service applications.

7 Related Work and Discussion

This section presents a survey of related work on modeling and analyzing web services. Particularly, we are interested in the following four topics: (1) modeling approaches for distributed systems, (2) description of global behaviors in distributed systems, (3) realizability analysis, and (4) automated analysis and verification of web services. At the end of this section we also present a discussion about our approach, identifying its limitations and possible extensions.

7.1 Modeling Approaches and Communication Semantics

Since the web service technology can be regarded as essentially a branch of distributed systems, we include a discussion of earlier models for describing interaction and composition of distributed systems. Traditionally, many modeling approaches use synchronous communication semantics, where sender and receiver of a message transmission have to complete the send and receive actions simultaneously. The typical examples include (but not limited to) CSP [23], I/O automata [29] and interface automata [2].

In the models which use asynchronous communication semantics FIFO queue is the most commonly used message buffer. Communicating Finite State Machines (CFSM) were proposed in early 1980's as a simple model with asynchronous communication semantics [7]. Brand et al. showed that CFSM can simulate Turing Machines [7]. Other related modeling approaches for distributed systems include Codesign Finite State Machine model [10], Kahn Process Networks [27], π-Calculus [30], and Microsoft Behave! Project [37]. Most of them, e.g., π-Calculus and Behave! Project, use or support simulation of asynchronous communication semantics.

7.2 Modeling Global Behaviors

In the conversation model, a global behavior is modeled as a sequence of send events. In many other modeling approaches, e.g., Message Sequence Charts (MSCs) [31], both send and receive events are captured. Such different modeling perspectives can lead to differences in the expressive power and in the difficulty of analysis and verification problems. We now briefly compare the conversation model and the MSC model [4]. This section is a summary of the more detailed discussion given in [21].

MSC model [31] is a widely used specification approach for distributed systems. A comparison with the basic MSC model would not be fair since using the MSC model one can only specify a fixed number of message traces. Instead, we compare our model with the more expressive MSC graphs [4], which are finite state automata that are constructed by composing basic MSCs. MSC graphs use asynchronous communication semantics. There are other MSC extensions such as the high level MSC (hMSC) [38]. However, hMSC is mainly used for studying infinite traces and the composition model used in [38] is synchronous. Therefore the MSC graph is a more suitable model for comparison.

An MSC consists of a finite set of peers, where each peer has a single sequence of send/receive events. We call that sequence the *event order* of that peer. There is a bijective mapping that matches each pair of send and receive events. Given an MSC M, its language $L(M)$ is the set of *linearizations* of all events that follow the event order of each peer. Essentially $L(M)$ captures the "join" of local views from each peer. A formal definition of MSC can be found in [4].

An MSC graph [4] is a finite state automaton where each node of the graph (i.e., each state of the automaton) is associated with an MSC. Given an MSC graph G, a word w is accepted by G, if and only if there exists an accepting path in G where w is a linearization of the MSC that is the result of concatenating the MSCs along that path.

The main difference between the MSC graph framework and the conversation oriented framework is the fact that the MSC model specifies the ordering of the receive events whereas the conversation model does not. In the conversation model the timing of a receive event is considered to be a local decision of the receiving peer and is not taken into account during the analysis of interactions among multiple peers.

Conversation protocols and MSC graphs are incomparable in terms of their expressive power [21]. For example, it is possible to construct two MSC graphs with different languages but identical conversation sets. This implies that there are interactions that can be differentiated using MSC graphs but not using conversation protocols. On the other hand there are interactions which can be specified using a conversation protocol but cannot be specified with any MSC graph. Hence, expressiveness of MSC graphs and conversation protocols are incomparable. It is also possible to show that the expressive power of the MSC graphs and the bottom-up specified web service compositions are incomparable [21].

One natural question is: which approach is better? Both approaches have pros and cons. In the conversation model the ordering of receive events is like a "don't care" condition which can simplify the specification of interactions. On the other hand realizability problem in the conversation model can be more severe since we focus on global ordering of send events. For example, the non-realizable conversation protocol $\{a_{A\to B}\ b_{C\to A}\}$ cannot be specified using MSCs.

