A Lightweight Dynamic Conversation Controller for E-Services

Harumi Kuno and Mike Lemon

Hewlett-Packard Laboratories
1501 Page Mill Road, MS 1U-14
Palo Alto, CA 94304-1126
harumi_kuno@hp.com and mike_lemon@hp.com

Abstract

As services become more loosely coupled and increasingly autonomous, heterogeneous distributed services should be able to discover and converse with each other dynamically, with or without human intervention. Current paradigms of service interaction require service developers to hardcode their logic to adhere strictly to pre-defined conversation policies. We propose here a mechanism for a Conversation Controller that can free service (and, to some extent, client) developers from having to couple business and conversation logic. We also describe our implementation of a prototype Conversation Controller. The Conversation Controller can direct and track spontaneous conversations between services and clients, thus enabling services to carry out an entire conversation without the service developers having to implement any explicit conversation control mechanisms.

1. Introduction

Electronic Commerce is driving distributed computing to evolve from intra-enterprise application integration, where application developers work together to develop and code to agreed upon method interfaces, to inter-enterprise integration, where E-Services may be developed by independent enterprises with completely disjoint computing infrastructures. E-Services should be inherently easy to integrate and should facilitate dynamic brokering and composition[6,8]. This requirement leads to certain technical challenges: E-Services must be able to "reflect" their functionality and APIs (or their equivalent); they must be able to communicate and exchange business data in a meaningful way; and finally, E-Services must have some degree of flexibility and autonomy with regard to their interactions.

For example, Figure 1, below, depicts an E-Service marketplace with two different enterprises. Suppose that a client service in one enterprise (Enterprise B) discovers some kind of storefront service in the other enterprise (Enterprise A). These services can communicate by exchanging messages using some common transport (e.g., HTTP) and message format (e.g., SOAP).

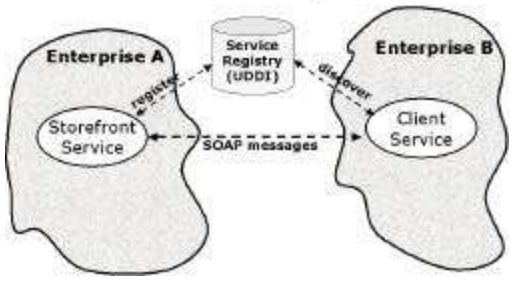

Figure 1. Two services belonging to different enterprises can dynamically discover and interact with each other.

However, now suppose that the storefront service expects the message exchanges to follow a specific pattern (conversation), such as the conversation depicted in Figure 2. Because the client and storefront services belong to different enterprises and have discovered each other dynamically, service developers are faced with several issues. For example, how does the client service know what conversations the storefront service supports? Does the storefront service developer have to code the conversation-controlling logic directly into the service? If so, do developers have to re-implement the client and storefront services each time a new message exchange is added to the supported conversation?

Our goal is to make it possible for service developers to create services without having to implement explicit conversation control. In this paper, we address the problem of how to enable E-Services from different enterprises to engage in flexible and autonomous, yet potentially quite complex, business interactions. We adopt an approach from the domain of software agents, modeling protocols for business interaction as *conversation policies,* but extend this approach to

exploit the fact that E-Service messages are XML-based business documents and can thus be mapped to XML document types. In particular, we propose a methodology by which a single third-party controller can leverage "reflected" XML-based specifications to direct the message exchanges of E-Services and their clients according to protocols without the service developers having to implement protocol-based flow logic themselves.

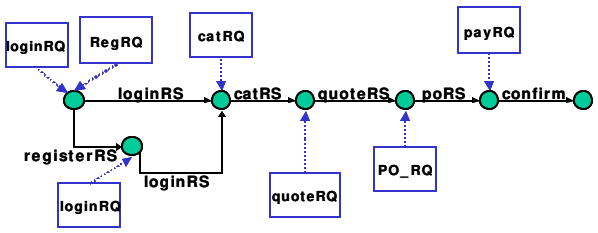

Figure 2 Output document types serve as transitions between interactions.

Current systems require services to participate in homogeneous marketplaces, in which participants code to matching conversation protocols; should a protocol change, all participants that support the protocol must be updated and recompiled. This reduces the likelihood that two services that discover each other will be able to converse spontaneously. In addition, existing systems also couple the message exchanges with the internal state of a service/agent; breaking this coupling between conversational logic and business logic allows us to exploit the document type properties of XML-based messages to produce an extremely lightweight conversation controller.

