Model Checking Interactions of Composite Web Services

Xiang Fu, Tevfik Bultan, and Jianwen Su
University of California at Santa Barbara

Abstract. There are two main challenges in the verification of composite web services: 1) Asynchronous messaging makes most interesting problems undecidable, and 2) rich data representation (XML) and data manipulation (e.g. XPath query) forbids direct application of model checking tools. In this paper, we present a top-down specification and verification approach to tackle both of these problems. In our framework, each peer (individual web accessible program) interacts with other peers via asynchronous messages. We define a *conversation* among the peers as the global sequence of messages exchanged by the peers. We propose a top-down approach where the set of desired conversations of a web service is specified as a guarded automaton, which we call a *conversation protocol*. Guards of the automata are XPath queries that manipulate message contents. We show that if four *realizability* conditions are satisfied by a conversation protocol, the projections of the protocol to each peer are guaranteed to preserve any LTL properties satisfied by the protocol. We also present the translation from a guarded conversation protocol to a Promela specification that can be verified by the SPIN model checker.

1 Introduction

Web services can be described as web accessible software that provide interfaces for service description, discovery and interaction [7]. By introducing a framework for decoupling service descriptions from their implementations, web services provide a promising technology for interoperability and integration. There are many challenges in developing web services: a) Web services implemented using different platforms (e.g. .Net or J2EE) should be able to interact with each other; b) It should be possible to modify an existing web service without modifying other services that interact with it; c) Web services should be able tolerate pauses in availability of other services and slow data transmission. Web services address these challenges with the following common characteristics: 1) standardized data transmission via XML, 2) loose coupling among interacting web services through standardized interfaces, 3) asynchronous message passing.

In this paper, we address the specification and verification of interaction in composite web services. We model a composite web service as a set of *peers* interacting with asynchronous messages in XML format. Due to the distributed nature of web services, it is not possible to assume that their interaction will be controlled by a central coordinating process. In a typical scenario, the interacting peers will be running autonomously, on different servers belonging to different organizations and no single peer will have control over the global interaction. Such a distributed nature and asynchronous message passing make it extremely hard to ensure the "correctness" of the interaction merely through the design of each peer individually. The solution we propose is to use a protocol which specifies the interaction patterns among communicating web services. We call such protocols *conversation protocols*.

In our model of a composite web service, each peer has a queue for all of its input messages and may send messages to the input queues of other peers. To model the global behavior of the composite web service we define a virtual *watcher* that records the sequence of messages as they are sent by the peers [4]. We call the message sequences observed by the watcher the *conversations* of the composite web service. A conversation protocol defines the set of allowed conversations of a composite web service [9]. Our contributions in this paper are: 1) We present a language for specification of guarded conversation protocols on XML messages. We use XPath expressions [16] for specification of the guards and the assignments of the protocols, and MSL type expressions [3] for the message type declarations. 2) We present and implement realizability tests for guarded conversation protocols. For a conversation protocol which passes the realizability tests, we automatically synthesize peers that generate only the conversations specified by the conversation protocol. 3) We present a translator that translates the conversation protocols to Promela (the input language of the SPIN model checker [10]), and we show that properties of the conversation protocols can be automatically verified using the SPIN model checker.

Related Work: In [11] verification and composition of web services specified in a Petri Net model is investigated. In [8] web service compositions are specified using message sequence charts and modeled using finite state machines. Both models characterize only individual behaviors of the peers in our model, rather than global behavior of peers which communicate with asynchronous message passing via unbounded queues. Also, XML data manipulation is not addressed in [11] or [8].

Realizability of Message Sequence Chart (MSC) Graphs [1] is similar to the realizability problem discussed in this paper with the following differences: 1) the MSC Graph model captures both “send” and “receive” events, while we are interested in the ordering of “send” events only, 2) our results are on protocols with XML messages and XPath guards, 3) even if the message contents are ignored, the realizability conditions used in this paper and in [1] are different, and 4) we implement the realizability checks for the specification language presented in this paper.

Our previous work on web services focused on formally characterizing conversations generated by composite web services [4] and realizability conditions for conversation protocols specified as finite state machines with asynchronous messages without data content [9]. This paper extends the earlier work with 1) a specification language for conversation protocols with messages with XML data content, 2) extension of the realizability conditions in [9] to protocols with XML messages, 3) implementation of the realizability checks and peer synthesis, and 4) a model checking technique for conversation protocols via translation to Promela.

2 Web Services

In this section we show how web service standards can be used to specify web services, using a “stock analysis” service (SAS) example. Fig. 1 shows the overall structure of

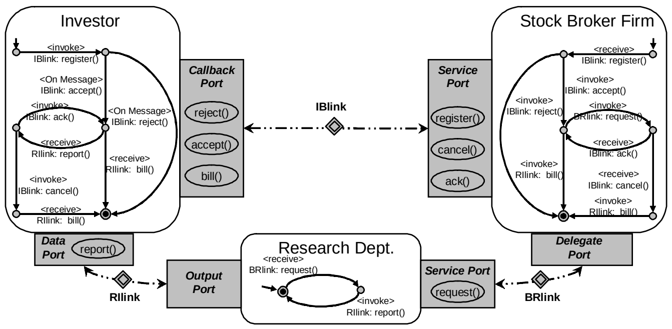


Fig. 1. Stock Analysis Service

the SAS. It involves three peers: Investor (Inv), Stock Broker Firm (SB), and Research Department (RD). Inv initiates the stock analysis service by sending a `register` message to SB. SB may `accept` or `reject` the registration. If the registration is accepted, SB sends an analysis `request` to RD. RD sends the results of the analysis directly to the Inv as a `report`. After receiving a `report`, Inv can either send an `ack` to SB or `cancel` the service. Then, SB either sends the `bill` for the services to Inv, or continues the service with another analysis `request`.

