Modeling Interactions of Web Software*

Tevfik Bultan
Department of Computer Science
University of California
Santa Barbara, CA 93106
bultan@cs.ucsb.edu

Abstract

Modeling interactions among software components is becoming increasingly important for analyzing behavior of web software. This is especially true for the area of web services where distributed interactions across organizational boundaries are the norm. In this paper, I will discuss three visual formalisms for modeling interactions: 1) Collaboration Diagrams, 2) Message Sequence Charts, and 3) Conversation Protocols. I will discuss the differences and similarities among these representations and identify two problems: realizability and synchronizability.

1. Introduction

Assume that a group of organizations decide to integrate their online businesses. One (or more) of them will provide a front end that enables user interaction via Web. However, in order to serve a user request, this front end will send and receive messages from software components that reside in different organizations. The typical example for this type of scenario is a travel agency that wants to integrate its online business with other organizations such as an airline, a hotel reservation system or a car rental company. The goal of web services standards and technologies that have been developed in recent years is to facilitate this type of business to business integration.

There are several interesting features of this type of distributed systems:

- First, some organizations may not want to share internal structures of their software components with other organizations they intend to do business with. This type of decoupling requires development of standardized and rich interface specification languages that will enable integration of software components that may be written using different implementation platforms. For example, Web Services Description Language (WSDL) [17] specifications and abstract Business Process Execution Language for Web Services (BPEL) [2] processes can be used to specify high level behavioral interfaces for software components.
- It is difficult to model/understand/analyze the global behavior of such systems since no single party may know the full details of all the components in such a system. The only global behavior that can be observed by different parties is the messages exchanged among different components. Local executions of different components are hidden behind the abstract interfaces that are visible from outside. This requires development of high-level specification mechanisms which do not require details of local behavior. For example, Web Service Choreography Description Language (WS-CDL) [16] enables specification of global interactions among multiple web services without requiring the availability of local behavior specifications.
- For this type of distributed systems it might be worthwhile to model the interactions among different software components before the components are written. This type of top-down design may help different organizations to better coordinate their development efforts.
- In the absence of detailed models for the distributed components that participate in such a system, desired behaviors have to be specified as constraints on the interactions among different components, since the interactions are the only observable global behavior.

Finding appropriate interaction modeling formalisms that address the above issues is an important problem for web software development. In this paper, I will discuss three different visual formalisms as candidates for modeling interactions among web software: 1) Collaboration Diagrams, 2) Message Sequence Charts, and 3) Conversation

*This work is supported by NSF grants CCF-0614002 and CCF-0341365.

Protocols. I will discuss the differences and similarities among these representations. I will also discuss two problems, called realizability and synchronizability, which arise in specification and analysis of interactions.

2. Modeling Interactions

Consider the following extremely simple example. A customer sends a purchase order to a vendor. After receiving the order, the vendor sends a shipment request to a warehouse. If the ordered item is in stock, the warehouse sends the shipment information to the vendor and the vendor sends a bill to the customer. If the ordered item is not in stock, the warehouse sends an out-of-stock message to the vendor and the vendor sends a message to the customer indicating that the ordered item is not currently available. In this example the customer could represent a web-based front end, and the vendor and the warehouse could be software components that run on different organizations' servers.

Let us assume that a distributed web application consists of a set of peers which communicate with messages. We can model the above example with three peers, Customer, Vendor, Warehouse, and six message types, *order*, *shipReq*, *shipInfo*, *outOfStock*, *bill*, and *notAvailable*. The interactions of the above example can be described as follows: Customer orders a product by sending the *order* message to the Vendor. After receiving the *order* message the Vendor sends the *shipReq* message to the Warehouse. After receiving the *shipReq* message the Warehouse sends either the *shipInfo* or the *outOfStock* message to the Vendor. If the Vendor receives the *shipInfo* message from the Warehouse, it sends the *bill* message to the Customer. If the Vendor receives the *outOfStock* message from the Warehouse, it sends the *notAvailable* message to the Customer.

Although it may not be the case for this very simple example, the description of dependencies among messages in a distributed system can easily become very complex and hard to understand. Below, I will discuss three visual formalisms for modeling interactions which are helpful in the specification and understanding of such dependencies.