We introduce here a model for a conversation controller for E-Services that focuses on conversation functionality, as opposed to a service's business functionality. Distinguishing between conversation logic and business logic enables service developers to delegate conversational responsibilities to some conversation controller service, both freeing the developers from having to implement explicit conversation control mechanisms and allowing services to interact even if they don't support precisely matching conversations. In Section 2, we present work related to our efforts. In Section 3, we describe our model for the problem space, and introduce the paradigm of our solution. We have implemented a prototype conversation controller, and in Section 4 we describe our implementation. Finally, we present conclusions and future work in Section 5.

2. Related work

In his survey of agent systems for E-Commerce, Griss [4] notes that researchers in the agent community have proposed a number of agent communication systems over the past decade, and indeed agent-based e-commerce systems seem like a natural model for the future of E-Services. Griss identifies several kinds of agent systems appropriate for E-Commerce, including personal agents, mobile agents and collaborative/social agents. Griss then lists seven properties that represent dimensions of agent-like behavior: adaptability, autonomy, collaborations, intelligence, mobility, persistence and personality/sociability. We believe that although E-Services exhibit some of these properties, E-Services are not necessarily adaptable, intelligent or anthropomorphic (they are not required to exhibit personality/sociability). However, since agents dynamically communicate via message exchanges that conform to specified protocols/patterns, agent-based conversations are recognized as an especially appropriate model for E-Service interactions.

Several existing agent systems allow agents to communicate following conversational protocols (or patterns). However, to the best of our knowledge, all of these are tightly coupled to specific agent systems, and require that all participating entities must be built upon a common agent platform. For example, the Knowledgeable Agent-oriented System (KaoS)[2] is an open distributed architecture for software agents, but requires agent developers to hard-wire conversation policies into agents in advance. Walker and Wooldridge [9] address the issue of how a group of autonomous agents can reach a global agreement on conversation policy; however, they require the agents themselves to implement strategies and control. Chen, et al. [3] provide a framework in which agents can dynamically load conversation policies from one-another, but their solution is homogeneous and requires that agents be built upon a common infrastructure. Our Conversation Controller is unique in that we require only that a participating service produce two XML-based documents – 1) a specification of the conversational flows it supports and 2) a specification of the service's functionality (describing how the service can be invoked).

A few E-Commerce systems support conversations between services. However, these all require that the client and service developers implement matching conversation control policies. RosettaNet's *Partner Interface Processes* (PIPs)[7] specify the roles and required interactions between two businesses. *Commerce XML* (cXML)[1] is a proposed standard being developed by more than 50 companies for business-to-business electronic commerce. cXML associates XML DTDs for business documents with their request/response processes. Both RosettaNet and CommerceXML require that participants pre-conform to their

standards. Our work is completely compatible with such systems, but is also unique in that we allow a service's clients to share the service's Conversation Controller dynamically – without having to implement the client to the specifications of the service.

Insofar as they reflect the flow of business processes, E-Service conversations also resemble workflows. However, as the authors of the E-Speak Conversation Definition Language (CDL[1]) [5] observe, workflows and conversations serve different purposes. Conversations reflect the interactions between services, whereas workflows delineate the work done by a service. A conversation models the externally visible commercial interactions of a service, whereas a workflow implements the service's business functionality. In addition, workflows represent long-running concurrent fully integrated processes, whereas E-Service conversations are loosely coupled interactions.

3. Model and paradigm

In this section, we describe the model and paradigm upon which our conversation controller is based. Together, these allow us to free service developers from having to handle conversational logic explicitly.

3.1. Document-based message model

In our model, EServices interact by exchanging messages. Each message can be expressed as a structured document (e.g., using XML) that is an instance of some document type (e.g., expressed using XML Schema). A message may be wrapped (nested) in an encompassing document, which can serve as an envelope that adds contextual (delivery or conversation specific) information (e.g., using SOAP). We define a conversation to be a sequence of message exchanges (interactions) between two or more services. We define a *conversation specification* (also known as a *conversation policy*) to be a formal description of "legal" message type-based conversations that a service supports.

We require the E-Service to communicate two pieces of information to the Conversation Controller:

- A specification of the structure of the conversations supported by the service (interactions, valid input and output message types of interactions, and transitions between interactions).
- A specification of the service's interfaces, mapping of document types to appropriate service entry points (for given interactions).

