Verifiable Web Services with Hierarchical Interfaces *

Aysu Betin-Can Tevfik Bultan
Department of Computer Science,
University of California, Santa Barbara, CA 93106
{aysu,bultan}@cs.ucsb.edu

Abstract

We propose an Hierarchical State Machine (HSM) model for specifying behavioral interfaces of peers participating in a composite web service. We integrate the HSM model to a design pattern which is supported by a modular verification technique that can 1) statically analyze the properties about global interactions of a composite web service and 2) check the conformance of the Java implementations of the participant peers to their interfaces. We extend the synchronizability analysis to HSMs to efficiently identify composite web services whose global interactions can be analyzed with respect to unbounded queues using finite state model checkers. We also discuss automated translation of behavioral interfaces specified as HSMs to BPEL specifications to be published and used by other services.

1 Introduction

Web services provide a framework for decoupling the interfaces of Internet accessible applications from their implementations. This framework facilitates interoperability and integration and enables development of composite services. Use of web services in critical business applications makes developing reliable composite web services and analyzing their interactions extremely important.

A composite web service consists of a collection of individual web services, called *peers*, working in a collaborative manner. In our work, we assume that the interaction among peers is established through *asynchronous* messages. In asynchronous communication, when a message is sent, it is inserted into a FIFO message queue, and the receiver consumes (i.e., receives) the message when the message reaches to the front of the queue. Asynchronous communication is necessary to prevent peers getting stuck during a send operation due to pauses in availability of the receiving peer or slow data transmission through the Internet. The interaction among the peers in a composite web service can be modeled as a *conversation*, the global sequence of messages that are exchanged among the peers [7, 11, 15].

To achieve interoperability among web services it is necessary to have a contractual agreement among the participating peers. The WSDL [26] standard is commonly used as a contract for specifying the operations and the port and message types of an individual web service. This kind of information, however, is not sufficient for developing composite services. WSDL is a connectivity contract which does not model the behavior [19, 23]. A number of standards have been proposed for describing the behavior of a web service, such as BPEL [5] and WSCI [25]. In [7], finite state machines are used for specifying behavioral interfaces and in [9] it is shown that other behavioral descriptions (such as BPEL) can be translated to finite state machines. Finite state machines are powerful enough to specify behavioral interfaces of typical web services and they are suitable for automated reasoning. However, due to explicit and flat representation of the states, behavioral interfaces represented as finite state machines may contain a large number of states and may be hard to understand since they lack high-level structure. In this paper, we propose using hierarchical state machines to specify the behavioral interfaces of peers participating in a composite web service.

A hierarchical state machine (HSM) is a finite state machine whose states can be composite or atomic states. HSMs are based on statecharts [12]; however, instead of the events used in statecharts, in our model, the transitions of HSMs are triggered by sending or receiving of asynchronous messages. The advantage of using HSMs over finite state machines is that HSMs can define a web service in a more compact way and represent the natural hierarchy of the service behavior. An HSM can have exponentially less number of states and transitions than a corresponding flat finite state machine. Moreover, with HSMs, we can specify concurrent executions of operations which are common in web service applications.

State machines are used as behavioral contracts also in [10] and [2]. Unlike our work, their goal is automatic web service composition. In [1] and [18] statecharts are used to describe service behavior, specifically to declare a service composition. Unlike these studies, we use HSMs not

*This work is supported by the NSF grant CCR-0341365.

only for specifying web services but also for verifying their global interaction properties.

In this paper, we use the design for verification approach proposed in [3, 4] to aid the development of reliable composite web services in Java. We use a design pattern that decouples operations related to the application logic from the communication details [4]. This pattern, called the Peer Controller Pattern, requires explicit specification of the peer interfaces during implementation. The peer interfaces are the behavioral contracts that the peer implementations have to obey.

A peer interface provides a high-level description of the peer behavior and as such it can be used as an abstraction for the verification purposes. Based on the peer interfaces and the proposed design pattern, we use a modular verification technique to statically check both properties of peer interactions and the conformance of the Java implementations of the peers to their interfaces [4]. In addition to their use during static verification, peer interfaces can also be used as a monitoring tool at run-time. In this paper we show that HSMs can be used to specify the peer interfaces and we extend the modular verification approach described in [4] to HSMs. Furthermore, we automatically translate the peer interfaces to BPEL specifications. The behavioral interface specification of a peer can be published in BPEL and used by other services that interact with that peer.

Our contributions in this paper can be summarized as follows: 1) we present an HSM model for specifying peer interfaces and we integrate this HSM model to a design pattern for developing reliable composite web services in Java, 2) we provide a tool for automated generation of BPEL specifications from the peer interfaces specified as HSMs, and 3) we extend the synchronizability analysis introduced in [9] to the synchronizability of HSMs. A composite web service is called synchronizable if its conversation set does not change when asynchronous communication is replaced with synchronous communication. Since the automated verification of asynchronously communicating finite state machines is an undecidable problem, the synchronizability analysis enables us to identify composite web services which *can* be verified using the synchronous communication semantics. The automated synchronizability check enables us to reason about the global behavior with respect to unbounded queues and to improve the efficiency of the behavior verification by removing the message queues, and hence, reducing the sate space without changing the behavior. The extended synchronizability analysis presented in this paper identifies the synchronizable peer interfaces efficiently without flattening the HSMs.

Recently, some researchers have used existing model checking tools to assure reliability of web services. Model checking is an automated verification technique that exhaustively searches the state space of a system to verify or falsify (by generating counter-examples) safety and liveness properties. In [21] and [9] the authors verify a given web service flow (specified in WSFL and BPEL respectively) by using the explicit state model checker SPIN [14]. In [13] the Labeled Transition System Analyzer (LTSA) is used for inferring the correctness of the web service compositions which are specified using message sequence charts. In [22], web services are verified using a Petri Net model generated from a DAML-S description of a service. An extended Petri Net model is proposed in [24] for verification of workflows and analyzing inheritance relations among models. These earlier verification efforts focus on the specification level and do not address the correctness of individual peer implementations. Verification of the communication flow does not guarantee that a composite web service behaves according to its specification unless we can ensure that each peer obeys its published contract (this requirement is called *conformance* in [19]). In our framework, both interaction behavior and interface conformance are verified.

