Design for Verification for Asynchronously Communicating Web Services

Aysu Betin-Can
Computer Science
Department
University of California
Santa Barbara, CA 93106,
USA

aysu@cs.ucsb.edu

Tevfik Bultan
Computer Science
Department
University of California
Santa Barbara, CA 93106,
USA

bultan@cs.ucsb.edu

Xiang Fu[*]
School of Computer and
Information Science
Georgia Southwestern State
University
Americus, GA 31709, USA

xfu@canes.gsw.edu

ABSTRACT

We present a design for verification approach to developing reliable web services. We focus on composite web services which consist of asynchronously communicating peers. Our goal is to automatically verify properties of interactions among such peers. We propose a design pattern that eases the development of such web services and enables a modular, assume-guarantee style verification strategy. In the proposed design pattern, each peer is associated with a behavioral interface description which specifies how that peer will interact with other peers. Using these peer interfaces we automatically generate BPEL specifications to publish for interoperability. Assuming that the participating peers behave according to their interfaces, we verify safety and liveness properties about the global behavior of the composite web service during behavior verification. During interface verification, we check that each peer implementation conforms to its interface. Using the modularity in the proposed design pattern, we are able to perform the interface verification of each peer and the behavior verification as separate steps. Our experiments show that, using this modular approach, one can automatically and efficiently verify web service implementations.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.4 [Software Engineering] Software/Program Verification – *Model checking, Formal methods*; H.5.3 [Information Systems] Group and Organization Interfaces – *Asynchronous interaction, Web-based interaction*

General Terms: Design, Verification

Keywords: Composite web services, asynchronous communication, design patterns, BPEL

1. INTRODUCTION

Web-based software applications which enable user interaction through web browsers have been extremely successful. Nowadays one can look for and buy almost anything online, from a book to a car, using such applications. A promising extension to this framework is the area of web services, i.e., Web accessible software applications which interact with each other through the Internet. Web services have the potential to have a big impact on business-to-business applications similar to the impact interactive web software had on business-to-consumer applications.

Web services provide a framework for decoupling the interfaces of Web accessible applications from their implementations, making it possible for the underlying applications to interoperate and integrate into larger, composite services. The following characteristics of web services are crucial for this purpose: 1) standardizing data transmission via XML [26], 2) loosely coupling interacting services through standardized interfaces, and 3) supporting asynchronous communication.

A fundamental problem in developing reliable web services is analyzing their interactions. The characteristics mentioned above present both opportunities and challenges in this direction. For example, decoupling interfaces and implementations, which is necessary for interoperability, also provides opportunities for modular analysis. On the other hand, asynchronous communication, which is necessary to deal with pauses in availability of services and slow data transmission, makes analysis more difficult.

In this paper, we present a design for verification approach for developing reliable composite web services. Earlier work on applying formal verification techniques to web services [9, 11, 21, 7, 20] focused on the specification level, whereas our work focuses on implementing reliable web services in Java and addresses the reliable implementation of individual services.

A composite web service consists of a collection of individual web services, called *peers*, working in a collaborative manner. As mentioned above, interaction among peers is established via *asynchronous messages*. In asynchronous communication, when a message is sent, it is inserted to a FIFO message queue, and the receiver consumes (i.e. receives) the message when it reaches to the front of the queue. 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 [6, 12, 14]. A typical peer implementation includes a code for the operations specific to the application, a code for the asynchronous communication mechanism, and an interface specification describing the peer behavior.

In this paper, we propose a behavioral design pattern called the *Peer Controller Pattern* for developing reliable web services. The Peer Controller Pattern separates the operations related to the application logic from the communication details. The communication component is responsible for asynchronous messaging. The component implementing the application logic uses the communication component to interact with other peers. This decoupling improves the code maintainability and reusability and supports our modular verification strategy.

In the Peer Controller Pattern, each peer has a behavioral interface description which captures the information needed by the other peers that interact with it. A *peer interface* is a Java class implementing a state machine which defines the order of send and receive operations that can be executed by that peer. The interface of a peer serves as a contract between that peer and the other peers. We automatically generate BPEL [4] specifications from the peer interfaces that can be published for interoperability.

The Peer Controller Pattern enables a modular, assume-guarantee style verification strategy. We show that if the developers use the proposed design pattern, then using model checking, they can automatically check the properties of a composite web service (behavior verification) and the conformance of the peer implementations to their interfaces (interface verification).

For interface verification we use the program checker Java PathFinder [5]. During interface verification, the interface of a peer is used as a stub for the communication component. This approach solves the environment generation problem in this context, and enables verification of each peer in isolation, improving the efficiency of the interface verification significantly.

For behavior verification we use the explicit and finite state model checker SPIN [13] which allows us to model asynchronous messaging. Given the peer interfaces, we *automatically* generate a Promela (the input language of the SPIN model checker) specification for the composite web service. We check safety and liveness properties of the composite web service specification using the conversation model.

Since SPIN is a finite state model checker, the size of the message queues needs to be bounded. Such bounded verification guarantees correctness only with respect to the set bounds. Moreover, model checking a composite web service that communicates asynchronously with unbounded queues is undecidable [10]. Note that, this is not just a theoretical problem. Asynchronous messaging with unbounded message queues is supported by messaging platforms such as Java Message Service (JMS) [16], Microsoft Message Queuing Service (MSMQ) [19], and Java API for XML Messaging (JAXM [15]).

We adapt the *synchronizability analysis* proposed in [11] to our framework to verify properties of composite web services in the presence of unbounded queues. A composite web service is called synchronizable if its conversation set does not change when asynchronous communication is replaced with synchronous communication. We check the sufficient conditions for synchronizability presented in [11] based on the Peer Controller Pattern. This automated synchronizability check enables us both 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 conversation behavior).

Related Work. To achieve interoperability among web services a contractual agreement among participating peers is a necessity. The WSDL [25] standard is commonly used as a contract for specifying the operations, port types, 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 [18]. A number of standards have been proposed for describing the behavior of a web service, such as BPEL [4], WSCI [24], and OWL-S (formerly DAML-S) [22]. In [6], state machines are used for this purpose and in [11] it is shown that other behavioral descriptions (such as BPEL) can be translated to state machines. Since state machines are powerful enough to specify the behaviors of web services and since they are

suitable for automated reasoning, in our framework, the behavioral contracts among peers are specified as state machines.