We assume that each service can produce its conversation specification using some conversation definition language (e.g., HP's CDL) upon demand. We also assume that a service can produce a document-based specification mapping valid input document types and service entry points to potential output document types.

3.2. Paradigm

Once the above requirements have been met, our conversation controller can act as a proxy to an E-Service, and track the state of an ongoing conversation, based on the types of messages exchanged. A conversation controller that acts as a proxy can perform the following tasks:

1. Once it has received a message on behalf of an E-Service, the Conversation Controller can dispatch the message to the appropriate service entry point, based on the state of the conversation and the document's type.
2. When forwarding the response from the E-Service to the client, the Conversation Controller includes a prompt indicating valid document types that are accepted by the next stage of the conversation. This prompt can optionally be filtered through a transformation appropriate to the client's type. (E.g., if the client is a web browser and has indicated that it would like form output, then the Conversation Controller may transform the response into an HTML form before sending it to the client.)

In addition, if the client requests it and specifies appropriate entry points, the Conversation Controller can also direct the client's side of the conversation. This means that neither the service nor the client developer must explicitly handle conversational logic in their code.

In order for an EService to use a Conversation Controller as a proxy, the service developer must do the following (note that the service developer does not need to implement code to handle the conversation flow logic):

- Document the service's conversation flow in a specification.
- Document the type-based inbound document handling entry points in a specification (ideally capturing both input and output document types).
- Advertise the service with an entry point going through the Conversation Controller.

[1] The E-Speak Conversation Definition Language (CDL) is *not* related to the Component Description Language (CDL).

Each time the Conversation Controller receives a message on behalf of the service, it will identify the current stage of the conversation and verify that the message's document type is appropriate; if not, then it will raise an exception. If the message is valid, then the Conversation Controller will invoke the service appropriately. It will then identify the document type of the response from the service, identify the new state and the valid input documents for that state, and format an appropriate response for the client. The Conversation Controller can also pass the response through an appropriate transformation, if requested by the client. For example, if the client is an HTML browser, then the Conversation Controller could return an HTML form prompting for appropriate input. Moreover, if the client is another service that can return a specification of its own service entry points, then the Conversation Controller could automatically send the output message to appropriate client entry points; if a valid input document for the new state is returned, the Conversation Controller could then forward it to the service, thus moving the conversation forward dynamically. As a result, the Conversation Controller can help a client and service carry out an entire conversation without either the client or the service developer having to implement any explicit conversation control mechanisms. This means that the client developer does not need complete knowledge of all the possible conversations supported by all the services with which the client might interact in the future. For example, each time the Conversation Controller receives a message on behalf of a service, it could implement the pseudo-code listed in Figure 3, below.

3.3. Client automation

An argument can be made that developers implementing E-Service clients will not want a conversation controller to direct their part of the conversation, both because they expect to hard-code the client parts of the conversation and also because they will find the idea of using a third-party to control conversation foreign[2]. However, decoupling conversation logic from business logic on the client side greatly increases the flexibility of a client by allowing it to interact dynamically with services even if their conversation policies do not match exactly. For example, the same client code could be used to interact with two services that support different conversation policies but common interfaces.

In order for a conversation controller to direct the client's part of a conversation, the controller must be able to dispatch messages the client receives from the server in order to generate documents that the server requests. This means that the client must be able to communicate its service interfaces to the Conversation Controller. For example, we can extend the process described in Figure 3 to allow the Conversation Controller to direct both the server and client sides of the conversation, producing the pseudo-code listed in Figure 4.

1. Look at the message header and determine the current state of the conversation. (Ask the service for specifications, if necessary.)
2. From the conversation specification, get the valid input document types for the current state.
3. Verify whether the current message is of a valid input document type for the current state.
4. If the received message is of a valid type, then look up the inbound document in the dispatch specification and dispatch the message to an appropriate service entry point. If more than one appropriate service entry point exists, then dispatch it to each entry point (in order specified by the service) until the service produces an output document of a valid document type. If no entry point exists or no valid output document is produced, then inform the client, also prompting for valid input document types.
5. From the conversation specification, calculate the conversation's new state, given the document type of the output document returned by the service. Look up the valid input documents for this new state.
6. Format the output document in a form appropriate to the client type, also prompting for the input document types that are valid in the new state.

Figure 3. The Conversation Controller can receive and handle a message on the behalf of a service.