The outline of the paper is as follows: Section 2 gives two motivating examples. Section 3 explains the Peer Controller Pattern. Section 4 presents the HSM model and Section 5 presents the semantics of composite web services based on the HSM model. Section 6 discusses the synchronizability analysis for HSMs. Section 7 explains the automated translation of peer interfaces to BPEL. Section 8 presents our automated verification technique and the experiments. Section 9 concludes the paper.

2 Motivating Examples

We will use two examples to illustrate the approach presented in this paper. The first one is a ***Travel Agency*** service. This service reserves a hotel, a car and a flight according to a request coming from a customer. A travel agent gets the travel information from the customer. The travel information is a set of alternative dates and vacation destinations. Then, the travel agent starts the accommodation and transportation reservations. These reservations are made concurrently and do not have any dependency between them. The transportation reservation books a car and a flight if the customer wants a transportation reservation. The bookings of the car and the flight also occur concurrently to complete the task in a timely manner. A flight reservation could be one-way or round-trip depending on the request. Finally, an itinerary is sent to the customer.

The *Travel Agency* service described above is composed of five individual services (peers): Customer, TravelAgent, CarReserve, HotelReserve, and FlightReserve. The Customer service provides the information about the customer request to the TravelAgent with the *place, date, process, transportResv*, and *transportNoResv* messages. The TravelAgent arranges reservations according to the requests of the Customer and responds to the Customer with the *travelInv* message. The other three services perform specific reservation requests. The HotelReserve communicates using the *reqHotel*, and *hotelInv* messages, the CarReserve using the

reqCar, and *carInv* messages and the FlightReserve using the *reqFlight*, and *flightInv* messages.

The second example we will discuss in this paper is a ***Purchase Order Handling*** service described in the BPEL 1.1 specification [5]. In this example, a customer makes a purchase order to a vendor. The vendor calculates the price for the order including the shipping fee, arranges a shipment, and schedules the production and shipment. The vendor uses an invoicing service to calculate the price, a shipping service to arrange the shipment, and a scheduling service to handle scheduling. To respond to the customer in a timely manner, the vendor performs these three tasks concurrently while processing the purchase order. There are two control dependencies among these three tasks that the vendor needs to consider: The shipping price is required to complete the final price calculation, and the shipping date is required to complete the scheduling. After these tasks are completed, the vendor sends an invoice to the customer.

The web service for this example is composed of five peers: CRelations, Purchasing, Shipping, Scheduling, and Invoicing. Customers order products using the CRelations peer and the *reqOrder* message. The CRelations peer communicates with the Purchasing peer which plays the role of the vendor described above. The Purchasing peer responds to the CRelations with the *replyOrder* message. The remaining services are the ones that the Purchasing peer uses to process the product order. The Shipping peer communicates with the *reqShipping*, and *schedule* messages, the Scheduling peer with the *productSchedule*, and *shippingSchedule* messages, and the Invoicing peer with the *initialize, shippingPrice*, and *invoice* messages.

In both services described above, the communication among the peers is through asynchronous messaging. These services can process more than one customer request at a time. Each customer request is processed according to the associated control logic described above.

We are interested in analyzing the interactions of multiple peers in composite web services such as the ones described above. In [7, 8, 9] interactions among peers in such a system is specified as a conversation, i.e., the sequence of messages exchanged among peers recorded in the order they are sent. The conversation set captures the global behaviors of a composite web service where each peer executes correctly according to its interface specification, and every message ever sent is eventually consumed. We assume that no messages are lost during transmission, which is a reasonable assumption based on the messaging frameworks such as [17, 20, 16].

3 Peer Controller Pattern

In this section we present the *Peer Controller Pattern* [4] which helps programmers in developing reliable web services that are composed of asynchronously communicating peers. The composite web services implemented based on this design pattern are amenable to automated verification.

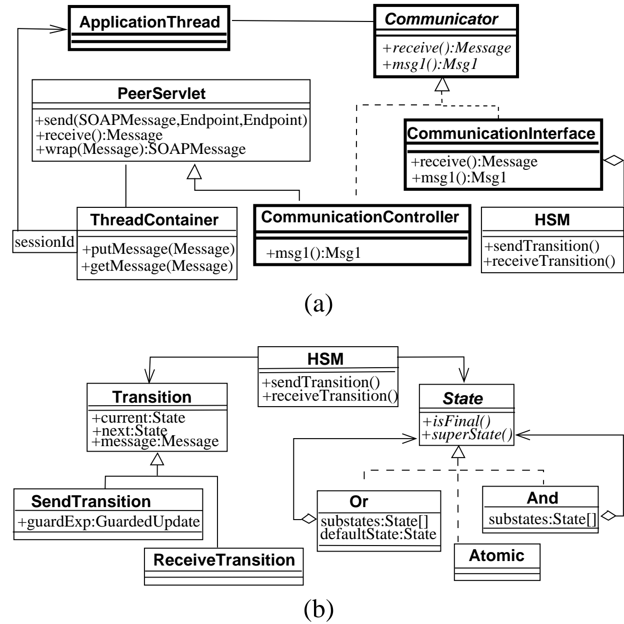

Figure 1. Class Diagrams: (a) Peer Controller Pattern (b) Hierarchical State Machines

The driving forces for the Peer Controller Pattern are as follows: 1) To achieve interoperability, the interface of a peer should be specified explicitly and should serve as a behavioral contract, specifying everything other peers need to know to interact with it. A peer interface should not be affected by the changes in the peer implementation that are irrelevant to this contract. 2) The application logic of a peer should be implemented separate from the communication logic handling the asynchronous communication. This separation is necessary for standardization of the communication and maintainability of the code. 3) The implementation should be amenable to automated verification. Due to their distributed nature and asynchronous communication, web services are prone to errors. There should be a scalable automated verification framework to ensure their correctness.

