Synchronizability of Conversations Among Web Services

Xiang Fu, Tevfik Bultan, Jianwen Su

Abstract

We present a framework for analyzing interactions among web services that communicate with asynchronous messages. We model the interactions among the peers participating to a composite web service as conversations, the global sequences of messages exchanged among the peers. This naturally leads to the following model checking problem: given an LTL property and a composite web service, do the conversations generated by the composite web service satisfy the property? We show that asynchronous messaging leads to state space explosion for bounded message queues and undecidability of the model checking problem for unbounded message queues. We propose a technique called synchronizability analysis to tackle this problem. If a composite web service is synchronizable, its conversation set remains the same when asynchronous communication is replaced with synchronous communication. We give a set of sufficient conditions that guarantee synchronizability and that can be checked statically. Based on our synchronizability results, we show that a large class of composite web services with unbounded message queues can be verified completely using a finite state model checker such as Spin. We also show that synchronizability analysis can be used to check realizability of top-down conversation specifications and we contrast the conversation model with the message sequence charts. We integrated synchronizability analysis to a tool we developed for analyzing composite web services.

Index Terms

web services, asynchronous communication, conversations, model checking, verification, synchronizability, realizability

I. Introduction

Browser-based web accessible software systems have been extremely successful in electronic commerce, especially for business-to-consumer applications. However, the difficulty of integrating business processes across heterogeneous platforms has been a major hurdle in extending this success to business-to-business applications. Web services address this problem by providing a framework for integration and interoperability of web accessible software applications regardless of implementation platforms and across boundaries of business entities [3], [13], [27], [33].

Since communicating web services can be deployed on different locations using different implementation platforms, agreeing on a set of standards for data transmission and service descriptions is clearly very important. Fig. 1 displays a collection of most frequently used standards for web services where Extensible Markup Language (XML) [12] forms the foundation. Web services interact with each other by exchanging messages in the XML format. XML Schema [44] provides the type system for XML messages and Simple Object Access Protocol (SOAP) [38] is a standard communication protocol for transmitting XML messages. Each web service has to publish its invocation interface, e.g., network address, ports, functions provided, and the expected XML message format to invoke the service, using the Web Services Description Language (WSDL) [43]. Although a WSDL specification defines the public interface of a web service, it does not provide any information about its behavior. Behavioral descriptions of web services are defined using higher level standards such as Business Process Execution Language for Web Services (BPEL) [7]. BPEL specifications can be used to define behavioral interfaces of web services. The Web Service Choreography Description Language (WS-CDL) [42] is used to specify the interactions among multiple web services. The specifications of web services are registered in a Universal Description, Discovery and Integration (UDDI) [40] registry, which allows registered web services to be discovered by other services. Web service development based on these standards is supported by different implementation platforms such as .Net [29]

This work is supported in part by NSF grants CCR-0341365, IIS-0101134, and IIS-9817432. X. Fu is with the Georgia Southwestern State University. T. Bultan and J. Su are with the University of California, Santa Barbara. Email: xfu@canes.gsw.edu, {bultan,su}@cs.ucsb.edu

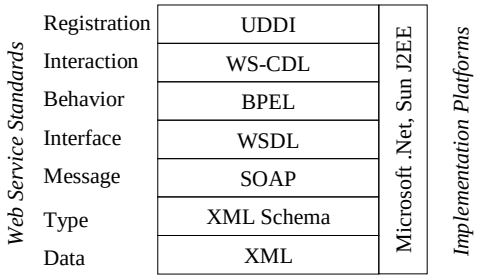

Fig. 1. Web Service Standards

and J2EE [41]. The use of standardized protocols allows automatic discovery, invocation and composition of web services regardless of the implementation platforms and programming languages used during their development.

Web services are essentially loosely coupled distributed systems. The loose coupling is partly achieved by the use of standardized interfaces and data formats mentioned above. Another factor in achieving loose coupling is *asynchronous* communication—during a message exchange, the sender and the receiver do not have to synchronize the send and receive actions. In asynchronous communication a message is stored in the message buffer (usually a FIFO queue) of its receiver until it is consumed and processed. Asynchronous communication is necessary for building robust web services [6]. It prevents the sending service getting stuck during a send operation due to pauses in availability of the receiving service or slow data transmission through the Internet. Asynchronous messaging is supported by message delivery platforms such as Java Message Service (JMS) [25] and Microsoft Message Queuing Service (MSMQ) [32].

The future success of web services framework will partly depend on the feasibility of constructing highly dependable services. For critical business applications, any design error during web service development can cause potentially great financial losses, and ad-hoc repairs after failures are not acceptable. It is desirable to statically ensure the correctness of web services before they are deployed. In this paper we investigate modeling and verification of interactions among web services. Our goal is to automatically verify properties of interactions among web services based on their behavioral descriptions. There are several challenges that need to be addressed to achieve this goal. First, we have to decide on how to characterize the interactions of web service compositions. For example, should both send and receive events be modeled? We need to resolve such questions in order to construct a formal model for interactions of web services. Another challenge is the handling of asynchronous messages. As we mentioned above, asynchronous communication is one of the advantages of the web service technology over traditional tightly coupled systems. However, asynchronous communication makes most verification and analysis problems about web services undecidable.

In this paper we tackle these challenges as follows: (1) We model the interactions among web services as *conversations*. A conversation is a sequence of send events recorded in the order they are sent. Linear Temporal Logic (LTL) can be naturally extended to specify desired properties on conversations. (2) We present a set of *synchronizability conditions*. When these conditions are satisfied by a set of web services, i.e., when a set of web services are synchronizable, they generate the same set of conversations under both asynchronous and synchronous communication semantics. Assuming that the behavioral descriptions of web services are specified as finite state machines, we show that verification of LTL properties of conversations of synchronizable web services can be performed using existing finite state model checkers.

We show that the synchronizability analysis is related to realizability analysis of conversation protocols [16]. A conversation protocol is a top-down specification which specifies the desired conversation set for a set of web services without describing the behaviors of individual services. We show that the synchronizability conditions can be used to show the realizability of conversation protocols when they are combined with one additional condition.

We built a tool called Web Service Analysis Tool (WSAT) which implements the analysis techniques presented in this paper. WSAT supports both bottom-up and top-down specifications. The front-end of WSAT translates web service specifications written in BPEL and WSDL to an internal state machine representation. The synchronizability and realizability analyses are performed on this state machine representation. The back-end of WSAT translates its internal state machine representation to Promela, the input language of the Spin model checker [22], [23]. We use

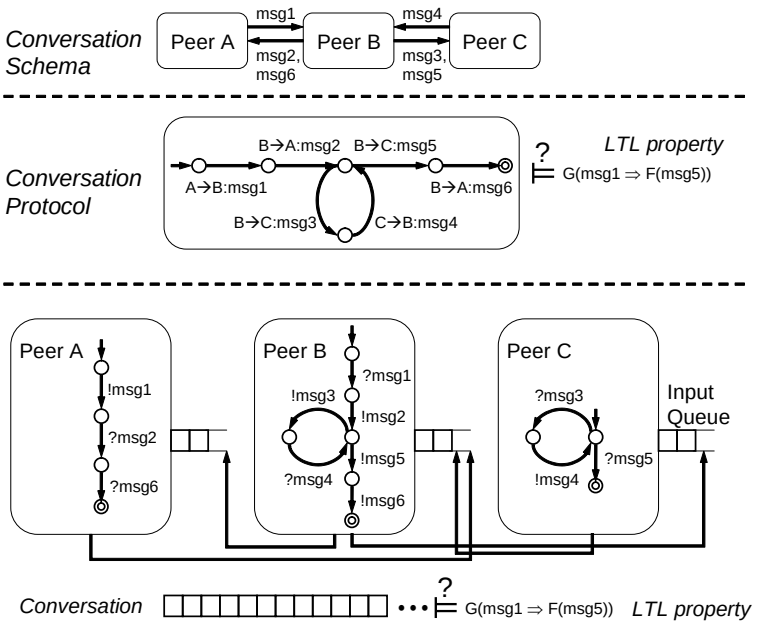

Fig. 2. A Simple Example Demonstrating Our Model

Spin as a back-end model checker to verify LTL properties of conversations.

