# Analysis of Interacting BPEL Web Services


Xiang Fu
fuxiang@cs.ucsb.edu

Tevfik Bultan
bultan@cs.ucsb.edu

Jianwen Su
su@cs.ucsb.edu

Department of Computer Science
University of California
Santa Barbara, CA 93106-5110



## ABSTRACT

This paper presents a set of tools and techniques for analyzing interactions of composite web services which are specified in BPEL and communicate through asynchronous XML messages. We model the interactions of composite web services as conversations, the global sequence of messages exchanged by the web services. As opposed to earlier work, our tool-set handles rich data manipulation via XPath expressions. This allows us to verify designs at a more detailed level and check properties about message content. We present a framework where BPEL specifications of web services are translated to an intermediate representation, followed by the translation of the intermediate representation to a verification language. As an intermediate representation we use guarded automata augmented with unbounded queues for incoming messages, where the guards are expressed as XPath expressions. As the target verification language we use Promela, input language of the model checker SPIN. Since SPIN model checker is a finite-state verification tool we can only achieve partial verification by fixing the sizes of the input queues in the translation. We propose the concept of *synchronizability* to address this problem. We show that if a composite web service is synchronizable, then its conversation set remains 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 input queues can be *completely* verified using a finite state model checker such as SPIN.



## Categories and Subject Descriptors

H.1.1 [**Models and Principles**]: Systems and Information Theory—*(E.4) formal models of communication*; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking*

## General Terms

Verification, Design

## Keywords

Web Service, asynchronous communication, conversation, synchronizability, model checking, XPath, BPEL, SPIN.




*WWW2004*, May 17–20, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.


## 1. INTRODUCTION

A fundamental goal of web services is to have a collection of network-resident software services accessible via standardized protocols, whose functionality can be automatically discovered and integrated into applications or composed to form more complex services. While several established and emerging standards bodies (e.g., [19, 24, 2, 23, 8] etc.) are rapidly laying out the foundations that the industry will build upon, there are many research challenges behind web services that are less well-defined and understood [13]. This paper attempts to address the aspect of "global behaviors" of interacting web services.

At an elementary level, a web service is composed of "activities" whose execution performs tasks of interest, and "messages" that enable the service to participate in a more complex web service. Activities resemble traditional programs; messages are necessary to allow individual web services to interact with each other while maintaining their autonomy. Our objective is to understand the role of messaging in composing web services and to develop tools for analyzing interactions of composite web services.

In our earlier work [5, 11], we introduced a global behavior model for interacting web services based on their "conversations", i.e., the global sequence of messages recorded in the order in which they are sent. Such message-oriented behavior modeling is not only simple, but more importantly, it requires web services to reveal the least amount of information that is necessary to make meaningful compositions. Thus complex internal states (e.g. in legacy systems) can be hidden. Interestingly, conversations immediately permit temporal properties to be expressed on and verified against composite web services.

A top-down specification approach based on *conversation protocols* was proposed in [5, 11]. A conversation protocol is a finite state automaton which specifies the desired set of conversations of a composite web service. The model used in [5, 11] does not have message contents and hence, abstracts away the data semantics. To capture data semantics, in our technical report [10], we developed the notion of a guarded automaton. Each transition of a guarded automaton is equipped with a guard that is expressed using an XPath [26] expression. The use of XPath expressions as guards allows us to express the manipulation of XML message contents in a conversation protocol.

Contrary to the top-down specification approach adopted in [5, 11, 10], this paper studies the bottom-up composition of BPEL [2] web services, and makes the following new contributions:

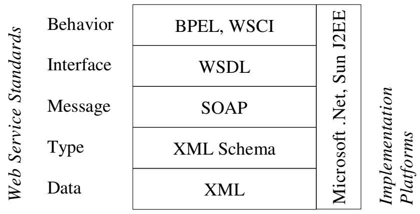



**Figure 1: Web Service Standards Stack**

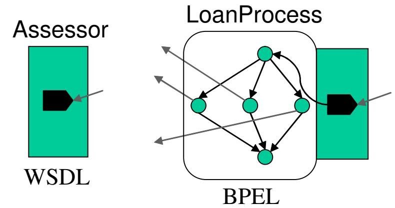



**Figure 2: WSDL Ports and BPEL Web Services**

1. To facilitate the translation of BPEL web services, we extend the guarded automata model in [10] by allowing the use of local XML variables in a guarded automaton. We develop a tool which translates BPEL web services to this extended guarded automata model.

2. We further extend the translation tool developed in [10] from guarded automata to Promela, the input language for the SPIN model checker [12]. The extension handles local variables in guarded automata, and supports asynchronous messaging with input queues. (In [10] a composite web service is specified in a top-down fashion starting with a single guarded automaton and input queues are not handled in the translation.) The combination of the two translation tools allows us to model check properties of BPEL web services without abstracting away their data semantics.

3. We develop sufficient conditions for the equivalence of conversations under synchronous and (the usual) asynchronous communication semantics. We demonstrate that "synchronizable" composite web services allow "complete" and more efficient verifications. We present an improvement to the autonomy condition. With a slight modification this result can also improve the realizability conditions given in [11].

Our results form a framework for developing analysis, verification, and design tools for web services. In particular, the use of the guarded automata model as an intermediate representation for composite web services results in a modular and extensible architecture for our web service analysis tool: multiple web service specification languages can be supported at the front-end, and various model checking tools (as well as the synchronizability analysis introduced in this paper) can be employed at the back-end.

The rest of the paper is organized as follows. Section 2 discusses general notions of web services and interaction models, which provides the context for the technical problems discussed in this paper. Section 3 defines a formal model for the technical development. Section 4 presents two translation algorithms from BPEL to guarded automata, and from guarded automata to Promela, respectively. Section 5 improves the results presented in Section 4 by proposing the notion of synchronizability, and gives sufficient conditions for synchronizability. Finally Section 6 concludes the paper.