These forces are resolved by the Peer Controller Pattern. Since the Peer Controller Pattern decouples the application logic of a peer and the communication component, the developer can focus on the application logic without worrying about the details of the asynchronous communication implementation. This pattern also facilitates explicit behavioral interface descriptions and is supported by an automated verification technique. The class diagram for the Peer Controller Pattern is shown in Figure 1(a). The classes a developer needs to write are drawn as bold. Other classes can be used as is, without modification.

The application logic is implemented with the `ApplicationThread`. Each instance of this thread is identified with a session number. The application thread communi-

cates asynchronously with other peers through the `Communicator` which is a Java interface that provides standardized access to the communication implementation. This Java interface provides a single receive operation and one send operation per message type.

The `CommunicationController` class is a servlet that performs the actual communication. Since it is tedious to write such a class, we provide a servlet implementation (`PeerServlet`) that uses JAXM [16] in asynchronous mode. This servlet deals with opening an asynchronous connection, creating SOAP messages, and sending and receiving SOAP messages through the JAXM provider. The `CommunicationController` class extends the `PeerServlet` and implements the `Communicator`.

The `PeerServlet` servlet is associated with a `ThreadContainer` which contains application thread references indexed by session identifiers. When a message with a session identifier is received from the JAXM provider, it is delegated to the thread indexed with that session number. I.e., there is one thread for each session and that thread is responsible for executing the application logic for that session.

In the Peer Controller Pattern, the behavioral interface of a peer is written as an instance of the `CommunicatorInterface` class. This class defines the peer interface which is the behavioral contract of the peer. This explicit definition of the behavioral contract is crucial both for interoperability and modular verification.

4 Hierarchical State Machines

We propose using Hierarchical State Machines (HSMs) for specifying behavioral interfaces of participant peers in a composite web service. Figures 2 and 3 show the HSMs specifying the peer interfaces for the Travel Agency and the Purchase Order Handling services, respectively.

The class diagram in Figure 1(b) shows the structure of the HSMs. An HSM state can either be an atomic state or a composite state with substates. There are two types of composite states: `And` and `Or` states. In Figures 2 and 3, composite states are shown as rectangles and atomic states are shown as circles. The substates of `And` states are separated using dashed lines. Each `Or` state has a unique default substate (shown with an arc without a source). Each state has an attribute called `final` which denotes if it is a final state (final states are shown with double lines). We assume that each HSM has a *root* state at the top of the state hierarchy, which is an `Or` state (we do not show this root state in the figures).

An HSM consists of a set of states and a set of transitions. There are two types of HSM transitions: send and receive transitions. Each transition has a source state, a target state, and a message. We denote a receive transition as $(r, ?m, r')$ where r is the source state, r' is the target state, and m is the message received.

A send transition has an additional attribute, a guarded

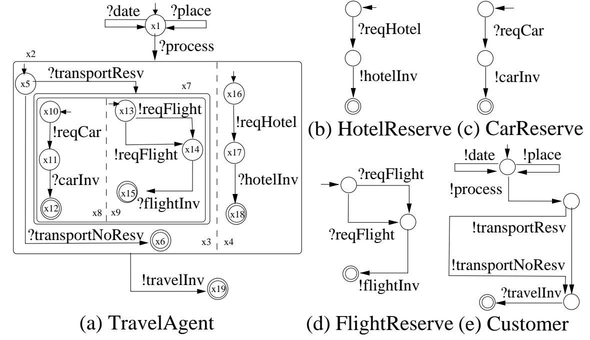

Figure 2. Interfaces for Travel Agency

command, which is a pair of conditions on message contents. A guarded command consists of a guard condition on the contents of the latest instances of the messages that have been transmitted, and an update condition on the contents of the message to be sent. A guard condition consists of predicates on the fields of the latest messages, combined with the logical operators $\wedge, \vee, \neg$. An update condition is a conjunction of predicates which assign values to the fields of the message that is being sent. We denote a send transition as $(r, !m[g/u], r')$ where r is the source state, r' is the target state, m is the message being sent, g is the guard condition and u is the update condition.

Consider the Travel Agency example. Recall that, in this example, a flight reservation could be for a one-way or a round trip flight depending on the request of the customer. The TravelAgent peer has two send transitions that correspond to these two cases with labels !*reqFlight*[*transportResv.flight = oneway/ reserve = oneway*] and !*reqFlight*[*transportResv.flight = roundtrip/ reserve = roundtrip*] where *reserve* is a field of the *reqFlight* message and *flight* is a field of the *transportResv* message. We assume that each message field is either boolean or enumerated. Note that, we only model the message fields that influence the interface behavior. There can be other message fields in the implementation which contain data. These fields, however, do not need to be a part of the interface model if they do not affect the interface behavior.

Finally, in HSMs, dependency arcs can connect states to transitions. A transition can only be taken if the current configuration of an HSM contains the states that are the sources of all the dependency arcs that point to that transition (formal description is given below). In Figure 3, there are two dependency arcs. One of them implies that the transition from a_9 to a_{10} can only be taken if the the current configuration contains the states a_9 and a_6. The other dependency arc implies that the transition from a_{13} to a_{14} can only be taken if the the current configuration contains a_{13} and a_7.