We now discuss how different parts of the web service in Fig. 1 are modeled using the web services standards. In particular the message contents are specified as XML documents with the message types in XML Schema and MSL, expressions on messages are specified in XPath, the communication ports are specified in WSDL, and the three peers are specified in BPEL.

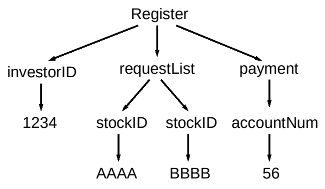
XML: Extensible Markup Language (XML) is a markup language for describing data [15]. As the universal data transfer format over the Internet, XML plays a central role in specifying semi-structured data in a way that is platform and language neutral, and in some sense, self-explanatory. It is widely agreed that messages exchanged among web services should be in the XML format. Similar to HTML, XML tags are written as `<tag>` followed by `</tag>`. However, tags in XML describe the content of the data rather than the appearance. Fig. 2(a) shows an XML document containing the data for a `register` message for the stock analysis service. A `register` message is sent from Inv to SB to register for the stock analysis service. It consists of a string containing the identification of the Inv, a list of stock identifiers that the investor is interested in, and payment information. XML documents can be modeled as trees where each internal node corresponds to a tag and leaf nodes correspond to basic types. The document in Fig. 2(a) corresponds to the tree in Fig. 2(b).

XML Schema and MSL: XML provides a standard way to exchange data over the Internet. However, the parties that exchange XML documents still have to agree on the *type* of the data, i.e., what are the tags that will appear in the document, in what order, etc. XML Schema [17] is a language for defining XML data types. Model Schema Language (MSL) [3] is a compact formal model that captures most features of XML Schema. We use a slightly simplified version of MSL with type expressions defined as:

$$g \rightarrow \epsilon \mid b \mid t\,[g] \mid g\{m,n\} \mid g\ ,\ g \mid g|g$$

```
<Register>
<investorID>
1234
</investorID>
<requestList>
<stockID>
AAAA
</stockID>
<stockID>
BBBB
</stockID>
</requestList>
<payment>
<accountNum>
56
</accountNum>
</payment>
</Register>
```

(a)



(b)

```
Register[
 investorID[xsd:int],
 requestList[
  stockID[xsd:string]{1,10}
 ],
 payment[
  creditCardNum[xsd:int] |
  accountNum[xsd:int]
 ]
]
```

(c)

Fig. 2. An XML document (a), the corresponding tree (b), and its type declaration in MSL (c)

where g is an XML type (i.e. an MSL type expression), ϵ is the empty sequence, b is a basic type such as string, boolean, int, etc., t is a tag, m and n are positive integers, and '`[]`', '`{}`', '`,`' and '`|`' are MSL type constructors. The semantics of MSL type constructors can be summarized as follows: t`[`g`]` denotes a type with root node labeled t with children of type g; $g\{m,n\}$ denotes a sequence of size at least m and at most n where each member is of type g; g_1 `,` g_2 denotes an ordered sequence where the first member is of type g_1 and the second member is of type g_2; and, $g_1 \mid g_2$ denotes a choice between type g_1 and type g_2, i.e., either type g_1 or type g_2, but not both.

Fig. 2(c) shows the MSL type declaration for the `register` message. According to the MSL declaration, a `register` message consists of an ordered sequence of `investorID`, `requestList`, and `payment`, where element `investorID` is an integer, `requestList` is an ordered sequence of elements of type `stockID` (the size of the sequence is at least 1 and at most 10), `stockID` is string, and `payment` is either a `creditCardNum`, which is an integer, or an `accountNum`, which is also an integer.

XPath: In order to write specifications or programs that manipulate XML documents we need an expression language to access values and nodes in XML documents. We use a subset of XPath [16] to navigate through XML trees and return the answer nodes. The fragment of XPath we use consists of the following operators: node name test (t), the child axis (`/`), the descendant axis (`//`), wildcard (`*`), self-reference (`.`), parent-reference (`..`), and qualifiers `[]`. An *XPath query* q is defined with the following grammar:

$$q \rightarrow \texttt{.} \mid \texttt{..} \mid b \mid t \mid \texttt{*} \mid \texttt{/}q \mid \texttt{//}q \mid q\texttt{/}q \mid q\texttt{//}q \mid q\texttt{[}q\texttt{]} \mid q\texttt{[}exp\texttt{]}$$

where *exp* denotes a predicate on basic types (i.e. on the leaf nodes of the XML tree), b denotes a basic type such as string, boolean, int, etc., and t denotes a tag. Semantics of each XPath operator can be defined as a function which takes an XML tree and a set of context nodes (in the same XML tree) as input and returns a set of nodes in the same XML tree as output. The result of an XPath query is obtained by evaluating its operators from left to right. The semantics of the XPath operators are defined as follows. Given a set of XML tree nodes: `.` returns the same nodes; `*` returns all of their children; `..` returns their parents; b returns the children that are of basic type b; t returns the children which are labeled with t; $/q$ executes q on input nodes; $//q$ executes q on any descendant of input nodes; q_1/q_2 executes q_1 first and then q_2 on the result of q_1;

q_1`//`q_2 executes q_1 first, and then executes q_2 on descendants of the results returned by q_1; q_1`[`q_2`]` selects from the nodes returned by q_1 which makes q_2 evaluate to at least one node; q`[`*exp*`]` returns the nodes in the result of q which makes the expression *exp* evaluate to true.

For example, given the XML document in Fig. 2, the query `//payment/*` returns the node labeled `accountNum`, `/Register/requestList/stockID/string` returns the nodes labeled `AAAA` and `BBBB`, `//stockID[string=AAAA]/string` returns the node labeled `AAAA`, and `//[int>0]` returns the nodes labeled `1234` and `56`.