2.1. Collaboration Diagrams

A Collaboration Diagram (called communication diagram in UML [15]) consists of a set of peers, a set of links among the peers showing associations, and a set of message send events among the peers. Let us assume that each peer is an active object with its own thread of control. Each message send event is shown by drawing an arrow over a link denoting the sender and the receiver of that message.

In a collaboration diagram each message send event has a unique sequence label. These sequence labels are used to declare the order the messages should be sent. Each sequence label consists of a (possibly empty) string of letters followed by a numeric part. The numeric ordering of the sequence numbers defines an implicit total ordering among the message send events with the same prefix. In addition to the implicit ordering defined by the sequence numbers, it is possible to explicitly state the events that should precede an event e by listing their sequence labels (followed by the symbol "/") before the sequence label of the event e.

Figure 1 shows the interactions among the peers of the example discussed above using two collaboration diagrams. Each collaboration diagram shows one execution scenario. Both scenarios are acceptable interactions for this example. Note that the messages in each collaboration diagram in Figure 1 have a total ordering since their sequence labels share the same prefix (the empty prefix). So the messages have to occur based on the numeric ordering in their sequence labels. The set of interactions described by these collaboration diagrams are:

- *order shipReq shipInfo bill*
- *order shipReq outOfStock notAvailable*

2.2. Message Sequence Charts

A Message Sequence Chart (MSC) (called sequence diagram in UML) consists of a set of peers and a total ordering of a set of send and receive events for each peer. For each message the send event at the sender is connected to the corresponding receive event in the receiver with an arrow. The set of interactions specified by an MSC is all possible send and receive event sequences that do not violate the local ordering of send and receive events of any peer.

Figure 2 shows the interactions for the running example using two MSCs. The set of interactions specified by these MSCs are:

- *!order ?order !shipReq ?shipReq !shipInfo ?shipInfo !bill ?bill*
- *!order ?order !shipReq ?shipReq !outOfStock ?outOfStock !notAvailable ?notAvailable*

where, given a message m, we use $!m$ to denote a send event that sends the message m, and $?m$ to denote a receive event that receives the message m.

Note that, the interaction language specified by collaboration diagrams and MSCs are different. Collaboration diagrams give an ordering of only send events whereas MSCs give an ordering of both send and receive events. Although this does not make a big difference for this example, one can construct examples demonstrating that collaboration diagrams and MSCs are not compatible [3].

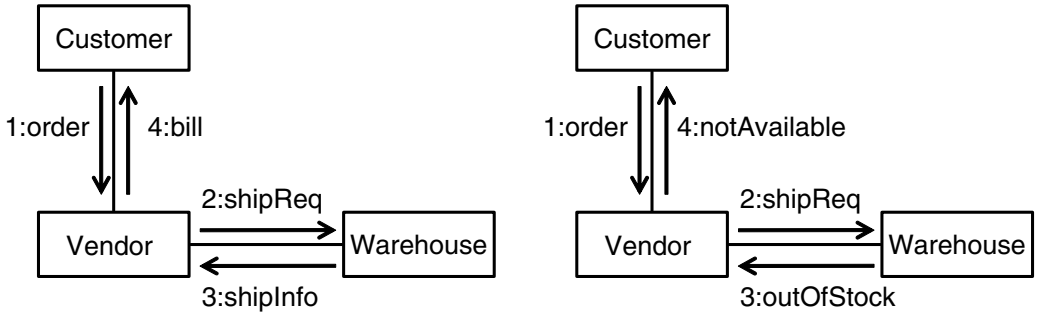

Figure 1. Collaboration diagrams.