HSMs are a variation of Statecharts [12]. The differences are, in HSMs 1) the transitions are triggered by the send or receive operations instead of events, 2) the transition guards and updates are defined on message contents, and 3) depen-

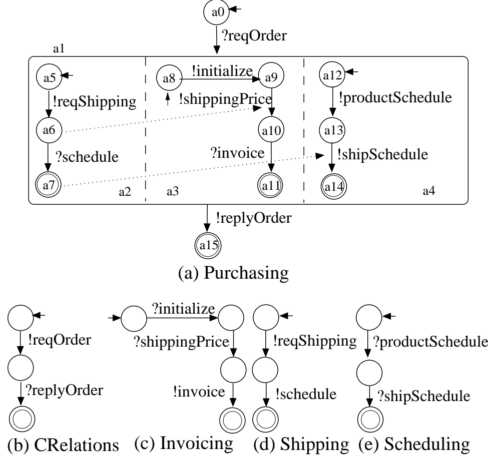

Figure 3. Interfaces for Order Handling

dency arcs are used instead of predicates on states in the guards of the transitions. We also restrict the HSMs so that: 1) the labels of the substates of an `And` state are disjoint, 2) there are no transitions among the substates of an `And` state, and 3) an HSM can leave a state only if that state is exit-ready which means that all of its substates are final states.

HSMs provide a compact model which represents the natural hierarchy in the interface behavior. Consider the peer interface of the TravelAgent peer given in Figure 2(a). (For readability, the guard and update conditions for the send transitions are not shown in the figure.) This interface is specified with 19 states (excluding the $root$ state) and 13 transitions. If we specify this interface with a flat finite state machine there would be 35 states and 78 transitions.

Now, we define the HSM model formally. An HSM is a tuple $P = (S, C, I, F, T, M, D)$ where S is the set of states (including the $root$ state), C is the set of configurations, $I \in C$ is the initial configuration, $F \subseteq C$ is the set of final configurations, T is the set of transitions, M is the set of messages, and D is the set of dependency arcs. We use $class(M)$ to denote the set of message classes and for each message $m \in M$ we use $class(m)$ to denote the message class of m. For each composite state $s \in S$, we use $sub(s)$ to denote its substates. For a substate s, $super(s)$ denotes its superstate. We use $sub^+(s)$ and $super^+(s)$ to denote all the descendants and ancestors (respectively) of the state s in the state hierarchy. We also define the following: $sub^*(s) = sub^+(s) \cup \{s\}$ and $super^*(s) = super^+(s) \cup \{s\}$. Given a state s, $type(s) \in \{$ `atomic`, `Or`, `And` $\}$ denotes its type.

A configuration $c \in C \cup C_{sub}$ is a tree whose nodes are states from S. We denote the root node of a configuration c as $root(c)$. We use C to denote the configurations whose root is the *root* state, and the rest of the configurations are called subconfigurations, denoted by C_{sub}. Given a configuration c and a state s that appears in c, $subconf(s, c)$ is the set of subtrees (subconfigurations) of c for which the root node is a substate of s, i.e., $c' \in subconf(s, c) \Rightarrow super(root(c')) = s$. We use $subconf^+(s, c)$ to denote all the subtrees of the state s in c. All configurations have to satisfy the following constraints: 1) The root state of all the configurations in C is $root$; 2) Children of a node in a configuration are substates of that node; 3) All the leaf nodes in a configuration are `Atomic` states; 4) Each `Or` state in a configuration has exactly one child which is one of its substates; and 5) Each `And` state in a configuration has all of its substates as its children.

Transition set T is partitioned to two sets $T = T_S \cup T_R$, where T_S is the set of send transitions and T_R is the set of receive transitions. Each send transition $t \in T_S$ is a tuple $t = (r, z, g, u, r')$ where $r \in S$ is the source state, $r' \in S$ is the target state, $z \in class(M)$ is the message class of the message that is being sent, g is the guard condition defined on the contents of the latest instances of the messages that have been transmitted, and u is the update condition defined on the contents of the message that is being sent. Each receive transition $t \in T_R$ is a tuple $t = (r, z, r')$ where $r \in S$ is the source state, $r' \in S$ is the target state, and $z \in class(M)$ is the message class of the message that is being received. Each dependency arc $d \in D$ is a state-transition pair $d = (s, t)$, i.e., $D \subseteq S \times T$. A transition can only be taken in a configuration which contains all the states that are the source of a dependency arc that points to that transition.

In the initial configuration I, all the states which are substates of an `Or` state are default states. A configuration c is a final configuration (i.e., $c \in F$) if all the states in c which are substates of an `Or` state are final states. We call a state s in a configuration c exit-ready if all the states in $subconf(s, c)$ which are substates of an `Or` state are final states. A configuration c is a final configuration if $root(c)$ is exit-ready. A transition from a state in a configuration can only be taken if that state in that configuration is exit-ready.

The execution of an HSM starts from the initial configuration and continues by transitioning to a next configuration of the current configuration until one of the final states is reached. In Figure 4 we show the next configuration computation for HSMs. Given a current configuration and a message, the **takeReceiveTrans** function computes all the next configurations after receiving that message, and the **takeSendTrans** function computes all the next configurations after sending that message using the recursive functions **next** and **construct**.