XPath queries can be combined with operators and predicates on basic types to form *XPath expressions*. Note that since XPath queries can result in a set of values, XPath expressions are also evaluated on a set of values. For example, `//[int>0] + 1` is an XPath expression which will evaluate to two values, `1235` and `57`, for the above example.

SOAP and WSDL: Simple Object Access Protocol (SOAP) [12] is a standard messaging protocol for exchanging XML documents. Built upon XML Schema and SOAP, WSDL (Web Services Description Language) [13] is an XML-based language used to publish the public “hook-up” interfaces of Web services. In WSDL, message types are declared using XML Schema. For example, the WSDL specification for the SAS example contains a message type declaration for the `Register` message XML Schema, which corresponds to the MSL type declaration in Fig. 2(c). In WSDL, each port can host one or more functions, and the input/output message types are specified for each function. For the SAS example, the WSDL specification for the SB contains a port declaration called `Service Port` with three input functions: `register`, `cancel`, and `ack`. Hence, using the WSDL type and port declarations we can specify the messages that are serviced by each peer.

BPEL: The simple function call model employed by WSDL makes it hard to capture long running complex composite web services. Many competing standards such as BPEL (Business Process Execution Language for Web Services) [2], WSCI (Web Service Choreography Interface) [14] have been proposed to address this problem.

In a BPEL specification, a partner link connects a pair of port declared in the WSDL specification. A BPEL specification for a peer also includes the control flow with the associated receive and send operations through the partner links and WSDL ports. BPEL has many control structures, such as “sequence”, “while”, “switch”, etc. The atomic operations in BPEL are: “invoke”, “receive”, and “reply” for sending and receiving messages; and “assign” for updating values of the variables. Assignments are conducted on complex type variables with XPath expressions to identify the sources and the destinations of the assignments statements. The state machines for the peers in Fig. 1 can be specified using one BPEL specification for each peer.

3 Conversation Protocols

Although parts of the SAS example in Fig. 1 can be specified using the existing standards no languages can specify the global interaction among them. We propose a top-down approach in which the interaction patterns among communicating web services are specified as conversation protocols: A *conversation protocol* is a guarded automaton

```
Conversation {
 Schema{
  PeerList{Customer,Broker,ResearchDept},
  TypeList{
   Register[
    investorID[xsd:int],
    orderID[xsd:int],
    requestlist[
     stockID[xsd:int]{1,10}
    ],
    payment [
     accountNum[xsd:int] |
     creditCard[xsd:int]
    ]],
   Reject[
    investorID[xsd:int],
    orderID[xsd:int],
    comment[xsd:string]],
   Accept[
    investorID[xsd:int],
    orderID[xsd:int],
    comment[xsd:string]],...
  },
  MessageList{
   register{Customer->Broker:Register},
   reject{Broker->Customer:Reject},
   accept{Broker->Customer:Accept},...
  }
 },
```

```
Protocol{
 States{s1,s2,...,s12},
 InitialState {s1},
 FinalStates{s3},
 TransitionRelation{
  t1{ s1 -> s2 : register, Guard{true} },
  t2{ s2 -> s3 : reject,
   Guard{ true =>
    $reject[
     //investorID:=$register//investorID,
     //orderID:=$register//orderID]}
  },
  t3{ s3 -> s4 : terminate,
   Guard{ true =>
    $terminate[
     //investorID:=$register//investorID,
     //orderID:=$register//orderID]}
  },
  t4{ s2 -> s5 : accept,
   Guard{ true =>
    $accept[
     //investorID:=$register//investorID,
     //orderID:=$register//orderID]}
  },
  ...
 }
}
}
```

Fig. 3. Conversation Specification for the Stock Analysis Service

that describes the set of desired conversations (sequence of messages) generated by a set of interacting web services.

A conversation specification consists of a conversation schema that specifies the peers and the communication channels, and a conversation protocol. Fig. 3 shows parts of the conversation specification for the SAS in Fig. 1. The abstract syntax of the language we developed for specification of conversations is shown in Fig. 4. The nonterminals *name*, *source*, *destination*, *type*, and *message* all denote strings. The nonterminal *StringList* denotes a list of strings separated by commas, and the nonterminal *MslExpList* denotes a list of MSL expressions separated by commas. The nonterminal *XPathExp* denotes an XPath expression. In a valid specification, *source* and *destination* should be state names, *type* should be name of an MSL type and *message* should be a message name all defined in the specification.

Conversation Schema: Formally, a *conversation specification* is a tuple $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$ where (P, M) is a conversation schema and $\mathcal{A}$ is a conversation protocol. A *conversation schema* is a pair (P, M) where P is a finite set of peers and M a finite set of message types. Each message type $c \in M$ is transmitted on exactly one peer-to-peer channel.

In our specification language messages are XML documents. For each message type $c \in M$, let $\textsc{dom}(c)$ denote all the XML documents that match to the type declaration of c. Given a set of message types M, we define the *message alphabet* as $\Sigma = \bigcup_{c \in M} \{c\} \times \textsc{dom}(c)$. Each element $m \in \Sigma$ is called a *message*. Let $\textsc{type}(m) \in M$ denote the type of message m. We say that m is an *instance* of $\textsc{type}(m)$.