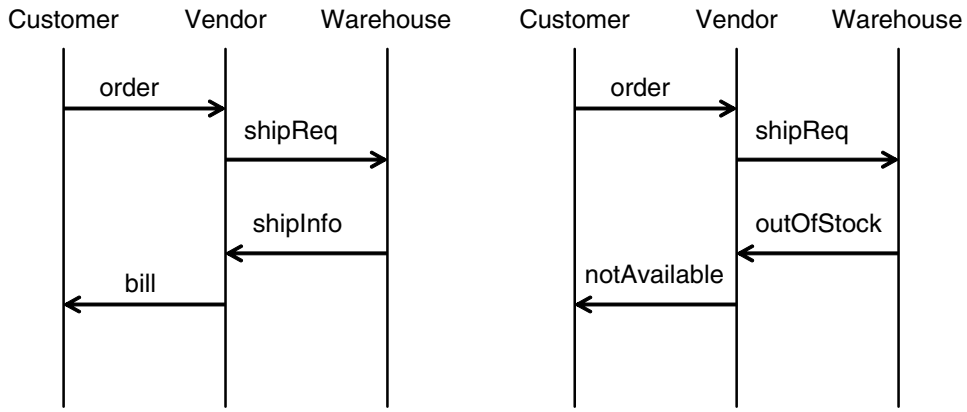

Figure 2. Message Sequence Charts (MSC).

2.3. Conversation Protocols

A conversation protocol is a finite state automaton used for specification of interactions among peers participating to a composite web service [7]. The alphabet of a conversation protocol is the set of messages that are exchanged among the peers and each transition in a conversation protocol corresponds to a message send event. Hence, each string recognized by a conversation protocol corresponds to a sequence of message send events. Like collaboration diagrams, conversation protocols also specify the ordering of only message send events.

Figure 3 shows the interactions for the running example using a conversation protocol. Note that, each transition in the protocol corresponds to a message send event. In order to make conversation protocols easier to understand, we write the sender and the receiver of each message before the message name. The set of interactions specified by this conversation protocol are:

- *order shipReq shipInfo bill*
- *order shipReq outOfStock notAvailable*

Note that the interactions specified by the collaboration diagrams in Figure 1 and the conversation protocol in Figure 3 are identical.

2.4. Comparison and Extensions

Collaboration diagrams and conversation protocols both only specify the ordering of message send events. In other words, for a collaboration diagram or a conversation protocol specification, the ordering of the message receive events is like a "don't care" condition—there is no constraint on how the receive events are ordered as long as they do not conflict with the specified ordering of the send events. MSCs on the other hand specify the ordering of both message send and receive events.

Another interesting characteristic of MSCs is the fact that they specify local orderings of the send and receive events for each peer. According to an MSC specification, any global ordering which obeys the local orderings of all the peers is an acceptable interaction. On the other hand, collaboration diagrams and conversation protocols directly specify the global ordering of message send events.

All three formalisms described above can be extended in several ways. One possible extension is the addition of conditions or guards which enable specification of conditional behavior. Such conditional behaviors may depend on the contents of the exchanged messages [9]. Also auxiliary variables can be used to store message values.

An important issue in modeling interactions is the type of communication used among the peers. Messages can be transmitted using synchronous or asynchronous communication. During synchronous message transmission, the sender and the receiver must execute the send and receive events simultaneously (similar to a remote procedure call). During an asynchronous message transmission, the send event appends the message to the input queue of the receiver, where it is stored until receiver consumes it with a receive event. Asynchronous communication is important for building robust distributed systems. Communication delays and pauses in availability of other components are less likely to effect the performance of a distributed system if asynchronous communication is used instead of synchronous communication. Asynchronous communication is supported by message delivery platforms such as Java Message Service (JMS) [12] and Microsoft Message Queuing Service (MSMQ) [13].

Note that, since collaboration diagrams and conversation protocols do not specify when a receive event for an asynchronous message will be executed, the type of behaviors that can be specified with these formalisms and the type of behaviors that can be specified with MSCs can be different. In fact, one can show that there are conversation protocols which specify interactions that cannot be specified by any MSCs and there are MSCs which specify interactions that cannot be specified by any conversation protocol [10]. Hence, the expressive power of conversation protocols and MSCs are not comparable.

On the other hand, conversation protocols are more powerful than the collaboration diagrams in terms of the interactions they can specify. Given a set of collaboration diagrams it is always possible to construct a conversation protocol which specifies the same set of interactions [3].

An MSC Graph [1] is a finite state automaton where each node of the graph (i.e., each state of the automaton) is associated with an MSC. The interactions specified by an MSC graph corresponds to all the interactions that are specified by the MSCs obtained by concatenating the MSCs along each path of the MSC graph. MSC Graphs can be used to specify interactions that cannot be specified by MSCs. As with the MSCs, expressive power of conversation protocols and MSC Graphs are not comparable [10].

