Using CDL in a UDDI Registry 1.0
UDDI Working Draft Best Practices Document

Overview

Electronic commerce is moving towards a vision of web-service based interactions, where corporate enterprises use web-services to interact with each other dynamically. For example, a service in one enterprise could spontaneously decide to engage a service fronted by another enterprise. In order for services to interact with each other dynamically, they must be able to do three fundamental things.

1. Clients must be able to discover services.
2. A service must be able to describe its abstract interfaces and protocol bindings so that clients can figure out how to invoke it.
3. A service must be able to describe the kinds of interactions (conversations) that it supports (e.g., that it expects clients to login before they can request a catalog) so that it can engage in complex exchanges with its clients.

The Universal Description Discovery and Integration (UDDI)[1-3,6] specifications address the first problem by defining a way to publish and discover information about Web services. The Web Services Description Language (WSDL)[4] addresses the second problem, defining a general purpose XML language for describing the interface and protocol bindings of network services. The Conversation Definition Language (CDL)[5,6] addresses the last problem, providing a standard way to model the public processes of a service, thus enabling network services to participate in rich interactions. Together, UDDI, WSDL, and CDL enable developers to implement web services capable of spontaneously engaging in dynamic and complex inter-enterprise interactions.

In this document, we clarify the relationship between CDL, UDDI, and WSDL. In particular, we describe how together these three components can be used to create an environment in which services can spontaneously discover each other and then engage in complicated interactions.

Modeling Web-Service Interactions

Web-services are much more loosely coupled than traditional distributed applications. This difference impacts both the requirements and usage models for web-services. Web-services are deployed on the behalf of diverse enterprises, and the programmers who implement them are unlikely to collaborate with each other during development. However, the purpose of web-services is to enable business-to-business interactions. Therefore, web-services must support very flexible, dynamic bindings. Web-services should be able to discover new services and interact with them dynamically without requiring programming changes to either service.

The prevalent model for web-service communication is that web-services will publish information about the specifications that they support. UDDI facilitates the publication and discovery of web-service information. The current version of WSDL (1.0) is an XML-based format that describes the interfaces and protocol bindings of web service

functional endpoints. WSDL also defines the payload that is exchanged using a specific messaging protocol; SOAP is one such possible messaging protocol. However, neither UDDI nor WSDL currently addresses the problem of how a service can specify the sequences of legal message exchanges (interactions) that it supports. (We use the term "conversation" to refer to a legal sequence of message exchanges.)

The Conversation Definition Language (CDL) addresses this issue, providing an XML schema for defining legal sequences of documents that web-services can exchange. CDL and WSDL are highly complimentary – WSDL specifies how to send messages to a service and CDL specifies the order in which such messages can be sent. The advantage of keeping the two distinct is that doing so allows us to decouple conversational interfaces (represented by CDL) from service-specific interfaces (represented by WSDL). This means that a single conversation specification can be implemented by any number of services, independent of the protocols supported by the various implementations.

Usage of CDL

CDL addresses the problem of how to enable E-Services from different enterprises to engage in flexible and autonomous, yet potentially quite complex, business interactions. It adopts an approach from the domain of software agents, modeling protocols for business interaction as *conversation policies,* but extends this approach to exploit the fact that E-Service messages are XML-based business documents and can thus be mapped to XML document types. Each CDL specification describes a single type of conversation from the perspective of a single participant. A service can participate in multiple types of conversations. Furthermore, a service can engage in multiple simultaneous instances of a given type of conversation.

Because a CDL specification describes the messages a service that supports it expects to receive as well as the messages it can send, other services can use the CDL specification to programmatically control interaction with that service. In addition, CDL offers 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. A service could register itself with a third-party service or service infrastructure, specifying CDL documents that it supports. The third-party could then take on responsibility for directing the service's conversations, making it possible for service developers to create services without having to implement explicit conversation control, such as handling exceptions when the service receives unexpected messages.