The rest of the paper is organized as follows. Section II introduces a model for web service compositions and their interactions. Section III presents the synchronizability analysis for bottom-up specifications. Section IV presents the realizability analysis for top-down specifications, and discusses the relationship between the synchronizability and realizability analyses. Section V presents the WSAT tool and some experimental results. Section VI discusses the related work, and presents the comparison with Message Sequence Charts. Section VII concludes the paper.

II. Modeling Interactions of Asynchronously Communicating Web Services

In this section we present a formal model for interacting web services [9], [15], [16]. 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. We consider the problem of how to characterize the interactions among peers participating in a web service composition, as well as how to reason about the correctness of these interactions. 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. We present some theoretical observations on conversation sets, which motivates the synchronizability analysis presented in the next section.

A. Composition Architecture

Fig. 2 shows a simple example demonstrating our model. The top part of Fig. 2 shows a composition schema. A composition schema specifies the set of peers and the set of messages exchanged among the peers. The middle part of Fig. 2 (labeled conversation protocol) shows a top-down specification for the interactions among the peers. We will discuss the top-down specification approach later in section IV. In this section we will focus on the bottom-up specification model shown at the bottom of Fig. 2. In the bottom-up model we are interested in analyzing the interactions of a set of peer implementations. 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. A conversation is the sequence of messages exchanged among the peers during an execution, recorded in the order they are sent. A conversation can be regarded as a linearization of the message events, similar to the approach used in defining the semantics of Message Sequence Charts [31] in [5]. We now formalize the above descriptions with the definitions below.

Definition 2.1: 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 (the input alphabet), M_i^{out} is the set of outgoing messages (the output alphabet), and $M_i = M_i^{in} \cup M_i^{out}$ is the alphabet of p_i. If $i \neq j$ then $M_i^{in} \cap M_j^{in} = M_i^{out} \cap M_j^{out} = \emptyset$. M satisfies the following:

$$\bigcup_{i \in [1..n]} M_i^{in} = \bigcup_{i \in [1..n]} M_i^{out} = M$$

The above definition implies that each message has a unique sender and a unique receiver, and a peer cannot send a message back to itself. Now, using the composition schema, we can define web service compositions as follows.

Definition 2.2: 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 each $\mathcal{A}_i$ is the *peer implementation* (or simply *peer*) for the corresponding peer prototype $p_i = (M_i^{in}, M_i^{out}) \in P$. The alphabet of $\mathcal{A}_i$ is $M_i = M_i^{in} \cup M_i^{out}$.

The above definition is general in the sense that it does not restrict the expressive power of peer implementations. However, in this paper, we focus on the web service compositions whose peers are specified using Finite State Automata (FSA).

B. Peers

The bottom part of Fig. 2 presents the peer implementations for the peer prototypes shown at the top. Note that, in order to model asynchronous communication, each peer is equipped with a FIFO queue to store incoming messages. Formally, a peer implementation is defined as follows.

Definition 2.3: 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* $\mathcal{A}_i$ which *implements* p_i is a nondeterministic FSA $\mathcal{A}_i = (M_i, T_i, s_i, F_i, \delta_i)$ where $M_i = M_i^{in} \cup M_i^{out}$, T_i is the finite set of states, $s_i \in T$ is the initial state, $F_i \subseteq T$ is the set of final states, and $\delta_i \subseteq T_i \times (M_i \cup \{\epsilon\}) \times T_i$ 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 $(t_1, !m_1, t_2)$ which sends out a message $m_1 \in M_i^{out}$, (2) a receive-transition of the form $(t_1, ?m_2, t_2)$ which consumes a message $m_2 \in M_i^{in}$ from its input queue, and (3) an ϵ-transition of the form (t_1, ϵ, t_2).

C. Conversations

Below we will formalize the notion of conversations to model the interactions among peers in a web service composition [9], [16]. Let $\mathcal{W} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be a web service composition. A *global configuration* (or simply configuration) is a $(2n)$-tuple of the form $(Q_1, t_1, ..., Q_n, t_n)$ where for each $j \in [1..n]$, $Q_j \in (M_j^{in})^*$, $t_j \in T_j$. Here t_i, Q_i denote the local state and the queue contents of peer $\mathcal{A}_i$ respectively.

Let the set of states, transition relation, etc. of a peer $\mathcal{A}_i$ be all labeled with subscript i, e.g., δ_i is the transition relation of peer $\mathcal{A}_i$. For two configurations $c = (Q_1, t_1, ..., Q_n, t_n)$ and $c' = (Q'_1, t'_1, ..., Q'_n, t'_n)$, we say that c *derives* c', written as $c \rightarrow c'$, if one of the following three conditions hold:

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

3) $t'_k = t_k$ for each $k \neq i$.

Consider the definition of the receive action. Intuitively, the above definition says that the peer $\mathcal{A}_i$ executes a receive action if there is a message at the head of its queue and a corresponding receive transition in its transition relation from its current state. After the receive is executed, the received message is removed from the queue of $\mathcal{A}_i$, the queues and local states of other peers remain the same, and the local state of $\mathcal{A}_i$ is updated accordingly. The ϵ-action (where a peer takes an ϵ-transition) and the send action (where a peer takes a send transition) are defined similarly. Now we can define the runs of a web service composition as follows:

Definition 2.4: 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 *partial run* of $\mathcal{W}$ if it satisfies the first two of the following three conditions, and γ is a *(complete) run* if it satisfies all the three conditions:

1) The configuration $c_0 = (\epsilon, s_1, \ldots, \epsilon, s_n)$ is the initial configuration where s_i is the initial state of $\mathcal{A}_i$ for each $i \in [1..n]$, 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 $i \in [1..n]$.

Now we define the send sequences and the conversations as follows:

Definition 2.5: Given a partial or complete run γ the *send sequence* generated by γ, denoted by $ss(\gamma)$, is defined recursively as follows:

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

A word w over M ($w \in M^*$) is a *conversation* of web service composition $\mathcal{W}$ if there exists a complete run γ such that $w = ss(\gamma)$, i.e., a conversation is the send sequence generated by a complete run. The *conversation set* of the 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 the web service composition in Fig. 2 can be captured by the regular expression: msg1 msg2 (msg3 msg4)* msg5 msg6

Given a web service composition, one interesting problem is to check if its conversations satisfy an LTL property. The semantics of LTL formulas can be naturally adapted to conversations by defining the set of atomic propositions as the power set of messages, i.e., 2^M. Standard LTL semantics is defined on infinite sequences [11] whereas we focus on finite conversations. We can adopt the standard LTL semantics to conversations by extending each conversation to an infinite string by adding an infinite suffix which is the repetition of a special termination symbol. For example, the composition in Fig. 2 satisfies the LTL property: $\mathbf{G}(\mathsf{msg1} \Rightarrow \mathbf{F}(\mathsf{msg5}))$, where **G** and **F** are temporal operators which mean "globally" and "eventually", respectively.

Notice that, due to the queue effects, not all web service compositions produce regular conversation sets like the example in Fig 2. In fact, conversations of a web service composition could be very difficult to analyze as demonstrated by the following negative result about the LTL verification of conversations of web service compositions [16]:

Theorem 2.1: Given a web service composition $\mathcal{W}$ and an LTL property ϕ, determining if all the conversations of $\mathcal{W}$ satisfy the LTL property ϕ is undecidable.

