

Modular Verification of Asynchronous Service Interactions Using Behavioral Interfaces

Aysu Betin-Can[†], *Member, IEEE*, Sylvain Hallé[‡], *Member, IEEE*, Tevfik Bultan[§], *Member, IEEE*

Abstract—A crucial problem in service oriented computing is the specification and analysis of interactions among multiple peers that communicate via messages. We propose a design pattern that enables the specification of behavioral interfaces acting as communication contracts between peers. This “peer controller pattern” provides a modular, assume-guarantee style verification strategy that consists of three phases. 1) Each individual peer is statically verified for conformance to its part of the contract, using software model checking. 2) Alternately, a runtime enforcement mechanism blocks the communication events that violate the interface specification at runtime. 3) Using either of these two mechanisms, it can be assumed that the participating peers behave according to their interfaces and safety and liveness properties about the global behavior of the composite web service can then be verified directly on the communication contract. The interface verification of each peer and the behavior verification are hence handled in separate steps. A Java implementation of this pattern is developed and tested on a series of examples; we show that by working in such a modular fashion, it is possible to automatically and efficiently verify properties about service interactions that would otherwise be impossible to verify.

Index Terms—Web services, automated verification, runtime enforcement, asynchronous communication, design patterns

1 INTRODUCTION

The advent of web services in the last decade provided a framework for decoupling the interfaces of applications from their implementations. A *composite* web service consists of a collection of individual web services, called *peers*, working in a collaborative manner. A set of core characteristics contributes to the growing success of composite web services: transmission of messages is standardized by the use of XML, interacting services are loosely coupled through standardized and documented interfaces, and communications can be performed asynchronously. As an example of this growing trend, bandwidth utilized by Amazon.com web services in the fourth quarter of 2007 was greater than bandwidth consumed in the same period by all of Amazon.com’s global websites combined [1].

A fundamental problem in developing reliable web services is to analyze their interactions. This has long been argued as a *sine qua non* condition for the emergence of compositions between services going beyond trivial request-response patterns. The characteristics mentioned above present both opportunities and challenges in this direction. A loose coupling of services allows more flexible interactions; on the other hand, interface descriptions often overlook constraints on the sequential ordering of the interactions. This can cause a service to diverge from the behavior expected by its peers. Similarly, without asynchronous communication, performance can be degraded due to latency, and the unresponsiveness of a peer can unnecessarily block the progress of another peer that is trying to

send a message. Yet, asynchronous communication makes verification of web services more challenging, as senders and receivers consume messages at arbitrary times and can drift into states inconsistent with one another. In this latter case the problem is not merely theoretical: asynchronous messaging with unbounded message queues is supported by messaging platforms such as Java Message Service (JMS), Microsoft Message Queuing Service (MSMQ) and Java API for XML Messaging (JAXM).

In this paper, we demonstrate that the challenges in verifying web services that use asynchronous communications can be remedied using a “design for verification” approach that preserves the advantages of asynchronous communication while enabling efficient verification. We propose a behavioral design pattern called the *Peer Controller Pattern* (PCP) that separates the operations related to the application logic from the communication details, and show that this design pattern enables a modular verification approach.

Typical implementations of peers include both application-specific code and asynchronous communication routines. A monolithic approach to contract compliance aims at ensuring that the global system made of all peer implementations interacts in a way that fulfills a predefined behavioral specification. This can be done through extensive testing of each peer (involving actual communications), or statically through software model checking (by exhaustively exploring the execution traces of this composite system).

However monolithic approaches are not feasible for realistic systems. Loosely coupled web services typically do not belong to the same organization. An exhaustive model checking of the closed system becomes impossible unless a running instance of each peer is made available specifically

[†]Middle East Technical University, 06531, Ankara, Turkey; e-mail: aysu@i.metu.edu.tr. [‡]Université du Québec à Chicoutimi, Canada G7H 2B1; e-mail: shalle@acm.org. [§]University of California, Santa Barbara, CA 93106-5110; e-mail: shalle@acm.org, bultan@cs.ucsb.edu. This work is supported by NSF grant CCF-1117708.

for that purpose; the same applies for live testing of a peer implementation. Even when such a situation occurs, in most cases the composition of each peer’s complete implementation generates a state space too large to explore in reasonable time. This is especially true of asynchronous communications, where the presence of unbounded message queues at each receiver makes *a priori* reasoning on their possible interactions impossible to check exhaustively in many situations.

In the PCP, the communication component responsible for asynchronous messaging is separated from the implementation and abstracted by an interface called a *Communicator*, one for each peer. The Communicator imposes the designer to specify a communication contract specifying acceptable sequences of incoming and outgoing messages. Compliance of the composite web service to this communication contract is then broken down into three subproblems.

First, the interface verification phase can be conducted using a *peer modular* approach where each peer is analyzed separately to guarantee that it conforms to its interface specification. The Communicator uses the specification to simulate the outside world and provide a closed environment over which software model checkers can then verify each peer in isolation. Second, the use of a well-defined behavioral interface for each peer allows the Communicator to perform *runtime enforcement*, by tracking messages at a peer’s interface, and only allowing those that conform to the specification to cross that interface. This is especially useful when model checking of a peer cannot be performed for various reasons. Finally, in the third step, *behavior* verification is performed directly on the global communication contract, assuming that each peer conforms to its interface specification. Any of the previous two steps guarantees this assumption, either statically or at runtime.

The contributions of this paper can be summarized as follows: The PCP provides a unique communication interface for three purposes: 1) simulate the outside environment for testing or software model checking of a single peer implementation; 2) act as a contract enforcement mechanism on each peer at runtime; 3) use these mechanisms in an assume guarantee strategy to verify properties of global communication contracts efficiently. We develop a Java package implementation of the PCP, and implement it on top of Java implementations of a set of web services, including a client for the Amazon.com E-Commerce Service (AWS-ECS). A series of experiments is made on these services and show that the PCP can be seamlessly integrated with an application to perform the previous tasks efficiently.

2 PEER INTERFACES AS CONTRACTS

A web service exposes an interface through which other peers can collaborate. Even when not explicitly documented, this interface imposes constraints, acting as a “contract”, that define valid interactions with other peers. In this section, we define the basic building blocks used to specify and analyze these contracts, and illustrate them on a set of examples.

2.1 Defining a Peer Interface

The *peer interface* is a formal description of the form, content, and sequences of operations representing a valid interaction with a given web service.