For example, Figure 1 depicts a simple purchase conversation type. The circles represent ReceiveSend interactions; the boxes represent inbound document types, and the arcs between the circles represent the transitions between interactions, which are driven by outbound document types. A service that supports this conversation type expects a conversation to begin with the receipt of a LoginRQ or a RegRQ document. Once the service has received one of these documents, then the conversation can progress to either

a "logged in" state or a "registered" state, depending on the type of message the service generates to return to the client.

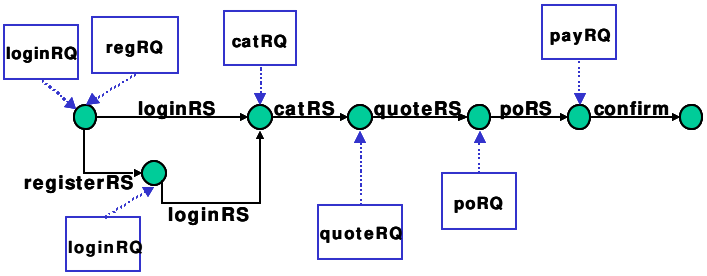

Figure 1 Output document types serve as transitions between ReceiveSend interactions.

Authoring CDL Conversation Specifications

There are three elements to a CDL specification:

- *Document type descriptions* specify the types (schemas) of XML documents that the service can accept and transmit in the course of a conversation.
- *Interactions* model the states of the conversation as document exchanges between conversation participants. CDL currently supports four types of interactions: *Send* (the service sends out an outbound document), *Receive* (the service receives an inbound document), *SendReceive* (the service sends out an outbound document, then expects to receive an inbound document in reply), and *ReceiveSend* (the service receives an inbound document and then sends out an outbound document).
- *Transitions* specify the ordering relationships between interactions. A transition specifies a source interaction, a destination interaction, and a document type that triggers the transition. CDL 1.0 also supports two special transitions: *Default Transition* and *Exception Transition.* A *default transition* is triggered if a valid inbound (for a *SendReceive* interaction) or outbound (for a *ReceiveSend* interaction) document is received for a given interaction, but no other transition is triggered. At most one default transition can be defined per source interaction.

We now examine an XML representation of the conversational pattern depicted in Figure 1, specified using CDL. The root *Conversation* element specifies a unique name to the conversation (e.g., a UDDI tModel), and contains two sub-elements: a *ConversationInteraction* element that contains a list of *Interaction* elements, and a *ConversationTransitions* element that consists of a list of *Transition* elements. (The full specification for Figure 1's example conversation is listed in Appendix A.)

```
<?xml version="1.0" encoding="UTF-8"?>
<Conversation conversationType="eSpeakSFS" id="conv123"
```

```
  name="storefrontConversation">
  <ConversationInteractions>
     +<Interaction . . . >
     +<Interaction . . . >
     . . .
  </ConversationInteractions>
  <ConversationTransitions>
    +<Transition>
</ConversationTransitions>
</Conversation>
```

The following XML code is a CDL representation of the initial (start) interaction.

```
<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>
```

Note that this interaction is assigned the identifier "Start." Also, note that each document type is assigned an identifier (e.g., "LoginRQ"). These identifiers are used when specifying the source and destination interactions for the conversation's transitions. The other interactions would be defined in a similar manner. For example, the following XML code is a CDL representation of the "LoggedIn" interaction, one of the interactions immediately following the "Start" interactions.

```
<Interaction StepType="ReceiveSend" id="Start" initialStep="false">
   <InboundXMLDocuments>
      <InboundXMLDocument
          hrefSchema="CatalogRQ.xsd" id="CatalogRQ">
      </InboundXMLDocument>
    </InboundXMLDocuments>
    <OutboundXMLDocuments>
      <OutboundXMLDocument
            hrefSchema=http://conv123.org/CatalogRS.xsd
            id="CatalogRS">
      </OutboundXMLDocument>
    </OutboundXMLDocuments>
```

```
 </Interaction>
```

Transitions indicate the progression of the conversation between interactions. For example, the following XML code specifies that there is a transition between the "Start" interaction and the "LoggedIn" interaction. If the conversation is in the "Start" state and the protagonist service outputs a document of type "ValidLoginRS," then the conversation will proceed from the "Start" interaction to the "LoggedIn" interaction.

```
<Transition>
      <SourceInteraction href="Start"/>
      <DestinationInteraction href="LoggedIn"/>
      <TriggeringDocument href="ValidLoginRS"/>
</Transition>
```

Although CDL specifies the valid inbound and outbound documents for an interaction, it does not specify how the conversation participants will handle and produce these documents. The CDL specification of a conversation is thus service-independent, and can be used (and reused) by any number of services.