3.4. Conversation controller state

The Conversation Controller that we have outlined above does not include any performance management, history, or rollback mechanisms. If one subscribes to the idea that intermediate states of an E-Service's conversation are *not* transactional, and one also supposes that Conversation Management functionality (including performance history, status of ongoing conversations, etc.) is distinct from Conversation Control functionality, then the

[2] Conversation with Kevin Smathers, 1/4/2001.

Conversation Controller can operate in a stateless mode.

1. Look at the message header and determine the current state of the conversation. (Ask the service for specifications, if necessary.)
2. From the conversation specification, get the valid input document types for the current state.
3. Verify whether the current message is of a valid input document type for the current state.
4. If the received message is of a valid type, then look up the inbound document in the dispatch specification and dispatch the message to the appropriate service entry point; otherwise, inform the client that the message is not a valid type and prompt for the input document types that are valid in the new state.
5. From the conversation specification, calculate the conversation's new state, given the document type of the output document returned by the service. Look up the valid input documents for this new state.
6. If the client wishes to be treated as a browser, then format the output document in an appropriate HTML form, also prompting for the valid input document types for the new state.
7. If the client wishes to be directed by the Conversation Controller and there are valid input documents for the new state, then look up outbound document types in the client's dispatch table, and invoke the appropriate client methods that could produce valid input documents.
8. If the client produces a valid input document, then send it to the service, invoking it through the Conversation Controller (recursion takes place here).
9. If the client does not produce any valid input documents, or if there were no valid input documents in the new state, then format and return the output document in an appropriate HTML form, also prompting for the new state.

Figure 4. The Conversation Controller can receive a message from a client on behalf of a service, dispatch it to the service, and then prompt the client for an appropriate response.

4. Prototype implementation

We have implemented a prototype Conversation Controller as well as example services and clients. Our goal was to implement a conversation controller that could receive messages on the behalf of the service, validate that each message was of an appropriate input document type for the current state of the conversation, dispatch each message to appropriate service entry point, and use the resulting output document types to identify the next appropriate interaction for the conversation.

We tried to leverage as much existing technology as we could in building this prototype:

- We used the Conversation Definition Language (CDL) from HP's E-Speak Organization to specify the conversations supported by the E-Services. CDL is an XML-based specification that defines a service interface in terms of a list of interactions (keyed by document type) and a list of transitions that describe legal interaction orderings.
- We implemented the services as Apache JServ servlets (they could also have easily been implemented using ESpeak, Chaiserver, Jini, or CORBA).
- We used Apache's Xerces and Xalan packages for their XML and XSL functionality.
- We used a subset of the Web Services Description Language (WSDL) to describe E-Service interfaces. WSDL is a standard proposed by IBM and Microsoft. WSDL describes the capabilities of a web service and specifies how to access that service's entry points. WSDL is intended to simplify software delivery by allowing software to be invoked remotely (e.g., via the web) rather than requiring the software to be installed locally.

We made a number of simplifying assumptions for this implementation. The Conversation Controller is not responsible for maintaining management history, transactional guarantees, multi-party conversations or the synchronization of multiple related conversations. We do not address security issues at this time.

4.1. Directing conversations

In order to track the state of conversations, the Conversation Controller needed to be able to perform the following functions:

- Given a message, determine the type of conversation (conversation specification) and the stage (interaction identifier) of the ongoing conversation.
- Given a message, identify its document type.
- Given a Conversation Specification and an interaction identifier from that specification,

return the document types accepted as valid input to that interaction.

- Given a Conversation Specification, an interaction identifier from that specification, and a document type (representing an input document), return a boolean indicating whether or not the document type would be accepted as valid input for that interaction.
- Given a Conversation Specification, a source interaction identifier from that specification, and a document type (representing an output document), return the target interaction represented by the transition from the source interaction given the output document type.

Our simple method of implementing the first two requirements was to give each message a special context element:

```
<Context>
  <ConversationId/>
  <In-Reply-To/>
  <Reply-With/>
  <DocumentType/>
</Context>
```

Each of these elements has an "owner" who controls the contents of the element value. The Conversation Controller owns the ConversationId field, which can be used to map to the conversation type identifier, the current interaction and the valid input document types for the current interaction of the current conversation. The message sender owns the Reply-With and DocumentType element. The In-Reply-To element's value should be the value of the Reply-With element of the message to which the current message is responding. Each party is responsible for protecting the contents of their fields from tampering, e.g., using encryption.