## 2. WEB SERVICE INTERACTIONS

Figure 1 displays the stack of standards for web services where XML [25] sits as the foundation. 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. Web services interact with each other using XML messages. XML Schema [27] provides essentially the type system for XML messages. Communication protocols such as SOAP [19] can then be used to transmit XML messages. The interfaces of web services can be described in WSDL [24] which, most importantly, defines the ports that web services can connect to in order to interact with each other. 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 can be defined using higher level standards such as BPEL [2], WSCI [23], BPML [3], DAML-S [8], etc. Web service development based on these standards is supported by different (and competing) implementation platforms such as .Net [21] and J2EE [15].

Consider a loan processing service (similar to the example in [2]) that consists of services for loan processing, risk assessment, etc., and a customer process. For example, the interface of the web service for risk assessment can be defined using WSDL ("Assessor" in Figure 2). The WSDL specification for the risk assessment service defines a port that a loan processing service can connect to. The loan processing service in Figure 2 (right) provides not only a WSDL port for the customer process to connect, but it also specifies the behavior of the loan processing service in BPEL, describing how it interacts with other services including the risk assessment service.

One can use pre- and post-conditions to associate behavioral descriptions for "atomic" web services that are specified in WSDL [17]. These pre- and post-conditions can be used to reason about the composed behaviors of web services [13]. However, BPEL provides more expressive behavioral descriptions. BPEL not only allows manipulation of XML data structures using XPath expressions [26], but also provides programming constructs such as sequence, conditional and case statements, parallelism, and loops.

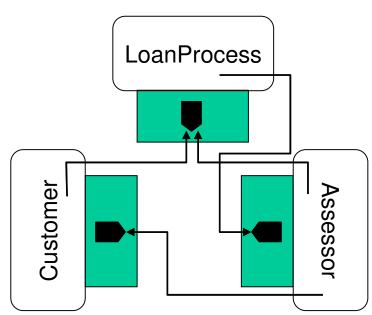



**Figure 3: Interacting BPEL Web Services**

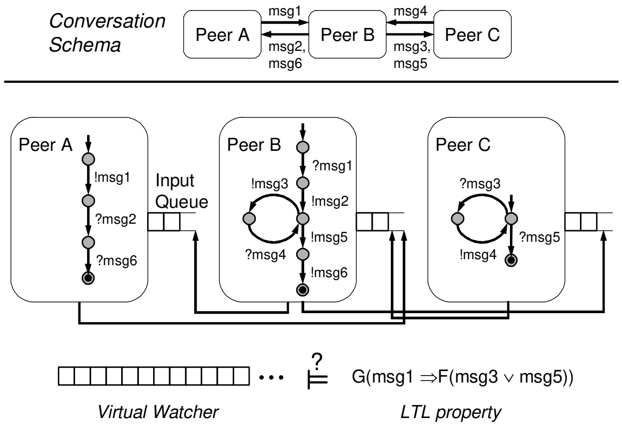



**Figure 4: A Simple Example Demonstrating Our Model**

Figure 3 shows a part of the composite loan processing web service that includes LoanProcess, Assessor, and Customer. The individual services may be specified in BPEL or, in simple cases, WSDL. The directed edges represent the communication links among the individual services. Note that Assessor may contact Customer directly. As a result, there is no single web service that can keep track of the "global" state of the service execution (i.e., there is no mediator process as described in [13]). Clearly, analyzing interactions of such web services presents a great challenge due to their distributed behavior [13].

Our goal in this paper is to analyze and verify properties of composite web services consisting of multiple BPEL web services communicating asynchronously such as the one shown in Figure 3.

# 3. A FORMAL MODEL FOR INTERACTING WEB SERVICES

In this section, we give a formal model for composite web services which consists of multiple peers communicating with asynchronous messaging. Figure 4 gives an informal illustration of our conversation based model. A composite web service consists of a conversation schema that specifies the set of peers and the messages transmitted among peers, and a set of guarded automata specifying the behavior of each individual peer. As communication among web services is asynchronous, each peer is equipped with a FIFO queue to store incoming messages. We assume that there is a *virtual watcher* which records the sequence of messages as they are sent by the peers. The sequence of messages recorded by the watcher is called a conversation. (Note that the virtual watcher is a construct we use to reason about the interactions among different peers and it is not implemented.) 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 [16] in [1].

Formally, a *composite web service* is a tuple $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, where $(P, M)$ is a conversation schema, $n = |P|$ and each $\mathcal{A}_i$ is an implementation for peer $p_i \in P$. A *conversation schema* is a pair $(P, M)$ where $P$ is a finite set of peers and $M$ is a finite set of message types. Each message type $c \in M$ is transmitted between only one pair of peers. In our model, messages are XML documents and types of messages correspond to XML Schemas. For each message type $c \in M$, let $\textsc{dom}(c)$ denote all the XML documents that are of type $c$. 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 $\textsc{type}(m)$.

## 3.1 Guarded Automata Peers

For each peer $p_i$, its implementation $\mathcal{A}_i$ is a guarded automaton $(M_i^{\text{in}}, M_i^{\text{out}}, L_i, T_i, s_i, F_i, \Delta_i)$, where $M_i^{\text{in}}$ ($M_i^{\text{out}}$) are incoming (outgoing) message types for $p_i$, $L_i$ is the set of local variables for $p_i$, and $T_i$, $s_i$, $F_i$ are the set of states, the initial state, and the set of final states, respectively. Like messages, we assume that local variables are also XML documents. For each local variable $l \in L_i$, we use $\textsc{dom}(l)$ to denote all the XML documents that match to the type declaration of $l$. Similar to message types, the types of local variables also correspond to XML Schemas.

Each transition $\tau \in \Delta_i$ of the guarded automaton for the peer $p_i$ 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. *local*-transition, $\tau = (q_1, g, q_2)$, where $g$ is the transition guard. The transition changes the state of the automaton from $q_1$ to $q_2$ and updates the local variables based on the guard $g$.

2. *receive*-transition, $\tau = (q_1, ?a, q_2)$, where $a \in M_i^{\text{in}}$. The transition changes the state of the automaton from $q_1$ to $q_2$ and removes the received message (of type $a$) from the input queue of peer $p_i$.

3. *send*-transition, $\tau = (q_1, (!b, g), q_2)$, where $b \in M_i^{\text{out}}$ and $g$ is the transition guard. The transition changes the state of the automaton from $q_1$ to $q_2$ and appends the sent message (of type $b$) to the input queue of the receiving peer.