Form and Content of Operations: We focus on SOAP interactions, where web services interact through the exchange of XML messages. Obviously, not all XML documents are valid input or output messages for a given service; this is why existing description languages such as WSDL refer to DTDs or XML Schemas specifying the precise structure of each message type.

A large fraction of XML Schemas can be defined by a compact notation called the Model Schema Language (MSL) [14]. We use a simplified version of MSL that handles all the portions of XML Schema we found necessary in our case studies, as follows:

$$g \rightarrow b \mid t[g] \mid g_+ \mid g^* \mid g_1, \dots, g_k$$

Here g, g_1, \dots, g_k are all MSL types; b is a basic data type such as Boolean, integer, or string; t is a tag. The MSL type expressions are interpreted as follows: $g \rightarrow b$ specifies a basic type b ; $g \rightarrow t[g_0]$ specifies a type with tag t at its root and with a sub-element whose contents are described by the type expression g_0 ; $g \rightarrow g_1, \dots, g_k$ specifies an ordered sequence, with each of the g_i s listed one after the other. Optionally, the cardinality of any MSL type can be constrained with the use of “+” (the element can appear one or more times) and “*” (the element can appear zero or more times).

Equipped with such a language, one can define a message structure as in the following example:

```
ItemSearchResponse[ Operation[string] ,
  Items[ Item[ ASIN[int] , Price[int] ]* ] ]
```

This expression describes a message called ItemSearchResponse, containing an Operation element holding a string value, and an Items element containing an arbitrary number of Item structures. In turn, each Item contains two integer elements labeled ASIN and Price. The correspondence between MSL and XML documents is straightforward. Such a message type can be used for two purposes: verify that a message follows a given type, and provide rules to generate a valid instance of a given type. These two facets are crucial in the framework we describe in Section 3: the first will be used in a communication component that performs runtime enforcement, while the second is utilized in an alternate *communication stub* that can simulate the outside environment.

Valid Sequences of Messages: 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. Consider for example the state machine in Figure 1(a). It specifies valid sequences of messages for one session of a Loan Approver service. The transitions are labeled either with *!message* or *?message*, denoting sending or receiving of a message, respectively. They provide conditions, called *guards*, made of two parts: the part before

the “/”, called the *guard condition*, states a constraint on the last message processed by the state machine, and the latter part, called the *update condition*, expresses a condition on the current message processed by the state machine. As an example, consider the transition labeled with `!approval[risk.level=high/accept=false]`. This transition is taken only if the *level* element of the last *risk* message is “high”. If this guard condition holds, the peer sends an approval message with the *accept* element set to “false”.

Using the peer interfaces, the 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 [12]. The interactions among peers in such a system are called *conversations*, i.e., a conversation is the sequence of messages exchanged among peers during the execution of a session of the composite web service, recorded in the order they are sent [22]. 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. For example, the following is a conversation of MSL messages that can be generated by the Loan Approval example in Figure 1(a): `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. Linear Temporal Logic (LTL) can be used to specify properties of conversations; a composite web service satisfies an LTL property if all the conversations generated by the service satisfy the property [19], [22], [23].

2.2 Examples

We shall now show, by means of a set of examples, how MSL types and guarded state machines can be used to describe behavioral interfaces.

Loan Approval: In the Loan Approval example, taken from the BPEL 1.1 specification [2], 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 scenario is composed of three individual services (peers): *CustomerRelations*, *LoanApprover* and *RiskAssessor*. Customers make loan requests using the *CustomerRelations* 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.

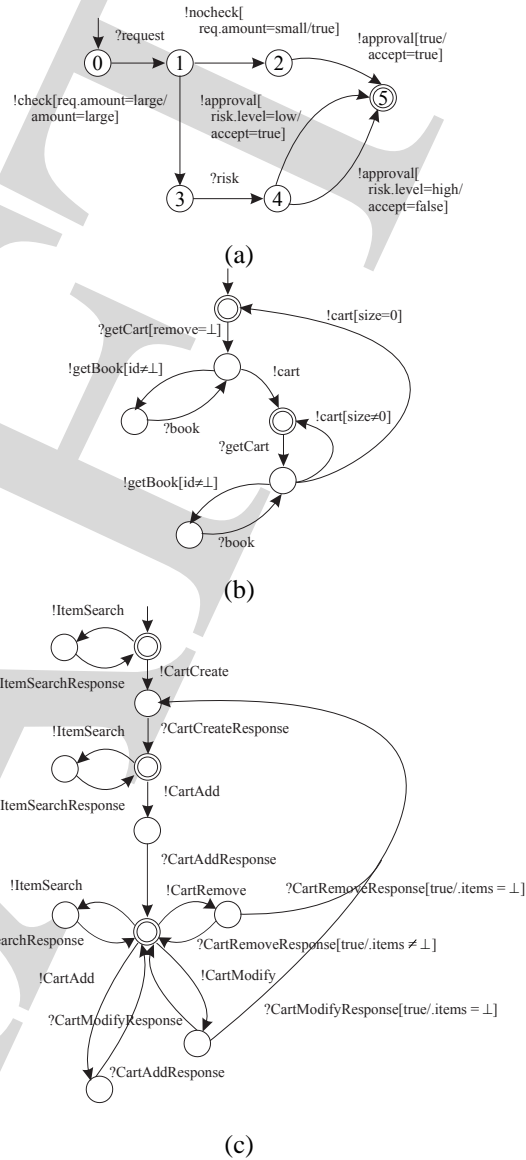


Fig. 1. Peer interfaces for (a) Loan Approver, (b) Duke’s Bookstore Cart, and (c) Amazon ECS Client.

Since this service is a composition of three services, one can specify the peer interfaces with three finite state machines. The state machine shown in Figure 1(a) represents the behavior of the Loan Approver peer.

The Duke’s Bookstore: The Duke’s Bookstore scenario is taken from the example of the same name in the Java EE 5 tutorial [27]. Our study focuses on three of its peers: a *Catalog* service managing the bookstore’s inventory, a *Cart* service taking care of a user’s shopping cart contents and manipulations, and a *FrontEnd* service acting as an interface between a client and the bookstore. For example, the state machine in Figure 1(b) describes the behavior of the *Cart* peer. In this case, the guarded transitions prevent nonsensical requests to be sent by the peers. For example, an item can only be deleted from the cart from a preceding cart listing. In the first state of the state machine, this listing does not precede the *getCart* request, hence the *getCart*

transition in this state is only enabled if the “remove” parameters of the message is unspecified (represented by the \perp symbol).