Berardi et al. [2] also use state machines as behavioral contracts. They focus on action sequencing rather than message sequencing in the composition. Unlike our work, their goal is to automatically synthesize composite web services. Benatallah et al. [1] use statecharts to describe service behavior, specifically to declare a service composition. They present a framework for implementing web services without addressing verification of service interactions.

Model checking has become a commonly used technique in automated software verification. Recently, some researchers have applied existing model checking tools to verification of composite web services. For example, in [20] and [11] the authors verify a given web service flow (specified in WSFL and BPEL respectively) by using the model checker SPIN. [9] presents an application of the Labeled Transition System Analyzer (LTSA) in inferring the correctness of the web service compositions which are specified using message sequence charts. In [21], web services are verified using a Petri Net model generated from a DAML-S description of a service. Unlike these earlier verification efforts, we consider the correctness of the individual peer implementations as well as the verification of the global properties of the composite web services. Verification of the communication flow does not guarantee that the composition behaves according to the specification unless we ensure that each individual service obeys its published contract (this requirement is called *conformance* in [18]).

Rajamani et al. [23] address the conformance of an implementation model to a specification for asynchronous message passing programs. Unlike the interface verification in our framework, their conformance check requires a model extraction from the implementation. Moreover, their approach does not separate the interface and the behavior verification steps.

Mehlitz et al. [17] suggest the design for verification approach which promotes using design patterns and exploiting the properties of the design patterns in improving the efficiency of the automated verification. The approach we propose in this paper is an application of the design for verification approach to web services.

In [3] we presented a design pattern and a modular verification approach for concurrency controllers. The work presented in [3] focuses on concurrent threads accessing shared data using synchronization statements. In this paper, we are focusing on interaction among web services and asynchronous communication; therefore, both the application domain and the underlying semantic model are different.

The rest of the paper is organized as follows: Section 2 presents an example composite web service. In Section 3, the Peer Controller Pattern is presented. Section 4 presents the proposed modular verification technique for composite web services based on the Peer Controller Pattern. Section 5 explains the automated BPEL specification generation. Section 6 discusses the experimental results, and Section 7 presents our conclusions.

2. AN EXAMPLE WEB SERVICE

To illustrate the approach presented in this paper, we use the *Loan Approval* example described in the BPEL 1.1 specification [4]. In this example a customer requests a loan for some amount. If the amount is small, the loan request is approved. For large amounts, a risk assessment service decides a risk level. The loan request is approved when the risk level is low and denied when the risk level is high.

The Loan Approval service is composed of three individual services (peers): CustomerRelations, LoanApprover and RiskAssessor (see Figure 1). Customers make loan requests using the Cus-
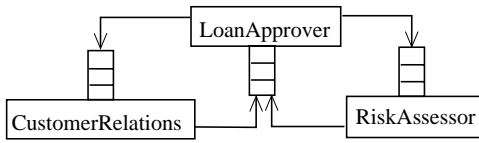
Figure 1: Loan Approval Service

tomerRelations service. This service sends a *request* message to the LoanApprover service. If the request is for a small amount, the LoanApprover service sends an *approval* message, with the accept field set to true, to the CustomerRelations service. Otherwise, the LoanApprover service sends a *check* message to the RiskAssessor service. Then, the LoanApprover service sends an *approval* message to the CustomerRelations service with the *accept* field set to true or false depending on the message received from the RiskAssessor service.

In this system, the communication among the peers is through asynchronous messaging. The Loan Approval service can process more than one customer application at a time. Each loan request generates a new session. The control logic described above is the same for each session.

2.1 Peer Interfaces as Contracts

To reason about a composite web service, we need behavioral contracts describing the behaviors of the individual services, i.e., peers. We use finite state machines to specify behaviors of the peers and we call these state machines peer interfaces. Let us consider the Loan Approval example. Since this service is a composition of three services, one can specify the peer interfaces with three finite state machines, as shown in Figure 2.

The state machines in Figure 2 (a), (b), and (c) specify the behavioral interfaces of the CustomerRelations, LoanApprover and RiskAssessor services respectively. These behaviors are specified for one session. The transitions are labeled either with *!message* or *?message*, denoting sending or receiving of a message, respectively. There are 5 message types: *request* with an *amount* field, *approval* with an *accept* field, *check* with an *amount* field, *nocheck* with no fields, and *risk* with a *level* field. As seen in Figure 2, send transitions are labeled with conditions on the message contents. We will discuss the syntax and semantics of these conditions in Section 3, however, as an example, consider the transition labeled with `!approval[risk.level=high/accept=false]` in Figure 2 (b). This transition is taken only if the `level` field of the last `risk` message is `high`. When this guarding condition holds, the
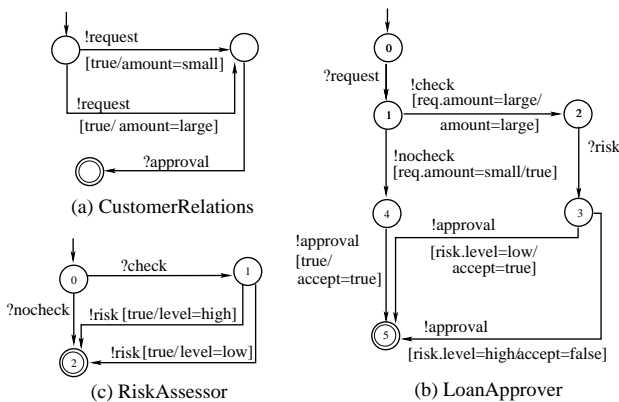

(a) CustomerRelations

(c) RiskAssessor

(b) LoanApprover

Figure 2: Peer Interfaces

LoanApprover peer sends an approval message with the `accept` field set to `false`.

2.2 Conversations

Using the peer interfaces, global behavior of a composite web service can be modeled as a set of state machines communicating with asynchronous messages, similar to the communicating finite state machine (CFSM) model. In [6, 10, 11] the interactions among peers in such a system are specified as a conversation, i.e., the sequence of messages exchanged among peers recorded in the order they are sent. A conversation is said to be complete if at the end of the execution each peer ends up in a final state and each message queue is empty. The notion of a conversation captures the global behavior 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 provided by the industry [16, 19, 15]). For example, the following is a conversation that can be generated by the Loan Approval example in Figure 2 (where the values of the message fields are shown in parentheses): *request(amount=large)*, *check(amount=large)*, *risk(level=high)*, *approval(accept=false)*.

The conversation model gives us a convenient framework for reasoning about and analyzing interactions of web services. Given this framework, a natural problem is verifying properties related to conversations. As discussed in [10], the 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.