Note that, sent messages are not received (consumed) synchronously, rather, they are instantaneously appended to the appropriate input queue. A message is received only after it moves to the head of the input queue.

A guard consists of a guard condition and a set of assignments. A send- or local-transition is taken only if the guard condition evaluates to true, and receive-transitions have no guards because they simply consume a message from the head of the input queue. If the transition is a send-transition, then the assignments of the guard specify the contents of the message that is being sent. If the transition is a local-transition, then the assignments of the guard update the values of the local variables.

Given a send-transition $\tau = (s, (!b, g), t)$ where peer $p_i$ is the sender for the message type $b$, guard $g$ is a predicate of the following form: $g(m, \vec{m} \times \vec{l}\,)$, where $m$ is the message being sent, the vector $\vec{m}$ contains the last instance of each message type that is received or sent by peer $p_i$ (i.e., $M_i^{\text{in}} \cup M_i^{\text{out}}$) and $\vec{l}$ represents the values of the local variables in $L_i$.

Given a local-transition $\tau = (s, g, t)$ for peer $p_i$, guard $g$ is a predicate of the following form: $g(\vec{l}, \vec{m} \times \vec{l}\,)$, where $\vec{m}$ and $\vec{l}$ are as described above, and the first vector denotes the values of the local variables immediately after the transition $\tau$ is executed, and $\vec{m} \times \vec{l}$ denotes the values of the messages and the local variables just before the transition $\tau$ is executed.

## 3.2 Conversations

As explained above, 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 [5].

Formally, given a composite web service $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, a *global configuration* of $\mathcal{S}$ is a $(3n+3)$-tuple of the form $(Q_1, \vec{l_1}, t_1, ..., Q_n, \vec{l_n}, t_n, w, \vec{s}, \vec{r})$ where

- for each $j \in [1..n]$, $Q_j \in \Sigma^*$ is the content of the input queue of peer $p_j$, the vector $\vec{l_j}$ denotes the values of the local variables of $p_j$, $t_j$ is the state of $p_j$, and
- $w \in \Sigma^*$ is the sequence of messages recorded by the global watcher, and
- message vectors $\vec{s}$ and $\vec{r}$ record the latest *sent* and *received* 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 $\tau$ in configuration $\gamma$ results in the configuration $\gamma'$ [11]. 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 $\gamma = \gamma_0, \gamma_1, \gamma_2, ..., \gamma_{|\gamma|-1}$ that satisfies the following conditions: 1) $\gamma_0 = (\epsilon, \vec{\perp}, s_1, ..., \epsilon, \vec{\perp}, s_n, \epsilon, \vec{\perp}, \vec{\perp})$ is the initial configuration, where $s_i$ is the initial state of $p_i$ for each $i \in [1..n]$ and $\vec{\perp}$ denotes uninitialized messages and local variables; 2) for each $0 \leq i < |\gamma| - 1$, $\gamma_i \rightarrow \gamma_{i+1}$; and 3) $\gamma_{|\gamma|-1} = (\epsilon, \vec{l_1}, s'_1, ..., \epsilon, \vec{l_n}, s'_n, w, \vec{s}, \vec{c})$, is a final configuration, where for each peer $p_i$, $s'_i \in F_i$. Note that the input queues for the peers are empty in a final configuration (i.e., every message that has been sent is received).

We call the message sequences observed by the watcher the *conversations* of the composite web service. Formally, a finite word $w \in \Sigma^*$ is a *conversation* of a composite web service $\mathcal{S}$ if there exists a run $\gamma = \gamma_0, \gamma_1, \gamma_2, ..., \gamma_{|\gamma|-1}$ of $\mathcal{S}$ such that, the value of the watcher in the final configuration $\gamma_{|\gamma|-1}$ is $w$. Let $\mathcal{L}(\mathcal{S})$ denote the set of conversations of $\mathcal{S}$.

We use the temporal logic LTL [18] to express properties of conversations [11]. We define the set of atomic propositions as follows: Each atomic proposition is either of the form $c$ where $c$ is a message type (i.e., $c \in M$), or *c.pred*, where $c \in M$ and *pred* is a predicate over the contents of $c$. We denote that a message $m \in \Sigma$ *satisfies* an atomic proposition $\psi$ by $m \models \psi$, where

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

LTL formulas are constructed from atomic propositions, logical operators $\wedge, \vee$, and $\neg$, and temporal operators $\mathbf{X}$ (meaning "next"), $\mathbf{G}$ ("globally"), $\mathbf{U}$ ("until"), and $\mathbf{F}$ ("eventually"). The semantics of LTL temporal operators can be easily defined on finite length conversations. Given a composite web service $\mathcal{S}$ and an LTL property $\phi$, we say that $\mathcal{S} \models \phi$, iff for all $w \in \mathcal{L}(\mathcal{S})$, $w \models \phi$.

For example, the LTL property

$$\mathbf{G}(msg_1 \Rightarrow \mathbf{F}(msg_3 \vee msg_5))$$

shown in Figure 4 denotes that every $msg_1$ will be eventually followed by a $msg_3$ or $msg_5$. The composite web service in Figure 4 satisfies this property since all the conversations generated by this composite web service satisfy the property.

We say that a composite web service $\mathcal{S}$ over a schema $(P, M)$ has *finite content* if for each message type $c \in M$, $\text{DOM}(c)$ is a finite set and for each local variable $l \in L_i$ for each $i \in [1..n]$, $\text{DOM}(l)$ is a finite set. We have the following undecidability result [11]:

> Given a composite web service $\mathcal{S}$ with finite content and an LTL property $\phi$, checking $\mathcal{S} \models \phi$ is undecidable.

Hence, even for protocols with finite content, verification of composite web services is an undecidable problem. This is due to presence of unbounded queues used for asynchronous communication. In the next section we will show that, if we put a bound on the sizes of the input queues we can use existing model checking tools to analyze properties of composite web services.