Spec $\rightarrow$ { *Schema* , *Protocol* }
Schema $\rightarrow$ `Schema{ PeerList{` *StringList* `},TypeList{` *MslExpList* `},`
    `MessageList{` *MessageList* `}}`
MessageList $\rightarrow$ *Message* | *Message* , *MessageList*
Message $\rightarrow$ *name* { *source* -> *destination* : *type* }
Protocol $\rightarrow$ `Protocol{ States{` *StringList* `},InitalState{` *StringList* `},`
    `FinalStates{` *StringList* `},TransitionRelation{` *TransitionList* `}}`
TransitionList $\rightarrow$ *Transition* | *Transition* , *TransitionList*
Transition $\rightarrow$ *name* { *source* -> *destination* : *message* , *Guard* }
Guard $\rightarrow$ `Guard{` *XPathExp* => *Update* }
Update $\rightarrow$ *name* { *AssignList* }
AssignList $\rightarrow$ *Assign* | *Assign* , *AssignList*
Assign $\rightarrow$ *XPathExp* := *XPathExp*

Fig. 4. Conversation Specification Syntax

Left hand side of Fig. 3 shows the conversation schema for the SAS example. Note that, in the conversation schema, the message types are declared using MSL, and then the sender and receiver for each message type are declared using the message list.

Conversation Protocol: Formally, a *conversation protocol* is a guarded automaton $\mathcal{A} = (M, T, s, F, \Delta)$, where M is the set of message classes, T a finite set of states, $s \in T$ the initial state, $F \subseteq T$ a set of final states, and Δ the transition relation. Each transition $\tau \in \Delta$ is of the form $\tau = (s, (c, g), t)$. Here, $s, t \in T$ are the source and the destination states of the transition τ, $c \in M$ is a message type and g is the *guard* of the transition. A guard consists of a guard condition and a set of assignments. A transition is taken only if the guard condition evaluates to true. The assignments specify the contents of the message that is being sent. Given a transition $\tau = (s, (c, g), t)$ where peer p is the sender of the message of type c, then guard g is a predicate of the following form: $g(m, \vec{m})$, where m is the message being sent, and the vector $\vec{m}$ contains the last instance of each message type that is received or sent by peer p.

The syntax for transition guards is shown in Fig. 4 (nonterminal $Guard$). Each guard consist of guard condition which is an XPath expression followed by an update expression which is a list of assignments. Update expressions start with the name of the message type that is being sent (in XPath expressions messages appear with `$` as prefix), and continue with a list of assignments (nonterminal $Update$ in Fig. 4). Each assignment has an XPath expression on the left hand side, which evaluates to a node of the XML message that is being sent. The right side of the XPath expression evaluates to a node of the last instances of XML messages received or sent by the sender of the current message.

Right hand side of Fig. 3 shows the conversation protocol for the SAS in Fig. 1. For example, on transition `t4` the conversation protocol goes from state `s2` to `s5`, and a message of type `accept` is sent from peer `Broker` to peer `Customer`. The assignment list declares that, in the `accept` message that is sent, the `investorID` and the `orderID` elements should be same as the `investorID` and the `orderID` elements in the last `register` message that is received by the `Broker` peer.

A conversation protocol is only able to remember the last sent message for each class. This is not a significant restriction based on the web services we have examined.

Note that more information about the sent and received messages can be stored in the states of the conversation protocol. Another approach would be to extend the guard definition so that the guards can refer to the last ℓ instances of each message class where ℓ is a fixed integer value. Such an extension would not effect the approach and the results discussed in this paper.

Conversations: Given a conversation specification $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$, a *configuration* of the conversation protocol $\mathcal{A} = (M, T, s, F, \Delta)$ is a tuple $(t, \vec{m})$ where $t \in T$ and message vector $\vec{m} \in \text{DOM}(c_1) \times \cdots \times \text{DOM}(c_k)$, $k = |M|$, keeps track of the latest instance of each message type. For a vector $\vec{m}$ and message type c, let $\vec{m}[c]$ denote its projection to $\text{DOM}(c)$. A configuration $(t_1, \vec{m}_1)$ is said to *derive* another configuration $(t_2, \vec{m}_2)$ *via* a message $m \in \Sigma$, written as $(t_1, \vec{m}_1) \overset{m}{\to} (t_2, \vec{m}_2)$, if there is a transition $(t_1, (c, g), t_2) \in \Delta$ such that

– message m and $\vec{m}_1$ satisfy the guard g, i.e., $g(m, \vec{m}_1)$ is true, and
– $\vec{m}_2[\text{TYPE}(m)] = m$ and $\vec{m}_1$ and $\vec{m}_2$ have same instances for all other message types, i.e., for all $c \in M$ such that $c \neq \text{TYPE}(m)$, $\vec{m}_1[c] = \vec{m}_2[c]$.

Let $\mathcal{A} = (M, T, s, F, \Delta)$ be a conversation protocol and let $w = w_0, w_1, \ldots$ be a finite word over Σ, i.e., $w \in \Sigma^*$. A *run* of $\mathcal{A}$ for the word w is a finite sequence of configurations $\gamma_0, \gamma_1, \ldots$ such that, $\gamma_0 = (s, (\perp, \ldots, \perp))$ where s is the initial state of the conversation protocol, $\perp$ denotes an uninitialized message, and for all $0 \leq i < |w|$, $\gamma_i \overset{w_i}{\to} \gamma_{i+1}$. A word $w \in \Sigma^*$ is *accepted* by the conversation protocol $\mathcal{A}$ if there exists a run γ for w, such that $\gamma_{|w|} = (t, \vec{m})$ and $t \in F$. We define, $\mathcal{L}(\mathcal{P})$, the set of conversations defined by a conversation specification $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$, as the set of words accepted by the conversation protocol $\mathcal{A}$.

We use the temporal logic LTL to express properties of conversations. We define the set of atomic propositions as follows: Each atomic proposition is either of the form c where c is a message class (i.e., $c \in M$), or *c.pred*, where $c \in M$ and *pred* is a predicate over the content of c. We denote that a message $m \in \Sigma$ *satisfies* an atomic proposition ψ by $m \models \psi$, where

$$m \models \psi \ \text{ iff} \begin{cases} \text{TYPE}(m) = \psi & \text{if } \ \psi \in M \\ \text{TYPE}(m) = c \ \wedge \ \textit{pred}(m) = \textit{true} & \text{if } \ \psi = c.\textit{pred} \end{cases}$$

LTL formulas are constructed from atomic propositions, logical operators $\wedge, \vee, \neg$, and temporal operators **X**, **G**, **U**, and **F**. The semantics of LTL temporal operators can be easily defined on finite length conversations [9]. Given a conversation specification $\mathcal{P}$ and each LTL property ϕ, we say that $\mathcal{P} \models \phi$, iff for all $w \in \mathcal{L}(\mathcal{P})$, $w \models \phi$.