5 Interactions of HSMs

In this section we discuss the semantics of a composite web service based on the Peer Controller Pattern whose interfaces are defined with HSMs. In our model, HSMs interact with each other by exchanging messages through FIFO queues. We assume that each HSM is associated with an input message queue and the messages are delivered without any error (i.e., no duplicate, lost or modified messages

```
takeSendTrans(m:Message, C:set of Configurations)
returns a set of Configurations
  N: set of Configurations
  for each c in C add all elements of next(c,m,"send") to N
  store the content of m as the latest transmitted message of its type
  return N

takeReceiveTrans(m:Message, C:set of Configurations)
returns a set of Configurations
  N: set of Configurations
  for each c in C add all elements of next(c,m,"receive") to N
  store the content of m as the latest transmitted message of its type
  return N

next(c:Configuration, m:Message, direction:String)
returns a set of Configurations
  T: set of Transitions; R, N: set of Configurations
  if direction = "send" then T := findSendTrans(c,m)
  else T := findReceiveTrans(c,m)
  if T ≠ φ then
    if type(root(c)) = atomic or c is exit-ready then
      for each t ∈ T where s' is the target of t
          find s_a s.t. s_a ∈ super*(root(c)) ∩ super*(s')
               and sub(s_a) ∉ super*(root(c)) ∩ super*(s')
          add construct(s',s_a) to R
      return R
  if T = φ and type(root((c)) = atomic then return φ
  for each c' ∈ subconf(root(c),c)
      N := next(c',m,direction)
      for each c'' ∈ N
          if super(root(c'')) = root(c) then
            make a copy of c replacing c' with c''
            add the copy to R
          else if c is exit-ready then add c'' to R
  return R

findSendTrans(c:Configuration, m:Message) returns a set of Transitions
  T: set of Transitions
  for each t = (s_1, class(m), g, u, s_2) ∈ T_S s.t. s_1 = root(c)
      if there is a dependency (s_d, t) ∈ D s.t. s_d is not in c then
        continue
      if the contents of the latest transmitted messages satisfy g
        and the contents of m satisfy u then
            add t to T
  return T

findRecvTrans(c:Configuration, m:Message) returns a set of Transitions
  T: set of Transitions
  for each t = (s_1, class(m), s_2) ∈ T_R s.t. s_1 = root(c)
      if there is a dependency (s_d, t) ∈ D s.t. s_d is not in c then
        continue
      add t to T
  return T

construct(s_t,s:State) returns a Configuration
  c: Configuration;
  set root(c) to s
  if type(s) = And then
    for each s' ∈ sub(s) add construct(s_t,s') to subconf(s,c)
  else if type(s) = Or then
    if s ∈ super+(s_t)
       find s' ∈ sub(s) s.t. s' ∈ super*(s_t)
    else let s' ∈ sub(s) be the default state
    add construct(s_t,s') to subconf(s,c)
  else subconf(s,c) := φ
  return c
```

Figure 4. Next Configuration Computation

during transmission). When a message m is sent from the peer i to the peer j, the message m is inserted to the end of the input message queue of the peer j, and the configuration of the peer i is updated according to the **takeSendTrans** function given in Figure 4. A peer receives a message by consuming the first element of its input message queue and updates its current configuration according to the **takeReceiveTrans** function given in Figure 4.

Formally, a composite service is a tuple $CS = (M, P_1, \ldots, P_k)$ where M is a finite set of messages, k is the number of interacting peers, for each $1 \leq i \leq k$, $P_i = (S_i, C_i, I_i, F_i, T_i, M_i, D_i)$ is an HSM defining the interface of peer i, and $M = \bigcup_{1 \leq i \leq k} M_i$.

We define the execution semantics of a composite service as a transition system $T(CS) = (IT, CT, RT)$ where CT is the set of configurations, $IT \subseteq CT$ is the set of initial configurations, and RT is the transition relation of the system. The set of configurations is defined as $CT = C_1 \times Q_1 \times \ldots \times C_k \times Q_k$ where k represents the number of peers in the composition and Q_i is the configuration of the message queue that holds the incoming messages to peer i.

We introduce the following notation. Given a message $m \in M$, *receiver*(m) denotes the peer that receives the message m. Given a configuration $c \in CT$ and a peer identifier i, $c(C_i)$ denotes the configuration of the peer P_i in configuration c, and $c(Q_i)$ denotes the configuration of the input queue Q_i in configuration c. We define two functions. The function *append*: $\text{DOM}(Q) \times \text{DOM}(Q) \to \text{DOM}(Q)$ is used for manipulation of the queue configurations, where *append*(Q_1, Q_2) appends Q_2 to the end of Q_1. The function *first* returns the first element in Q. $\langle\rangle$ denotes an empty queue and $\langle m \rangle$ where $m \in M$ denotes a queue containing a single message m.

The set of initial configurations of $T(CS)$ is defined as $IT = \{c \mid c \in CT \ \wedge\ (\forall 1 \leq i \leq k, c(Q_i) = \langle\rangle \ \wedge\ c(C_i) = I_i)\}$

We define the following relation for a send transition which sends a message m:

$$
\begin{aligned}
&RT_{!m} = \{(c, c') \mid c, c' \in CT \wedge (\exists 1 \leq i \leq k, \\
&c'_i \in P_i.\mathit{takeSendTrans}(m, c_i) \wedge c(C_i) = c_i \wedge c'(C_i) = c'_i \\
&\wedge(\forall 1 \leq j \leq k, j \neq i, c'(C_j) = c(C_j))) \\
&\wedge \mathit{receiver}(m) = P_p \ \wedge\ c'(Q_p) = \mathit{append}(c(Q_p), \langle m \rangle) \\
&\wedge(\forall 1 \leq l \leq k, l \neq p, c'(Q_l) = c(Q_l))\}
\end{aligned}
$$

We define the following relation for a receive transition which receives a message m:

$$
\begin{aligned}
&RT_{?m} = \{(c, c') \mid c, c' \in CT \wedge (\exists 1 \leq i \leq k, \\
&c'_i \in P_i.\mathit{takeReceiveTrans}(m, c_i) \wedge c(C_i) = c_i \\
&\wedge c'(C_i) = c'_i \wedge (\forall 1 \leq j \leq k, j \neq i, c'(C_j) = c(C_j)) \\
&\wedge \mathit{first}(c(Q_i)) = m \wedge \mathit{append}(\langle m \rangle, c'(Q_i)) = c(Q_i) \\
&\wedge(\forall 1 \leq l \leq k, l \neq i, c'(Q_l) = c(Q_l)))\}
\end{aligned}
$$

Finally, the transition relation RT for the $T(CS)$ is

$$RT = \bigcup_{m \in M} (RT_{!m} \cup RT_{?m})$$

Having defined the execution semantics of a composite service, we can define the conversations generated by executions of a composite service. An execution sequence $e = c_0, c_1, ...$ is a sequence of configurations where for each $i \geq 0$, $(c_i, c_{i+1}) \in RT$ and $c_0 \in IT$. The conversation $conv(e)$ generated by an execution sequence e is defined recursively as follows: The conversation $conv(c_0)$ is the empty sequence. The conversation $conv(c_0, c_1, ..., c_n, c_{n+1})$ is equal to $conv(c_0, c_1, ..., c_n), m$ if there exists a Q_j such that $c_{n+1}(Q_j) = append(c_n(Q_j), \langle m \rangle)$, and it is equal to $conv(c_0, c_1, ..., c_n)$ otherwise. A conversation is a complete conversation if in the last configuration of the execution sequence each peer is in a final configuration and all the message queues are empty.