Relevant UDDI Structures

A UDDI business registration is an XML document that describes a business entity and its web services. The UDDI XML schema defines four core types of service information: business information (such as business name and contact information), business service information (general technical and business descriptions of web services), binding information (specific information needed to invoke a service), and service specification information (associating a service's binding information with the business service information it implements).

Programmers and programs can use the UDDI Business Registry to locate technical information about services, such as the protocols and specifications that they implement. More importantly, the UDDI Business Registry also serves as a registry for abstract (service-independent) specifications. Services can refer indirectly to the UDDI registrations for specifications they implement, which makes it straightforward to identify the business service information that represents a given service.

The UDDI *tModel* is a meta-data construct that uniquely identifies reusable service-related technical specifications for reference purposes. A service publishes tModelInstanceDetails, which is a list of tModelInfo elements that refer to the tModels that the service supports. A tModel is composed of a unique key, a name, an optional description, and a URL for the specification itself.

For example, suppose we wanted to register a CDL specification of the "storefront" conversation depicted in Figure 1 in a UDDI registry. The following XML code is a UDDI tModel reference for a CDL specification for a service conversation.

```
<tModel authorizedName="XXXX" operator="YYYY" tModelKey="ZZZZ">
    <name>storefrontConversation</name>
```

```
    <description xml:lang="en">
      CDL description of a simple storefront conversation
    </description>
    <overviewDoc>
      <description xml:lang="eng">CDL source document.</description>
      <overviewURL>http://foo.org/specs/storefrontCDL.xml</overviewURL>
    </overviewDoc>
</tModel>
```

This "storefront conversation" tModel can now be referenced by the tModelInstanceInfo of any service that implements that conversation type:

```
<businessService>
  (. . .)
  <bindingTemplates>
    <bindingTemplate>
      (. . .)
      <accessPoint urlType="http">http://www.foo.com/</accessPoint>
      <tModelInstanceDetails>
        <tModelInstanceInfo tModelKey="ZZZZ">
          <instanceDetails>
            <overviewDoc>
                  http://www.foo.com/overview.html
            </overviewDoc>
          </instanceDetails>
  </tModelInstanceInfo>
      </tModelInstanceDetails>
      (. . .)
    <bindingTemplate>
      (. . .)
  <bindingTemplates>
 <businessService>
```

Relevant WSDL Structures

As noted before, CDL specifications are conversation-specific. CDL describes the structures (types) of documents a service expects to receive and produce, as well as the order in which document interchanges will take place, but does not specify how to dispatch received documents to the service. This is partially addressed by WSDL. WSDL documents describe the abstract interface and protocol bindings of a network service. WSDL specifications that describe abstract protocol interfaces are reusable and thus are registered as UDDI tModels.