Collaboration diagrams can similarly be extended to Collaboration Diagram Graphs (CDG) [3] where each node in a CDG is a collaboration diagram. The set of interactions specified by a CDG is obtained by concatenating the collaboration diagrams on each path of the CDG. CDGs can express interactions that cannot be expressed by collaboration diagrams [3]. Interestingly, it is possible to show that CDGs and conversation protocols are equivalent in terms of their expressive power. Given a CDG one can construct a conversation protocol which specifies exactly the same set of interactions specified by the CDG and given a conversation protocol one can construct a CDG which specifies exactly

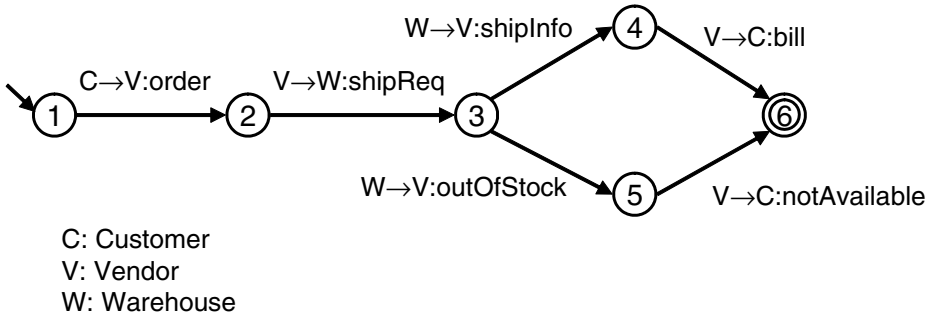

Figure 3. A Conversation Protocol.

the same set of interactions specified by the conversation protocol [3].

It is possible to formalize the semantics of collaboration diagrams [3], MSCs [1] and conversation protocols [7]. Based on these formal semantics one can automatically analyze and verify properties of interaction specifications. Hence, interaction specifications can be fixed before they are used during the development of distributed components which will implement them.

3. Realizability and Synchronizability

As we discussed above, collaboration diagrams, message sequence charts and conversation protocols can be used to specify interactions of distributed web applications. Given such a specification, one important question is if the set of interactions specified by the given specification can be implemented. It is possible that the dependencies among the message send events in a given specification may not implementable. This problem is called the *realizability problem*. If a specification is realizable, then there exists an implementation which generates the same set of interactions specified by the specification.

In order to formalize the realizability problem one has to identify a formal computational model for the peer implementations. One basic approach is to use finite state machines which communicate with asynchronous or synchronous messages as the computational model for the peer implementations and then to investigate the realizability problem on this model. For example, Figure 4 shows the peer implementations for the running example. Note that each peer is specified as a finite state machine. Each transition corresponds to either a message send event or a message receive event. These peer implementations generate the interactions specified in Figures 1, 2 and 3.

Realizability problem has been investigated for collaboration diagrams [3], MSCs [1] and conversation protocols [7]. The realizability problem for collaboration diagrams and collaboration diagram graphs can be reduced to the realizability problem of conversation protocols since they can be converted to conversation protocols [3]. However, existing realizability analyses for MSCs and conversation protocols have different characteristics due to the differences between these specification formalisms [10].

Using a simple implementation model (such as finite state machines) makes it possible to use automated verification techniques (such as model checking [4]) for analyzing interaction properties [14, 6, 8, 5]. For example, in order to make sure that a Customer does not indefinitely wait for a reply after sending an order, we may want to verify that whenever an *order* message is sent, eventually either a *bill* message is sent or a *notAvailable* message is sent. Such properties about interactions can be specified as temporal logic properties [7]. If synchronous communication is the only communication mechanism, and peer implementations are modeled as finite state machines, existing model checking tools (such as [11]) can be used to verify such properties. However, if asynchronous communication is used, then verification complexity can increase significantly with increasing size of the message queues. In fact, the verification problem is undecidable for unbounded message queues.