Amazon ECS: The last example is taken from the Amazon E-Commerce Service (ECS). The ECS is a free service that exposes Amazon’s product data, and provides a number of operations to search for items in Amazon’s catalog, create and manipulate the contents of a *shopping cart* that can eventually be “checked out” for payment directly on Amazon’s web site. Any third-party developer can obtain a free Amazon account and create a web application that interacts in the background with the ECS, through SOAP request-response messages. The documentation of the ECS is also publicly available.

Although each of the shopping cart operations in Amazon’s ECS is intended as a simple request-response pattern of interaction, the semantics of each operation is such that the *ordering* of the operations is also important. For example, it does not make sense (and is actually an error, according to Amazon’s documentation) to add contents to a shopping cart if no shopping cart has first been created. Similarly, one cannot delete an item from a shopping cart before adding anything to it. We produced the behavioral specification shown in Figure 1(c) from the constraints on message sequences stated in Amazon’s online documentation.

2.3 Discussion

While a formal description of message types is generally easy to find, thanks to the widespread use of WSDL, the behavioral part of the specification, in the form of a state machine, is generally not readily available. In the previous examples, we had to peruse the documentation of the respective examples, and generate a state machine specification for each peer by hand; yet, all the information to build that state machine was available. For example, the shopping cart constraints in Amazon ECS can all be found in the online documentation for the web service intended for developers. We argue that this interface specification step does not represent an undue burden for the application developer. First, one can see from the examples that the state machines are in general small, even for real-world examples such as the Amazon ECS. Second, the developer of an application should be aware of the protocol of interaction in order to successfully interact with another peer. For example, the Amazon ECS returns an error message when a client tries to modify an empty cart. Hence, the required use of state machine specifications in our framework simply forces the developer to explicitly provide a specification for a protocol that should anyway be known by the developer, and can be seen as an incentive for a good development practice.

In all systems, the communication among the peers is done 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. Similarly, the Duke’s Bookstore can process multiple

clients, which each use a different shopping cart instance. Finally, the Amazon ECS also serves multiple clients at a time. For the cases where multiple sessions are possible, our behavioral specification is *session modular* —that is, the same behavior is assumed for every session. Therefore, one instance of state machine is required to keep track of each session individually. Virtually all web services exhibit this behavior; as a matter of fact, the opposite is not appealing, since it would require a peer communicating with some service to have information about other transactions occurring in parallel to correctly interact with it.

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 message queues grow without a bound. In fact, model checking conversations of asynchronously communicating finite state machines is an undecidable problem [22]. In Section 5.2 we discuss conditions under which asynchronous communication (with unbounded message queues) can be replaced with synchronous communication without changing the conversation set generated by the composite web service.

3 THE PEER CONTROLLER PATTERN

In this section we present the *Peer Controller Pattern* (PCP), 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.

By separating the application logic from the communication component and by requiring the explicit identification of the peer interfaces, PCP enables the modular verification approach presented in the next section, and, hence, is an instance of design for verification [15].

3.1 Overview of the PCP

The class diagram of the PCP is shown in Figure 2. The main elements of this pattern are provided by the Java package `edu.ucsb.cs.pcp`, which we briefly review below. The classes a developer needs to write are drawn in bold. Other classes are helper classes that can be used as is, without modification. As was discussed previously, the proposed pattern is session based. The application logic of a peer is the same for each session, and is implemented in the `ApplicationThread` class.

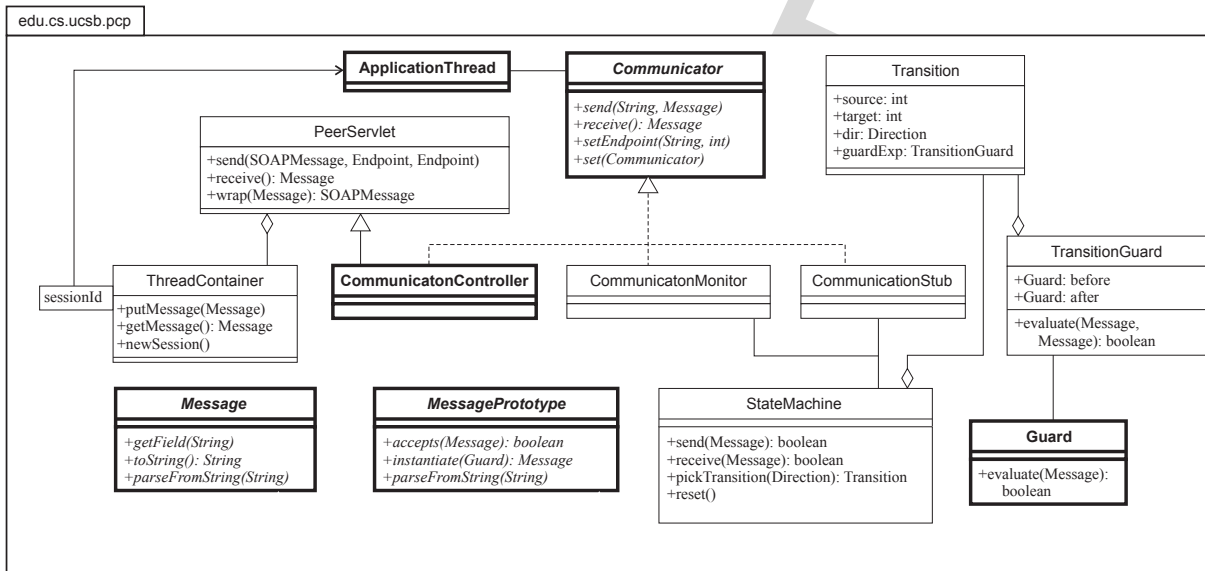


Fig. 2. Class Diagram for the Peer Controller Pattern

Communicator: The Communicator is the interface specifying the main communication functionalities. It provides a send method, which takes a String representation of an endpoint (such as an URL) and a Message to send as arguments. Similarly, the next message in the input queue of a Communicator can be fetched through the receive method.

Communication Controller: This is the first implementation of the Communicator interface and it is the only one performing actual communication with the outside world. Since it is tedious to write such a class, we provide a servlet implementation (PeerServlet) that uses JAXM 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. The helping servlet is associated with a ThreadContainer that contains application thread references indexed by the session identifier. Web services supporting multiple sessions (such as the Amazon ECS) provide a session identifier in their messages that can be used to dispatch an incoming message to the thread indexed with that session ID.