A reusable WSDL document consists of four components: document type, message, portType, and binding definitions. For example, the "storefront" conversation shown in Figure 1 requires that a service implementing the "Start" interaction provide some sort of endpoint that can accept a *LoginRQ* or *RegistrationRQ* document and output either a *LoginRS* or a *RegistrationRS* document. The reusable WSDL documents for a SOAP implementation of the interface to a *Login* method might look something like the following:

```
<?xml version="1.0"?>
```

```
<definitions name="Login"
      targetNamespace=http://foo.com/login.wsdl
      xmlns:tns=http://foo.com/login.wsdl
      xmlns:xsd1=http://foo.com/login.xsd
      xmlns=http://schemas.xmlsoap.org/wsdl>
  <types>
    <import namespace="xsd1"
                location="http://foo.com/schemas/loginRQ.xsd">
    <import namespace="xsd1"
                 location="http://foo.com/schemas/loginRS.xsd">
</types>
<message name="LoginRQ">
   <part name="body" element="xsd1:LoginRQ"/>
</message>
<message name="LoginRS">
   <part name="body" element="xsd1:LoginRS"/>

<portType name="LoginPortType">
   <operation name="Login">
      <input message="tns:LoginRQ"/>
      <output message="tns:LoginRS"/>
   </operation>
</portType>

<binding name="LoginSoapBinding"
   type="tns:LoginPortType:">
   <soap:binding style="document"
      transport=http://schemas.xmlsoap.org/soap/http/>
   <operation name="Login">
      <soap:operation
            soapAction=http://example.com/GetLastTradePrice/>
      <input>
            <soap:body use="literal"
                  namespace="http://conv123.org/LoginRQ.xsd"

      encodingStyle=http://schemas.xmlsoap.org/soap/encoding/ />
      </input>
      <output>
            <soap:body use="literal"
                  namespace="http://conv123.org/LoginRS.xsd"

      encodingStyle=http://schemas.xmlsoap.org/soap/encoding/ />
      </output>
    </operation>
  </binding>
</definitions>
```

WSDL associates an endpoint with at most one input and one output document type. Because the *Start* interaction accepts both *LoginRQ* and *RegistrationRQ* documents, we must also define an endpoint that accepts *RegistrationRQ* documents. The reusable WSDL documents for a SOAP implementation of the interface to this *Register* method might look something like the following:

```
<?xml version="1.0"?>
```

```
<definitions name="Register"
      targetNamespace=http://foo.com/login.wsdl
      xmlns:tns=http://foo.com/login.wsdl
      xmlns:xsd1=http://foo.com/login.xsd
      xmlns=http://schemas.xmlsoap.org/wsdl>
  <types>
    <import namespace="xsd1"
                location="http://foo.com/schemas/RegistrationRQ.xsd">
    <import namespace="xsd1"
                 location="http://foo.com/schemas/RegistrationRS.xsd">
</types>
<message name="RegistrationRQ">
   <part name="body" element="xsd1:RegistrationRQ"/>
</message>
<message name="RegistrationRS">
   <part name="body" element="xsd1:RegistrationRS"/>

<portType name="RegisterPortType">
   <operation name="Register">
      <input message="tns:RegistrationRQ"/>
      <output message="tns:RegistrationRS"/>
   </operation>
</portType>

<binding name="RegisterSoapBinding"
   type="tns:RegisterPortType:">
   <soap:binding style="document"
      transport=http://schemas.xmlsoap.org/soap/http/>
   <operation name="Register">
      <soap:operation
            soapAction=http://example.com/GetLastTradePrice/>
      <input>
            <soap:body use="literal"
                  namespace="http://conv123.org/RegistrationRQ.xsd"

      encodingStyle=http://schemas.xmlsoap.org/soap/encoding/ />
      </input>
      <output>
            <soap:body use="literal"
                  namespace="http://conv123.org/RegistrationRS.xsd"

      encodingStyle=http://schemas.xmlsoap.org/soap/encoding/ />
      </output>
    </operation>
  </binding>
</definitions>
```

These documents can then be registered as tModels, and referred to by a service's tModelInstanceInfo, as was done with the CDL example specification above.

Discussion

A key advantage of specification languages such as CDL and WSDL is that they pave the way for the creation of service frameworks that will enable service implementers to offload the responsibility for conversation-related tasks to infrastructure. For example, a

service developer could create WSDL specifications documenting their service's endpoints and CDL specifications that document the legal sequences of document exchanges that their service can support. These specifications could then be registered as UDDI tModels, and referred to by services' businessService entries in the UDDI registry. A service could then present such specifications to a third-party Conversation Controller service, which would provide such functionality as document type validation, conversational exception handling, conversation tracking, and message dispatching.