Note that, during the execution of the Loan Approval service which generates the above conversation, the input queue of each peer contains at most one message. However, this may not always be the case. It is easy to write specifications with infinite state spaces where the queues are not bounded. In fact, model checking conversations of asynchronously communicating finite state machines is an undecidable problem [10]. In the following sections we will show that using a set of synchronizability conditions we can identify composite web services which can be verified using finite state model checking techniques by replacing asynchronous communication (with unbounded message queues) with synchronous communication without changing the conversation set generated by the composite web service.

3. PEER CONTROLLER PATTERN

In this section we present the *Peer Controller Pattern* which resolves the following design forces that arise in the development of reliable composite web services: 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 about a peer to interact with it. The interface of a peer should not be affected by the changes in the peer implementation that are not relevant to this contract. 2) The application logic of a peer should be implemented independent 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.

The Peer Controller Pattern resolves the above design forces. In the Peer Controller Pattern, the application logic of a peer and the
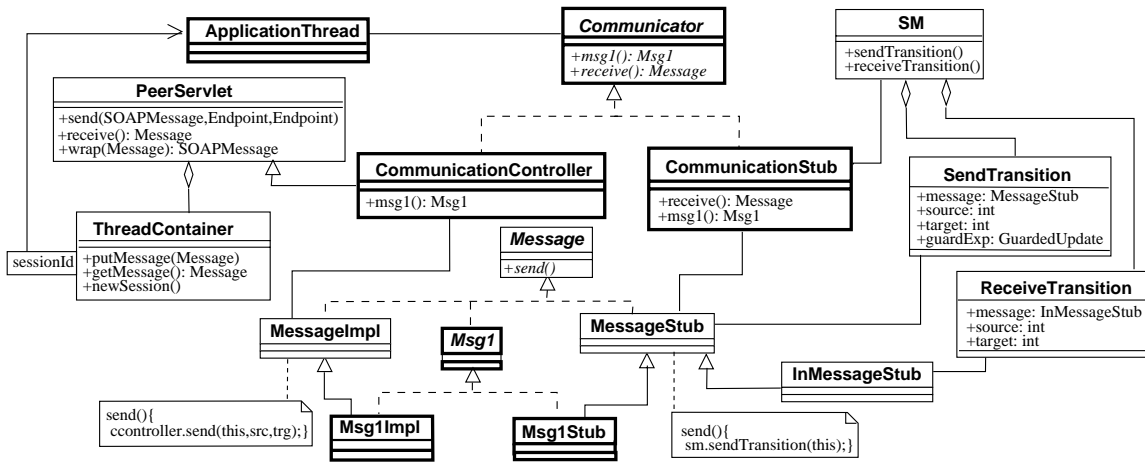
Figure 3: Class Diagram for the Peer Controller Pattern

communication component are separated. This separation enables the developer to focus on the application logic without worrying about the details associated with the implementation of the asynchronous communication. This pattern also requires the developer to define the peer interface, which is the behavioral contract of the peer, explicitly. The peer interface is specified within the communication component. This explicit definition of the behavioral contract is crucial both for interoperability and modular verification.

The class diagram of the Peer Controller Pattern is shown in Figure 3. The classes a developer needs to write are drawn as bold. Other classes can be used as is, without modification.

The proposed pattern is session based. The application logic of a peer is the same for each session. This logic is implemented in the `ApplicationThread`. The application thread communicates asynchronously with other peers through the `Communicator`. The `Communicator` is a Java interface that provides standardized access to the actual asynchronous communication implementation and its stub. The actual communication is performed via the `CommunicationController` and customized message implementation classes (e.g. `Msg1Impl`). The peer interface is written as a state machine via the `CommunicationStub`. This class uses message stubs and a nondeterministic state machine implementation (SM). Note that in the Peer Controller Pattern, the communication component is more than a Business Delegator [8]. This component contains the behavioral contract of a peer, and plays a crucial role in verification.

Communication Controller. 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 [15] in asynchronous mode. This helping servlet deals with opening an asynchronous connection, creating SOAP messages, and sending/receiving a SOAP message through the JAXM provider. The `CommunicationController` class extends the `PeerServlet` and implements the `Communicator` interface. An interface method in this class returns a new actual outgoing message instance. Figure 4 shows a `CommunicationController` implementation for the LoanApprover peer.

The helping servlet is associated with a `ThreadContainer` that contains application thread references indexed by the session identifier. Whenever a message with a session identifier is received from the JAXM provider, it is delegated to the thread indexed with that session number. We use buffers for this message delegation.

```
public class ApproverServlet extends PeerServlet
  implements LAServlet{
 public void init(ServletConfig servletconfig)
      throws ServletException{
  super.init(servletconfig);
  urn="urn:LoanApprover";
 }
 public ApprovalMessage request(int sessionId){
   return new ApprovalMessageImpl(this,sessionId); }
 public CheckMessage check(int sessionId){
   return new CheckMessageImpl(this,sessionId); }
 public NoCheckMessage nocheck(int sessionId){
   return new NoCheckMessageImpl(this,sessionId); }
}
```

Figure 4: ApproverServlet

If there is no thread for the specified session, this container class creates a new application thread instance and starts that thread.

Messages. The peers interact with each other with customized messages implemented in Java. Each message implementation consists of one actual message type definition, one message stub, and one Java interface to provide uniform access to these classes. The abstract class for the customized message types is shown as `MessageImpl` in Figure 3. Its `send` operation uses the sending method of the `PeerServlet`. The abstract class for the message stubs is `MessageStub` class. Its `send` operation uses the `sendTransition` method of SM, which is explained below. This abstract class has a subclass that serves as a stub for incoming messages. Finally, the Java interface which unifies the actual message types and their stubs is called `Message` in Figure 3.

As an example, consider the *approval* message used by the Loan-Approver peer. For this message type, we need to implement one Java interface, one message class which is used for communication, and one stub class which is used during verification. In Figure 3, these implementations correspond to the `Msg1`, `Msg1Impl` and `Msg1Stub`, respectively.

In our framework, the fields of message classes are categorized as control data and non-control data. The data fields that are used in the interface specifications are called *control data*. A control data field can be of enumerated or boolean type. Since Java does not have enumerated type, we provide an `Enumerated` class. We use this separation between control and non-control data in reducing the states of the message stubs. The stub for a message class only stores the values of the control data.

Communication Stub. The `CommunicationStub` class is a special class that specifies the *peer interface*. A peer interface specifies the behavioral contract of the peer and it is also used during the verification process. The `CommunicationStub` class contains two representative variables for each message type. One holds the last value, and the other holds the current value of a message.