Communication Monitor: This is the second implementation of the Communicator interface and its purpose is to perform *runtime enforcement* of a peer interface by keeping track of the current state of a peer in its state machine. When a message is to be sent by a peer, if the transition is indeed allowed by the state machine, the monitor calls the send method of its linked Communicator and the state machine is updated according to the appropriate transition; otherwise, the monitor can either fail an assertion, or report an error back to its servlet, or discard the message. The monitor filters incoming messages in the same way, preventing messages that are incompatible with the receiver's current state to be consumed.

State Machine: This class is a straightforward implementation of a state machine, whose transitions are labeled with TransitionGuards made of two parts identifying the guard and the update conditions for that transition. The pickTransition method is used for selecting a transition that is consistent with the current state.

Communication Stub: The CommunicationStub is the third implementation of the Communicator interface. It is used during the static interface verification stage and enables verification of each peer in isolation. Like the monitor, the stub updates an internal state machine and filters outgoing messages. However, the stub does not relay these messages to another communicator, nor does it fetch incoming messages from an external source. Instead, calling its receive method makes it randomly pick a receive transition (using the StateMachine's pickTransition method) from its current state, and generate a message that fulfills the guard conditions on this transition. This way, the stub can simulate the outside world based on the behavioral interface definition.

Messages and Prototypes: The peers interact with each other with messages implementing the Message interface. A Message must be constructable from a String representation, and must support a way of fetching values from a query string using the getField method. Depending on the particular message implementation, the syntax and meaning of query strings will vary.

Each message instance is expected to follow a description of its structure called a MessagePrototype. The CommunicationMonitor uses a prototype's accepts method to check whether a given message follows its expected structure. The CommunicationStub's receive method uses the prototype's instantiate method to generate a message that fulfills both the prototype, and the additional conditions expressed by the guard for the transition it picked. This way, the stub can not only choose a transition, but also generate a realistic

instance of a message for that transition.

MSL Package: In addition to these basic classes, we developed a companion package, edu.cs.ucsb.pcp.msl, providing implementations for the Messages and MessagePrototype, and Guard where messages and conditions over messages are expressed in the Model Schema Language described earlier. MSL Guards are conjunctions of individual conditions of the form element = value. Nested elements of a message can be fetched using a period; for example, to specify that the Operation element in an ItemSearchResponse message is not null, one writes !ItemSearchResponse.Operation = null. An additional frontend is provided to convert MSL messages to/from their XML equivalents.

3.2 Semantics of Composite Web Services

In this section we formalize the semantics of the composite web services developed based on the PCP. We use the 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 to simplify our presentation. 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 $W = (M, P_1, \dots, P_k)$. 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 $m \in M$, $sender(m) \in \{P_1, \dots, P_k\}$ and $receiver(m) \in \{P_1, \dots, P_k\}$ denote the peers that send and receive the message m , respectively. 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 $r_1, r_2 \in SP_i$. Each $t \in TP_i$ is in the form of $t = (r_1, !m, r_2)$ where $sender(m) = P_i$, or $t = (r_1, ?m, r_2)$ where $receiver(m) = P_i$.

Note that the peer interface is the semantic model for the state machines used by the CommunicationMonitor and the CommunicationStub. Although we use finite state machines in this semantic model, a different representation can also be used as long as it implements the Communicator interface. Therefore, our verification approach can in principle be used with other languages, such as propositional and first-order temporal logics [19], [24], and choreography languages such as *Let's Dance* [18].

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 $ST = SP_1 \times Q_1 \times \dots \times SP_k \times Q_k$ 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 $e = s_0, s_1, \dots$ is a finite 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 $conv(s_0, s_1, \dots, s_n, s_{n+1})$ is equal to $conv(s_0, s_1, \dots, s_n), m$ 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 $conv(s_0, s_1, \dots, s_n)$ 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 AN ASSUME GUARANTEE VERIFICATION STRATEGY FOR THE PCP

In this section we present a modular verification approach for composite web services implemented based on the Peer Controller Pattern (PCP). Our approach follows assume-guarantee style reasoning where the verification process is

divided to interface conformance and behavior verification phases. The properties verified during the behavior verification phase hold for the composite web service if each peer behaves according to its interface, and this assumption is checked during the interface verification phase.

4.1 Interface Conformance

During the *interface verification* phase we check that each peer implementation conforms to its interface —which defines the order that a peer can send and receive messages. 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 before getting a loan request with a large amount.

Interface conformance can be ensured in two different ways: 1) Statically, by model checking the code of the peer communicating with a stub of the actual communication component, and making sure that every possible execution sequence conforms to the interface. 2) At runtime, by having a component dynamically monitor the messages at the peer's interface and block any non-compliant behavior on-the-fly.

4.1.1 Software Model Checking

A first option is to perform static interface verification directly from a peer's source code. To this end, we use the program checker Java PathFinder (JPF) [13]. JPF is an explicit state model checker for Java. It 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 StateMachine class. The assertions that JPF checks are embedded in the StateMachine 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 exploit two types of modularity based on the PCP: 1) Peer-modular interface verification: We check interface conformance for each peer in isolation. The communication and message stubs provide an environment model for each peer enabling us to isolate them during interface verification. This is crucial for the efficiency and feasibility of interface verification. 2) Session-modular interface verification: We check the peer implementations for a single session. Since, in the PCP, 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. Note that the semantic model we discussed in Section 3.2 is inherently session modular since it considers the semantics of a single session. The communication

controller in the PCP implements the session modular semantics by delegating messages to the appropriate session based on the session ID of each incoming message.

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.

In the program that is given to JPF as input, each send action that the application thread performs is directed to the send method of the state machine class. 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.

Each receive action performed by the application thread is directed to the receive method of the state machine. 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).

4.1.2 Runtime Enforcement

When performed successfully, static verification guarantees the conformance of a peer's implementation to its interface specification. This is due to the fact that all possible execution paths are searched for violations of the state machine. However, there are cases where static verification is not appropriate.

First, the CommunicationStub used to close the environment assumes that the outside world communicates according to the peer interface; therefore, static verification will only find errors that are caused by the peer under study. This assumption is reasonable if all peers involved in a composite service are statically checked for conformance. However, since web services can be dynamically composed at runtime, a specific composition might involve a peer whose conformance to its part of the contract is not known in advance. In such a case, sequences of messages can be sent to a peer that violate the interface defined in the CommunicationStub, and can lead that peer into an

execution path that was not explored by JPF for that very reason.