Conversation model gives us a convenient framework for analyzing interactions of web services. Given this framework, a natural problem is verifying properties related to conversations. As discussed in [8], temporal logic LTL can be extended to specify properties of conversations. A composite web service satisfies an LTL property if all the conversations generated by the service satisfy the property.

In general, model checking conversations of asynchronously communicating finite state machines is an undecidable problem [8]. In [9], a set of synchronizability conditions are introduced to identify composite web services which can be verified using finite state model checking techniques. In the next section, we discuss the synchronizability analysis of HSMs.

6 Synchronizability Analysis

Synchronizability analysis is a technique for identifying systems that generate the same conversation set under both asynchronous and synchronous communication ([8, 9]). If a system of asynchronously communicating finite state machines is synchronizable, we can verify the system with synchronous semantics by omitting the input queues. This reduces the state space of the system to a finite set and enables automated verification. Since the generated conversation set remains the same and the properties are defined over conversations, the verification results hold for the asynchronous communication semantics.

Synchronizability analysis uses two sufficient conditions: 1) synchronous compatibility and 2) autonomous condition. Synchronous compatibility requires that when one peer is ready to send a message, the receiver should be in a state where it can receive it. The autonomous condition requires that at any state a peer has exactly one of the following three choices: 1) to send, 2) to receive, or 3) to terminate, i.e., a peer cannot have both receive and send transitions from the same state. Note that, the autonomous condition still allows a certain level of nondeterminism. If a peer is in a state where it can only send a message, the choice of which message to send can be nondeterministic.

Below we discuss the synchronizability of a composite

```
autonomy() returns boolean
  for each s ∈ sub(root)
    status := simpleAutonomy(s)
    if status = "fail" return false
    if s is final return finalStateCheck(s)
  return true

simpleAutonomy(s:State) returns String
  mystatus:String
  mystatus := "none"
  if there is a send transition from s then
    mystatus := "send"
  if there is a receive transition from s then
    if mystatus = "send" then return "fail"
    else mystatus := "receive"
  for each s' ∈ sub(s)
    status := simpleAutonomy(s')
    if status = "fail" then return "fail"
    if there exists a transition t from s'
      s.t. target of t is not in sub+(s) then
      if status = "send" and mystatus = "receive" then return "fail"
      if status = "receive" and mystatus = "send" then return "fail"
      if status ≠ "none" then mystatus := status
  return mystatus

finalStateCheck(s:State) returns boolean
  if there exists a transition from s then return false
  for each s' ∈ sub(s)
    if s' is final and finalStateCheck(s') = false then return false
  return true
```

Figure 5. Autonomy Check

web service which consists of peers specified as HSMs. We check the above two conditions without flattening the HSMs. Since, on average, the number of states and transitions in an HSM are less than an equivalent flat finite state machine, the analysis performed on HSMs is more efficient.

Checking Autonomous Condition. The autonomy checking algorithm is given in Figure 5. The base condition is that given a state, all the transitions originating from that state should be either send transitions or receive transitions. If this condition is satisfied, then we investigate whether all of the substates of that state satisfy this condition as well.

The function **simpleAutonomy** in Figure 5 first checks this base condition. If there are no failures, it examines the transitions whose source and target states do not share the same superstate. For each composite state s, the transition type from s should be the same as the transition type from its final substates to the states that are not the substates of s. For example, adding the transition $(a_{14}, ?m, a_{15})$ to Figure 3(a) would violate the autonomy. On the other hand, adding the transition $(a_{14}, !m[g/u], a_{15})$ does not violate the autonomous condition.

The autonomy check algorithm first invokes the **simpleAutonomy** for all substates of the $root$. If no failures are reported, the algorithm checks that there are no configurations in C that follow the configurations in the final configuration set (F) of the HSM in question. If no violations are found, the algorithm concludes that the HSM is autonomous.

```
synchronous(Peers: set of HSMs) returns boolean
  inspect:set of Configuration arrays, a, b:Configuration array
  sendConf, recvConf:set of Configurations, legal:boolean
  inspect := φ
  for each P_i ∈ Peers a[i] := P_i.I
  add a to inspect
  while inspect ≠ φ do
    remove one element from inspect and write it to a
    for each Message m ∈ M
      for each P_i ∈ Peers
        legal := false
        sendConf := P_i.takeSendTrans(m, {a[i]})
        if sendConf ≠ φ then
          for each P_j ∈ Peers s.t. P_j ≠ P_i
            recvConf := P_j.takeReceiveTrans(m,{a[j]})
            if recvConf ≠ φ then
              legal=true
              for each sc ∈ sendConf
                for each rc ∈ recvConf
                  let b be a copy of a
                  set b[i] to sc and b[j] to rc
                  add b to inspect
          if legal=false then return false
  return true
```

Figure 6. Synchronous Compatibility Check

Checking Synchronous Compatibility Condition. To check the synchronous compatibility of the peers in a composite web service, we search for the illegal configurations in the Cartesian product of the peer interfaces. The configurations of this product are tuples with the domain $C_1 \times \ldots \times C_k$ where k is the number of peers in the composition. Let c_i denote the configuration for peer P_i at a product configuration. A configuration in the product is illegal if P_i.**takeSendTrans**(m, c_i) returns a nonempty set while for all other peers P_j.**takeReceiveTrans**(m, c_j) is empty. The synchronous compatibility checking algorithm is given in Figure 6.