The `CommunicationStub` encodes the state machine defining the contract by using the provided `SM` class. In the constructor, developer defines the transitions of the state machine. There are two kinds of transitions: send and receive transitions. A send transition is defined as a `SendTransition` instance. This instance stores the message, the source and the target states, and the guarded update for that transition. A guarded update is defined as an anonymous inner class implementing the `GuardedUpdate` Java interface. The guard defines the condition when the transition is available. The update condition specifies the contents of the message to be sent when the guard holds.

The syntax of guard and update conditions are defined as follows:

$$guard \rightarrow term \mid guard \text{ \&\& } guard \mid guard \mid\mid guard \mid \text{ !}guard$$
$$update \rightarrow term \mid update \text{ \&\& } update$$
$$term \rightarrow varname \text{ == } value$$

In the guard condition the *varname* is in the form of `last_msg.fieldname()` where `last_msg` is the name of the variable that holds the last value of the message `msg`. In other words, in guard conditions the equalities are defined on the control fields of last messages. In an update condition, the equalities are defined on the message to be sent, i.e., *varname* is in the form of `msg.fieldname()`. As an example, consider the transition (3, *!approval[risk.level=low/ accept=true]*, 5) in Figure 2(b). This send transition is implemented as:

```
GuardedUpdate gc=new GuardedUpdate(){
  public boolean guard(){
    return last_risk.level()==Low;}
  public boolean update(){
    return approval.accept()==true;}};
SendTransition outT=
           new SendTransition(approval,3,5,gc);
```

A receive transition is defined with a `ReceiveTransition` instance. This instance holds the message, the source and the target states. For example, in the LoanApprover peer the receive transition (0,*?request*, 1) in Figure 2(b) is specified as `ReceiveTransition inT=new ReceiveTransition(request,0,1);` where the variable `request` has the current value of the request message.

The `SM` class is a nondeterministic finite state machine implementation. This class has two important methods: `sendTransition` and `receiveTransition`. The method `sendTransition(message)` computes the set of next states from the current states, asserts that this set is not empty, and updates the current state. Each next state must be the target of a send transition whose 1) label is of the same message type, 2) guard and update conditions are satisfied, 3) and source state is in the current state. The `receiveTransition()` method computes the set of possible receive transitions available in the current state, and asserts that this set is not empty. Then, it chooses one of these transitions nondeterministically, updates the current state, and returns the chosen incoming message stub instance. The incoming message stubs (`InMessageStub`) are generated in the preprocessing phase of the interface verification. These classes have an `instance` method which returns an instance with control field values chosen nondeterministically from the possible values that other peers can set.

The nondeterminism is achieved by the `Verify.random` function which is a special function of the program checker Java PathFinder (JPF) [5]. This function forces JPF to search exhaustively every possible nondeterministic choice during interface verification.

3.1 Semantics of Composite Web Services

In this section we formalize the semantics of a composite web service based on the Peer Controller Pattern. We use interface of a peer as the specification of its behavior. This semantic definition is the formal model we use during behavior verification. We omit the guard and update conditions in the semantic model. However, it is easy to extend the model below to handle guard and update conditions by adding the control data for the messages to the model. In fact, as long as the control data is of finite domain, one can model the semantics of a composite web service with guard and update conditions using the semantic model given below by storing the contents of the messages in the states of the finite state machines.

In our semantic model, a composite web service is a tuple $W = (M, P_1, \ldots, P_k)$. M is a finite set of message types and P_i is the interface of the peer i, where $1 \leq i \leq k$ and k is the number of peers in the composition. For each $m \in M$, $sender(m) \in \{P_1, \ldots, P_k\}$ denotes the peer that sends the message m, and *receiver*$(m) \in \{P_1, \ldots, P_k\}$ denotes the peer that receives the message m. We assume that there is one sender and one receiver for each message type.

Each peer interface $P_i = (SP_i, TP_i, IP_i, FP_i)$ is a finite state machine specifying the behavior of the peer i. SP_i is the set of states, $IP_i \in SP_i$ is the initial state, and $FP_i \subseteq SP_i$ is the set of final states. The transition relation TP_i is defined as follows. Let $m \in M$ and $r_1, r_2 \in SP_i$. For each $t \in TP_i$, t is in the form of $t = (r_1, !m, r_2)$ where *sender*$(m) = P_i$, or $t = (r_1, ?m, r_2)$ where *receiver*$(m) = P_i$.

The semantics of a composite web service is a transition system $T(W) = (IT, ST, RT)$ where ST is the set of states, $IT \subseteq ST$ is the set of initial states, and RT is the transition relation of the system. The set of states is defined as $ST = SP_1 \times Q_1 \times \cdots \times SP_k \times Q_k$ where k is the number of peers in the composition and Q_i is the configuration of the message queue that holds the incoming messages to peer P_i for $1 \leq i \leq k$.

We introduce the following notation. Given a state $s \in ST$ and a peer identifier i, $s(SP_i)$ denotes the state of the peer P_i in state s, and $s(Q_i)$ denotes the configuration of input queue Q_i in state s. We define two functions. The function *append*: $\text{DOM}(Q) \times \text{DOM}(Q) \rightarrow \text{DOM}(Q)$ is used for manipulation of the queue configurations, where *append*(Q_1, Q_2) appends Q_1 to the front of Q_2. The function *first* returns the first element in the 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 states of $T(W)$ is defined as $IT = \{s \mid s \in ST \wedge (\forall 1 \leq i \leq k, s(Q_i) = \langle\rangle \wedge s(SP_i) = IP_i)\}$

We define the following relation for a send operation:

$$RT_{(r,!m,r')} = \{(s, s') \mid s, s' \in ST \wedge (\exists 1 \leq i \leq k,$$
$$(r, !m, r') \in TP_i \wedge s(SP_i) = r \wedge s'(SP_i) = r'$$
$$\wedge (\forall 1 \leq j \leq k, j \neq i, s'(SP_j) = s(SP_j)))$$
$$\wedge receiver(m) = P_p \wedge s'(Q_p) = append(s(Q_p), \langle m \rangle)$$
$$\wedge (\forall 1 \leq l \leq k, l \neq p, s'(Q_l) = s(Q_l))\}$$