The different modeling perspectives on global behaviors leads to different realizability analysis techniques. Alur et al. investigated the weak and safe realizability problems for sets of MSCs and the MSC graphs [3, 4]. They showed that determining realizability of a set of MSCs is decidable, however it is not decidable for MSC graphs. They gave one sufficient and necessary condition for realizability of MSC graphs. The sufficient and necessary condition looks very similar to the *lossless join* condition in the realizability analysis on the conversation model. However there are key differences: 1) In the MSC model, the condition is both sufficient and necessary whereas in the conversation model, lossless join is a sufficient condition only, and, 2) it is undecidable to check the condition for MSC graphs. Alur et al. introduced another condition called *boundedness* condition, which ensures that during the composition of peers the queue length will not exceed a certain preset bound (on the size of the MSC graph). This condition excludes some of the realizable designs. Note that the realizability conditions in the conversation model do not require queue length to be bounded. However, notice that the realizability analysis on conversation model does not subsume the realizability analysis on MSC graphs. There are examples which can pass the realizability analysis on MSC graphs but are excluded by the realizability analysis we presented for the conversation model.

7.3 Realizability and Synchronizability

Interest in the realizability problem dates back to 1980's (see [1, 35, 36]). However, the realizability problem means different things in different contexts. For example, in [1, 35, 36], realizability problem is defined as whether a peer has a strategy to cope with the environment no matter how the environment decides to move. The concept of realizability studied in this chapter is rather different. We are investigating realizability in a closed system that consists of multiple peers interacting with each other. Our definition of realizability requires that the implementation generates exactly the same set of global behaviors as specified by the protocol. A closer notion to the realizability problem in this chapter is the "weak realizability" of MSC graphs studied in [4]. Different communication assumptions can lead to different realizability analysis. For example, realizability problem for high-level MSC (hMSC) is studied in [38].

To the best of our knowledge, synchronizability analysis was first proposed in [15]. The relationship between synchronizability (for bottom-up specifications) and realizability (for top-down specifications) was discussed in [21].

7.4 Verification of Web Services

Application of automated verification techniques to web services has been an active area. Narayanan et al. [34] model web services as Petri Nets and investigate the simulation, verification and composition of web services using the Petri Net model. Foster et al. [12] use LTSA (Labeled Transition System Analyzer) to verify BPEL web services using synchronous communication semantics and MSC model. Nakajima [33] proposes an approach in which a given web service flow specified in WSFL is verified using the model checker SPIN. The approach presented by Kazhamiakin et al. [28] determines the simplest communication mechanism necessary to verify a web service composition, and then verifies the composition using that communication mechanism. Hence, if a web service is not synchronizable it is analyzed using asynchronous communication semantics.

7.5 Discussion

We conclude this section with a discussion of possible limitations of the presented framework and possible extensions.

We believe that an important limitation of the presented analyses techniques is the fact that they do not handle dynamic service creation or establishment of dynamic connections among different services. In the model discussed here we assume that interacting web services do not dynamically create communication channels or instantiate new business processes. Since dynamic service discovery is an important component of service oriented computing, in order to make the approach presented in this chapter applicable to a wider class of systems, it is necessary to handle dynamic instantiation of peers and communication channels. Extending synchronizability and realizability analyses to such specifications is a promising research direction.

So far we have only applied the presented analysis techniques to protocols with a modest number of states. This is due to the fact that most web service composition examples we have found do not have a large number of control states. In the future, it would be interesting to investigate the scalability of the presented techniques for specifications with large number of states. Generally, we believe that the presented techniques will be scalable as long as the specifications are deterministic, and, therefore, the cost of determinization can be avoided.

Currently we do not have an implementation of symbolic synchronizability and realizability analyses for handling specifications in which message contents influence the control flow. At this point, the WSAT tool only performs skeleton analyses to guarded automata specifications. This makes the synchronizability and realizability analyses conditions quite restrictive and using symbolic techniques can relax these conditions. However, it is necessary to find a symbolic representation for XML data in order to implement symbolic analyses, which could be a difficult task. If successful, such a symbolic representation can also be used for symbolic verification of web services as opposed to the explicit state model checking approach we are currently using.

Finally, the synchronizability and realizability conditions presented in this chapter are sufficient conditions and it could be possible to relax them. Finding necessary and sufficient conditions for synchronizability and realizability of conversations is an open problem.