To meet the last three requirements, each time the Conversation Controller reads a new conversation specification (expressed in CDL), it populates two hash tables: one that maps from interaction identifiers to valid input document types, and one that maps from source interaction identifier/transition document types to target interaction identifiers. The Conversation Controller uses the first table to look up the valid input document types for a given interaction. It uses the second table to determine when a conversation has progressed from one interaction state to another (given the document type of an output document and a source interaction identifier).

4.2. Dispatching messages to services

In order to forward messages to appropriate service entry points, the Conversation Controller needed to map input and output document types to service entry points. For this, we created a WSDL specification for each service. Each time the Conversation Controller reads a WSDL specification, it populates two hash tables: one that maps from input document types to service entry point and output document types, and one that maps from output document types to service entry point and input document types.

One open issue is how to couple the dispatch and the conversation flow specifications. Our current implementation uses these tables to find actions that can handle incoming document types and produce output documents of appropriate document types. That is to say, currently the Conversation Controller does not consider the state of the conversation when dispatching the message. This is because the WSDL and CDL are completely independent. For the future, we are considering associating a WSDL specification with each interaction.

4.3. Architecture

The prototype Conversation Controller as currently implemented is stateless, and can direct both servers and clients in CDL-specified conversations (using Algorithm 2). Figure 5 sketches the components of our initial prototype, from the perspective of how the Conversation Controller handles a message from a client to a service.

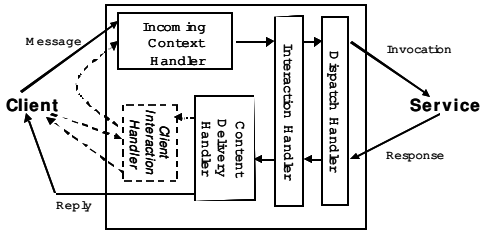

Figure 5 The Conversation Controller handles messages on the behalf of services.

The *Incoming Context Handler* has logic that handles message structure, and is responsible for unpacking contextual information from incoming message headers. The *Outgoing Content Delivery Handler* is responsible for packing contextual information into outgoing message headers and composing outgoing messages. The *Interaction Handler* parses and queries conversation definitions (specified in CDL), for example to validate document types or calculate new conversation states. The *Dispatch Handler* parses and queries service descriptions (specified in WSDL), then uses that information to forward messages to services. The

optional *Client Interaction Handler* and its interactions (drawn with dotted lines) can dispatch the reply from the service to the client and then forward the client's response back to the service (via the *Conversation Controller*).

Figure 6 details how the Conversation Controller uses these components to handle a message it receives on the behalf of a service.

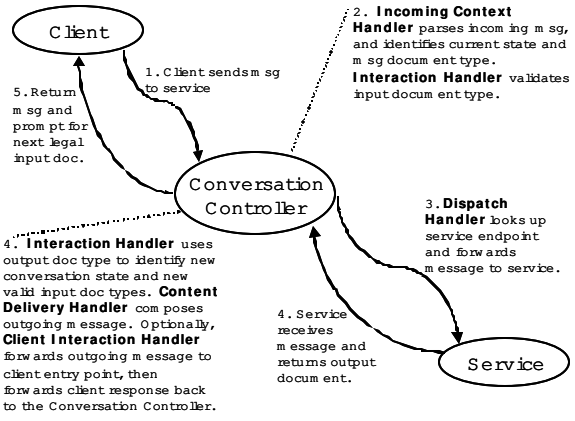

Figure 6 The Conversation Controller acts as a proxy to services to services.

Each time the Conversation Controller receives a message on behalf of a service, the *Incoming Context Handler* parses the incoming message and extracts (or initializes) its Context element. The *Interaction Handler* uses this context element to identify the current state, conversation specification (specified in CDL) and document type represented by the incoming message, then validates whether or not the document type of the incoming message is valid for the current state. If the incoming message is of a legitimate type, then the *Dispatch Handler* parses the service's specification and forwards the message to an appropriate service entry point. When the service returns a response message, the *Interaction Handler* uses the document type of the response message (along with the Conversation Specification) to identify the next state of the Conversation as well as the new state's valid document types. The *Outgoing Content Handler* builds an outgoing message context element from this information, and composes (incorporating the response message from the service) an outgoing message to return to the client. If the client service has requested that the Conversation Controller direct its side of the conversation and has also provided a service specification for itself, then the *Dispatch Handler* identifies and dispatches to appropriate client entry point(s) that could produce an appropriately typed document using the client specification. If the client produces a valid document for the new state, then the *Dispatch Handler* will forward that message back to the Conversation Controller on behalf of the service (thus starting the cycle again).