The web service composition model we described above is essentially a system of Communicating Finite State Machines (CFSM) and it is known that CFSMs can simulate Turing Machines [8]. 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.

III. Bottom-up Approach and Synchronizability

The undecidability of LTL verification for web service compositions is caused by the asynchronous communication. Notice that if synchronous communication is used instead, the set of configurations of a web service

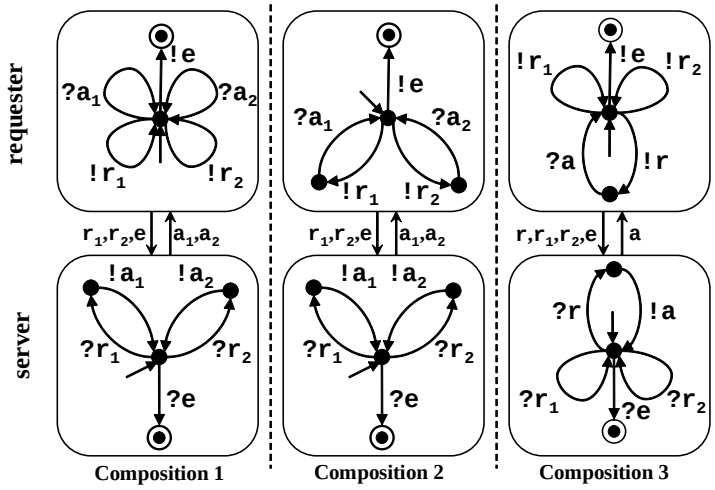

Fig. 3. Three Motivating Examples

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 present a technique called *synchronizability analysis* which identifies web service compositions that 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 will hold for the asynchronous communication semantics. We start with three motivating examples, and then we introduce the technical results for the synchronizability analysis.

A. Motivating Examples

Consider the three web service compositions shown in Fig. 3. Each composition consists of two peers: a requester and a server. For each "request" message (r_i) the server will respond with a corresponding "acknowledgment" message (a_i). However the consumption of the acknowledgments at the requester may not be immediate (e.g. in Composition 1). Finally the "end" message (e) concludes the interaction between the requester and the server.

Assume that we want to use the finite state model checker Spin to verify the properties of the examples shown in Fig. 3. In order to translate these examples to Promela (the input language of the Spin model checker) we need to bound the sizes of the input queues (communication channels in Promela), since Spin is a finite state model checker. In fact, based on the undecidability of LTL verification we discussed above, it is impossible to verify the behaviors of web service compositions with unbounded queues automatically. In general, best we can do is *partial* verification, i.e., to verify the behaviors of a web service composition using queues with a fixed length. Note that the absence of errors using such an approach does not guarantee that the web service composition is correct. Interestingly, in this section we will show that, Compositions 2 and 3 shown in Fig. 3 are different than Composition 1, in that the properties of Compositions 2 and 3 can in fact be verified for unbounded message queues, whereas for Composition 1 we can only achieve partial verification.

First, note that in Composition 1 the requester can send an arbitrary number of messages before the server starts consuming them. Due to this queuing effect, the conversation set of Composition 1 is not a regular set [9]. Actually it is a subset of $(r_1|r_2|a_1|a_2)^*e$ where the number of r_i and a_i messages are equal and in any prefix the number of r_i messages is greater than or equal to the number of a_i messages. Hence, it is not surprising that we cannot map the behavior of Composition 1 to a finite state process. Another problem with Composition 1 is the fact that its state space (i.e., reachable configurations) increases exponentially with the sizes of the input queues. Hence, even partial verification for large queue sizes becomes intractable.

In Composition 2 the requester and server processes move in a lock-step fashion, and it is easy to see that the conversations generated by Composition 2 is $(r_1a_1 \mid r_2a_2)^*e$, i.e., a regular set. In fact, the web service composition described in Composition 2 has a finite set of reachable states. During any execution of Composition 2, at any state, there is at most one message in each queue. Based on the results we will present in this section, we can

statically conclude that properties of Composition 2 can be verified using synchronous communication (in other words, using input queues of size 0).

Unlike Composition 2, Composition 3 has an infinite state space as Composition 1. In other words, the number of messages in the input queues for Composition 3 is not bounded. Similar to Composition 1, the state space of Composition 3 also increases exponentially with the sizes of the queues. However, unlike Composition 1, the conversation set of Composition 3 is regular. Although Composition 3 has an infinite state space, we will show that the properties of Composition 3 can also be verified for arbitrary queue sizes using finite state verification techniques.

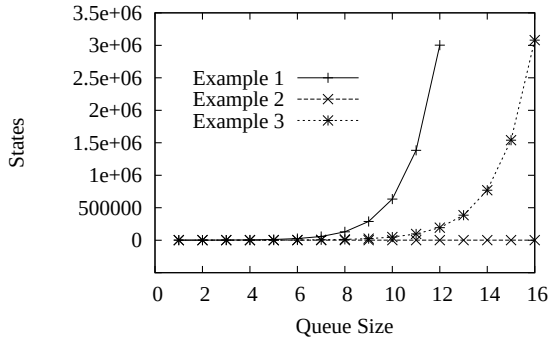

Fig. 4. Increase in the State Space with respect to the Queue Size

We can experimentally demonstrate how state spaces of the examples in Fig. 3 change with the increasing queue sizes. In Fig. 4 we show the sizes of the reachable state spaces of the examples in Fig. 3 computed using the Spin model checker for different input queue sizes. The *x*-axis of the figure shows the size of the input queues, and *y*-axis shows the number of reachable states computed by Spin. As shown in the figure, the state space of Composition 2 is fixed (always 43 states), however the state spaces of Compositions 1 and 3 increase exponentially with the queue size. Below we will show that we can verify properties of Compositions 2 and 3 for arbitrary queue sizes, although best we can do for Composition 1 is partial verification. In particular, we will show that the communication among peers for Compositions 2 and 3 are "synchronizable" and we can verify their properties using synchronous communication and guarantee that the verified properties hold for asynchronous communication with unbounded queues.

B. Synchronous Communication

To further explore the differences of Compositions 2 and 3 from Composition 1, we define an alternative "synchronous" semantics for web service compositions different than the one given in Section II. Intuitively, the synchronous semantics dictates that the sending and receiving peers take the send and receive actions concurrently. Therefore, there is no need to have the input message queues.

Recall that a web service composition $\mathcal{W}$ is a tuple $\mathcal{W} = \langle (P, M), \mathcal{A}_1, ..., \mathcal{A}_n \rangle$ 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 $\mathcal{A}_j$.

For two syn-configurations $c = (t_1, ..., t_n)$ and $c' = (t'_1, ..., t'_n)$, we say that c *derives* c', written as $c \rightarrow_{syn} c'$, if one of the following two conditions hold:

- Two peers execute a *send* action (denoted as $c \xrightarrow{m}_{syn} c'$), i.e., there exist $1 \leq i, j \leq n$ and $m \in M_i^{out} \cap M_j^{in}$ such that:
  1) $(t_i, !m, t'_i) \in \delta_i$,
  2) $(t_j, ?m, t'_j) \in \delta_j$,
  3) $t'_k = t_k$ for each $k \neq i$ and $k \neq j$.
- One peer executes an ϵ-action (denoted as $c \xrightarrow{\epsilon}_{syn} c'$), i.e., there exists $1 \leq i \leq n$ such that:
  1) $(t_i, \epsilon, t'_i) \in \delta_i$,

2) $t'_k = t_k$ for each $k \neq i$.