# 4. MODEL CHECKING INTERACTING BPEL WEB SERVICES

In this section we discuss the use of the SPIN model checker [12] for verification of interacting BPEL web services. The input language of SPIN is called Promela, a modeling language for finite-state concurrent processes. SPIN model checker verifies (or falsifies, by generating counter-examples) LTL properties of Promela specifications using an exhaustive state space search [12]. Given a set of interacting web services specified as BPEL processes with WSDL ports, we generate a Promela specification which corresponds to the composite web service. The Promela specification contains a set of concurrent processes which communicate via asynchronous channels. We implement the translation in two phases: (1) from BPEL processes to guarded automata model described in Section 3, and (2) from the guarded automata model to Promela processes with asynchronous communication channels. The guards in the intermediate guarded automata model are XPath expressions manipulating the XML documents with XML Schema types.

This two step translation has several advantages: (a) The intermediate guarded automata model enables us to implement static analysis techniques such as synchronizability analysis described in the next section; (b) We are able to use our translator from guarded automata to Promela described in [10] in implementing the back-end translation; (c) Decoupling the front- and back-ends of the translator will enable us to target multiple web service languages at the front-end (e.g., WSCI [23], DAML-S [8]) and multiple verification languages (e.g., SMV [7], Action Language Verifier [6]) at the back-end in the future.

## 4.1 Translation from BPEL to Automata

Given a set of BPEL process specifications and the related WSDL port declarations, we can construct a corresponding composite web service specified using guarded automata.

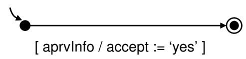
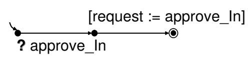
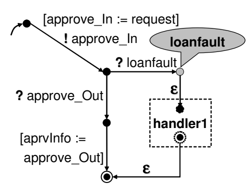
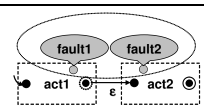
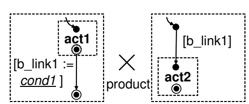
| BPEL | Sample Code | Translation |
|---|---|---|
| assign | <assign ...><br><copy><br><from="yes"/><br><to var="aprvInfo"<br>part="accept"/><br></copy><br></assign> |  |
| receive | <receive ...<br>operation="approve"<br>variable="request"<br>/> |  |
| invoke | <scope><br><invoke ...<br>operation="approve"<br>invar="request"<br>outvar="aprvInfo /><br><catch ...<br>faultname="loanfault"<br>< ... handler1 ... /><br></catch><br></scope> |  |
| sequence | <sequence ... ><br>< ... act1 ... ><br>< ... act2 ... ><br></sequence> |  |
| flow | <flow ...><br>< ... act1 ...><br><source<br>linkname="link1"<br>condition="cond1"/><br></act1><br>< ... act2 ...><br><target<br>linkname="link1/><br></act2><br></flow> |  |

**Figure 5: From BPEL to Guarded Automata**

We first construct the conversation schema, and then translate the control flow of each BPEL process. As BPEL process specifications are fed as input, the peer list $P$ of the schema tuple $(P, M)$ is already known. Message types $M$ are extracted from WSDL files. For each input/output/fault parameter of an operation in each port and each service link of each BPEL process, a message type is declared. For example if a BPEL process `loanProcess` has an operation `approve` declared in a port `aprvPT`, and its input parameter is of WSDL message format `creditInfo`, a message type `loanProcess_aprvPT_approve_In` will be declared in the composition schema, and `creditInfo` is used as its domain type. When the name of an operation is unique among ports, our tool will omit the port name in the generated message name for simplicity (e.g. the `approve_Out` in Figure 5). In BPEL, the type of the contents of a message can be defined using WSDL message declaration constructs, or SOAP definition or XML Schema, we translate all possible formats to MSL [4], a formal model for XML Schema.

Next we discuss the translation of BPEL control flow and data manipulation. In Figure 5 we present the guarded automata translation for some typical language constructs in BPEL language. As shown in the figure, each BPEL language construct is translated into a guarded automaton with one single initial state and one single final state. For example, the `assign` statement is translated to a one-transition automaton where the XPath expression guard of the transition expresses the assignment. Note that BPEL has several different approaches for navigating messages (e.g. the keyword `part` used in the example or using XPath expressions). We translate all of them to equivalent XPath expressions, and these XPath expressions are then embedded into the guards of the generated transitions. The `receive` statement is translated into a two-transition automaton, where the first transition receives the message and the second transition assigns the input variable. Similarly, the main body of the `invoke` statement is translated to an automaton where the first transition sends the input message for the operation that is being invoked, and the following two transitions receive the response and assign the output variable (assuming there are no exceptions). Note that, exceptions might arise during `invoke`, and we have to generate additional transitions to handle them. For each fault there is a transition which leads to an "exception exit", where the information about the fault is associated with the exception exit. When a fault handler is wrapped around an `invoke` statement, our translator connects the fault handler with the corresponding exception exit.

BPEL control flow constructs such as `sequence`, `switch`, and `while` are used to compose atomic constructs we discussed above. In Figure 5 we display the translation for `sequence`. We connect the final state and initial state with local transitions, and unmark the final state of all activities except the last one. The information about exception exits are recollected and properly maintained. Other control flow constructs can be handled similarly by embedding the control flow to the transitions of the guarded automata. Finally, for `flow` construct (which is the concurrent composition of its branches), we simply construct the Cartesian product of all its branches. There might be control dependency links among the activities in different flow branches. We map each link into a boolean variable, and their semantics are reflected in the guards of the transitions appended to each activity.

Translation of the control flow of BPEL to finite state machines or petri-nets has been discussed in [9, 17]. The difference in our work is that we handle XML based data manipulation using guarded automata with guards expressed as XPath expressions. This enables us to verify properties about XML data manipulation. Such analysis cannot be done using approaches presented in [9, 17] since they abstract away the data content.

## 4.2 Translation from Automata to Promela

Given a composite web service specified using interacting guarded automata, we translate it into a Promela specification which consists of a set of concurrent processes, one for each guarded automaton. Each concurrent process is associated with an asynchronous communication channel storing its input messages.