8 Conclusions

Conversations are a useful model for specification of interactions among web services. By analyzing conversations of web services one can investigate properties of the interactions among them. However, asynchronous communication semantics makes verification and analysis of conversations difficult. We discussed two techniques that can be used to overcome the difficulties that arise in verification due to asynchronous communication. Synchronizability analysis identifies web service compositions for which the conversation behavior does not change when different communication mechanisms are used. Using the synchronizability analysis one can verify properties of conversations using the simpler synchronous communication semantics without giving up the benefits of asynchronous communication. Realizability analysis is used to make sure that for top-down web service specifications asynchronous communication does not create unintended behaviors. Realizable conversation protocols enable analysis and verification of conversation properties at a higher level of abstraction without considering the asynchronous communication semantics. As we discussed, it is also possible to extend synchronizability and realizability analyses to specifications in which message contents influence the control flow.

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. L. D. Alfaro and T. A. Henzinger. Interface automata. In *Proc. 9th Annual Symp. on Foundations of Software Engineering*, pages 109–120, 2001.
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. Adam Bosworth. Loosely speaking. *XML & Web Services Magazine*, 3(4), April 2002.
6. Business Process Execution Language for Web Services (Version 1.0). `http://www.ibm.com/developerworks/library/ws-bpel`, 2002.
7. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the 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. M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, and A. San giovanni Vincentelli. A formal specification model for hardware/software codesign. In *Proc. Intl. Workshop on Hardware-Software Codesign*, October 1993.
11. E.M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

12. 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.
13. X. Fu. *Formal Specification and Verification of Asynchronously Communicating Web Services.* PhD thesis, University of California, Santa Barbara, 2004.
14. X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proc. 8th Int. Conf. on Implementation and Application of Automata*, volume 2759 of *LNCS*, pages 188–200, 2003.
15. X. Fu, T. Bultan, and J. Su. Analysis of interacting web services. In *Proc. 13th Int. World Wide Web Conf.*, pages 621 – 630, New York, May 2004.
16. 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.
17. X. Fu, T. Bultan, and J. Su. Model checking XML manipulating software. In *Proc. 2004 ACM/SIGSOFT Int. Symp. on Software Testing and Analysis*, pages 252–262, July 2004.
18. X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. In *Proc. 2004 IEEE Int. Conf. on Web Services*, pages 96–203, July 2004.
19. X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of web service compositions. In *Proc. 16th Int. Conf. on Computer Aided Verification*, volume 3114 of *LNCS*, pages 510–514, July 2004.
20. 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.
21. X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, December 2005.
22. J. E. Hanson, P. Nandi, and S. Kumaran. Conversation support for business process integration. In *Proc. of 6th IEEE Int. Enterprise Distributed Object Computing Conference*, 2002.
23. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
24. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, Boston, Massachusetts, 2003.
25. IBM. Conversation Support Project. `http://www.research.ibm.com/convsupport/`.
26. Java Message Service. `http://java.sun.com/products/jms/`.
27. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP 74*, pages 471–475. North-Holland, 1974.
28. Raman Kazhamiakin, Marco Pistore, and Luca Santuari. Analysis of communication models in web service compositions. In *Proc. of 15th World Wide Web Conference (WWW)*, pages 267–276, 2006.
29. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. Principles of Distributed Computing*, pages 137–151, 1987.
30. R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.
31. Message Sequence Chart (MSC). ITU-T, Geneva Recommendation Z.120, 1994.
32. Microsoft Message Queuing Service. `http://www.microsoft.com/windows2000/technologies/communications/msmq/default.mspx`.

33. Shin Nakajima. Model checking verification for reliable web service. In *Proc. of the 1st International Symposium on Cyber Worlds (CW 2002)*, pages 378–385, November 2002.
34. S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proc. International World Wide Web Conference (WWW)*, 2002.
35. 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.
36. 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.
37. S. K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In *Proc. 8th Static Analysis Symposium*, pages 375–394, July 2001.
38. 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.
39. Web Service Analysis Tool (WSAT). `http://www.cs.ucsb.edu/~su/WSAT`.
40. Web Service Choreography Description Language (WS-CDL). `http://www.w3.org/TR/ws-cdl-10/`, 2005.
41. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.