The definition of the *derivation* relation between two syn-configurations is different than the asynchronous case so that 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 II is called the *asynchronous semantics*.

The definitions of a run, a partial run, a send sequence and a conversation for synchronous semantics is similar to those of the asynchronous semantics given in Section II (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 3.1: 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. Below we will give sufficient conditions for synchronizability. Based on these conditions, we can show that Compositions 2 and 3 in Fig. 3 are indeed synchronizable. However, before giving the sufficient conditions for synchronizability, we would like to show the following property:

Theorem 3.1: Given a web service composition $\mathcal{W}$ its conversation set with respect to synchronous semantics is a subset of its conversation set with respect to asynchronous semantics, i.e., $\mathcal{C}_{\text{syn}}(\mathcal{W}) \subseteq \mathcal{C}(\mathcal{W})$.

Proof: We will show that for each syn-send-sequence generated by a web service composition with synchronous semantics, there exists a run with asynchronous semantics which generates the same send-sequence. The idea is to simulate the execution of a web service composition with synchronous semantics using asynchronous semantics. Given a syn-send-sequence, consider a syn-run that generates that sequence. Now, we can construct a run for the asynchronous semantics where each send action in the syn-run is replaced with the same send action immediately followed by the corresponding receive action for that message and ϵ-actions are left as they are. It is easy to show that such a run exists based on the derivation relations of the synchronous and asynchronous semantics. In this constructed run each message is received immediately after it is sent, i.e., at any given time during the execution there is at most one message queue that is not empty and there is at most one message in it. We call such a run, *a run with immediate receives*. Note that, the send-sequence generated by the constructed run is same as the syn-send-sequence. Hence, every conversation generated by the synchronous semantics is also generated by the asynchronous semantics. ∎

It is easy to show that the converse of the above property does not hold. For example, consider a web service composition of two peers A and B, which can exchange two messages m_1 (from A to B) and m_2 (from B to A). The implementation of A accepts one word $!m_1?m_2$ and the implementation of B accepts one word $!m_2?m_1$. Obviously, if asynchronous semantics is used there exists a run which generates the conversation m_1m_2. However, note that, when synchronous semantics is used there is no run which generates the same conversation. In fact, there is no run with immediate receives for the asynchronous semantics which generates the conversation m_1m_2. Note that in any run which generates the conversation m_1m_2, when B sends out m_2, the message m_1 is in B's input queue and is not received immediately. In the following we will give conditions which, when they hold, guarantee that for any conversation generated with the asynchronous semantics there exists a run with immediate receives which generates the same conversation.

C. Synchronizability Analysis

We propose two sufficient conditions for synchronizability which can be used in identifying synchronizable web service compositions.

Synchronous compatible condition: This condition requires that in each syn-configuration of the web service composition that is reachable from the initial state based on the synchronous semantics, if there is a peer which has a send transition for a message m from its local state in that configuration, then the receiver for that message should have a receive transition for that message either from its local state in that configuration or from a configuration reachable from that configuration via ϵ-actions. Below we give an algorithm for checking the synchronous compatible condition:

Procedure isSynchronousCompatible($\mathcal{W} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$)

1. Let $CS = \{c\}$ where c is the initial syn-configuration of $\mathcal{W}$ which is a tuple $c = (t_1, \ldots, t_n)$ where for each $i \in [1, n]$ t_i is initial state of $\mathcal{A}_i$.
2. Until there are no more configurations to add to CS, do the following:
3. If there are syn-configurations c, c' s.t. $c \in CS$, $c' \notin CS$ and $c \rightarrow_{syn} c'$,
4. then add c' to CS.
5. For every syn-configuration $c \in CS$ where $c = (t_1, \ldots, t_n)$, do the following:
6. Let ϵ-closure(c) be the set of configurations that are reachable from c via ϵ-actions.
7. If there are two peers p_i and p_j and a message $m \in M_i^{out} \cap M_j^{in}$ s.t. there is a send transition of p_i which originates from t_i and sends m, however, for each c' in ϵ-closure(c) there is no c'' in CS s.t. $c' \xrightarrow{m}_{syn} c''$.
8. c is illegal, return `false`.
9. Return `true` if no illegal state is found in CS.

The basic idea of the above algorithm is to construct the product (i.e., the synchronous composition) of all peers. Each state (i.e., syn-configuration) of the product 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. When all states of the product have been examined, the algorithm returns true. The 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.

Using the above algorithm we can show that Compositions 2 and 3 in Fig. 3 satisfy the synchronous compatible condition, however, Composition 1 does not. Starting from the initial syn-configuration of Composition 1, we can go to another syn-configuration after the requester sends and the server receives r_1 (concurrently). Now in this new configuration the requester has a transition to send out another r_1, however, the server is in a local state where it can send a_1 but does not have a transition to receive r_1. Hence Composition 1 in Fig. 3 does not satisfy the synchronous compatible condition.

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. Notice that the autonomous condition allows a peer the choice of sending one of many messages. However, autonomous condition does not permit a choice between send and receive actions. The following algorithm checks the autonomous condition:

Procedure isAutonomous($\langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$)

1. For each A_i do the following:
2. Determinize A_i and remove all the ϵ-transitions.
3. For each state t in A_i do the following:
4. Return `false` if there are both send and receive transitions originating from t.
5. Return `false` if there is a transition originating from t and t is a final state.
6. Return `true` if each state of each peer passes the above check.

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. 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 due to the determinization.

Using the above algorithm we can show that Composition 2 and 3 in Fig. 3 are autonomous. However, Composition 1 is not autonomous because in the initial state requester can either send r_1 or r_2 or receive a_1 or a_2.

We now present the key result concerning the synchronizability analysis. Note that the property and the discussion below is with respect to asynchronous semantics.

Theorem 3.2: 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 with immediate receives which generates the same conversation.

Proof: We will prove this property by induction on the number of sends in a conversation. We will actually prove the following property which implies the property above: Let $w = m_0 m_1 \ldots m_{|w|-1}$ be a conversation generated by a synchronous compatible and autonomous web service composition, then, for all n such that $0 \leq n \leq |w|$, there exists a run which generates the same conversation in which the first n sends are *immediately received* (i.e., between the send and receive actions of the first n messages there are no other actions).

Base Case: For $n = 0$ the property holds vacuously.

Inductive Step: Now we will assume that for n, $0 \leq n < |w|$ there exists a run γ which generates the conversation w in which the first n sends are immediately received. Assume that the $n+1$st message in the send sequence is $m_{n+1} \in M_i^{out} \cap M_j^{in}$. Let c be the configuration in γ right before m_{n+1} is sent, we now prove that *during* γ, *the first non-*ϵ *action that* p_j *takes after* c *is to receive* m_{n+1}.

Since, according to the inductive assumption, the first n messages in γ are *immediately received*, we can generate a partial syn-run from γ which reaches the configuration c by collapsing the consecutive send and receive actions for the first n messages. Let t_j be the state of p_j in configuration c. Since p_i is ready to send m_{n+1} at c, according to the synchronous compatible condition, peer p_j can execute a receive transition for the message m_{n+1} from t_j, or from a state reachable from t_j via ϵ-transitions. Since $\mathcal{W}$ is autonomous, from the above statement, we also know that the first non-ϵ action that p_j can perform after configuration c has to be a receive action. Combined with the fact that each queue is FIFO, we can infer that the first action that p_j can execute after configuration c in γ is to receive m_{n+1}.