We define the following relation for a receive operation:

$$RT_{(r,?m,r')} = \{(s, s') \mid s, s' \in ST \wedge (\exists 1 \leq i \leq k,$$
$$(r, ?m, r') \in TP_i \wedge s(SP_i) = r \wedge s'(SP_i) = r'$$
$$\wedge (\forall 1 \leq j \leq k, j \neq i, s'(SP_j) = s(SP_j))$$
$$\wedge first(s(Q_i)) = m \wedge append(\langle m \rangle, s'(Q_i)) = s(Q_i)$$
$$\wedge (\forall 1 \leq l \leq k, l \neq i, s'(Q_l) = s(Q_l)))\}$$

The transition relation RT for the $T(W)$ is defined as

$$RT = \bigcup_{(r,!m,r') \in TP_i, 1 \leq i \leq k} RT_{(r,!m,r')}$$
$$\cup \bigcup_{(r,?m,r') \in TP_i, 1 \leq i \leq k} RT_{(r,?m,r')}$$

An execution sequence $e = s_0, s_1, ...$ is a sequence of states where $(s_i, s_{i+1}) \in RT$ and $s_0 \in IT$. The conversation $conv(e)$ generated by an execution sequence e is defined recursively as follows: The conversation $conv(s_0)$ is the empty sequence. The conversation $conv(s_0, s_1, ..., s_n, s_{n+1})$ is equal to $conv(s_0, s_1, ..., s_n), m$ if there exists Q_j such that $s_{n+1}(Q_j) = append(s_n(Q_j), \langle m \rangle)$, and it is equal to $conv(s_0, s_1, ..., s_n)$ otherwise. A conversation is a complete conversation if in the last state of the execution sequence each peer is in a final state and all the message queues are empty.

4. VERIFICATION

In this section, we present a modular verification technique for composite web services implemented based on the Peer Controller Pattern. During the *interface verification*, we check that each peer implementation conforms to its interface which defines the order that a peer can send and receive messages. During the *behavior verification*, assuming that the peers behave according to their interfaces, we check a set of LTL properties on the global behavior of the composite web service using the conversation model. This verification strategy solves the environment generation problem for the peers and improves the efficiency of the verification and, hence, makes automated verification of realistic web services feasible.

4.1 Interface Verification

A peer implementation conforms to its interface, if all the call sequences to the Communicator are accepted by the finite state machine defining the peer interface. For example, in the Loan Approval service, the LoanApprover peer should not send a *check* message to the RiskAssessor peer before getting a loan request with a large amount.

We use the program checker Java PathFinder (JPF) [5] for the interface verification. 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 to look for assertion violations. Using JPF, we can check whether a peer implementation generates a call sequence that is not allowed by the peer interface. As discussed in Section 3, the peer interface is specified by the CommunicationStub which encodes the finite state machine using the SM class. The assertions that JPF checks are embedded in the SM class. Since these stubs are finite state machines and abstract the asynchronous messaging with other peers, the efficiency of the interface verification is improved significantly.

During the interface verification, we check the peer implementations for a single session. Since in the Peer Controller Pattern each session is independent and does not affect other sessions, it is sufficient to check the peer implementations for a single session, which improves the efficiency of the verification further.

To perform the interface verification, the communication controller and the message instances are replaced with the communication stub and the message stubs by a source-to-source transformation. With this transformation, the asynchronous communication mechanism (which cannot be handled by JPF) is abstracted away. However, we still need to write a small driver to instantiate the service. The reason is that JPF requires standalone programs as input, but a peer is a servlet without a main method. The simple driver class contains only a main method that consists of three

statements: 1) instantiating the communicator stub, 2) instantiating an application thread, and 3) starting the application thread. After these steps, the resulting program is given to JPF to search for interface violations. Note that, using the stubs and the driver we close the environment of a peer. Hence, our verification approach solves the environment generation problem and enables verification of each peer in isolation.

In the program that is given to JPF as input, each send action that the application thread performs is directed to the sendTransition of the SM class described in Section 3. This method computes the set of next states from the current state. If the set of next states is empty, it means that the application thread executed an illegal send action and JPF gives an assertion violation. Otherwise, the current state is updated, the previous value and the current value of the message to be sent is stored. Note that the message stubs only preserve the valuations of the control data. The non-control data (for example, the customer name in the Loan Approval service) is abstracted away. Hence, the message stubs reduce the state space further.

Each receive action performed by the application thread is directed to the receiveTransition of the SM class. This method computes the set of possible receive transitions available in the current state and asserts that this set is not empty. If JPF does not report an assertion violation, this means that a receive action is legal at this state. One of the receive transitions is chosen nondeterministically and the associated incoming message is returned. All of the nondeterministic choices are made using the Verify.random function which is a special method of the program checker JPF that forces JPF to search every possible choice exhaustively (i.e., this is an exhaustive search not random testing).

4.2 Bounded Behavior Verification

We use the explicit and finite state model checker SPIN [13] for the behavior verification. SPIN provides a structure called *channel* which is suitable for modeling the asynchronous messaging among the peers.

During the behavior verification, we assume that each peer obeys its interface. Based on this assumption, we can verify safety and liveness properties of a composite web service by using only the peer interfaces to characterize the peer behaviors, and ignoring the peer implementations (implementations are checked during the interface verification). Recall that, the peer interfaces are actually finite state machines (such as the ones shown in Figure 2) which are specified using the CommunicationStub classes (Figure 3). We have implemented a translator that takes these state machine specifications and the message stubs as input, and automatically generates a specification in Promela, which is the input language of SPIN.

Consider the Loan Approval service which consists of three peers. Given the message stubs and the communication stubs encoding the interfaces in Figure 2, our translator generates a Promela specification with three process types. Below is an excerpt from the generated specification.

```
#define size 5
mtype ={requestType, approvalType, nocheckType,
        checkType,riskType} //message names
/*data domains*/
mtype ={undef1,small,large}//amount domain
mtype ={undef2,low,high}//level domain
/*message types*/
typedef approval{ bool accept; }
typedef request{ mtype amount; }
typedef risk{ mtype level; }
...
message lastmsg;//holds the last send message
/*channels*/
```

```
chan customerQ=[size] of {mtype,message}
chan approverQ=[size] of {mtype,message}
chan assessorQ=[size] of {mtype,message}
proctype LoanApprover(){
  short state=0;
  nocheck nocheckmsg;
  check checkmsg;
  approval  approvalmsg;
  risk riskmsg;
  request requestmsg;
  message msg;
  do
  ::state==0 ->
    if
    ::approverQ?[requestType,msg]->/*receive*/
      approverQ?requestType,msg;
      requestmsg.amount=msg.requestmsg.amount;
      state=1;
    fi
  ::...
  ::state==3 ->
      if
      ::riskmsg.level==low ->/*guard*//*send*/
        approvalmsg.accept=true;/*update*/
        msg.approvalmsg.accept=approvalmsg.accept;
        atomic{
         lastmsg.approvalmsg.accept=approvalmsg.accept;
         customerQ!approvalType,msg; }
        state=5;
      ::riskmsg.level==high ->/*guard*//*send*/
        approvalmsg.accept=false;/*update*/
        msg.approvalmsg.accept=approvalmsg.accept;
        atomic{
         lastmsg.approvalmsg.accept=approvalmsg.accept;
         customerQ!approvalType,msg; }
        state=5;
      fi
  ::state==5 ->break; /*final state*/
  od;
}
proctype CustomerRelations(){...}
proctype RiskAssessor(){...}
init{ atomic{run CustomerRelations();
     run LoanApprover(); run RiskAssessor();}}
```