For example, two sample LTL properties of the conversation protocol in Fig. 1 are:

1. **G**(`$register` $\to$ **F**(`$accept` $\vee$ `$reject`)): Each `register` will be eventually followed by a `reject` or `accept` message.
2. **G**(`$register//stockID` $== a \to$ **F**(`$request//stockID` $== a$)): Every `stockID` appeared in the `register` message will eventually appear in the `request` message from Broker to ResearchDept.

Note that in the second property a denotes a constant value.

A conversation specification $\mathcal{P}$ over a schema (P, M) is *finite state* if for each $c \in M$ message type c has a finite domain. Given a finite state specification, we can translate the guarded automaton of a finite state conversation protocol to a standard automaton without any guards and then use the results for finite state LTL model checking. In particular, we can prove the following:

Theorem 1. *Given a finite-state conversation specification* $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$ *where* $M = \{c_1, ..., c_k\}$*, and an LTL property* ϕ*, determining* $\mathcal{P} \models \phi$ *takes* $|T| \text{DOM}(c_1)| \cdots |\text{DOM}(c_k)| 2^{O(|\phi|)}$ *time.*

The above result follows from the size of the standard automaton constructed from the given conversation protocol specification and the results on LTL model checking [5]. Note that the exponential complexity in the size of the LTL formula is not a prohibitive factor since sizes of typical LTL formulas tend to be small. In section 5 we exploit the above result by translating conversation protocols to input language of the SPIN model checker and then use SPIN model checker to verify properties of conversations.

4 Realizable Protocols and Peer Synthesis

In this section, we give a formal model for a composite web service which consists of multiple peers communicating with asynchronous messaging. A *composite web service* is a tuple $\mathcal{S} = \langle (P, M), \mathcal{A}_1, ..., \mathcal{A}_n \rangle$, where (P, M) is a conversation schema, $n = |P|$ and each $\mathcal{A}_i$ is an implementation for peer $p_i \in P$. For each peer p_i, its implementation $\mathcal{A}_i$ is a guarded automaton $(M_i^{\text{in}}, M_i^{\text{out}}, T_i, s_i, F_i, \Delta_i)$, where M_i^{in} (M_i^{out}) are incoming (outgoing) message types for p_i, and T_i, s_i, F_i are the set of states, the initial state, and the set of final states, resp. Each transition $\tau \in \Delta_i$ has a source state $q_1 \in T_i$ and a destination state $q_2 \in T_i$ and is in one of the following three forms: 1) ϵ-move where $\tau = (q_1, \epsilon, q_2)$, which only changes the state, 2) consuming an input message where $\tau = (q_1, ?a, q_2)$ and $a \in M_i^{\text{in}}$, which changes the state and removes the consumed message from the input queue, or 3) producing an output message where $\tau = (q_1, (!b, g), q_2)$ and $b \in M_i^{\text{out}}$ and g is the transition guard, which changes the state and appends the produced message to the input queue of the receiving peer.

The global configuration of a composite web service $\mathcal{S}$ is a $(2n+3)$-tuple of form $(Q_1, t_1, ..., Q_n, t_n, w, \vec{s}, \vec{c})$ where for each $j \in [1..n]$, $Q_j \in \Sigma^*$ is the content of the input queue of peer p_j, t_j is the state of p_j, $w \in \Sigma^*$ is the sequence of messages recorded by the global watcher, and message vectors $\vec{s}, \vec{c}$ record the latest *sent* and *consumed* instances (resp.) for each message type. It is straightforward to define a derivation relation between two configurations based on the transition relations of the peers such that $\gamma \rightarrow \gamma'$ if and only if there exists a peer p_i and a transition $\tau \in \Delta_i$ such that executing the transition τ in configuration γ results in the configuration γ' [9]. Note that each send operation appends the message 1) to the input queue of the receiver and 2) to the global watcher at the same time.

A *run* of $\mathcal{S}$ is a finite sequence of configurations $\gamma = \gamma_0, \gamma_1, \gamma_2, ...$ that satisfies the following conditions: $\gamma_0 = (\epsilon, s_1, ..., \epsilon, s_n, \epsilon, [\perp, ..., \perp], [\perp, ..., \perp])$ is the initial configuration, where s_i is the initial state of p_i for each $i \in [1..n]$, and for each $i \geq 0$, $\gamma_i \rightarrow \gamma_{i+1}$, and $\gamma_{|\gamma|} = (\epsilon, s_1', ..., \epsilon, s_n', w, \vec{s}, \vec{c})$ is the final configuration, where for each peer p_i, $s'_i \in F_i$. A finite word $w \in \Sigma^*$ is a *conversation* of a composite web service $\mathcal{S}$ if there exists a run $\gamma = \gamma_0, \gamma_1, \gamma_2, ...$ of S such that, the value of the watcher in the final configuration is w. Let $\mathcal{L}(\mathcal{S})$ denote the set of conversations of $\mathcal{S}$.

Given a composite web service $\mathcal{S}$ and an LTL property ϕ, we say that $\mathcal{S} \models \phi$, iff for all $w \in \mathcal{L}(\mathcal{S})$, $w \models \phi$. We say that a composite web service $\mathcal{S}$ over a schema (P, M) has *finite message content* if for each message type $c \in M$, $\text{DOM}(c)$ is finite. In contrast to Theorem 1 we have the following property: Given a composite web service $\mathcal{S}$ with finite message content and an LTL property ϕ, testing $\mathcal{S} \models \phi$ is undecidable [9].