Second, there exist situations where static verification of a peer is not possible. This can be due to the complexity of the peer’s source code, which makes exhaustive analysis of its execution paths intractable. Moreover, while the `CommunicationStub` closes a peer’s environment with respect to its communications, its global environment might still not be closed (for example, if the peer uses a back-end database system, which is a common scenario), and hence impossible to verify without further abstracting its behavior. For example, the JPF model checker is unable to handle native code and can only verify pure Java programs automatically. So, any peer implementation that interacts with a database or even writes to a file cannot be automatically verified using JPF. Such peer implementations would require manually written stubs that replace native methods. This results in a semi-automated partial search of the state space which can be effective in identifying interface violations but cannot guarantee interface correctness.

A complementary approach to static verification of interface conformance is to dynamically monitor the messages sent and received by a peer, and enforce the interface contract, given as a state machine in the PCP, at runtime. Independently of the previous considerations, such runtime enforcement of peer interfaces can also be seen as a good practice. As we shall see in Section 5.1, a runtime enforcement mechanism incurs only a small processing overhead; moreover, when one of the peers violates an interface, runtime enforcement acts as a barrier, which will prevent a bad message from being processed by its receiver, or even prevent that message from being sent in the first place.

The PCP is designed to support runtime enforcement. In Section 3.1, we described the `CommunicationMonitor`, which, while working transparently like the `CommunicationController` when every peer is well-behaved, can also block incoming or outgoing messages that violate a peer interface.

The runtime enforcement mechanism is complementary to the static interface verification and does not replace the need for the software model checking approach discussed in the previous section. We eliminate as many interface violations as feasible using static interface verification and then use the runtime enforcement mechanism to block the execution of any violations that are not caught during the static interface verification step. Moreover, our runtime enforcement mechanism will not catch an interface error that results from a peer terminating in a non-final state. It is possible to extend runtime enforcement using timeouts to identify such problems, however this is not currently implemented in our runtime enforcement mechanism. Finally, in our current implementation of runtime enforcement, we assume that the peer interface are deterministic. Runtime enforcement of non-deterministic interfaces can be done by keeping track of all possible states of the interface.

As discussed earlier, the handling of runtime interface violations can be done in multiple ways, most of which re-

sult in blocking the message, and aborting the conversation, failing an assertion or terminating the execution of the peer. Compensating interface violations at runtime is out of the scope of this paper. However, even without compensation, runtime enforcement can ensure interface conformance, and isolate a misbehaving peer from affecting other peers.

4.2 Modular Behavior Verification with the PCP

The first phase of our analysis based on the Peer Controller Pattern (PCP) ensures that every peer conforms to its interface. During the *behavior verification* phase, 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 (since we can safely substitute the interface machines as stubs for peers based on the guarantee provided by the interface verification and enforcement steps) and improves the efficiency of the verification and, hence, makes automated verification of realistic web services feasible.

4.2.1 Bounded Behavior Verification

We use the explicit and finite state model checker SPIN [26] 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 use the peer interfaces to characterize the peer behaviors, and ignore the peer implementations (since conformance of peer implementations to the peer interfaces are checked during the interface verification step). Recall that, the peer interfaces are actually finite state machines (such as the ones shown in Figure 1) which are specified using the `CommunicationStub` classes (Figure 2). 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.

The generated Promela specification consists of three parts. 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. 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 processes. 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 `CommunicationStub` (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 (which is enabled if the state is a final state) 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.

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.”

4.2.2 Unbounded Behavior Verification

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. Hence, bounded verification using SPIN can only give web service developers a certain level of confidence—it cannot ensure freedom from violations (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. A technique called the *synchronizability analysis* can be used to identify composite web services where replacing asynchronous communication with synchronous communication does not change the generated conversations [22]. A set of sufficient synchronizability conditions on the control flows of the peer state machines can be checked [22], 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.

We implemented the synchronizability analysis based on the PCP. Given the communication stubs defining the peer interfaces, we automatically check the synchronizability conditions by passing the peer interface definitions to a tool called WSAT, which returns a Boolean verdict whether the service is synchronizable or not [22]. If the composite service is synchronizable, the Promela code with synchronous communication semantics is generated. Note that, when the synchronizability conditions are not met, bounded verification can still be used. Our experiments in Section 5.2 show how synchronizability analysis greatly simplifies the static verification of behavioral properties.

The Promela code generated for a synchronizable service has two differences from our earlier description. 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.

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.

4.3 Discussion

In general neither the interface verification step nor the behavior verification step can be done in a sound and complete manner since automated verification of programs and communicating finite state machines are both undecidable problems. However, our modular assume-guarantee strategy decomposes the verification problem to two independent steps of interface and behavior verification. So, for example, during behavior verification if the given behavior specification is synchronizable, we can achieve sound and complete verification. The verified properties will hold for any implementation as long as the peer implementations obey the interfaces. On the other hand, if a property is violated during behavior verification, that means that there exists an implementation that obeys the interfaces and violates the property. However, if a behavior specification is not synchronizable, then all we can achieve is bounded verification and soundness cannot be guaranteed.

For interface verification, sound and complete verification can only be achieved if the peer implementations are finite state, in which case JPF can terminate its exhaustive search. However, for many systems this may not be the case resulting in a bounded search for interface violations, hence, soundness cannot be guaranteed. On the other hand runtime enforcement mechanism is a complementary approach to static interface verification that blocks interface violations at runtime.

Peer	Time (s)	Memory (gc)	Memory (new)
CustomerRelations	0.70	15	1,586
LoanApprover	0.75	25	2,209
RiskAssessor	0.70	24	1,787

TABLE 1
Interface Verification Performance for the Loan Approver example

5 EXPERIMENTS

We performed a set of experiments on the three examples described in Section 2, corresponding to each step of the verification strategy described in the previous section.

We first tried to verify the whole Loan Approval service in a non-modular fashion 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 and ran all peers in a single JVM. 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.

This demonstrates that a monolithic verification approach is unsuccessful even for small systems. In our experiments we evaluate the appropriateness of the proposed modular verification strategy based on PCP for resolving this problem by answering the following questions: 1) Does the use of CommunicationStub enable peer-modular automated interface verification by closing each peer’s environment and how efficient is it? 2) Does the runtime enforcement mechanism succeed in blocking interface violations at runtime without inducing significant overhead? 3) Does the use of modular verification strategy based on PCP enable verification of properties about the global exchange of messages between peers where monolithic verification fails?

5.1 Interface Conformance

Static Verification: During static interface verification, we used JPF to check whether all the peer implementations in the various examples obey their interfaces as described in Section 4.1.1. The interface verification performance for the three peers of the Loan Approval example is given in Table 1. Memory (gc) gives the number of objects that have been garbage-collected during the run of the program, while Memory (new) is the memory consumption in kilobytes.