An example Promela output that is generated by our translator is shown in Figure 6. The first part of the Promela code consists of type declarations and global variable definitions. Each MSL type declaration used in conversation schema is mapped into a record type (`typedef`) in Promela. As in our guarded automata model, strings are used as constants only, they are mapped to `mtype`, the enumerated type in Promela (e.g. the element `name` in `creditInfo` is originally a string). As shown in Figure 6, each message type in a conversation schema has three corresponding global vari-

```
/* type declaration */
typedef creditInfo{
 mtype name; ...
}
...
/* message declaration */
creditInfo aprv_In_s, aprv_In_r, stub_aprv_In;
...
/* enumerate type of msgs and states of peers*/
mtype = {m_aprv_In, ....
         m_loanaprv_s1, ... }
mtype msg;
...
/* channels */
chan ch_loanaprv = [8] of {mtype, creditInfo, appeal};
chan ch_customer= [8] of {mtype, aprvInfo};
...
proctype loanaprv(){
 mtype state;
 /* definition of local variables */
 creditInfo request; ...
 /* definition of auxiliary variables used
    to evaluate XPath expressions */
 bool bVar_0, ...

 do::
  /* evaluation of transition conditions */
  ... bCond1 = true; ...

  /* nondeterministically select transitions to fire */
  if
   /* transition t1: s1 -> s2, ?aprv_In */
   ::state == m_loanaprv_s1 && bCond1 &&
     ch_loanaprv ? [m_aprv_In] ->
      atomic{
       ch_loanaprv ?
          m_aprv_In, aprv_In_r, stub_appeal;
       state = m_loanaprv_s2 ;
      }

   /* transition t2: s2 -> s3, !aprv_Out,
     [cond2 => aprv_Out//accept = 'yes' ] */
   ::state == m_loanaprv_s2 && bCond2 ->
     atomic{
      aprv_Out_s.accept = m_yes;
      ch_customer ! m_aprv_Out, aprv_Out_s;
      state = m_loanaprv_s3;
      msg = m_aprv_Out
     }
   ...
   /* may jump out if it is a final state */
   :: state == m_final -> break;
  fi;
 od;
}
proctype customer(){ ... }
proctype assessor(){ ... }
proctype approver(){ ... }
init{
 /* initialization */
 ...
 atomic{
  run loanaprv(); run customer(); ...
 }
}
```

**Figure 6: An Example Promela Translation**

ables declared: one for recording its last sent instance (e.g. `aprv_In_s` for message type `aprv_In`), one for recording its last received instance(e.g. `aprv_In_r`), and one "stub" variable used in channel operations (e.g. `stub_aprv_In`). For each message type, we also declare a corresponding enumerated constant, e.g., `m_aprv_In` for `aprv_In`. The set of all these enumerated constants constitutes the domain of enumerated variable `msg`, which is used to store the type of the latest transmitted message.

A channel variable is declared for each peer to simulate its input queue. For example channel `ch_loanaprv` is the queue of peer `loanaprv` and its length is 8. The contents of a channel includes all input message types of that peer. In this example, peer `loanaprv` has two input message types: `aprv_In` and `appeal`. Note that in each send/receive operation of a channel, we actually send one message only, and other elements have to be filled with stub messages. The first `mtype` element in a channel content indicates the message type that is being transmitted.

In the Promela code, each automaton is translated to a process type (`proctype`). In the example shown in Figure 6, we have four process types `loanaprv`, `customer`, `approver` and `assessor`. The default main process in Promela is called `init`. The `init` process in Figure 6 initializes all global variables (initialization can be non-deterministic) and spawns four processes, creating one process instance for each process type.

Inside each `proctype` the local variables are declared first, followed by the auxiliary variables used for the evaluation of XPath expressions. An enumerated (`mtype`) variable `state` is used to record the current state of the automaton. The main body of the process is a single loop. In each iteration of the loop, first enabling condition of each transition guard is evaluated and the result is stored in the corresponding boolean variable for that condition. For example, the `cond1` in Figure 6 records the evaluation results for the enabling condition of transition `t1`.

In Promela, `if` statements can have multiple branches with a test condition for each branch, similar to a switch statement. One of the branches of the `if` statement with a test condition that evaluates to true is nondeterministically chosen and executed. In the Promela translation for a guarded automaton, each transition of the automaton is translated into a branch of the `if` statement inside the main `do` loop body. The test condition for each branch checks whether the current `state` is the source state of the corresponding transition, and whether the enabling condition of the corresponding transition evaluates to true. For receive-transitions, we check if the head of the channel contains the right message type by testing the first element of the channel content. (Note that Promela statement `channel ? messages` has side effects and cannot be used as a boolean condition, hence we have to use `channel ? [...]` statement, which checks the receive executability only but does not execute the receive operation.) If the head of the channel matches the message type of the receive operation, we consume the message, do the assignment, and update the local variable `state`. The handling of send-transitions is similar, and the only difference is that we need to update global variable `msg` while sending the message. Finally, if the state is a final state, a nondeterministic choice can be made to jump out of the loop and terminate.

The translation of XPath expression to Promela is not a trivial problem. For example, consider the `aprv_In` message and suppose that it has an element `preferred_term` which is a list of desired (APR/termLength) pair proposed by the customer, and the maximal length of the list is 10. Then the XPath expression

```
aprv_In / preferred_term / termLength == 6
```

is translated into the following code:

```
bool bResult = false;
int i=0;
do
:: i<10 ->
   if
   :: aprv.preferred_term[i].termLength == 6
      -> bResult = true
   :: else -> skip;
   fi;
:: else -> break;
od;
```

The XML Schema element `preferred_term` is translated into an array type in Promela. The key step in the generated code above is to search for the proper array index which satisfies the condition evaluation. When XPath functions such as `last()` and `position()` need to be handled, the translation becomes more complicated. The translation of XPath expressions to Promela code is discussed in [10] and we will omit the details here.

## 4.3 A Case Study

Using the translation tools described above, we translated the "Loan Processing Example" in the BPEL document [2] to Promela, and used the SPIN model checker to verify it. The source code (BPEL and WSDL files) of the example is taken from the IBM BPWS4J tool sample set [14]. Since no BPEL files are provided for the other three peers (customer, loan assessor, and loan approver) involved in the interaction, we supplied their BPEL implementations.