We now formalize the relationship between a conversation protocol and a composite web service. A composite web service $\mathcal{S}$ *conforms* to a conversation specification $\mathcal{P}$ iff $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{P})$, and $\mathcal{S}$ *realizes* $\mathcal{P}$ iff $\mathcal{L}(\mathcal{S}) = \mathcal{L}(\mathcal{P})$.

Given a conversation specification $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$, let $\Pi_i(\mathcal{A})$ denote the *projection* of the conversation protocol $\mathcal{A}$ onto a peer $p_i \in P$. $\Pi_i(\mathcal{A})$ is a guarded automaton obtained from $\mathcal{A}$ by replacing each move for a message that is not related to peer p_i with an ϵ-move, and dropping guards of the transition edges that are labeled with incoming messages. Let $\Pi(\mathcal{P}) = \langle (P, M), \Pi_1(\mathcal{A}), ..., \Pi_n(\mathcal{A}) \rangle$ be the composite web service derived from the conversation specification $\mathcal{P}$. Of course the interesting question is, if $\mathcal{L}(\Pi(\mathcal{P})) = \mathcal{L}(\mathcal{P})$, i.e., is the conversation specification and the composite web service which corresponds to its projection generate the same set of conversations? Below we present a set of realizability conditions which ensures that this is the case.

Realizability Conditions: In [9] we defined three realizability conditions for guardless conversation protocols. Here we extend these conditions by presenting an extra condition for the transition guards (i.e., that concerns the message contents). Then the realizability check is implemented by checking the first three realizability conditions on the skeleton of the conversation protocol (i.e, without considering the guards), and then checking the fourth realizability condition on the guards.

For a conversation protocol $\mathcal{A} = (M, T, s, F, \Delta)$, its *skeleton* is a standard (guardless) automaton $\mathcal{A}' = (M, T, s, F, \Delta')$ where each transition $\tau' \in \Delta'$ is generated by dropping the guard of a corresponding transition τ in Δ. Note that the language recognized by skeleton $\mathcal{A}'$, i.e., $L(\mathcal{A}')$ is a subset of M^*, while $L(\mathcal{A}) \subseteq \Sigma^*$.

The skeleton of a conversation protocols is realizable if it satisfies the following three realizability conditions [9]:

1) Synchronous compatibility: A conversation protocol skeleton is *synchronous compatible* if when we project the skeleton onto each peer and construct the Cartesian product of the projections, the resulting automaton does not contain a state where a peer p_i is ready to send a message to peer p_j but peer p_j is not ready to receive the message.

2) Autonomy: A conversation protocol skeleton is *autonomous* if for each peer p_i and each finite prefix w of a conversation, at most one of the following three conditions hold: a) the next transitions of p_i (including transitions that are reachable through ϵ-transitions) are all send operations, b) the next transitions of p_i (including transitions that are reachable through ϵ-transitions) are all receive operations, or c) p_i is either in a final state or it can reach a final state through ϵ-transitions.

3) Lossless join: A conversation protocol skeleton is a *lossless join* if we project the conversations allowed by the protocol to the alphabet of each peer and compute the join of the projected sets, the result is equal to the original conversation set.

Finally, we need a new fourth realizability condition that restricts the guards of a conversation protocol as follows:
4) Deterministic guards: To check the deterministic guard condition, for each peer p_i, we analyze the projected automaton $\Pi_i(\mathcal{A})$. We determinize $\Pi_i(\mathcal{A})$ as a guardless standard FSA, ignoring the guards. Each state in this determinized automaton corresponds to a set of states in $\Pi_i(\mathcal{A})$. For each state in the determinized automaton we collect all the transitions which have the same source, destination and message classes in the corresponding states in $\Pi_i(\mathcal{A})$. We require that at each state for all send transitions at that state there can be at most one guard for each message type, i.e., if there are two send transitions with the same message type, their guards have to be identical.

Theorem 2. Let $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$ be a conversation specification where $\mathcal{A}$ has deterministic guards and the skeleton of $\mathcal{A}$ is synchronous compatible, autonomous, and lossless join. Then $\mathcal{L}(\Pi(\mathcal{P})) = \mathcal{L}(\mathcal{P})$.

The proof can be easily extended from our results for conversation protocols without message contents and guards in [9].

5 Translation to Promela

In this section, we discuss the translation of a conversation protocol to Promela, the input language of the SPIN model checker [10]. We first introduce the mapping of an MSL type declaration to a Promela type definition, and then we show how to translate an XPath expression into an equivalent Promela code based on this mapping.

Translating MSL Type Expressions: Each MSL simple type has a counterpart in Promela. For example, “`xsd:int`” and “`xsd:boolean`” are mapped to Promela types “`int`” and “`bool`”, respectively. As strings are used solely as constants in our framework, all string constants in a protocol are collected and declared as symbolic constants using an “`mtype`” declaration in Promela and the type “`xsd:string`” is mapped to type “`mtype`”. MSL complex types are mapped to “`typedef`” record constructs in Promela. In the following we present the Promela translation for the `Register` type declared in Fig. 2(c).

```
typedef t2_requestlist{
  int stockID [10];
  int stockID_occ;
}
typedef t3_payment{
  int accountNum;
  int creditCard;
  mtype choice;
}
mtype {m_accountNum, m_creditCard};
```

```
typedef Register{
  int investorID;
  int orderID;
  t2_requestlist requestlist;
  t3_payment payment;
}
```

In the above Promela code, two additional type declarations are generated for elements `requestlist` and `payment` (prefixes `"t2_"` and `"t3_"` are generated to avoid name collisions). Since element `stockID` has max occurrence of 10, it is declared as an array in the translation, and an additional variable `stockID_occ` is used to record the actual occurrence of `stockID`. Also, note that, in the mapping of `payment`, an additional variable `choice` is declared to record which element of `payment` (`accountNum` or `creditCard`) is selected in each instance.