5. Conclusions / future work

We have proposed here a mechanism for a conversation controller that can act as a proxy to an E-Service, enabling the service to engage in complex interactions with other services. Our solution is unique in that we distinguish between the *routing* (e.g., flow control) and *management* (e.g., quality of service) of a conversation. We also make a distinction between *conversation logic* and *business logic*. These distinctions allow us to provide an extremely lightweight conversation controller capable of directing a service's conversations with other services or clients. This conversation controller can dynamically execute a service's conversation logic given a minimum amount of information – a specification of the conversations the service supports, and a specification service's functions. We have successfully implemented a prototype conversation controller, used it to control both client/service and user (web browser)/service conversations.

In the future, we plan to use this mechanism as a test bed for rapid development of conversation-based prototypes (for example, experimentation with automatic negotiation policies). We also hope to see this mechanism incorporated into a complete E-Service marketplace. For example, we would like to be integrate our prototype with a service management service. In addition, we propose to extend our model to include multi-party conversations.

Acknowledgements

We thank Alan Karp, Meichun Hsu, Qiming Chen, and Kevin Smathers for input and comments regarding this work. We owe special thanks to Alan Karp, who first recommended that the Conversation Controller act as a proxy to the service it represents.

References

[1] Web page: http://www.cxml.org

[2] Bradshaw, J.M., KAoS: An Open Agent Architecture Supporting Reuse, Interoperability, and Extensibility. Knowledge Acquisition for Knowledge-Based Systems Workshop, 1996 URL: http://spuds.cpsc.ucalgary.ca/KAW/KAW96/bradshaw/KAW.html

[3] Chen, Q., Dayal, U., Hsu, M., and Griss, M., Dynamic Agents, Workflow and XML for E-

Commerce Automation. First International Conference on ECommerce and Web-Technology, 2000.
http://www.hpl.hp.com/org/stl/dmsd/publications/qchen_EC2000.pdf
[4] Griss, M., My Agent Will Call Your Agent . . . But Will It Respond?. Software Development Magazine, 2000. (Also available as technical report HPL-1999-159)
[5] HP ESpeak Operations (updated by Alan H. Karp), Conversation Definition Language Specification for UDDI version 1.0, Nov, 2000
[6] Kuno, H., Surveying the E-Services Technical Landscape. International Workshop on Advanced Issues of ECommerce and Web-Based Information Systems (WECWIS), 2000. (Also available as technical report HPL-2000-22)
[7] Web page: http://rosettanet.org
[8] Seybold, P. B. Preparing for the E-Services Revolution, 1999.
http://www.hp.com/e-services/pdfs/seybold.pdf
[9] Walker, A. and Wooldridge, M., Understanding the emergence of conventions in multi-agent systems. First International Conference on Multi-Agent Systems, 1995.

Appendix A. Example CDL specification

The following XML document is an example of a CDL specification for a conversation supported by a storefront service (example taken from [5]). Figure 2 sketches the transition table expressed in this specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<Conversation conversationType="eSpeakSFS" id="conv123"
  name="simpleConversation">
  <ConversationInteractions>
    <Interaction StepType="ReceiveSend" id="Start" initialStep="true">
      <InboundXMLDocuments>
        <InboundXMLDocument
 hrefSchema="http://conv123.org/LoginRQ.xsd" id="LoginRQ">
        </InboundXMLDocument>
        <InboundXMLDocument
 hrefSchema="RegistrationRQ.xsd" id="RegistrationRQ">
        </InboundXMLDocument>
      </InboundXMLDocuments>
      <OutboundXMLDocuments>
        <OutboundXMLDocument
 hrefSchema="http://conv123.org/ValidLoginRS.xsd"
 id="ValidLoginRS">
        </OutboundXMLDocument>
        <OutboundXMLDocument
hrefSchema="http://conv123.org/RegistrationRS.xsd" id="RegistrationRS">
        </OutboundXMLDocument>
      </OutboundXMLDocuments>
    </Interaction>
    <Interaction StepType="ReceiveSend" id="LoggedIn" initialStep="false">
      <InboundXMLDocuments>
        <InboundXMLDocument
 hrefSchema="http://conv123.org/CatalogRQ.xsd" id="CatalogRQ">
        </InboundXMLDocument>
      </InboundXMLDocuments>
      <OutboundXMLDocuments>
        <OutboundXMLDocument
 hrefSchema="http://conv123.org/CatalogRS.xsd" id="CatalogRS">
        </OutboundXMLDocument>
      </OutboundXMLDocuments>
    </Interaction>
    <Interaction StepType="ReceiveSend" id="Registered"
 initialStep="false">
```