7 BPEL Generation

When a composite web service is implemented based on the Peer Controller Pattern, we can generate BPEL specifications for each participating peer. We have implemented a translator that takes HSMs defining the peer interfaces and automatically creates BPEL specifications to publish. The translator first synthesizes WSDL specifications, which are the connectivity contracts, and then generates the BPEL specifications. A generated BPEL specification contains partner link definitions to access other peers, variable declarations, and behavior description of the peer.

To reflect the hierarchy of the peer interface, the behavior description in a generated BPEL specification consists of nested scopes. Each scope has two local variables. The variable `state` represents current state in the scope, and the variable `exit` is used for exiting the scope. Each scope is a `while` loop that terminates depending on `exit`. The body of the loop is a `switch` activity whose `case` conditions are on the value of the `state`.

A case block implementing the operations at the corresponding state, consists of two activities. The type of the first activity depends on the type of the state. If the state is of type `And`, a `flow` is generated which has one subactivity per its substate. If the state is of type `Or`, a new scope is generated and this inner scope's local `state` is set to the name of the default substate. The second activity corresponds to the transitions originating from this state. The mappings of send and receive transitions to BPEL code fragments are similar to the mapping discussed in [4], which are sending or receiving appropriate messages and setting the current state. In this second activity, if the state is an exit-state then the local `exit` variable is set.

The following is an excerpt from the BPEL specification generated from the peer interface in Figure 2(a). This excerpt shows the code fragment synthesized for state x_2.

```
<case condition="state='x2'">
  <sequence>
   <flow>
     <scope>
       <variables> ...</variables>
       <!-- define fresh state and exit variable-->
       <sequence>
         ...<!--initilize state and exit variables-->
         <while condition="exit!='yes'">
           <switch>
             <case condition="state='x7'">
              <sequence>
               <flow>
                ...<!--car reservation-->
                ...<!--flight reservation-->
               </flow>
               <assign><copy>
                 <from expression="'yes'"/>
                 <to variable="exit"/>
               </copy></assign>
              </sequence>
             </case>
             ...<!--cases when state is 'x5' or 'x6'-->
           </switch>
         </while>
       </sequence>
     </scope>
     <scope>
       ...<!--hotel reservation -->
     </scope>
   </flow>
     ...<!--send travelInv, and assign 'x19' to state-->
  </sequence>
</case>
```

If there are transitions whose source and target states are not siblings, the translator identifies these transitions during preprocessing and flattens that portion of the HSM.

When there are dependency arcs in a peer interface, the translator generates one `link` [5] per arc. The translator places the link target and source inside the corresponding send or receive fragments. Consider the dependency arc in Purchasing (Figure 3(a)) which implies that to take the transition from a_9 to a_{10} the current configuration has to contain the state a_6. For this dependency arc, the translator puts the link source inside the activity that sets the state to a_6, and puts the link target into the fragment corresponding to the transition from a_9 to a_{10}.

8 Automated Verification and Experiments

Based on the peer interfaces specified with HSMs and the Peer Controller Pattern, we can check that each peer implementation conforms to its interface (interface verification), and verify properties of the peer interactions using the conversation model (behavior verification). In this section, we present this modular verification technique and the verification results for the two motivating examples.

During the *interface verification*, we check each peer implementation separately. We use the peer interfaces implemented as `CommunicationInterface` instances (Section 3) with the HSMs defining the order of messages a peer can send and receive. A peer implementation conforms to its interface if all the call sequences to the `Communicator` are accepted by its HSM. For example, in the Travel Agency service, a thread implementing the CarReserve peer conforms to its interface if it sends a *carInv* message to the TravelAgent if and only if after receiving a *reqCar* message.

For the interface verification we use the program checker Java PathFinder (JPF) [6]. JPF is an explicit state model checker for Java and supports property specifications via assertions that are embedded in the source code. JPF exhaustively traverses all possible execution paths. Using JPF we can verify arbitrary Java threads without any restrictions on data types.

To perform the interface verification with JPF, we substitute the actual communication instance with the `CommunicationInterface`, so that each send and receive operation is directed to the `sendTransition` and `receiveTransition` methods of the HSM. We input the modified program to JPF to look for assertion violations. These assertions, which are provided in the HSM implementation, are as follows: 1) the result of the next configuration computation should not be empty, 2) when a receive operation is invoked at a configuration, the set of possible messages to receive at that configuration should not be empty. A peer implementation conforms to its interface if JPF does not report any assertion violations. Since the peer interfaces are HSMs with finite states and abstract the asynchronous messaging, the efficiency of the interface verification is improved significantly.

We implemented the Travel Agency service based on the Peer Controller Pattern. We checked that the application threads conform to their interfaces given in Figure 2. The interface verification of the Customer peer took 5.63 seconds and used 4.92 MB memory, the HotelReserve peer took 4.76 seconds and used 3.95 MB memory, the CarReserve peer took 4.61 seconds and used 3.74 MB memory, the FlightReserve peer took 4.83 seconds and used 3.89 MB memory, and the TravelAgent peer took 9.72 seconds and used 19.69 MB memory when implemented as one thread. One can use multiple threads for implementing the TravelAgent peer to exploit the concurrency introduced by the `And` state in the interface specification. For such an implementation, we need to verify that the combined behavior of the concurrent threads conform to the peer interface. In this case one can use the modular verification approach presented in [3] and verify the concurrent threads separately.

Assuming that the peers behave according to their interfaces, during *behavior verification*, we verify safety and liveness properties on the global interaction behavior of a composite system using the conversation model. For behavior verification we use the model checker SPIN [14]. SPIN is an explicit finite state model checker for concurrent systems and provides a structure called *channel* which is suitable for modeling message queues. Using only the peer interfaces, we automatically generate a specification that represents the composite service in the input language of the SPIN. Then, using SPIN, we verify the composite service with respect to the properties defined on the conversations. A sample property for the Travel Agency example is "Whenever a travel request is sent, eventually a travel invoice will be sent."