Translating XPath expressions without function calls: Before discussing the translation algorithm, let us first study one example. Given two messages `regInfo1` and `regInfo2` of type `Register`, the query "Is there any `stockID` that appears in both messages?" can be captured by the boolean XPath expression `$regInfo1//stockID == $regInfo2//stockID`. The translation of this XPath expression to Promela is as follows, and note that prefix "v_" is attached to message name to avoid name collisions.

```
bool b1 = false;
int i1=0, i2=0;
do :: i1 < v_regInfo1.requestlist.stockID_occ ->
      do :: i2 < v_regInfo2.requestlist.stockID_occ ->
         if ::v_regInfo1.requestlist.stockID[i1] == v_regInfo2.requestlist.stockID[i2]
             -> b1 = true;
         :: else -> skip;
         fi;
         :: else -> break;
      od;
   :: else -> break;
od;
```

The main body of the above Promela code is a nested loop that searches for the array indices which makes the equality test succeed. Integer variables `i1` and `i2` are used as indices, and boolean variable `b1` is used to record the final result.

The translation algorithm we used in our implementation generates the above code in two stages. In the first stage, each XPath query is symbolically evaluated while recording the required information to generate the Promela code for enumerating all the nodes that match the query. The second stage generates the Promela code by traversing the trees generated in the first stage.

An XPath expressions consists of a set of XPath queries (as described in Section 2) which are combined using operations and predicates on basic types (such as addition, equality, etc.). Each XPath query consists of a set of XPATH operators. In the first stage of the algorithm, we traverse each XML query in the XPath expression from left to right, one operator at a time. During this traversal we generate a tree. The root node of the tree (at level 0) corresponds to the root node of the XML tree. A node in level i is obtained by applying the operator i to a node in level $i-1$. The leaf nodes of the generated tree correspond to XML nodes which match the corresponding query. Some nodes in the generated tree correspond to arrays and represent all the elements in an array. We call such nodes *parameterized* nodes. At the end of the first stage we end up with one tree for each XPath query in the XPath expression. We can generate Promela code which traverses all the XML nodes that match an XPath query by traversing the tree generated for it in the first stage. Note that, in order to to do that, we need to generate loops for the parameterized nodes.

The second stage generates the Promela code for the XPath expression using the trees generated in the first stage. The generated code consists of a set of nested loops which are used for traversing arrays. The innermost statement in each nested loop block corresponds to the input XPath expression where each XPath query in the XPath expression is replaced with a string that corresponds to the XML nodes which match that query. For the above example, the innermost statement is:

```
v_regInfo1.requestlist.stockID[i1] == v_regInfo2.requestlist.stockID[i2] -> b1 = true;
```

which corresponds to the XPath expression

```
$regInfo1//stockID == $regInfo2//stockID
```

where XPath queries

`$regInfo1//stockID` and `$regInfo2//stockID`
are replaced with strings
`v_regInfo1.requestlist.stockID[i1]` and `v_regInfo2.requestlist.stockID[i2]`

Note that, the indices `i1` and `i2` indicate that this statement is generated within a pair of nested loops which will enumerate all the elements in the two arrays.

Translating XPath expressions with function calls: We support two function calls: `position()` and `last()`. They are useful to enumerate and traverse through XML documents. For example, XPath expression

`$request//stockID = $register//stockID[position()==last()]`

assigns `stockID` of a `request` with the last `stockID` in `register`. The Promela code generated for the above query is given below. The key point in the translation is to map each appearance of a `position()` and `last()` to an integer variable, and update values of these variables in the translated code. We omit the details of the code generation for function calls due to lack of space.

```
/* calculate last() */
//last() replaced by i1
int i1=0;
int i2=0;
//position() replaced by i3
int i3=0;
do
:: i2 <
   v_register.requestlist.stockID_occ
   -> i1++;
      i2++;
:: else -> break;
od;
```

```
/* calculate the statement */
i2 =0;
do
:: i2 < v_register.requestlist.stockID_occ
   -> i2++;
   /* update position() */
   i3++;
   if
   :: (i3==i1) ->
      v_request.id =
      v_register.requestlist.stockID[i2];
   :: else -> skip;
   fi;
:: else -> break;
od;
```

Translating a Conversation Protocol: Finally, we briefly present the layout of the Promela translation of a guarded conversation protocol below.

```
/*1. type declaration */
typedef Register{ ... }

/*2. message variables */
Register v_register; ...

/*3. system variable */
mtype = {m_register, m_reject ... };
mtype state;
mtype msg;

/*4. local variables */
int i1, i2 ...
```

```
active proctype GProtocol{
 /* 5. initialize all variables */
 do
    ::
    /* 6. evaluate last() calls */
    /* 7. evaluate guards of each transition */
      ...  bGuard_1 = true ...
    if
      /* 8. each transition */
         //transition 1
      :: state == s1 && bGuard1
        -> state = s2;
           msg = m_register;
        ...  assignmentlist ...
      /* 9. termination code for final states */
      :: state == s3 -> break;
    fi;
 od;
}
```

The first part of the translation contains type declarations which is followed by variable declarations for messages. There are two system variables: "`state`" is used to record the current state of the guarded automaton, and "`msg`" is used to record the current message that is being transmitted. The main body of the protocol is a "`do`" loop. During each iteration of the loop, first the enabling conditions of all the transitions are evaluated and the results are stored in boolean variables (one for each transition). Then, using a nondeterministic "`if`" statement an enabled transition is nondeterministically chosen and executed. Finally, if the current state is a final state, the protocol can jump out of the loop and terminate.