Based on the facts established above, we can alter γ by 1) shifting the ϵ-actions by peer p_j that are between the send action for message m_{n+1} and the receive action for message m_{n+1} right before the send action for m_{n+1} and, 2) moving the receive action for message m_{n+1} right after the send action for m_{n+1}, ahead of all the actions by peers other than p_j that are between the send action for m_{n+1} and the receive action for m_{n+1}. This altered run is a run of $\mathcal{W}$ which generates the same send sequence and the first $n+1$ sends in the altered run are immediately received. ∎

Note that, for each send sequence generated with a run with immediate receives based on asynchronous semantics, there exists a syn-run which generates the same send-sequence with synchronous semantics. Such a syn-run can be obtained from the run with immediate receives by merging the consecutive send and receive actions. Then based on Theorem 3.2 we get the following result:

Theorem 3.3: 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.

Note that if synchronous semantics is used the set of configurations generated by a collection of finite state peers is finite. Hence, LTL verification with respect to synchronous semantics is a finite state model checking problem and therefore it is decidable. For example, both Compositions 2 and 3 in Fig. 3 are synchronizable whereas Composition 1 is not (it violates the autonomous condition). Hence, we can verify the properties of Compositions 2 and 3 using synchronous communication semantics (which can be achieved in Spin by restricting the communication channel lengths to 0) and the results we obtain will hold for behaviors generated using asynchronous communication with unbounded queues.

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 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 [20] a synchronizable web service composition has a run (with asynchronous semantics) that leads to a deadlock if an only if it has a syn-run (with synchronous semantics) that leads to a deadlock.

IV. Top-down Approach and Realizability

In this section we discuss the relationship between the top-down and bottom-up specification approaches of web service compositions. In our earlier work [9], we investigated using a top-down approach in which the desired interaction patterns among the peers are specified directly as finite state *conversation protocols* and the specification of the local behaviors of the individual peers are left blank. The middle part of Fig. 2 shows the conversation protocol for the simple web service composition we discussed in Section II. This approach enables the verification of the interaction properties on finite state conversation protocols using standard model checking techniques. However this top-down approach introduces a *realizability* problem since a conversation protocol may not be realizable by a set of asynchronously communicating finite state peers [16]. Below we will formalize these concepts.

Definition 4.1: 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 on alphabet M. We let $L(\mathcal{R}) = L(\mathcal{A})$, i.e., the language recognized

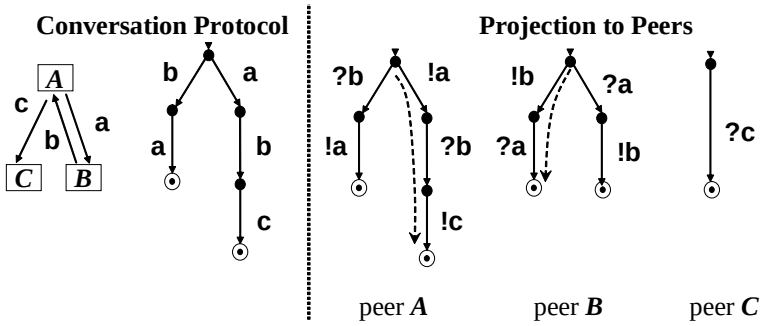

Fig. 5. Ambiguous Execution of a Conversation Protocol which Generates an Unspecified Conversation

by $\mathcal{A}$.

The top-down specification approach based on conversation protocols causes the *realizability* problem which is defined as follows.

Definition 4.2: 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 $\mathcal{R}$.

First we look into the following question: given a conversation protocol $\mathcal{R}$, is it always possible to construct a web service composition that realizes $\mathcal{R}$, i.e., are all conversation protocols realizable? In the following we show that the answer is negative.

Example 4.1: Let (P, M) be a composition schema which consists of four peers p_1, p_2, p_3 and p_4. Its alphabet M consists of two messages a, which is from p_1 to p_2, and b, which is from p_3 to p_4. Suppose $\mathcal{R}$ is a conversation protocol over (P, M) where $L(\mathcal{R}) = \{ab\}$. It is clear that any peer implementation which generates conversation ab can also generate ba as well, because there is no way to let p_3 and p_1 synchronize their send operations. Hence the conversation protocol $\mathcal{R}$ is not realizable. ∎

A more involved example (adapted from the Büchi protocol example in [16]) is shown in Fig. 5. On the left side of the figure is the conversation protocol, and on the right side we show its projection to each peer. Think about one possible execution of the web service composition shown on the right side of Fig. 5 with dashed arrows. At the beginning, peer B sends a message b to peer A, and b is stored in the input queue of peer A. Then peer A sends message a to peer B, and ba is the current partial conversation. Now peer B continues to execute the left path of the protocol, consumes the a in the queue; and peer A executes the right path of the protocol, consumes the b, and sends out c. Eventually, a non-specified conversation bac is generated, without any of the peers noticing it. This abnormal conversation is the result of "ambiguous" understanding of the protocol by peers, and the racing between A and B at the initial state is the main cause.

Below we will show that realizability of conversation protocols can be shown using the synchronizability analysis and an extra condition. First we need to introduce notions of projections and join which relate conversations with local views of the peers. 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:

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

Note that, $L = \text{JOIN}(L_1, \ldots, L_n) \;\Rightarrow\; \forall i \in [1..n] : \pi_i(L) \subseteq L_i$.

Now we can define the lossless join condition for conversation protocols as follows:

Lossless join condition: A conversation protocol $\mathcal{R}$ is *lossless join* if

$$L(\mathcal{R}) = \text{JOIN}(\pi_1(L(\mathcal{R})), \ldots, \pi_n(L(\mathcal{R}))),$$

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. The lossless join property can be checked as follows:

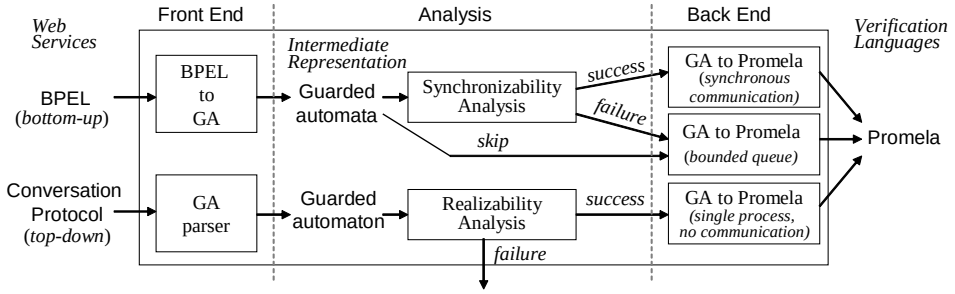

Fig. 6. WSAT architecture

Procedure isLosslessJoin($\mathcal{R} = \langle (P, M), \mathcal{A} \rangle$)

1. For each peer p_i:
2. Let $\mathcal{A}_i$ be the projection of $\mathcal{A}$ to peer p_i constructed by replacing each transition that is labeled with a message that is not in M_i with an ϵ-transition.
3. Let $\mathcal{A}'$ be the product automaton constructed by taking the product of the projections $\mathcal{A}_1, \ldots, \mathcal{A}_n$.
4. If the product automaton $\mathcal{A}'$ is equivalent to $\mathcal{A}$ then return `true` otherwise return `false`.

Intuitively, the lossless join property requires that the protocol should be realizable under synchronous communication semantics. The above algorithm simply projects the conversation protocol to each peer, 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 due to the equivalence check on two nondeterministic finite state machines.