The control logic of the loan approval process is as follows: If the loan amount in the request sent by the customer is lower than a certain amount, and if the loan assessor gives a "low-risk" assessment, the loan approval process approves the loan request; otherwise it will wait for the approver to make the final decision. The main body of the loan approval process is a `flow` construct, where its branches consist of `invoke`, `receive`, `reply`, `assign` statements. There are six dependence links among these concurrently running branches, and an exception handler is wrapped around the `flow` construct.

We considered the following two properties of the interactions. The first is satisfied by the composite web service, while the second is not.

$$\mathbf{G}(\texttt{LA\_aprv\_In\_s} \rightarrow \mathbf{F}(\texttt{LA\_aprv\_Out\_s} \vee \texttt{LA\_aprv\_Fault}))$$
$$\mathbf{G}(\texttt{LA\_aprv\_In\_s.amount>2} \rightarrow \mathbf{F}\ \texttt{LA\_aprv\_Out\_s.accept!= m\_yes})$$

Here `LA_aprv_*` are messages related to the `approve` operation of the loan approval process. The first property states that when the customer sends a request to the loan approval process, eventually it will either get an output message or a fault. The second property states that if the request amount is greater than 2 (since our domain is 4 here), it will eventually get rejected. (This property is false because the loan approval process passes the decision to the loan approver, and loan approver can still approve the request.)

Our tool translates the four BPEL files and corresponding WSDL files into one Promela specification. In the generated Promela code there are seven integer variables, which leads to a large state space. To limit the state space, we set the integer domain to 4, and change the condition of request amount in the specification accordingly.

The first property is verified in about 2 minutes with 980 thousand states explored. We tried different channel sizes from 1 to 10, and the verification cost changes very little. But if we increase the integer domain, the verification cost increases exponentially and uses up the memory resource.

For the second property, SPIN identifies the error very quickly. When integer domain is 4, it only takes 0.2 seconds to find and generate the error trace. The verification cost does not increase with the integer domain, which should be attributed to the depth-first search approach used in SPIN to locate error (however for the correct property we have to exhaust the whole state space). The cost also does not increase with channel sizes.

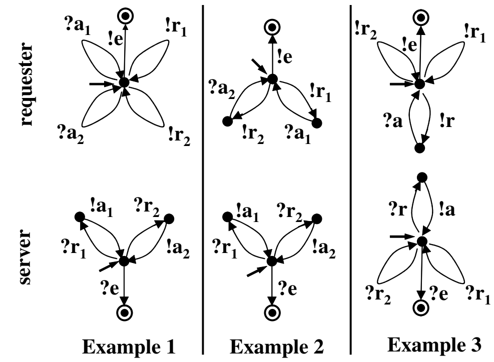



**Figure 7: Three Composite Web Service Examples**

# 5. SYNCHRONIZABILITY

Consider the three example composite web services given in Figure 7. Each composite web service consists of two peers: a requester and a server. For each "request" message (represented as $r_i$) sent by the requester, the server will respond with a corresponding "acknowledgment" ($a_i$). However this response may not be immediate (e.g. in Example 1). Finally the "end" message ($e$) concludes the interaction between the requester and the server.

We can verify properties of these examples by translating them to Promela as described in the previous section. However, as discussed above, we need to bound the sizes of the input queues (communication channels in Promela) to be able to verify a composite web service using SPIN, since it is a finite state model checker. In fact, based on the undecidability of LTL verification (Section 3), it is generally impossible to verify the behavior of a composite web service with unbounded queues. In general, best we can do is *partial* verification, i.e., to verify behavior of a composite web service for queues with a fixed length. Note that the absence of errors using such an approach does not guarantee that the composite web service is correct. Interestingly, in this section we will show that, Examples 2 and 3 are different from Example 1 in Figure 7 in that the properties of Examples 2 and 3 can in fact be verified for unbounded message queues, whereas for Example 1 we can only achieve partial verification.

First, note that in Example 1 the requester can send an arbitrary number of messages before the server starts consuming them. Hence the conversation set of Example 1 is not a regular set [5]. 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 [5]. It is not surprising that we cannot map the behavior of Example 1 to a finite state process. Another problem with Example 1 is the fact that its state space increases exponentially with the sizes of the input queues. Hence, even partial verification for large queue sizes becomes intractable.

In Example 2 the requester and server processes move in a lock-step fashion, and it is easy to see that the conversations generated by Example 2 is $(r_1a_1 \mid r_2a_2)^*e$, i.e., a regular set. In fact, the composite web service described in Example 2 has a finite set of reachable states. During any execution of Example 2 at any state, there is at most one message in each queue. Based on the results we will present below, we can statically conclude that properties of Example 2 can be verified using synchronous communication (in other words, using input queues of size 0).

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

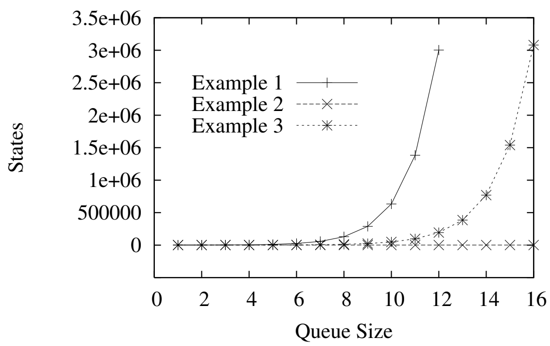



**Figure 8: State Space and Queue Size**

We can experimentally demonstrate how state spaces of the examples in Figure 7 change with the increasing queue sizes. In Figure 8 we present the size of the reachable state space for the examples in Figure 7 computed using the SPIN model checker for different input queue sizes. The $x$-axis of the figure is the size of the input queues, and $y$-axis displays the number of reachable states computed by SPIN. As shown in the figure, the state space of Example 2 is fixed (always 43 states), however the state spaces of Examples 1 and 3 increase exponentially with the queue size. Below we will show that we can verify behaviors of Examples 2 and 3 for arbitrary queue sizes, although best we can do for Example 1 is partial verification. In particular, we will show that the communication among peers for Examples 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.

## 5.1 Synchronous Communication

To further explore the differences of Examples 2 and 3 from Example 1, we define an alternative "synchronous" semantics for composite web services different than the one in Section 3. Intuitively, the synchronous semantics restricts that each peer consumes its incoming messages immediately. Therefore, there is no need to have the input message queue.