6 Experimental Results and Conclusions

We ran experiments on samples from major web service solution vendors and web service standard specifications. The following table summarizes our datasets and the corresponding experimental results. All conversation protocol source files, Promela translations, and verification results can be downloaded at web address [6].

Problem Set		**Size**			**Realizability Check**				**Verification**			
Source	**Name**	**#msg**	**#states**	**#transition**	**C1**	**C2**	**C3**	**C4**	**Promela length**	**StateVector(Byte)**	**Time(sec)**	**Depth**
-	SAS	9	12	15	$\surd$	$\surd$	$\surd$	$\surd$	1904	204	166	405
IBM Conversation Support Project	ConvSetup	4	4	4	$\surd$	$\surd$	$\surd$	$\surd$	340	32	43	187
	MetaConv	4	4	6	$\surd$	$\times$	$\surd$	$\surd$	-	-	-	-
	Chat	2	4	5	$\surd$	$\surd$	$\surd$	$\surd$	354	36	0	243
	BuyStuff	5	5	6	$\surd$	$\surd$	$\surd$	$\surd$	436	40	0.08	245
	Haggle	8	5	8	$\surd$	$\times$	$\surd$	$\surd$	-	-	-	-
	AMAB	8	10	15	$\surd$	$\surd$	$\surd$	$\surd$	783	52	0.01	475
BPEL1.1 spec.	shipping	2	3	3	$\surd$	$\times$	$\surd$	$\times$	-	-	-	-
	loan	6	6	6	$\surd$	$\times$	$\surd$	$\surd$	-	-	-	-
	auction	9	9	10	$\surd$	$\surd$	$\surd$	$\surd$	767	64	71	543
Collaxa.com	StarLoan	6	7	7	$\surd$	$\surd$	$\surd$	$\surd$	624	80	0.02	357
	auction	5	7	6	$\surd$	$\surd$	$\surd$	$\surd$	462	64	78	254

As shown in the table, four of six examples from IBM, one of three examples from BPEL, and two examples from Collaxa.com satisfy all the realizability conditions. All failed examples do not satisfy autonomous property, mostly due to scenarios where two peers are in a racing to send each other a message. Since neither the BPEL standard nor the IBM conversation project clarify the assumptions about the underlying communication infrastructure, we suspect that these failed examples may be originally designed assuming synchronous communication.

We generated Promela translations for all examples which satisfy realizability conditions, and checked one property for each example. For example, we checked the following LTL property for the SAS protocol, where $index$ and $value$ are two predefined constants.

$$\mathbf{G}((index < \texttt{v\_register.requestlist.stockID\_occc}$$
$$\quad \texttt{\&\& v\_register.requestlist.stockID}[index] == value) \rightarrow \mathbf{F}(\texttt{v\_request.stockID} == value))$$

This property states that if some `stockID` appears in the registration request, then it should eventually appear in the request from Broker to ResearchDept. For all other examples, we checked LTL properties in the form "$\mathbf{G}(p \rightarrow \mathbf{F}q)$". For some of the examples we intentionally checked properties that were not satisfied by the specification. The rightmost four columns of the table present the performance of the SPIN. Since we used the bitstate hashing optimization provided by SPIN, memory consumption for all examples are similar (around 20MB), so we gave the size of the State Vector to indicate the size of the state space. As suggested by the table, verification cost increases with the number of states, and transitions, and especially with the Promela translation size. SAS example has the highest verification cost, because its XPath expressions are the most complicated among all examples. There are some examples with verification time less than one second, this is because SPIN detect the error and ends the exhaustive search early. All the realizability tests were completed in a fraction of a second for all the examples.

In this paper, we studied verification of composite web services in terms of their conversations (i.e. message interactions). The novelty of the model beyond our earlier models is a specification language that uses XML schema and XPath, which brings our model much closer to standards such as BPEL. We obtained sufficient conditions for realizability of conversation protocols and implemented the algorithm to check realizability. We also developed an algorithm to translate conversation specifications to Promela.

Design of composite web services has many challenging problems and developing tools and techniques for studying global behavior of web services is an interesting topic. This paper presents a promising approach to meeting some of the challenges.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *Proc. 28th Int. Colloq. on Automata, Languages, and Programming*, 2001.
2. Business process execution language for web services (BPEL), version 1.1. *available at* http://www.ibm.com/developerworks/library/ws-bpel.
3. A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL a model for W3C XML Schema. In *Proc. of 10th World Wide Web Conference (WWW)*, pages 191–200, 2001.
4. T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proceedings of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 403–410, May 2003.
5. E.M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
6. Experimental data sets. *available at* http://cs.ucsb.edu/~fuxiang/tacas_data/.
7. Christopher Ferris and Joel Farrell. What are web services? *Comm. of the ACM*, 46(6):31–31, June 2003.
8. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering Conference (ASE 2003)*, 2003.
9. 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 (CIAA 2003)*, volume 2759 of *LNCS*, pages 188–200, 2003.
10. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, 2003.
11. S. Narayanan and S. A. Mcllraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th International World Wide Web Conference*, 2002.
12. Simple object access protocol (soap) 1.1. W3C Note 08, May 2000. (http://www.w3.org/TR/SOAP/).
13. W3C. Web services description language (WSDL) version 1.1. *available at* http://www.w3.org/TR/wsdl, 2001.
14. Web Service Choreography Interface (WSCI) Version 1.0. http://www.w3.org/2003/01/wscwg-charter.
15. Extensible markup language (XML). *available at* http://www.w3c.org/XML.
16. XML Path Language version 1.0. *available at* http://www.w3.org/TR/xpath.
17. XML Schema. *available at* http://www.w3c.org/XML/Schema.