Now we have the following result which connects the synchronizability analysis and the realizability analysis:

Theorem 4.1: Given a conversation protocol $\mathcal{R} = \langle (P, M), \mathcal{A} \rangle$ where $n = |P|$, let web service composition $\mathcal{W} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ include the determinized projections of $\mathcal{A}$ to each peer, s.t. for each $i \in [1..n]$, $L(\mathcal{A}_i) = \pi_i(L(\mathcal{R}))$, and $\mathcal{A}_i$ is a determinized FSA. 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 3.3 and the fact that the synchronous composition of a set of peers accepts the join of their languages. Theorem 4.1 shows the interesting relationship between the synchronizability analysis discussed in this paper and the realizability analysis introduced in [16]. In [16] three realizability conditions were given to guarantee the realizability of a conversation protocol. Here Theorem 4.1 shows that the realizability of a conversation protocol comes from the synchronizability of its projections to each peer. Therefore Theorem 4.1 is a stronger result than the realizability results in [16].

In the last three sections, we presented two formal models which characterize the top-down and bottom-up specification approaches for web service compositions. These two specification approaches reflect the characteristics of different web service languages and standards. For example, WSDL and BPEL essentially take the bottom-up approach. These two standards are used to describe the invocation signatures and control flows of individual web services where each individual web service can be composed of a set of atomic web services. On the other hand, WS-CDL is based on a top-down specification approach and is used to specify the public interaction contracts that should be followed by all participants involved in a web service composition. Our research presents a formalization of these two different approaches and reveal several semantics and analysis issues that should be known by the web service designers. In the bottom-up approach the use of asynchronous communication can cause unexpected behaviors and leads to undecidability of verification. In the top-down approach some global behavior specifications may not be realizable by a set of distributed peers. Our results on synchronizability and realizability can be used to address these problems. For bottom-up specifications, if the peers satisfy the two synchronizability conditions, the composition is guaranteed to be a finite state system and hence can be analyzed automatically. For a top-down specification, if the three realizability conditions are satisfied, the specification is guaranteed to be realized by its projections to each peer.

V. Web Service Analysis Tool

In this section we briefly discuss the Web Service Analysis Tool (WSAT) [19] which implements the synchronizability and realizability analyses discussed above. We also present experimental results about the synchronizability and realizability analyses of 12 web service composition examples.

Fig. 6 shows the architecture of WSAT. WSAT uses an intermediate representation called Guarded Automata (GA) for web services. 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 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.

We implemented the synchronizability and realizability analyses in WSAT. When the analysis succeeds, LTL verification can be performed using the synchronous communication semantics instead of asynchronous communication semantics. WSAT also implements extensions to the synchronizability and realizability analyses to handle the guards of the transitions in the GA model [18]. We developed and implemented algorithms for translating XPath expressions to Promela code [17], and we use the model checker Spin [22], [23] at the back-end of WSAT to check LTL properties.

A. Experimental Results

We applied WSAT to a range of examples, including six conversation protocols converted from the IBM Conversation Support Project [24], five BPEL services from BPEL standard and Collaxa.com, and the SAS example from [17]. We applied either the synchronizability analysis or the realizability analysis to each example, depending on whether the specification is bottom-up or top-down. Our intent was to examine the applicability of the sufficient conditions for synchronizability and realizability analyses to realistic examples. We believe that the set of web service specifications we collected represent a diverse set of realistic applications. The information and the synchronizability/realizability analyses results for these examples are shown in Fig. 7. Other issues relating to LTL verification of web service compositions (e.g., translation of input specifications to intermediate guarded automata representations, handling of XML data and XPath semantics, using SPIN as a back-end model checker) are out of the scope of this article and are discussed in [15], [17], [19].

Problem Set		Size			Pass	S-Analysis/ R-Analysis
Source	Name	#msg	#states	#trans		
ISSTA'04	SAS	9	12	15	yes	R
IBM Conv. Support Project	CvSetup	4	4	4	yes	R
	MetaConv	4	4	6	no	R
	Chat	2	4	5	yes	R
	Buy	5	5	6	yes	R
	Haggle	8	5	8	no	R
	AMAB	8	10	15	yes	R
BPEL spec	shipping	2	3	3	yes	S
	Loan	6	6	6	yes	S
	Auction	9	9	10	yes	S
Collaxa. com	StarLoan	6	7	7	yes	S
	Auction	5	7	6	yes	S

Fig. 7. WSAT experimental results

The size columns in Fig. 7 indicate the sizes of the examples. The #states is either the size of the product automata (i.e., synchronous composition) of all peers, or the size of the conversation protocol. The last column indicates which analysis is applied, e.g., ”R” and ”S” denote the realizability and synchronizability analysis, respectively.

We did not list the analysis cost in Fig. 7 because the cost is trivial for these examples. Generally, we believe that the synchronizability analyses will be scalable as long as all peers are deterministic, and the cost of determinization can be avoided. As reported in the table, only 2 of the 12 examples violate the conditions for realizability (both violate the autonomous condition). For example, the “Meta Conversation” example in [24] allows two peers to race at the beginning and decide the initiator of the conversation. Unfortunately, our autonomous condition forbids such racing. However, the fact that 10 out of 12 examples satisfy the presented sufficient conditions implies that they are not restrictive and they are able to capture a significant portion of practical web service applications.

VI. Related Work

In this section we compare the framework presented in this paper with other modeling approaches used for web services. We particularly focus on comparison between the state machine based specification approaches and Message Sequence Charts (MSC). First, we briefly discuss other state-based models related to our approach, then we concentrate on the comparison of our model with the MSC Graph model and discuss both the expressiveness and the synchronizability and realizability analyses for both models.

A. State-Based Models

State based models such as CSP [21], I/O automata [28] and interface automata [2], have been extensively used for modeling concurrent and distributed systems. However, in most traditional specification approaches synchronous communication semantics is used, i.e., the communicating processes execute a send and a corresponding receive action synchronously. In the communication model used in this paper, messages are stored in a FIFO buffer before they are consumed by the receiver. This is essentially the approach taken by the Communicating Finite State Machines (CFSM) model presented in [8]. However, in [8] each pair of communicating machines use isolated communication channels and each channel has its own queue. The communication model we are using is similar to a variation of CFSM called Single-Link Communicating Finite State Machines (SLCFSM) [34]. Notice that, the simplification from one queue per channel to one queue per peer does not weaken the expressive power of the CFSM model. Also, the synchronizability (and realizability) analysis presented in this paper can be extended to the model with one queue per channel. Other related models include Codesign Finite State Machine [10], Kahn Process Networks [26], π-Calculus [30], and Microsoft Behave! Project [37]. Recently, Foster *et al.* uses LTSA (Labeled Transition System Analyzer) to verify BPEL web services [14], and their assumption about communication is synchronous. The most significant difference between our work and the earlier work on CFSM and other related models is our focus on conversations as a model of interactions among multiple peers.

B. Comparison with Message Sequence Charts

We now discuss the expressive power of three specification approaches for web services: the top-down conversation protocol, the bottom-up web service composition, and the MSC graph model. The motivation for this comparison is that MSC [31] is a widely used scenario specification approach for concurrent and distributed systems. MSC model has also been used in modeling and verification of web services [14]. Comparison with the MSC model leads to several interesting observations which provide insight for web service designers in practice. Notice that comparison with basic MSC will not be fair, because basic MSC can only specify a fixed number of message traces. Instead we compare our model with MSC graph [5], which is a finite state automaton that allows composing basic MSCs. There are similar MSC extensions, e.g., the high level MSC (hMSC) [39], which follows the same weak sequential composition semantics as MSC Graph. However, hMSC in [39] is mainly used for studying infinite traces and the composition model used in [39] is synchronous. Therefore the MSC Graph is a more suitable comparison object here.

MSC Graphs:

An MSC consists of a finite set of peers, where each peer has a single sequence of send/receive events (e.g. the sequence for peer B in Fig. 8(a) is $?a\ !b$). We call that sequence the *event order* of that peer. There is a bijective mapping (represented using arrows in Fig. 8) 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 [5].