Recall that a composite web service $\mathcal{S}$ is a tuple $\mathcal{S} = \langle (P, M), \mathcal{A}_1, ..., \mathcal{A}_n \rangle$ where each guarded automaton $\mathcal{A}_i$ describes the behavior of a peer. In a global configuration $(Q_1, \vec{l_1}, t_1, ..., Q_n, \vec{l_n}, t_n, w, \vec{s}, \vec{r})$ of $\mathcal{S}$, $Q_j$'s ($j \in [1..n]$) are the configurations of the input queues. We now define a *configuration* of a composite web service *with the synchronous communication semantics*, or *sc-configuration*, as a tuple $(\vec{l_1}, t_1, ..., \vec{l_n}, t_n, w, \vec{s}, \vec{r})$, which differs from a configuration by dropping all input queues.

When peers interact with each other through asynchronous communication, a send operation inserts a message to the input queue of the target peer and a receive operation removes the message at the head of the input queue. The definition of the *derivation* relation between two sc-configurations is modified from the asynchronous case so that a send transition can only be executed instantaneously with a matching receive operation, i.e., sending and receiving of a message occur synchronously. We call this semantics the *synchronous communication semantics* for a composite web service.

The definitions of the watcher and the conversation set are modified accordingly. In particular, given a composite web service $\mathcal{S}$, let $\mathcal{L}_{\rm syn}(\mathcal{S})$ denote the conversation set under the synchronous communication semantics. A composite web service is *synchronizable* if its conversation set remains the same when the synchronous communication semantics is used, i.e., $\mathcal{L}(\mathcal{S}) = \mathcal{L}_{\rm syn}(\mathcal{S})$.

Clearly, if a composite web service is synchronizable, then we can verify its behavior without any input queues and the results of the verification will hold for the behaviors of the composite web service in the presence of asynchronous communication with unbounded queues. In Section 5.2 we will give sufficient conditions for synchronizability. Based on these conditions, we can show that Examples 2 and 3 in Figure 7 are indeed synchronizable.

## 5.2 Synchronizability Analysis

For a guarded automaton $\mathcal{A} = (M^{\rm in}, M^{\rm out}, L, T, s, F, \Delta)$, its *skeleton* is a standard (i.e. guardless) finite state automaton $sk(\mathcal{A}) = (M^{\rm in}, M^{\rm out}, T, s,\ F, \Delta')$ where all local variables are removed and each transition $\tau' \in \Delta'$ is generated by dropping the guard of a corresponding transition $\tau$ in $\Delta$. (The local-transitions of the guarded automata become $\epsilon$-transitions in the skeleton.) Note that the language recognized by the skeleton $sk(\mathcal{A})$, i.e. $L(sk(\mathcal{A}))$, is a subset of $M^*$, while $L(\mathcal{A}) \subseteq \Sigma^*$.

Given a set of guarded automata, their composition is synchronizable if the following three conditions are satisfied by their skeletons:

**1) Synchronous compatibility:** If we construct the synchronous composition (i.e., Cartesian product) of the skeletons, 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:** For each peer $p_i$ and each state $s_j$ of $p_i$, at most one of the following three conditions hold: (a) the next transitions at $s_j$ (including transitions that are reachable through $\epsilon$-transitions) are all send operations, (b) the next transitions at $s_j$ (including transitions that are reachable through $\epsilon$-transitions) are all receive operations, or (c) $s_j$ is either a final state or it can reach a final state through $\epsilon$-transitions.

**3) Lossless composition:** Mark the initial state of each peer skeleton a final state, and construct the Cartesian product of the peer skeletons. The projection of the Cartesian product to each peer should be equivalent to the original skeleton (with the initial state marked as a final state).

THEOREM 1. Let $\mathcal{S} = \langle (P, M), \mathcal{A}_1, ..., \mathcal{A}_n \rangle$ be a composite web service where their skeletons $sk(\mathcal{A}_1), ..., sk(\mathcal{A}_n)$ satisfy the above three synchronizability conditions, then $\mathcal{S}$ is synchronizable.

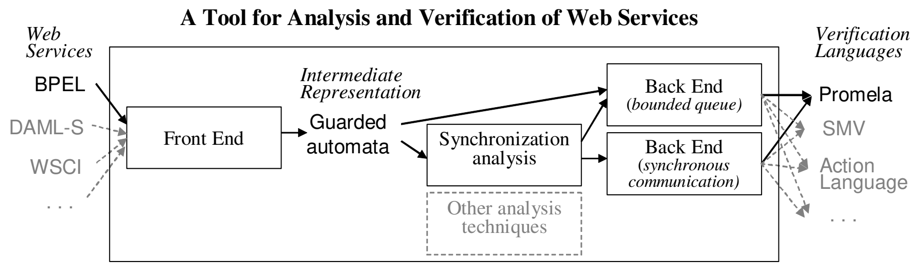



**Figure 9: Our Web Service Analysis and Verification Tool**

The synchronizability conditions listed above are derived from the realizability conditions for conversation protocols that we presented in [11]. A proof along the lines of the proof for realizability conditions given in [11] can be used to establish Theorem 1. Intuitively, we can show that, when a composite web service satisfies the above three conditions, the input queue of each peer is empty whenever the peer sends out a message. Based on this observation, we can show that for each conversation there exists a corresponding execution in which the peers communicate synchronously.

Note that both Examples 2 and 3 in Figure 7 are synchronizable whereas Example 1 is not (it violates the autonomy condition). Hence, we can verify the properties of Examples 2 and 3 using synchronous communication (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.

**Relaxing the Autonomy Condition.** As we discussed in Section 4, the `flow` construct in BPEL specification generates the Cartesian product of its flow branches when it is translated to the guarded automata. Unfortunately, such `flow` constructs are likely to violate the autonomy condition given above. For example, assume that there are two branches inside a flow statement, and each branch is a single `invoke` operation which first sends a request and then receives response. In the guarded automaton translation, there will be a state with one transition for sending out the request for one of the branches and another transition for receiving the response for the other branch. Note that such a state violates the autonomy condition. However, even the corresponding peer sends out a message while its input queue is not empty, since the Cartesian product of the flow branches includes all the permutations of the transitions in different branches we can show that there is an equivalent computation where the send operation is executed when the queue is empty, after the receive operation. We can generalize this scenario and we can relax the autonomy condition to single-entry single-exit permutation blocks. A permutation block has no cycles and no final states and contains all the permutations of the transitions from its entry to its exit. Then, we relax the autonomy condition by stating that all the states in a permutation block (including the entry but excluding the exit) satisfy the autonomy condition.