Synchronizability analysis is a technique for identifying systems whose interactions can be verified efficiently [6, 10]. A distributed system is synchronizable if the set of interactions it generates is the same when asynchronous communication is replaced with synchronous communication. Hence, if a system is synchronizable, during the verification of its interactions we can use synchronous communication instead of asynchronous communication. Since interaction behavior is not supposed to change, the verification results we obtain using synchronous communication will hold for asynchronous communication.

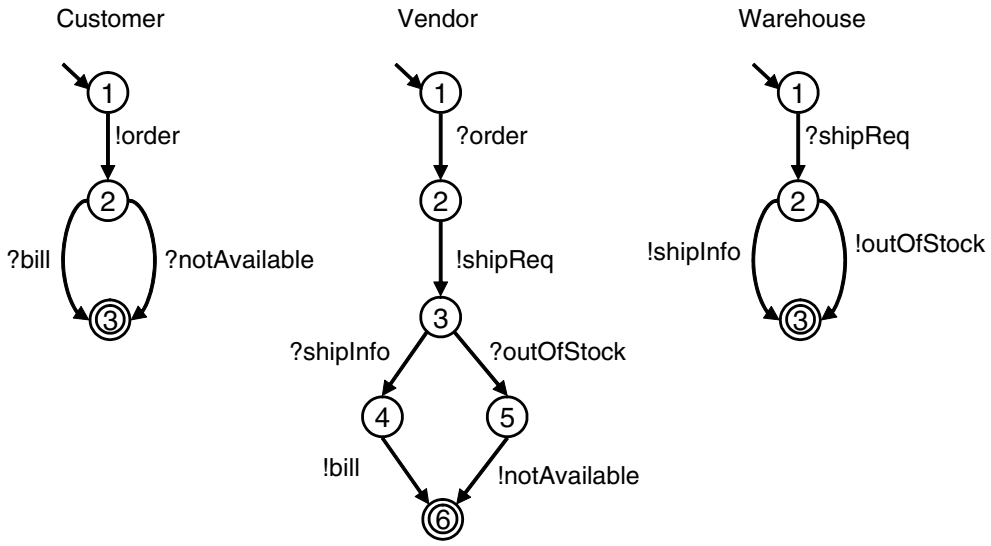

Figure 4. Peer implementations.

4. Conclusions

Specification and analysis of interactions in distributed web applications and services is an important problem. There are visual formalisms such as collaboration diagrams, message sequence charts and conversation protocols that can be used for specification of such interactions. Using these formalisms interactions can be specified precisely. Based on these precise specifications of interactions it is possible investigate analysis problems such as realizability and synchronizability, and to use automated verification techniques such as model checking to verify properties of interactions.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *Proc. 28th Int. Colloq. on Automata, Languages, and Programming*, pages 797–808, 2001.

[2] Business process execution language for web services (BPEL), version 1.1. `http://www.ibm.com/developerworks/library/ws-bpel`.

[3] T. Bultan and X. Fu. Realizability of interactions in collaboration diagrams. Technical report, Computer Science Department, University of California, Santa Barbara, September 2006.

[4] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[5] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based analysis of obligations in web service choreography. In *Proc. IEEE Int. Conf. on Internet and Web Applications and Services*, 2006.

[6] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. 13th Int. World Wide Web Conf.*, pages 621–630, May 2004.

[7] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and analysis of reactive electronic services. *Theoretical Computer Science*, 328(1-2):19–37, November 2004.

[8] X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of web service compositions. In *Proc. 16th Int. Conf. on Computer Aided Verification*, volume 3114 of *LNCS*, pages 510–514, July 2004.

[9] X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. *International Journal of Web Services Research*, 2(4):68–93, 2005.

[10] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, December 2005.

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

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

[13] Microsoft Message Queuing Service. `http://www.microsoft.com/windows2000/technologies/communications/msmq/d%efault.mspx`.

[14] S. Nakajima. Model checking verification for reliable web service. In *Proc. of the 1st International Symposium on Cyber Worlds (CW 2002)*, pages 378–385, November 2002.

[15] UML 2.0 superstructure specification. `http://ww.uml.org/`, October 2004.

[16] Web Service Choreography Description Language (WS-CDL). `http://www.w3.org/TR/ws-cdl-10/`, 2005.

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