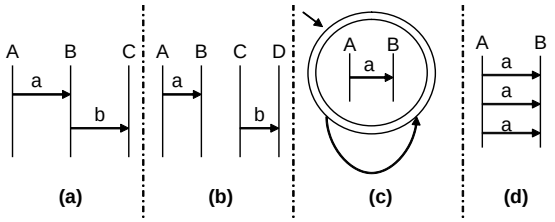

Fig. 8. MSC Examples

Example 6.1: Fig. 8 (a) and (b) are two MSCs, and let them be M_a and M_b respectively. Obviously $L(M_a)$ = $\{!a\ ?a\ !b\ ?b\}$, and $L(M_b)$ = $\{\ !a\ ?a\ !b\ ?b,\ \ !a\ !b\ ?a\ ?b,\ \ !a\ !b\ ?b\ ?a,\ \ !b\ ?b\ !a\ ?a,\ \ !b\ !a\ ?b\ ?a,\ \ !b\ !a\ ?a\ ?b\ \}$. ∎

An MSC Graph [5] 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.

Example 6.2: Fig. 8(c) presents an MSC graph G which consists of a single state. A linearization $!a\ !a\ ?a\ ?a\ !a\ ?a$ belongs to $L(G)$ because if we run G by traversing the transition twice, we shall connect the MSC associated with the state three times, and the resulting MSC is Fig. 8(d). Obviously $!a\ !a\ ?a\ ?a\ !a\ ?a$ is one of the linearizations of Fig. 8(d). Actually the language $L(G)$ can be described using the following [5]:

$$L(G) = \{(!a \mid ?a)^* \mid |!a| = |?a| \wedge \text{ for any prefix } |!a| \geq |?a|\}.$$

Note that the semantics of MSC graphs do not correspond to concatenations of the languages of the MSCs from each state along a path. ∎

At the first glance, the difference between the MSC graph framework and the conversation oriented framework is trivial—the MSC model specifies the ordering of the receive events while the conversation model does not. In the conversation model the timing of the receive event is considered a local decision of the receiving peer and is not taken in to account during the analysis of interactions among multiple peers. This seemingly trivial difference, however, leads to interesting and significant differences in the technical results for realizability analysis. But first let us investigate the relative expressive power of the two models.

MSC Graphs vs. Conversation Protocols:

Example 6.3: Let M_a and M_b be the two MSCs in Fig. 9 (a) and (b), respectively, and let operator π_{send} denote the "projection to send events" which removes all receive events from the linearizations of MSCs. Obviously $\pi_{send}(L(M_a)) = \pi_{send}(L(M_b)) = \{ab, ba\}$. But $L(M_a) \neq L(M_b)$ because in M_a $?a$ precedes $?b$, while in M_b $?b$ is before $?a$. ∎

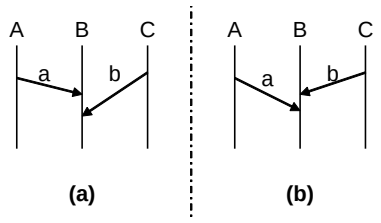

Fig. 9. Two MSC examples

Example 6.3 can be summarized as follows.

Proposition 6.1: There exist two MSCs M_1 and M_2, and one conversation protocol $\mathcal{R}$ where $L(M_1) \neq L(M_2)$ however $\pi_{send}(L(M_1)) = \pi_{send}(L(M_2)) = L(\mathcal{R})$.

Consider a conversation protocol for three peers A, B, and C which specifies a single conversation $m_{A\to B}\ m'_{C\to A}$ where message $m_{A\to B}$ is sent by A received by B and message $m'_{C\to A}$ is sent by C received by A. No MSC graph can capture the executions described by this conversation protocol, because there is no way to enforce the send of m to precede m'. Hence we have the following proposition.

Proposition 6.2: There exists a conversation protocol $\mathcal{R}$ where for any MSC graph G, the following is true: $\pi_{send}(L(G)) \neq L(\mathcal{R})$.

The above two Propositions imply that conversation protocols and MSC graphs are incomparable with regard to their expressive power: there are two different MSC graphs where conversation protocols cannot distinguish the difference; whereas there are conversation protocols which have no equivalent MSC graphs.

MSC Graphs vs. Bottom-up Web Service Compositions:

Now we show that the MSC Graph model and the bottom-up web service composition model are incomparable in terms of their expressive power.

Proposition 6.3: There exists an MSC graph G, such that, for any web service composition $\mathcal{W}$, $\mathcal{C}(\mathcal{W}) \neq \pi_{send}(G)$.

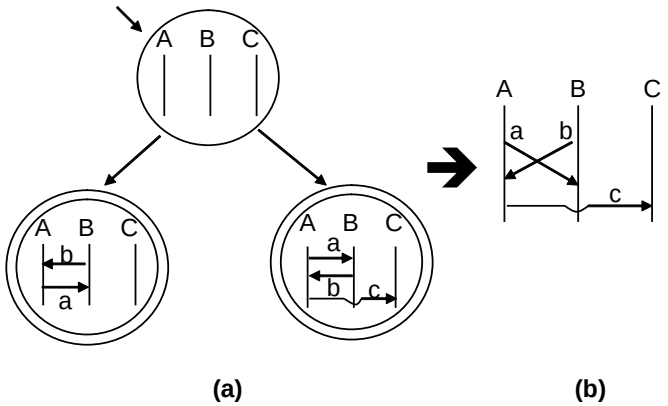

Fig. 10. The MSC Graph for Proposition 6.3

Proof: Let G be the MSC graph presented in Fig. 10(a). Obviously, $\pi_{send}(L(G)) = \{ba, abc\}$. Recall that $\{ba, abc\}$ is essentially the language discussed in Fig. 5, any web service composition that generates ba and abc will generate bac as well. Hence, there does not exist a web service composition $\mathcal{W}$ such that $\mathcal{C}(\mathcal{W}) = \pi_{send}(L(G))$. ∎

A converse question is: given a bottom-up web service composition $\mathcal{W}$, can its conversation set always be captured by an MSC graph? The following proposition gives a negative answer.

Proposition 6.4: There exists a web service composition $\mathcal{W}$, such that, for each MSC Graph G: $\mathcal{C}(\mathcal{W}) \neq \pi_{send}(L(G))$.

Proof: Recall that the web service composition in Fig. 3(a) produces a conversation set $\mathcal{L} \subseteq (a_1|r_1|a_2|r_2)^* e$ where for each word in $\mathcal{L}$, $|a_1| = |r_1|$, and $|a_2| = |r_2|$, and for any prefix $|a_1| \leq |r_1|$, and $|a_2| \leq |r_2|$. Now assume that there is an MSC graph G such that $\pi_{send}(L(G)) = \mathcal{L}$. Given an MSC graph G, we can show that there exists an FSA $\mathcal{A}$ such that the join of the projections of $L(\mathcal{A})$ to all peers (called join closure) is $\pi_{send}(L(G))$ [20]. The generation of such an FSA from an MSC graph G is straightforward. We simply convert each MSC M in G into an FSA that accepts $\pi_{send}(L(M))$, and then connect all neighboring FSAs using ϵ transitions. Now, since the join of all projections (called "join closure") of $L(\mathcal{A})$ is $\mathcal{L}$, $L(\mathcal{A})$ must be a subset of $\mathcal{L}$. However, it is not hard to see that for any subset of $\mathcal{L}$, its join closure produces itself. Hence, $L(\mathcal{A})$ must be $\mathcal{L}$, which is not a regular language. This contradicts with the fact that $\mathcal{A}$ is an FSA. Therefore the web service composition in Fig. 3(a) has no corresponding MSC Graph G. ∎

Combining Proposition 6.3 and 6.4, we have the conclusion that the expressive power of MSC graphs and the bottom-up web service compositions are not comparable.