**Loan Processing Service Revisited.** The "Loan Processing Example" discussed in Section 4 passes our synchronizability condition test, and we can safely verify the two LTL properties on its synchronous composition by setting the channel size to 0 in its Promela translation. The verification cost of channel size 0 is not significantly lower than that of the larger channel sizes. However even when the synchronous verification with synchronizability analysis does not reduce the state space much, it is still better than the partial verification because we are ensured that the verified properties work for asynchronous composition with unbounded queue size, while the partial verification can not. More examples of realizability/synchronizability analysis are available on our tool site [22].

There are other ways to achieve decidability in analyzing queued or buffered systems, for example, the realizability analysis of MSC graphs [1], and the use of *sound* WF-net (a variation of bounded petri-net) to analyze workflows [20]. Both in [1] and [20], the main idea is to provide sufficient conditions which guarantee that the queue (or buffer) sizes are bounded for any execution of the system. However, the synchronizability approach used in this paper is different. For example, in Example 3 of Figure 7, queue size of the server is not bounded during its composition with the requester but it is still synchronizable.

# 6. CONCLUSIONS AND FUTURE WORK

Our contributions in this paper and [10] together form a system for analyzing and verifying web services, as shown in Figure 9. Specifically, the input to our tool is a composite web service specification and some LTL properties. Our tool can be used to check if the composite web service satisfies the LTL properties. Currently, our tool accepts BPEL specifications as input and translates them to guarded automata in our model. The guarded automata can be either translated directly to Promela with bounded queues or synchronizability analysis can be used. Depending on the result of the synchronizability check, the system generates a Promela specification either with synchronous communication (if the check succeeds) or with bounded queues (otherwise). The SPIN model checker can then be invoked to do the verification on the Promela specification.

Clearly, the system can be extended in the future, as illustrated in Figure 9. First, translation from other web service specification languages (such as DAML-S, WSCI, etc.) can be added to the front-end. This immediately allows verification of web services defined in these languages without changing the back-end. Second, different verification tools can be targeted at the back-end either to make verification more efficient or to have more expressive verification languages. Finally, it is possible to add additional analysis techniques, such as automated abstraction, to the tool. Hence our guarded automata model provides a flexible intermediate representation that can be used for different analysis.

There also exist other challenges for applying model checking techniques to web services concerning extending the current model of guarded automata. In our current model, we assume that service links (channels) among peers are "predetermined" and established prior to the interaction starts. Hence some advanced features in BPEL, for example, the endpoint references (to dynamically determine the peer to talk to), cannot be captured in the current model. Also our model does not handle dynamic process instantiation and correlation sets. Extending our model to address these issues is an interesting direction for future research.

## Acknowledgments

Bultan was supported in part by NSF Career award CCR-9984822; Fu was partially supported by NSF grant IIS-0101134 and NSF Career award CCR-9984822; Su was also supported in part by NSF grants IIS-0101134 and IIS-9817432.

## 7. REFERENCES

[1] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *Proceedings of 28th International Colloquium on Automata, Languages, and Programming*, volume 2076 of *LNCS*, pages 797 – 808. Springer, 2001.

[2] Business Process Execution Language for Web Services (BPEL), Version 1.1. `http://www.ibm.com/developerworks/library/ws-bpel`, May 2003.

[3] Business Process Modeling Language (BPML). `http://www.bpmi.org`.

[4] A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL a model for W3C XML Schema. In *Proceedings of 10th World Wide Web Conference (WWW)*, pages 191–200, 2001.

[5] 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 12th World Wide Web Conference (WWW)*, pages 403 – 410, May 2003.

[6] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, pages 382–386, 2001.

[7] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In *IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[8] DAML-S (and OWL-S) 0.9 Draft Release. `http://www.daml.org/services/daml-s/0.9/`, May 2003.

[9] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings the 18th IEEE International Conference on Automated Software Engineering Conference (ASE 2003)*, 2003.

[10] X. Fu, T. Bultan, and J. Su. Model checking interactions of composite web services. UCSB Computer Science Department Technical Report (2004-05). (Available at `http://www.cs.ucsb.edu/~su/tmp/Map2SPIN.pdf`).

[11] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proceedings of 8th International Conference on Implementation and Application of Automata (CIAA)*, volume 2759 of *LNCS*, pages 188–200, 2003. Full version to appear in a special issue of *Theoretical Computer Science*.

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

[13] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: A look behind the curtain. In *Proceedings of 22nd ACM Symposium on Principles of Database Systems (PODS)*, pages 1 – 14, 2003.

[14] IBM. IBM Business Process Execution Language for Web Service Java Run Time (BPWS4J). `http://www.alphaworks.ibm.com/tech/bpws4j`.

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

[16] Message Sequence Chart (MSC). ITU-T, Geneva Recommendation Z.120, 1994.

[17] 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.

[18] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.

[19] Simple Object Access Protocol (SOAP) 1.1. W3C Note 08, May 2000. `http://www.w3.org/TR/SOAP/`.

[20] W. M. P. van der Aalst and et al. Verification of workflow task structures: A petri-net-based approach. *Information Systems*, 25(1):43–69, 2000.

[21] Joseph Williams. The web services debate J2EE vs. .Net. *Communications of the ACM*, 46(6):59–63, June 2003.

[22] Web Service Analysis Tool (WSAT). `http://www.cs.ucsb.edu/~su/WSAT`.

[23] Web Service Choreography Interface (WSCI) 1.0. `http://www.w3.org/TR/2002/NOTE-wsci-20020808/`, August 2002.

[24] Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.

[25] Extensible Markup Language (XML). `http://www.w3c.org/XML`.

[26] XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`, 1999.

[27] XML Schema. `http://www.w3c.org/XML/Schema`.