The first part of this specification declares the constants, the types and the global channels. The message name domain is defined with `mtype` which is the enumerated type in Promela. The domains of the control data are defined similarly. The message types are declared as type constructs (`typedef`) holding the control data values. The global variable `lastmsg` holds the last message transmitted. This variable is of type `message` which combines all the message types. The global channel variables are the asynchronous communication channels simulating the input message queues of the peers. In this example the sizes of the channels are restricted to 5 which is given as an input to the specification generator. These channels are defined to store elements consisting of a message name and a `message`.

The second part is a set of process type definitions. In Promela, the `proctype` keyword is used for defining concurrent process. One concurrent process definition is generated for each peer. These definitions are used for defining the behavior of a peer by implementing the state machine specified by the communication stub (i.e., the peer interface). In the generated code, each process definition has a local variable called `state`. This variable holds the current state of the state machine. Each process also has one local variable for each message type it sends or receives. The body of each process is a single loop which nondeterministically chooses an operation to execute depending on the `state`. At each state, there is a conditional selection which chooses a send, a receive or a termination operation to execute. During execution, one of the operations whose enabling condition is true is selected nondeterministically. The last part is the `init` block which instantiates the concurrent processes.

Let us consider the process type `LoanApprover`. The body of this process encodes the finite state machine given in Figure 2(b). This excerpt shows one example for each of the receiving, sending and terminating operations in that order. When the `state` is 0, the process can execute a receive operation. If there is a *request* type message at the head of the message queue, the receive operation is enabled. This condition is checked using the `approverQ?[requestType,msg]` statement which does not alter the queue contents. When this condition holds, the request message is removed from the queue, the message contents are stored in the local request message variable, and the state is updated. This fragment corresponds to the transition from state 0 to state 1 with label *?request* in Figure 2(b).

When the `state` is 3 the process has two send operations to choose from. The first choice corresponds to the *!approval* transition with the guard condition *last_risk.level()==low* and the update condition *approval.accept()==true*. Note that, this guarded update is described in the communication stub of the LoanApprover peer. This fragment checks the risk level, sets the accepting field of the approval message to true, and constructs a message to be sent. Next, `lastmsg` is atomically updated and the approval message is sent. Finally, the state is updated. The second choice corresponds to the *!approval* transition with the guard *last_risk.level()==high* and the update condition *approval.accept()==false*.

State 5 is a final state of the loan approver peer. Therefore, when the `state` is 5, the loop is terminated.

Using this automatically generated specification, we can check the LTL properties about the global behavior of the composite web service using the conversation model. The LTL properties are not generated automatically, they have to be specified by the user. An LTL property is specified using the atomic properties, the boolean logic operators ($\wedge$, $\vee$, $\neg$) and the temporal logic operators (G: globally, F: eventually, U: until). The atomic properties are predicates on the messages. An example property for the Loan Approval service is as follows: "Whenever a request message with a large amount is sent, eventually an approval message (with accept field set to true or false) will be sent."

SPIN is a finite state model checker, and therefore, the sizes of the channels need to be bounded. For example, in the above specification, the sizes of the channels are bounded with the `size` constant. Bounding the sizes of the communication channels, however, poses a problem since the verification results only hold as long as the channel sizes remain within the set bounds. In the next section we address this problem.

4.3 Unbounded Behavior Verification Using the Synchronizability Analysis

Bounded verification using SPIN can only give web service developers a certain level of confidence—it cannot ensure freedom from bugs (with respect to the specified LTL properties). On the other hand, the general problem of model checking a composite web service which uses asynchronous communication with unbounded queues is undecidable. In [10, 11], a technique called the *synchronizability analysis* is developed to identify asynchronously communicating finite state machines which can be verified automatically. The main idea is to find a set of sufficient synchronizability conditions on the control flows of the state machines, so that when these conditions are satisfied, the state machines generate the same set of conversations under both the synchronous and asynchronous communication semantics. Since the LTL properties are defined over the conversations, if these synchronizability conditions are satisfied, the verification results obtained using the synchronous semantics also hold for the asynchronous semantics. Note that,

verification of a system which consists of synchronously communicating finite state machines is decidable since the state space of the composed system is finite. In fact, the state space of the composed system can be constructed by taking the Cartesian product of the states of the individual state machines.

The synchronizability analysis developed in [11] uses two sufficient conditions to restrict the control flows of the state machines: 1) synchronous compatibility and 2) autonomous condition. Synchronous compatibility condition requires that, if we construct a state machine which is the Cartesian product of the peer interfaces, there should not be a state in the product machine in which one peer is ready to send a message, but the receiver for that message is not 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. Note that the autonomous condition still allows nondeterminism. A peer can choose which message to send nondeterministically.

We implemented the synchronizability analysis based on the Peer Controller Pattern. Given the communication stubs defining the peer interfaces, we automatically check the synchronizability conditions. If the composite service is synchronizable, the Promela code with synchronous communication semantics is generated. Otherwise, a reason for a condition violation is displayed. Note that, when the synchronizability conditions are not met, bounded verification can still be used.

The Promela code generated for a synchronizable service has two differences from the Promela code given in Section 4.2. First, the queue size is fixed to 0, which means that the processes synchronize when exchanging messages. The other difference is the implementation of the receive operations. Instead of inquiring the queue contents, the messages are received first and the appropriate action is performed depending on the message type. We need this modification because when the channel size is 0, the channels do not store messages. Therefore the inquiry `peerQ?[messagetype, msg]` always returns false.