From the above comparison, we see that the difference in expressiveness between two models results from the decision on whether to record the *receive* events in the behavior modeling of web services. In our approach the ordering of the receive events is considered a local decision and is not taken into account while analyzing the global interactions. Hence, in our approach the ordering of receive events is like a "don't care" condition which can simplify the specification of interactions. For instance, a single word protocol ab is more concise than the two MSCs needed in Fig. 9. On the other hand realizability problem in our approach can be more severe since we focus on global ordering of send events. For example, the non-realizable conversation protocol $\{m_{A\to B}\ m'_{C\to A}\}$ cannot be specified using MSC at all.

C. Related Work on Realizability and Synchronizability

The realizability problem with the top-down specification approach discussed in this paper is not an isolated problem. It exists for many prevalent global behavior specification paradigms, e.g., Message Sequence Charts (MSC) [31] and its extensions such as high-level MSC (hMSC) [39] and MSC Graphs [5]. The unspecified conversations generated in Example 4.1 and Fig. 5 are similar to the "implied scenarios" in [39] and the "implied MSC" in [5]. Notice that the use of input queues is not the sole reason which introduces unspecified behaviors. For example, the hMSC model in [39] has the realizability problem when synchronous communication semantics is used. The asynchronous communication semantics can lead to more complex unspecified behaviors like the ones in Fig. 5.

As we have shown in Theorem 4.1, synchronizability and realizability are closely related. While to the best of our knowledge, synchronizability was not discussed by any previous work, the notion of realizability has been

an interesting topic in concurrent and distributed system research since 1980's (see [1], [35], [36]). However, the term "realizability" may have different semantics under different contexts. 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 paper is rather different. In our model, the environment of an individual peer consists of other peers whose behaviors are also governed by portions of the protocol relevant to them. In addition, our definition of realizability requires that implementation should generate *all* (instead of a subset of) behaviors as specified by the protocol. A closer notion to the realizability definition used in this paper is the "weak realizability" of Message Sequence Chart (MSC) Graphs studied in [5]. However, as discussed above, the MSC Graph model captures both send and receive events, while in our web service composition model we are interested in the ordering of send events only. We have shown above that the two models are not comparable in terms of their expressive power. Now we will show that the realizability analysis for the two models are also different.

In [4], [5], weak and safe realizability problems were raised on MSCs (a finite set of MSCs) and MSC graphs respectively. Alur et al. showed that the decision of realizability for MSCs is decidable, however it is not decidable for MSC graphs. They gave the following sufficient and necessary conditions for realizability.

1) An MSC (MSC graph) M is weakly realizable if and only if $L(M)$ is the *complete* and *well-formed* join closure of itself. (Here *complete* means each send event has a matching receive event, and *well-formed* means each receive event has a matching send event.)
2) An MSC (MSC graph) M is safely realizable if and only if condition 1 is satisfied and *prefix*$(L(M))$ is the *well-formed* join closure of itself. Safe realizability is the weak realizability plus deadlock freedom during composition of peers.

These two conditions look very similar to our *lossless join* property. However there are key differences here: 1) In the MSC model, the conditions are sufficient and necessary conditions, while in conversation based model, lossless join is a sufficient condition only, and, 2) it is undecidable to check the above two conditions for MSC graphs, because of the requirement of well-formedness and completeness. Alur et al. introduces a third 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 can be restrictive, for example, Fig. 8(c) does not satisfy the boundedness condition because its queue length is not bounded. Note that the realizability conditions in our conversation model do not require queue length to be bounded. In addition each of the realizability conditions in our conversation model can be checked independently, and the decision procedure is decidable.

In the following, we present another example to illustrate the difference between the realizability conditions for the two models. Fig. 10(a) is an equivalent MSC graph of the conversation protocol in Fig. 5, and Fig. 10(b) is an MSC implied by the MSC graph (this implied MSC is generated by taking the event sequence for A and C from the right branch of Fig. 10(a), and taking the event sequence for B from the left branch). Consider how Fig. 10(a) and Fig. 5 violate the realizability conditions in the two models. Fig. 10(a) violates the condition (1) for weak realizability, because the implied MSC (i.e., Fig. 10(b)) is included in its join closure. However, the conversation protocol at Fig. 5 does not violate the *lossless join* condition, rather it violates the autonomy condition. Hence the lossless join conditions for the two models are essentially different.

VII. Conclusions and Future Directions

This paper presents a state machine based specification approach for asynchronously communicating web services. The notion of conversation is used to model the interactions among the peers participating to a web service composition, where the ordering of the send (but not receive) events are recorded. To avoid the undecidability of the verification, a technique called synchronizability analysis is developed. If a web service composition is identified to be synchronizable, it produces the same set of conversations under both asynchronous and synchronous communication semantics. Hence the LTL verification for synchronizable web service compositions can be conducted using the synchronous communication semantics and is guaranteed to be decidable (when each peer in the composition is specified using a finite state machine). In our earlier work, a realizability analysis was developed for top-down specification of composite web services. Our results in this paper reveal a close relationship between the synchronizability and realizability analyses. Synchronizability and realizability analyses are implemented as part of the Web Service Analysis Tool (WSAT) and play a crucial role in the automated verification framework for web services.

While the formal models presented in this paper nicely capture the control flows of static web services, they do not represent dynamic behaviors, e.g., dynamic instantiation of business processes and dynamic establishment of communication channels. We plan to extend our formal models to capture these dynamic behaviors in the future. This will lead to new research problems. For example, how do we specify a top-down conversation protocol for a web service composition where new peers can dynamically join? How will the dynamic behaviors effect the realizability and synchronizability analyses? How do we verify systems with dynamic process instantiations? We believe that symbolic model checking and automated abstraction can be useful for tackling these problems.

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

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

[5] 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.

[6] Adam Bosworth. Loosely speaking. *XML & Web Services Magazine*, 3(4), April 2002.

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

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

[9] 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.

[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] World Wide Web Consortium. Extensible markup language (XML). `http://www.w3c.org/XML`.

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

[14] 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.

[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] Xiang Fu. *Formal Specification and Verification of Asynchronously Communicating Web Services*. PhD thesis, University of California, Santa Barbara, 2004.

[21] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[22] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[23] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, 2003.

[24] IBM. Conversation Support Project. `http://www.research.ibm.com/convsupport/`.

[25] Java Message Service. `http://java.sun.com/products/jms/`.

[26] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP 74*, pages 471–475. North-Holland, 1974.

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

[28] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. Principles of Distributed Computing*, pages 137–151, 1987.

[29] Gerry Miller. .Net vs. J2EE. *Comm. of the ACM*, 46(6):64–67, June 2003.

[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/msmq/i`.

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

[34] W. Peng. Single-link and time communicating finite state machines. In *Proc. 2nd Int. Conf. on Network Protocols*, pages 126–133, 1994.

[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] Simple Object Access Protocol (SOAP) 1.1. W3C Note 08, May 2000. `http://www.w3.org/TR/SOAP/`.
[39] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifi cations and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology*, 13:37–85, 2004.
[40] Universal description, discovery and integration (uddi) protoocl. `http://www.uddi.org`.
[41] Joseph Williams. The Web Services Debate J2EE vs. .Net. *Comm. of the ACM*, 46(6):59–63, June 2003.
[42] Web Service Choreography Description Language (WS-CDL). `http://www.w3.org/TR/2004/ws-cdl-10-20041217`, 2004.
[43] Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.
[44] XML Schema. `http://www.w3c.org/XML/Schema`.