Although CDL and WSDL complement each other nicely in this regard, there are a number of issues that must be addressed before such a framework could be implemented. In the remainder of this section, we discuss two of these issues, and propose ways in which WSDL or CDL could be modified to become more compatible.

Cardinality of Input and Output Document Types

CDL captures the logic of a service's public processes, and defines an interaction as consisting of the exchange of a single input (or output) document for a single output (or input) document. Associating a CDL interaction with multiple input (or output) document types indicates that the interaction may result in the production of any one of the input (output) document types. For example, the start interaction of our CDL example is capable of outputting either a *LoginRS* or a *RegistrationRS* document. This powerful and extensible way to describe conversational interactions abstracts external process logic from application logic.

WSDL, as noted above, captures strongly-typed application logic, and does not provide a natural way for service developers to specify that a service endpoint can accept or emit any one of a number of document types. For example, in the above WSDL code, a service endpoint can take as input a single *LoginRQ* message and output a single *LoginRS* message, or it can take as input a single *RegistrationRQ* message and output a single *RegistrationRS* message. If we wanted to use WSDL to document a service endpoint that was capable of outputting a message of either of these types, we would have to create a composite complexType such as the following code.

```
<complexType name="Composite">
    <choice>
         <element name="LoginRS" minOccurs="1" maxOccurs="1"
             type="tns:LoginRSType"/>
         <element name="RegistrationRS" minOccurs="1" maxOccurs="1"
             type="tns:RegistrationRSType"/>
    </choice>
</complexType>
```

This is not a natural way to express the concept because ideally we would like to abstract conversation-specific logic from application-specific logic. It should be possible for some conversational infrastructure to check the type of the incoming message against a conversational specification and dispatch it to an appropriate service endpoint simply by looking at some document type identifier in the message header. However, wrapping multiple document types in a complex type means that either the conversational infrastructure must look at the message content (opening the "envelope") to determine the

actual type or else the service developer must assume responsibility for these tasks.

One solution for this problem would be to extend WSDL so as to be able to specify multiple alternative input and output document types. Another alternative would be to constrain service developers with an injunction that endpoints cannot produce multiple alternative input and output document types. Or, perhaps conversation frameworks will be implemented to recognize and deal with complex types. Finally, perhaps CDL could be extended with a null transition and modified to allow at most one input document type and at most one output document type to be associated with an interaction.

Mapping endpoints to interactions

Another issue is that we need a mechanism to map WSDL endpoints to CDL interactions. For example, suppose that a client wishes to engage in a business exchange with a service. Using the CDL specifications to which the service's *tModelInstanceDetails* refer, the client can identify the document types that the service will accept as conversation initiators. Using the WSDL specifications (also referred to by the service's *tModelInstanceDetails*), the client can identify service endpoints that can accept those document types as input and produce appropriate document types as output. However, suppose that more than one such service endpoint exists, and that the service developer wants to associate just one of them with a particular state of a conversation. We then would need some means of associating specific WSDL port/binding tModels with specific *interaction* elements of CDL conversation specification tModels. This need could be addressed easily by the specification of a mapping language that associates the two.

Acknowledgements

We would like to thank Alan Karp, Kannan Govindarajan and Gregory Pogossiants for their comments and suggestions regarding this work.

References

[1] Ariba, International Business Machines Corporation, Microsoft Corporation, *UDDI Technical White Paper*, Sep 6, 2000.

[2] Boubez, T., Hondo, M., Kurt, C., Rodriguez, J., and Rogers, D., *UDDI Programmer's API 1.0*, Sep 20, 2000.

[3] Boubez, T., Hondo, M., Kurt, C., Rodriguez, J., and Rogers, D., *UDDI Data Structure Reference V1.0,* Sep 30, 2000.

[4] Web page: *Web Services Description Language (WSDL) 1.0*. URL: http://msdn.microsoft.com/xml/general/wsdl.asp

[5] Hewlett-Packard, Co. *Service Framework Specification Version 2.0.* 2001.

[6] HP E-Speak Operations, *Conversation Definition Language Specification for UDDI version 1.0*, Nov, 2000.

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>
```