```
          <InboundXMLDocuments>
            <InboundXMLDocument
     hrefSchema="http://conv123.org/LoginRQ.xsd" id="LoginRQ">
            </InboundXMLDocument>
          </InboundXMLDocuments>
          <OutboundXMLDocuments>
            <OutboundXMLDocument
     hrefSchema="http://conv123.org/ValidLoginRS.xsd"
     id="ValidLoginRS">
            </OutboundXMLDocument>
          </OutboundXMLDocuments>
        </Interaction>
        <Interaction StepType="ReceiveSend" id="Catalogued"
     initialStep="false">
          <InboundXMLDocuments>
            <InboundXMLDocument
   hrefSchema="http://conv123.org/QuoteRQ.xsd" id="QuoteRQ">
            </InboundXMLDocument>
          </InboundXMLDocuments>
          <OutboundXMLDocuments>
            <OutboundXMLDocument
     hrefSchema="http://conv123.org/QuoteRS.xsd" id="QuoteRS">
            </OutboundXMLDocument>
          </OutboundXMLDocuments>
        </Interaction>
        <Interaction StepType="ReceiveSend" id="Quotation"
initialStep="false">
          <InboundXMLDocuments>
            <InboundXMLDocument
     hrefSchema="http://conv123.org/PurchaseOrderRQ.xsd"
     id="PurchaseOrderRQ">
            </InboundXMLDocument>
          </InboundXMLDocuments>
          <OutboundXMLDocuments>
            <OutboundXMLDocument
    hrefSchema="http://conv123.org/InvoiceRS.xsd"
     id="InvoiceRS">
            </OutboundXMLDocument>
          </OutboundXMLDocuments>
        </Interaction>
        <Interaction StepType="ReceiveSend" id="Invoiced" initialStep="false">
          <InboundXMLDocuments>
            <InboundXMLDocument
   hrefSchema="http://conv123.org/AuthorizePaymentRQ.xsd"
     id="AuthorizePaymentRQ">
            </InboundXMLDocument>
          </InboundXMLDocuments>
          <OutboundXMLDocuments>
            <OutboundXMLDocument
     hrefSchema="http://conv123.org/ConfirmationRS.xsd"
     id="ConfirmationRS">
            </OutboundXMLDocument>
          </OutboundXMLDocuments>
        </Interaction>
        <Interaction StepType="ReceiveSend" id="end" initialStep="false">
          <InboundXMLDocuments/>
          <OutboundXMLDocuments/>
```

```
    </Interaction>
  </ConversationInteractions>
  <ConversationTransitions>
    <Transition>
      <SourceInteraction href="Start"/>
      <DestinationInteraction href="LoggedIn"/>
      <TriggeringDocument href="ValidLoginRS"/>
    </Transition>
    <Transition>
      <SourceInteraction href="Start"/>
      <DestinationInteraction href="Registered"/>
      <TriggeringDocument href="RegistrationRS"/>
    </Transition>
    <Transition>
      <SourceInteraction href="Registered"/>
      <DestinationInteraction href="LoggedIn"/>
      <TriggeringDocument href="ValidLoginRS"/>
    </Transition>
    <Transition>
      <SourceInteraction href="LoggedIn"/>
      <DestinationInteraction href="Catalogued"/>
      <TriggeringDocument href="CatalogRS"/>
    </Transition>
    <Transition>
      <SourceInteraction href="Catalogued"/>
      <DestinationInteraction href="Quotation"/>
      <TriggeringDocument href="QuoteRS"/>
    </Transition>
    <Transition>
      <SourceInteraction href="Quotation"/>
      <DestinationInteraction href="Invoiced"/>
      <TriggeringDocument href="InvoiceRS"/>
    </Transition>
    <Transition>
      <SourceInteraction href="Invoiced"/>
      <DestinationInteraction href="End"/>
      <TriggeringDocument href="ConfirmationRS"/>
    </Transition>
  </ConversationTransitions>
</Conversation>
```