For the Duke’s Bookstore exhaustive interface verification with JPF was not readily possible. The bookstore implementation uses a back-end database, both for the management of the catalog and each instance of the shopping

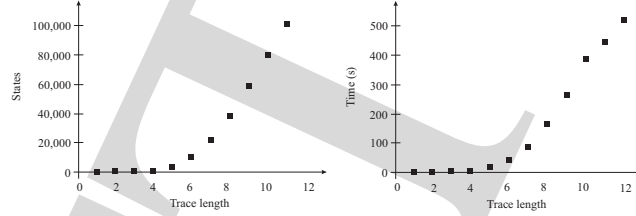


Fig. 3. Number of explored states by JPF (left) and running time (right) for the Cart peer implementation in the Duke’s Bookstore example.

cart. Therefore, it is not possible to close the environment relative to each peer without first providing additional stubs for database communications. The same applies to the Amazon client, which manipulates shopping cart elements based on Amazon’s online catalog.

However, the definition of message prototypes can be used to simulate a back-end database or catalog. Consider for example the prototype for the ItemSearchResponse message in Amazon ECS described earlier. By modeling the item ASIN and price as two bounded integers, the CommunicationStub can generate arbitrary lists of catalog elements that reasonably simulate a query to Amazon’s actual database. The boundedness of primitive data types is readily supported by our framework.

Yet, since the behavioral interface in these two examples contains loops, message traces can be arbitrarily long. In addition, any combination of items and prices is considered as a different message and will be explicitly considered by JPF. For larger examples, such an approach yields an exponential blow-up in the number of states explored.

As an illustration, we computed the running time and number of states visited by JPF for the Cart peer in the Duke’s Bookstore example; the results are shown in Figure 3. These results are computed for data domains of size 4 (i.e. the catalog simulated by the CommunicationStub contains at most 4 different integer values). Even under such a restricted simulation, both figures quickly grow: when stopping the simulation at 12 messages exchanged, it still takes JPF almost 10 minutes to explore the entire state space. In addition, because of the boundedness, both in terms of data domains and trace length, the simulation does not guarantee that the peer follows its behavioral interface for all traces with arbitrary values.

Runtime Enforcement: As explained in Section 4.1.2, for the cases such as Duke’s Bookstore and Amazon ECS where static interface verification is not feasible, runtime enforcement of interfaces can be used as an alternative approach. However, unlike the static verification approach, which does not require any modification to the system during execution, runtime enforcement needs the communication component to monitor the messages at the interface of a given peer, in order to block non-compliant behavior. Therefore, this method incurs an execution overhead which must be taken into account in the total cost of the approach.

In our experiments, we use a simple implementation of an Amazon ECS client, communicating with a stub of the

Amazon ECS simulating its shopping cart manipulations. A web service running remotely on an Apache web server receives the requests, and keeps track of the shopping cart on its side. It returns the appropriate responses corresponding to the requests received by the client. On its side, the Java client is programmed to generate random traces of catalog searches and shopping cart manipulations, and keeps track of the shopping cart contents locally. All generated traces are semantically sound, i.e. they follow the constraints described earlier. For example, a `CartRemove` operation cannot be randomly chosen by the client if the cart is currently empty. All communication is done through standard sockets, using the SOAP protocol over HTTP and XML messages following the same structure as the ECS documentation prescribes.

100 random traces of 1,000 requests each were produced by the client. For all these traces, we compared run time and memory consumption of a plain implementation of the ECS client using HTTP sockets, with the same implementation, sending its messages through the Communicator interface before relaying them to the socket. Java’s pseudo-random number generator was used with predefined seeds so that the same random traces could be reproduced for both implementations. All experiments were conducted in the Eclipse IDE under Windows XP, on an Intel Quad Xeon PC running at 2.66 GHz. Running times and memory consumption were recorded as reported by the Java System class methods. Figures 4 and 5 show the distribution of memory and time overhead incurred by the use of the CommunicationController over the 100 message traces. The same experiment has been performed on Java peer implementations for the two other scenarios shown in this paper. The results are summarized in Table 2.¹

A first metric to analyze is memory consumption. The interface specification for each peer is *session modular*: this entails that every session requires its own instance of the state machine to keep track of messages pertaining to that particular session. According to the formal definition of the PCP in Section 3, each peer controller keeps in memory the last instance of each message, along with its data parameters; it also stores an explicit representation of the finite state machine, including its states, transitions and guards over transitions. The memory consumed averages 24.1 kilobytes per Communicator.

A second metric we measured was execution time overhead, using the same experimental setting. Again, one can see that, due to the small size of the state machines, processing overhead is kept to a minimum; whether a message is blocked or let through is irrelevant. In some cases, the time difference is even below the JVM’s clock resolution.

Moreover, session modularity entails that this processing cost per message is constant for a given state machine

1. The Loan Approver example was statically verified with Java PathFinder, and therefore in theory no runtime enforcement is required. However, we also measured the overhead of a runtime enforcement monitor on peers from that example, to have a wider range of experimental data.

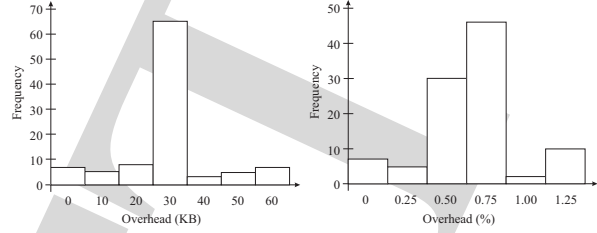


Fig. 4. Absolute (left) and relative (right) RAM overhead for the Amazon ECS Java client using the PCP.

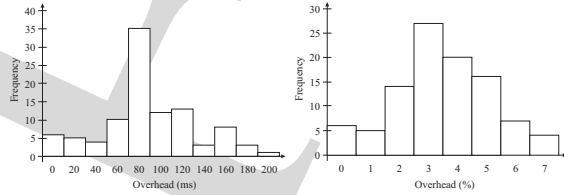


Fig. 5. Absolute (left) and relative (right) time overhead for the Amazon ECS Java client using the PCP.

and does not depend on the number of parallel sessions. We recall that sessions are not inter-related—that is, the correctness of each session with respect to the peer interface depends only on the messages exchanged in that session. Therefore, each parallel session on a peer requires one instance of the CommunicationMonitor, running independently of any other instance. The only overhead associated with multiple sessions is the fact that each incoming message must be dispatched to one (and only one) monitor instance, depending on the session it belongs to. However, this filtering must be done by the server code anyway, either by passing a session ID as an XML element in the messages, or as a cookie in the HTTP request that carries this message. The associated cost is negligible, and hence the presence of concurrent sessions is irrelevant to the performance overhead per session.

These figures allow us to conclude that runtime enforcement is indeed an adequate substitute to static verification for achieving interface conformance. In situations where static verification is not possible, the addition of a verification layer at runtime prevents violations of the peer interface from occurring, while imposing a reasonable overhead. Typical network latencies for major Internet service providers are more than 1,000 times larger than the overhead associated with the CommunicationMonitor.²

5.2 Behavior Verification

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 results are shown in

2. <http://www.verizonbusiness.com/about/network/latency/>

Peer	Memory (KB)	Time (μ s)
CustomerRelations	16.63 (3 %)	0.000 (0 %)
LoanApprover	33.84 (6 %)	0.000 (0 %)
RiskAssessor	35.18 (6 %)	0.000 (0 %)
Catalog	35.18 (6 %)	0.020 (2 %)
Cart	33.84 (6 %)	0.000 (0 %)
FrontEnd	16.54 (3 %)	0.000 (0 %)
Amazon ECS Client	23.53 (1 %)	0.795 (3 %)

TABLE 2

Average overhead, both absolute and relative, for PCP implementations of the three examples.

Scenario	Memory (MB)	Time (s)
Loan Approval	2.302	0.015
Duke's Bookstore	2.302	0.031
Amazon ECS	2.40	0.039

TABLE 3

Behavior verification performance

Table 3. 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.

We applied the same process to the Duke's Bookstore example. We used SPIN to verify the property "Whenever a *getCart* message is sent, eventually a *cart* message is received", which was shown to be true in less than one tenth of a second. Again, since we were able to verify that this model fulfills the synchronizability conditions, the results obtained with a queue size of 1 imply that the same results hold for unbounded queues as well. Finally, we ran a similar experiment on the Amazon ECS, with the property "A *CartModify* message cannot be issued while the cart is empty" and obtained similar results.

We shall stress that these properties cannot be proved by considering a single component, since the interface specifications for both ends of the communication must be compatible to ensure any communication at all. Nothing prevents the interface at one end to stipulate sending message A, while the interface at the other end expects message B. Only through model checking of the *composite* system can we be sure that the properties like those above are indeed fulfilled.

5.3 Discussion

Our experiments show that the modularity in the verification process based on the PCP improves the efficiency of the verification of composite web services significantly. Simple properties of conversations, which were impossible to check with JPF, now become verifiable in fractions of a second, thanks to the modular decomposition provided by the PCP. 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.

6 RELATED WORK

Preliminary results from this paper were reported by Betin-Can et al. [10]. This paper extends these earlier results by 1) discussing the modular verification approach we developed based on the Peer Controller Pattern and identifying and differentiating the types of modularity used in our analysis (i.e., assume-guarantee reasoning used in the behavior verification and the peer and session modularity used in interface verification), 2) extending the peer controller pattern for supporting runtime enforcement and presenting a new runtime enforcement mechanism to guarantee interface conformance at runtime, 3) providing an extended experimental evaluation of the proposed modular verification approach, and 4) providing an extended discussion on related work below.

Behavior Specification: A number of approaches have been proposed for describing behaviors of web services including standards such as BPEL [2] or WSFL [28], a language called DyLOG based on agent theory for reasoning about conversations between services [4], extensions of temporal logics for specifying sequential and data dependencies between messages [19], [24], and a graphical choreography description language called *Let's Dance* [18].

State machines have been extensively studied for specifying behaviors of web services. Fu et al. [22] show that other behavioral descriptions (such as BPEL) can be translated to state machines. Berardi et al. [8] use state machines as behavioral contracts. Unlike our work, their goal is to automatically synthesize composite web services. Benatallah et al. [7] 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.

None of the earlier results mentioned above, address verification of interactions among web services implemented as Java programs. Although analysis of specifications written in languages such as BPEL is an important research direction, it does not eliminate the need to verify web service implementations that are developed directly in a

programming language such as Java without using such specification languages.

Static Verification: There has been earlier attempts at applying model checking techniques to verification of composite web services [3], [21], [22], [29]–[31], [34]. 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.

De Alfaro et al. [17] introduce interface automata to formalize temporal aspects of software component interfaces. Later, this formalization is used for specifying interfaces as a set of constraints and algorithms for interface compatibility checking is presented by Chakrabarti et al. [11]. We use finite state machines to specify interfaces and our approach to interface checking can be seen as a special case of the interface compatibility checking. However, unlike Chakrabarti et al., we do not require each method to be annotated with interface constraints and also our goal is to verify both the controller behavior and conformance to interface specifications.

Rajamani et al. [32] 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 that the actual implementation of the service be modeled in the same formalism as the specification, so that the two can be compared. The PCP works directly on top of the actual implementation of a service and does not require this implementation to be formalized in any way. Moreover, their approach does not separate the interface and the behavior verification steps.

An alternate approach is followed by Rouached et al. [33], who map BPEL constructs into equivalent Event Calculus (EC) expressions. Properties can then be proved statically against the resulting expression, which then apply directly to the original BPEL code as well. Properties can also be monitored at runtime; in such a case, the running web service generates events which are used to evaluate the EC expression; should the expression evaluate to false, a violation of the specification is discovered at runtime. However, this approach is limited to web services composed with BPEL. In addition, the approach does not suggest the use of a behavioral specification as a stub to simulate the external environment.

Modular Verification and Environment Generation: Flanagan et al. [20] present a thread-modular reasoning and verifies each thread separately with respect to safety properties. This is similar to peer-modular reasoning used in our approach, however, in their approach the effects of other threads are modeled as environment assumptions whereas we use stubs and drivers to reflect these effects. Besides, we combine peer-modular reasoning with assume-guarantee reasoning and check the peer behavior against the interface contract and leave the property verification to the behavior verification phase.

Godefroid et al. [16] convert an open reactive program into to a closed program by inserting nondeterminism into

the code and eliminating procedure arguments. Unlike this work, we have restrictions on the environment interactions caused by controllers via interfaces. Stoller [35] transforms distributed programs communicating with RMI into one program for model checking. Unlike this centralization approach, we apply peer-modular model checking, decouple the remote processes, and reduce the state space. The techniques presented by Tkachuk et al. [36] generate environments for software components by using side effect and points-to analyses. Although the techniques we discuss for isolating peers are similar to these, we base our techniques on the behavioral interfaces that are supported by the Peer Controller Pattern.

Use of a design pattern to achieve modular verification of concurrent programs has been proposed before [9]. However these earlier results focus on concurrent threads accessing shared data using synchronization statements. In this paper, we are focusing on interactions among web services and asynchronous communication; therefore, both the application domain and the underlying semantic model are different.

Development tools like JMock allow developers to create “mock objects” for Java classes, that are intended to simulate the behavior of some external component in order to test a class that uses that component. The Communication-Stub class defined in our Peer Controller Pattern fits that definition. The process of providing an external object to a class under test is called *dependency injection*. However, mocks still need to be implemented manually, while in the PCP the formal definition of the peer interface is sufficient for the stub to automatically simulate the environment without further manual intervention.

Runtime Enforcement: A runtime monitoring tool called JavaMOP has been presented by Hills and Roşu [25]. By treating a servlet as a special type of Java program, JavaMOP can be used to perform runtime enforcement properties. However, the state machines in JavaMOP do not handle guards and conditions on data parameters; to this end one must use PTCaRet, an input language derived from temporal logic.

Halle et al. [23] present a runtime monitoring tool for Ajax applications, using temporal logic as the specification language; however, the tool only applies for client-side verification in web browsers, and not to web services in general. The Runtime Monitoring Language (RTML) has been used to specify and monitor properties of web services [5]. Also systems that turn design-time assertions on the behavior of a service into runtime checks have been developed before [6]. None of these works use runtime enforcement as a way to complement and improve static behavior verification. Using the same interface specification, our Peer Controller Pattern can be used either as a runtime monitor, or as a communication stub to perform static verification of a web service.

7 CONCLUSION

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 [22], we verified conversations of composite web services with respect to unbounded queues and improved the efficiency of the behavior verification. We showed that bounded 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, they improve the efficiency of the interface verification. We also presented a runtime enforcement mechanism that can prevent interface violations without incurring significant overhead. Another benefit of the presented approach is the explicit specification of the peer interfaces, which can be used to improve interoperability.

REFERENCES

- [1] Amazon.com Q4 2007 earnings call transcript.
- [2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services, version 1.1, 2003.
- [3] J. Arias-Fisteus, L. S. Fernández, and C. D. Kloos. Applying model checking to BPEL4WS business collaborations. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 826–830, 2005.
- [4] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about interaction protocols for web service composition. *Electr. Notes Theor. Comput. Sci.*, 105:21–36, 2004.
- [5] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *Proceedings of the 2006 IEEE International Conference on Web Services*, pages 63–71, 2006.
- [6] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Software*, (6):219–232, 2007.
- [7] B. Benatallah, Q. Z. Sheng, A. H. H. Ngu, and M. Dumas. Declarative composition and peer-to-peer provisioning of dynamic web services. In *Proceedings of the 18th International Conference on Data Engineering*, pages 297–308, 2002.
- [8] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proceedings of the First International Conference on Service-Oriented Computing*, pages 43–58, 2003.
- [9] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 248–257, 2004.
- [10] A. Betin-Can, T. Bultan, and X. Fu. Design for verification for asynchronously communicating web services. In *Proceedings of the 14th international conference on World Wide Web*, pages 750–759, 2005.
- [11] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *Proceedings of the 14th International Conference on World Wide Web*, pages 148–159, 2005.
- [12] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.
- [13] G. Brat, K. Havelund, S. Park, and W. Visser. Java Pathfinder: Second generation of a Java model checker. In *Proceedings of the Workshop on Advances in Verification*, 2000.
- [14] A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL: a model for W3C XML Schema. In *Proceedings of the 10th International World Wide Web Conference*, pages 191–200, 2001.
- [15] T. Bultan and A. Betin-Can. Scalable software model checking using design for verification. In *Verified Software: Theories, Tools, Experiments, Proceedings of the IFIP TC 2/WG 2.3 Conference*, volume 4171, pages 337–346, 2005.
- [16] C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically closing open reactive programs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 345–357, 1998.
- [17] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, 2001.
- [18] G. Decker, J. M. Zaha, and M. Dumas. Execution semantics for service choreographies. In *Proceedings of the 3rd International Workshop on Web Services and Formal Methods*, pages 163–177, 2006.
- [19] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven web services. In *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 90–99, 2006.
- [20] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proceedings of the 10th International SPIN Workshop*, pages 213–224, 2003.
- [21] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 152–163, 2003.
- [22] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Trans. Software Eng.*, 31(12):1042–1055, 2005.
- [23] S. Hallé and R. Villemaire. Browser-based enforcement of interface contracts in web applications with BeepBeep. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 648–653, 2009.
- [24] S. Hallé, R. Villemaire, and O. Cherkaoui. Specifying and validating data-aware temporal web service properties. *IEEE Transactions on Software Engineering*, 35(6), November/December 2009.
- [25] M. Hills and G. Rosu. A rewriting approach to the design and evolution of object-oriented languages. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 827–828, 2007.
- [26] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
- [27] E. Jendrock, J. Ball, D. Carson, I. Evans, S. Fordin, and K. Haase. *The Java EE 5 Tutorial, Third Edition*. Prentice Hall, 2006.
- [28] F. Leymann. Web services flow language (WSFL), version 1.0, 2001. <http://www306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>.
- [29] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing interacting BPEL processes. In *Proceedings of the 4th International Conference on Business Process Management*, pages 17–32, 2006.
- [30] S. Nakajima. Verification of web service flows with model-checking techniques. In *Proceedings of International Symposium on Cyber Worlds*, page 0378, 2002.
- [31] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th International World Wide Web Conference*, pages 77–88, 2002.
- [32] S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 166–179, 2002.
- [33] M. Rouchéd, O. Perrin, and C. Godart. Towards formal verification of web service composition. In *Proceedings of the 4th International Conference on Business Process Management*, pages 257–273, 2006.
- [34] B.-H. Schlingloff, A. Martens, and K. Schmidt. Modeling and model checking web services. *Electr. Notes Theor. Comput. Sci.*, 126:3–26, 2005.
- [35] S. D. Stoller and Y. A. Liu. Transformations for model checking distributed Java programs. In *Proceedings of the 8th International SPIN Workshop*, pages 192–199, 2001.
- [36] O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu. Automated environment generation for software model checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 116–129, 2003.