With the aid of the automated synchronizability analysis, we can both reason about the global behavior with respect to unbounded queues and improve the efficiency of the behavior verification. Since the messages are not buffered, the state space of the specification is reduced which can lead to a significant improvement in the behavior verification.

5. BPEL GENERATION

Given a composite service whose peers are implemented based on the Peer Controller Pattern, we generate BPEL specifications from the peer interfaces automatically. As discussed earlier, the peer interfaces are specified with finite state machines. In this section, we discuss the BPEL generation from the peer interfaces.

Before creating the BPEL files, our generator creates one WSDL specification which contains all the message type definitions and one WSDL specification per peer defining its port type, partner link types and bindings. These WSDL specifications are used in the BPEL files. Then, we create one BPEL file per peer. The BPEL specification contains partner links definitions to access other peers, variable declarations, and behavior description of the peer. One of the declared variables is used to store the state of the peer. We also declare one variable per message type to store the contents of the last message of that type.

Below we discuss the mapping of the send and receive transitions to the BPEL activities. Consider the transitions originating from a state. If these transitions are send operations, the corresponding BPEL fragment consists of a `switch` clause which has one `case` for each different send transition. The condition of each

case corresponds to the guard expression of a send transition. The inner activity of each `case` block contains an assignment which corresponds to the update condition of the send transition, an invoke statement, and another assignment statement that updates the state variable, in this order.

Consider the (3, *approval[risk.level = low/ accept =true]*, 5) transition from the Loan Approver service. The code generated for this transition is:

```
<case condition="getVariableData('risk', 'level')='low'">
 <sequence>
  <assign> <copy>
     <from expression="'true'"/>
     <to variable="approval" part="accept"/>
  </copy> </assign>
  <invoke partnerLink="CustomerRelations"
     portType="ns1:CustomerRelationsPT"
     operation="ns1:approval"
     inputVariable="approval">
   </invoke>
  <assign> <copy>
     <from expression="'5'"/> <to variable="state"/>
  </copy> </assign>
 </sequence>
</case>
```

In the case of receive transitions originating from a state, there are two kinds of resulting code fragments. If there is a single receive transition, a `receive` statement is generated. For example, for the (2,*?risk*,3) transition, the generated code is:

```
<sequence>
 <receive partnerLink="RiskAssesor"
   portType="ns3:RiskAssessorPT"
   operation="ns2:risk" variable="risk">
 </receive>
 <assign> <copy>
     <from expression="'3'"/> <to variable="state"/>
 </copy> </assign>
</sequence>
```

If there are multiple receive transitions, we use the `pick` construct which has one `onMessage` per transition. For example, the initial state of Figure 2(c), defining the interface of Risk Assessor peer, has two outgoing receive transitions: (0, *?check*, 1) and (0, *?nocheck*, 2). The generated code fragment is:

```
<pick>
 <onMessage partnerLink="LoanApprover"
   portType="ns2:LoanApproverPT"
   operation="ns3:check" variable="check">
    <assign> <copy>
     <from expression="'1'"/> <to variable="state"/>
    </copy> </assign>
 </onMessage>
 <onMessage partnerLink="LoanApprover"
   portType="ns2:LoanApproverPT"
   operation="ns3:nocheck" variable="nocheck">
    <assign> <copy>
     <from expression="'2'"/> <to variable="state"/>
    </copy> </assign>
 </onMessage>
</pick>
```

If the peer interface is nondeterministic, we generate abstract BPEL processes. There are three situations that requires nondeterminism: 1) when there are both send and receive transitions originating from one state, 2) when the guard conditions associated with the send transitions originating from one state are not disjoint, 3) when there are transitions originating from a final state. We create nondeterminism by declaring an extra variable, using *opaque* assignments to that variable, and making choices based on the value of that variable. According to the BPEL 1.1 specification, an opaque assignment to a variable sets a nondeterministic value chosen from the value space of the variable.

The following is an excerpt from the BPEL specification generated for the peer interface of LoanApprover given in Figure 2(a).

```
<process name="LoanApprover" ...>
 <partnerLinks>
   <partnerLink name="LoanApprover"
        partnerLinkType="ns2:LoanApproverLinkType"
        myRole="LoanApprover"/>
   <partnerLink name="CustomerRelations"
        partnerLinkType="ns1:CustomerRelationsLinkType"
        partnerRole="CustomerRelations"/>
      ....
 </partnerLinks>
 <variables>
    <variable name="approval"
      messageType="ns0:approvalMessage"/>
    <variable name="request"
      messageType="ns0:requestMessage"/>
    <variable name="state" type="xsd:string"/>
    <variable name="exit" type="xsd:string"/>
 </variables>
 <sequence>
    ....<!--receive loan request, set '1' to state
        and create and instance -->
   <assign>
     <copy><from expression="'no'"/>
          <to variable="exit"/></copy>
   </assign>
   <while condition="exit!='no'">
     <switch>
     <case condition="state='1'">
       <sequence>
       ...<!--send check or nocheck message -->
       </sequence>
      </case>
     ...<!--other choices -->
     <case condition="state='5'">
        <assign><copy>
          <from expression="'yes'"/>
          <to variable="exit"/>
        </copy></assign>
     </case>
     </switch>
   </while>
 </sequence>
</process>
```

The specification contains two special variables. The variable `state` represents the current state, and the variable `exit` is used for the termination condition. The process description is a `while` loop that terminates depending on the value of the `exit` variable. The body of the loop selects a `case` block based on the value of the `state` variable. Each block contains a subactivity which is the corresponding send or receive fragment if the state is not a final state. Otherwise, there are two possibilities: 1) if there are no transitions originating from this final state, the local `exit` variable is set to terminate the loop; 2) otherwise, there is a nondeterministic choice between message transition activity and terminating the loop.

6. EXPERIMENTS

We implemented the Loan Approval service based on the Peer Controller Pattern. We verified the implementation using the modular verification approach presented in this paper. In our approach, exploiting the structure of the Peer Controller Pattern, verification of the peer implementations with respect to their interfaces is performed separately from the verification of the global behavior of the composite web service. As we demonstrate below, this is crucial for the feasibility of the automated verification of composite web services.

During interface verification, we used JPF to check whether all three peer implementations in the Loan Approval service obey their behavioral contracts. During interface verification of each peer, we used its communication stub which encodes the state machine defining the peer interface. With the aid of our state machine implementation, JPF investigates every possible execution of a peer implementation. We also used the message stubs and a simple

Table 1: Interface Verification Performance

Peer	Time (sec)	Memory (MB)
CustomerRelations	8.86	3.84
LoanApprover	9.65	4.70
RiskAssessor	8.15	3.64

driver to enclose the environment of the application threads in a peer implementation. Therefore, we were able to check each peer implementation separately. The interface verification performance for the Loan Approval system is given in Table 1.

We also tried to verify the whole Loan Approval service using JPF without separating the interface and the behavior verification steps. The first problem is that JPF cannot handle asynchronous communication among peers. To overcome this problem, we wrote some Java code which simulates the JAXM provider and the asynchronous input queues. In this simulation, for each peer (aside from the application thread) there is a concurrent queue instance and a thread which is activated whenever a message arrives in the queue. We ran JPF on this simulation program for only one session. JPF ran out of memory without producing a conclusive result. Hence, without using the modular approach proposed in this paper, JPF is unable to verify properties of the Loan Approval service.

Based on our modular verification approach, we verified the global behavior of the Loan Approval service with the SPIN model checker using the conversation model. An example property we verified during behavior verification is the following: "Whenever a *request* message with a small amount is sent, eventually an *approval* message accepting the loan request will be sent." The behavior verification took less than one second and used 1.49MB memory. During the behavior verification, we observed that the reachable state space of the Loan Approval system is finite (154 states). Independent of the size of the message queues, during any execution, there is at most one message in each queue at any state; therefore, increasing the size of message queues did not increase the state space. Note that, this experimental observation is not a proof of the fact that the results we obtained using bounded verification will hold for the Loan Approval service when unbounded message queues are used. To guarantee this, we used synchronizability analysis. Our automated synchronizability analyzer identified Loan Approval service as synchronizable. Therefore, we were able to verify Loan Approval service using synchronous communication, and since this service is synchronizable, the verification results are guaranteed to hold when unbounded message queues are used.

Let us consider another web service example where the number of messages in the message queues is not bounded. This example consists of one client and one supplier peer. The client peer places arbitrary number of Product1 and Product2 orders. After ordering the products, the client issues a *PayRequest* message. The supplier calculates the total price and sends a bill to the client. Client sends the payment and gets a receipt from the supplier. The state ma-
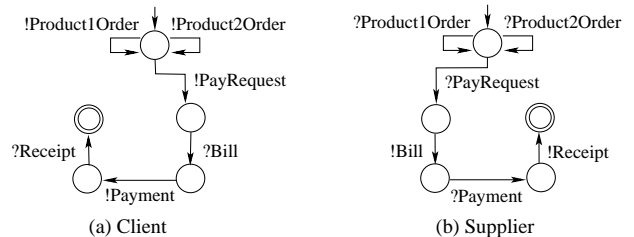

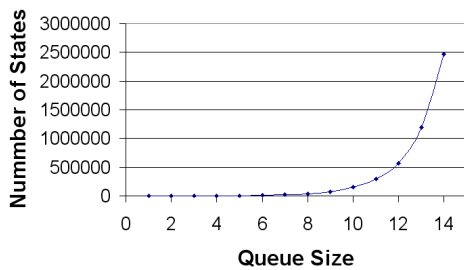**Figure 5: Client-Supplier Example**

Figure 6: Effect of the Queue Size on the State Space

chines defining the interfaces of these peers are shown in Figure 5. We verified the behavior of this example with different queue sizes. As shown in Figure 6, the state space increases exponentially with the size of the queues. In fact, the number of reachable states for this example is infinite if unbounded queues are used. The exponential growth in the state space affects the performance of SPIN significantly. SPIN ran out of memory when the queue size was set to 15. On the other hand, our automated synchronizability analyzer identified this example as synchronizable. Therefore, we can verify this service by replacing asynchronous communication with synchronous communication without changing the conversation set generated by this composite web service. With synchronous communication there are only 68 states and the behavior verification succeeds and uses 1.49 MB memory.

Our experiments show that the modularity in the verification process based on the Peer Controller Pattern improves the efficiency of the verification of composite web services significantly. We can verify asynchronously communicating web service implementations using reasonable amount of time and memory which are otherwise too large for a Java model checker to handle. With the aid of the synchronizability analysis, during the behavior verification, we can reason about the global behavior with respect to unbounded queues and perform the behavior verification efficiently. Furthermore, the usage of the stubs during the interface verification causes a significant reduction in the state space, thus improving the performance of the verification process.

7. CONCLUSIONS

In this paper, we presented a verifiable design pattern for developing reliable composite web services. Based on this pattern, we developed a modular verification approach that enables developers to check both the global behaviors of the composite web services (behavior verification) and the conformance of peer implementations to their interfaces (interface verification). We showed that the global behavior of a composite web service can be verified using the SPIN model checker, by automatically translating the peer interfaces to a Promela specification. By adapting the synchronizability analysis proposed in [11], we verified conversations of composite web services with respect to unbounded queues and improved the efficiency of the behavior verification. We showed that the interface verification can be performed with the program checker JPF using the peer interfaces as stubs for the communication component. Since these stubs are finite state machines and abstract the asynchronous messaging, we achieved a significant improvement in the efficiency of the interface verification. Another benefit of the presented approach is the explicit specification of the peer interfaces, which can be used to improve interoperability. To support this, we automatically generated BPEL specifications from the peer interfaces.

9. 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. 18th 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 the 19th IEEE Int. Conf. on Automated Software Eng.*, 2004.

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

[5] 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.

[6] 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 the 12th Int. World Wide Web Conf.*, 2003.

[7] Z. Cheng, M. P. Singh, and M. A. Vouk. Verifying constraints on web service compositions. In *Real World Semantic Web Applications*, June 2002.

[8] W. Crawford and J. Kaplan. *J2EE Design Patterns*. O'Reilly and Associates Inc., Sebastopol,California, 2003.

[9] 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 Eng.*, 2003.

[10] 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*, 2003.

[11] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. of the 13th Int. World Wide Web Conf.*, 2004.

[12] 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.

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

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

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

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

[17] P. C. Mehlitz and J. Penix. Design for verification using design patterns to build reliable systems. In *Proc. of 6th Workshop on Component-Based Software Eng.*, 2003.

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

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

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

[21] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proc. of the 11th Int. World Wide Web Conf.*, 2002.

[22] OWL-S: Semantic markup for web services. http://www.daml.org/services/owl-s/1.0/owl-s.html, 2003.

[23] S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *Proc. of the Int. Conf. on Computer Aided Verification*, 2002.

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

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

[26] Extensible markup language. http://www.w3.org/XML/