We verified the global behavior of Travel Agency service by translating the HSMs to the input language of SPIN. Translation of composite web services with interfaces specified as finite state machines is discussed in [4]. We extended this translation algorithm to HSM interfaces. First we investigated the verification of the Travel Agency service using asynchronous communication. During this behavior verification we used different input queue sizes. We observed that for the Travel Agency service the state space increases exponentially with the size of the queues. This exponential growth affected the performance of the SPIN model checker and it ran out of memory for queue size 12. In fact, the reachable state space for this example is infinite when the queues are unbounded.

We applied the synchronizability analysis to the Travel Agency example, and identified that it is synchronizable. Therefore, we were able to verify the Travel Agency service with synchronous communication without affecting its conversation set. With the synchronous communication state space contains only 8911 states, and the behavior verification with SPIN took 0.38 seconds and used 5.15 MB memory. With synchronous communication, the size of the input queues are set to 0, and messages are consumed as soon as they are sent. Since the messages are not buffered, the state space of the specification is reduced and this reduction leads to a significant improvement in the efficiency of the behavior verification and avoids the state space explosion.

We also implemented the Purchase Order Handling example based on the Peer Controller Pattern and verified it with our modular verification technique. The interface verification of the CRelations peer took 4.63 seconds and used 3.73 MB memory, the Invoicing peer took 4.54 seconds and used 4.22 MB memory, the Shipping peer took 4.81 seconds and used 3.91 MB memory, the Purchasing peer took 5.00 seconds and used 7.69 MB memory, and the Scheduling peer took 4.83 seconds and used 3.76 MB memory. The be-

havior verification with asynchronous communication used 6.78 MB memory in 0.59 seconds for different queue sizes. We observed that during any execution there is at most one message at each queue at any state; therefore, increasing the size of message queues did not increase the state space (10105 states). This experimental observation is not a proof of the fact that the results we obtained using bounded verification will hold for this example when unbounded message queues are used. To guarantee this, we applied the synchronizability analysis and identified that the five HSMs given in Figure 3, are synchronizable. With synchronous communication there are only 1562 states, and behavior verification with SPIN took 0.08 seconds and used 2.01 MB memory.

9 Conclusions

We presented an HSM model for specifying behavioral interfaces of peers in a composite web service. With HSMs we achieved a compact representation that 1) reflects the natural hierarchy of the service behavior, 2) can specify concurrent executions of operations, and 3) is suitable for automated verification. We integrated this model to a design pattern that facilitates the development of verifiable web services that communicate with asynchronous messaging. Based on this pattern, we analyzed both the conformance of the peer implementations to their interfaces and the global interaction in the composition with a modular verification technique. Our experiments show that our modular verification approach can check asynchronously communicating web service implementations and their interactions using reasonable amount of time and memory. We extended the synchronizability analysis proposed in [9] to HSMs to efficiently identify synchronizable peer interfaces. With this analysis we verified conversations of composite web services with respect to unbounded queues. We also discussed automated translation of peer interfaces specified as HSMs to BPEL to be published for interoperability.

References

[1] B. Benatallah, M. Dumas, Q. Z. Sheng, and A. H. H. Ngu. Declarative composition and peer-to-peer provisioning of dynamic web services. In *Proc. of Int. Conf. on Data Eng.*, 2002.

[2] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proc. of the 1st Int. Conf. on Service Oriented Computing*, 2003.

[3] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proc. of Automated Software Eng.*, 2004.

[4] A. Betin-Can, T. Bultan, and X. Fu. Design for verification for asynchronously communicating web services. In *Proc. of World Wide Web Conf.*, 2005.

[5] Business process execution language for web services version 1.1. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, May 2003.

[6] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder: Second generation of a Java model checker. In *Proc. Workshop on Advances in Verification*, 2000.

[7] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proc. of World Wide Web Conf.*, 2003.

[8] X. Fu, T. Bultan, and J.Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proc. of the 8th Int. Conf. on Implementation and Application of Automata (CIAA)*, 2003.

[9] X. Fu, T. Bultan, and J.Su. Analysis of interacting BPEL web services. In *Proc. of World Wide Web Conf.*, 2004.

[10] C. E. Gerede, R. Hull, O. H. Ibarra, and J. Su. Automated composition of e-services: Lookaheads. In *Proc. of Int. Conf. on Service Oriented Computing*, 2004.

[11] J. E. Hanson, P. Nandi, and S. Kumaran. Conversation support for business process integration. In *Proc. of the 6th Int. Enterprise Distributed Object Computing Conf.*, 2002.

[12] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3), June 1987.

[13] H.Foster, S. Uchitel, J.Magee, and J.Kramer. Model-based verification of web service compositions. In *Proc. of. the 18th Int. Conf. on Automated Software Engineering*, 2003.

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

[15] Conversation support for agents, e-business, and component integration. http://www.research.ibm.com/convsupport.

[16] Java API for XML messaging (JAXM). http://java.sun.com/xml/jaxm/.

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

[18] Z. Maamar, B. Benatallah, and W. Mansoor. Service chart diagrams - description & application. In *Proc. of Int. World Wide Web Conf.*, 2003.

[19] L. Meredith and S. Bjorg. Contracts and types. *Communications of ACM*, 46(10):41–47, October 2003.

[20] Microsoft Message Queuing Service. http://www.microsoft.com/msmq.

[21] S. Nakajima. Verification of web service flows with model-checking techniques. In *Int. Symposium on Cyber Worlds: Theories and Practice*, 2002.

[22] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proc. of World Wide Web Conf.*, 2002.

[23] K. Sivashanmugam, K. Verma, A. P. Sheth, and J. A. Miller. Adding semantics to web services standard. In *Proc of ICWS*, 2003.

[24] H. M. W. Verbeek. *Verification of WF-nets*. PhD thesis, Technische Universiteit Eindhoven, Netherlands, 2004.

[25] Web service choreography interface 1.0. http://www.w3.org/TR/wsci/, August 2002.

[26] Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl.