WS-CDL
Primer

Date: 17th January 2005
Last Modified: 26th April 2005
Authors: Steve Ross-Talbot, Tony Fletcher
Version: 4/8/05 4:19 pm

1 INTRODUCTION 4

1.1 STRUCTURE OF THE PRIMER 4

2 AN OVERVIEW OF WS-CDL 5

2.1 USING WS-CDL 7
2.2 WHY USE WS-CDL? 7
2.3 THE STRUCTURE OF WS-CDL 8

3 GETTING STARTED WITH WS-CDL 10

3.1 AN EXAMPLE 10
3.2 INTERACTIONS ORIENTED DESIGN 12
3.2.1 INTERACTIONS 13
3.2.2 ROLES 16
3.2.3 RELATIONSHIPS 17
3.2.4 INFORMATION TYPES 18
3.2.5 TOKENS 19
3.2.6 CHANNELS 20
3.3 CHOREOGRAPHIES, SEQUENCES, CHOICES AND WORKUNITS 22
3.3.1 CHOREOGRAPHIES 22
3.3.2 SEQUENCES 24
3.3.3 REPEATING WORKUNITS 25
3.3.4 CHOICES 27
3.3.5 COMPLETE EXAMPLE 29

4 ADVANCED TOPICS 31

4.1 DEPENDENT WORKUNITS 31
4.2 ADVANCED CHANNELS 34
4.2.1 USAGE 34
4.2.2 CHANNEL PASSING 34
4.3 BUSINESS EXCEPTIONS 34
4.3.1 EXCEPTIONS 34
4.3.2 EXCEPTIONS AS MESSAGES 34
4.4 COMPENSATIONS 34
4.4.1 FINALIZERS AND FINALIZATION 34
4.5 MODULARIZATION 34
4.5.1 CHOREOGRAPHIES AND SUB-CHOREOGRAPHIES 34
4.5.2 PERFORMING A CHOROEOGRAPHY 34
4.6 PARALLEL AND CONCURRENT 34
4.6.1 MANAGING JOIN CONDITIONS 34
4.7 SILENT ACTIONS AND CONDITIONS 34
4.8 NOACTIONS 34
4.9 TIME 34
4.10 ISOLATION LEVELS 34

5 AN EAI EXAMPLE 34

6 IMPLEMENTATION CONSIDERATIONS 35

6.1 END POINT PROJECTION 35
6.1.1 JAVA 35
6.1.2 WS-BPEL 35
6.1.3 RUNTIME MONITORING 35
6.2 WSDL 35
6.2.1 WSDL 1.1 35
6.2.2 WSDL 1.2 35
6.3 WS-ADDRESSING 35
6.3.1 CHANNEL REPRESENTATION 35

APPENDIX 1 – SIMPLE WS-CDL ENCODING OF THE EXAMPLE. 36

1 Introduction

1.1 Structure of the primer

This primer is intended to give an overview of WS-CDL and can be read by WS-CDL business users (e.g. a business analyst) and WS-CDL implementers (e.g software engineer) alike. The first 5 sections are intended for both audiences while the last is intended primarly for WS-CDL implementors.

Section 2 provides an overview of WS-CDL. The first half of Section 3 describes an example using UML sequence diagrams. The second half of Section 3 walks through building the WS-CDL description of the example. Section 4 examines WS-CDL at a deeper level describing how more advanced features can be employed through the use of the same example. Section 5 described the use of WS-CDL within an organization. Section 6 describes some of the implementation considerations for likely implementers.

2 An Overview of WS-CDL

It is essential in understanding Web Services Choreography Description Language (WS-CDL) to realize that there is no single point of control. There are no global variables, conditions or workunits. To have them would require them to be somewhere and that somewhere would be, by definition, a centralization point. WS-CDL is a language for specifying peer-to-peer protocols where each part wishes to remain autonomous and in which no party is master over any other – i.e. no centralization point. WS-CDL does permit a shorthand notation to enable variables and conditions to exist in multiple places, but this is syntactic sugar to avoid repetitive definitions. There is also an ability for variables residing in one service to be aligned (synchronized) with the variables residing in another service, giving the illusion of global or shared state.

WS-CDL is an XML-based language that can be used to describe the common and collaborative observable behavior of multiple services that need to interact in order to achieve some goal. WS-CDL describes this behavior from a global or neutral perspective rather than from the perspective of any one party and we call a complete WS-CDL description a global model.

Services are any form of computational process with which one may interact, examples are a buying process and a selling process that are implemented as computational services in a Service Oriented Architecture (SOA) or indeed as a Web Services implementation of an SOA. The distinction between SOA and Web Services is that the latter has its interface described using WSDL whereas the latter may not. Because WS-CDL is not explicitly bound to WSDL it can play the same global model role for both SOA services and Web Services.

Common collaborative observable behavior is the phrase we use to indicate describe the behavior of a system of services, for example buyer and seller services, from a global perspective. Each service has an observable behavior that can be described today using WSDL or some other interface description language (e.g. Java). Such observable behavior is described as a set of functions, possibly with parameters, that a service offers coupled with error messages or codes that indicate failure along with the return types for the functions offered. If we used abstract BPEL along with WSDL we can also describe the valid sequences of functions that cannot be done with WSDL or Java alone.[SS1] We refer to this set as the “observable behavior” for a service. This level of “observable behavior” does not describe the order in which functions may be used. It may be the case in many services that a service requires a session open function and a session close function that bracket a long lived interaction. If we captured such

ordering rules then we would have the observable behavior fully specified.

Individual service behaviors can be used in the composition of wider collaboration in which a set of services with their own behaviors could be gainfully employed and guaranteed to work. In order to do so a global model that described the peer to peer observable interactions of such a set of services is required to ensure that the services will in-fact cooperate to a commonly understood script. That script is the global model and that script is what WS-CDL is used to describe.

A global model, ensures that the common collaborative observable behavior is not biased towards the view of any one of the services. Instead it describes as peers the entire collaborative observable behavior of all of the services such that no one service can be said to exert any control over any other service. In effect it described the services as a complete distributed application in which each service plays a distinct role and has distinct relationships with its peer services.

One may think of WS-CDL as a language for describing the observable activities of a set of services some of which are synchronized through some common understanding realized by a specific business interaction between the services or by a declaration of interest in the progress of one service by another (e.g. has the buyer accepted the price offered by the seller). The least interesting scenario is one in which WS-CDL can be used to describe a set of services that never synchronize at all; that is there is no observable relationships and no statement of an unobservable relationship that exists between the services. In this case the services perform a choreography, but effectively on different stages and thus need no form of coordination (e.g. a buyer and seller choreography for WallMart versus a Bloomberg Reuters choreography for the exchange of news items). In all other cases the synchronization is what makes life interesting (e.g. a buyer seller choreography coupled with a seller credit check choreography or indeed a seller shipper choreography).

In WS-CDL the mechanisms for describing the common observable behavior range from specific information alignment (e.g. when a buyer and seller record the fact that an order has been accepted in variables that reside at the buyer and at the seller), interaction (e.g. when a buyer requests a price from a seller and receives a price as a response from the seller) and a declaration of interest in the progress of a choreography (e.g. has the bartering choreography between buyer and seller “started” or has it “finished”). In the first two cases synchronization is explicit and visible as a business related activity (e.g. the observable recording of information and it’s alignment and the description of an information exchange between a buyer and seller) and in the last case (e.g. choreography has “started” or

"finished") it is implicit based on the progress of a choreography and not any business relationships.

2.1 Using WS-CDL

WS-CDL is a description and not an executable language, hence the term "Description" in it's name. It is a language that can be used to unambiguously describe observable service collaborations, we might also refer to this a business protocols within and across domains of control that govern how the services interact.

When WS-CDL is focused on describing collaboration within a domain of control (e.g. a single company or enterprise) WS-CDL is used to describe the internal workflows that involves multiple services (also called end-points) that constitute observable collaborative behavior. The value in so doing is to encourage conformance of services to a negotiated choreography description and to improve interoperability of services through an agreed choreography description. This is no more than describing a business protocol that defines an observable collaboration between services. You can think of it as a way of ensuring services are well behaved with respect to the goals you wish to achieve within your domain.

When the focus of WS-CDL is across domains of control, WS-CDL is used to describe the ordering of observable message exchanges across domains such as the those that govern vertical protocols such as fpML, FIX, TWIST and SWIFT. These protocols have some form of XML data format definition and then proceed to describe the ordering of message exchanges using a combination of prose and UML sequence diagrams. WS-CDL provides an unambiguous way of describing the ordering of message exchanges and in so doing ensure that the services that participate in the observable collaborations based on such vertical standards conform to the choreography description. You can think of it as a way to ensure that services are well-behaved with respect to their common goals across domains.

2.2 Why use WS-CDL?

WS-CDL can be used to ensure interoperability within and across domains of control to lower interoperability issues, such as downtime, and create solutions within and across domains of control.

WS-CDL can be used to ensure that the total cost of software systems in a distributed environment, within a domain of control and across the world-wide-web is lowered by guaranteeing that the services that participate in a choreography are well behaved on a continuous basis.

Both of these benefits translate into greater up-time and so increase top line profits. At the same time they translate into less testing time and so reduce cost of delivery which decreases bottom line costs.

2.3 The Structure of WS-CDL

WS-CDL is a layered language that provides different levels of expressability to describe a choreography. These levels are illustrated below in Figure 1.

At the top most level for any WS-CDL there is a package that contains all other things. All choreographies described in WS-CDL will include as a minimum a set of Roles that are defined as some sort of behavior (i.e. a WSDL description) and so represent our WS-CDL notion of a service, Relationships between those roles, Channels used by roles to interact and a Choreography block that uses channels to describe Interaction. What the choreography describes at this level is a basic set of typed and unambiguous service connections that enable the various roles to collaborate in order to achieve some common goal.

Adding further ordering rules through Structured composition allows Interactions and Choreographies (which are just logical groupings of interactions) to be combined into sequences, parallel activities and so on.

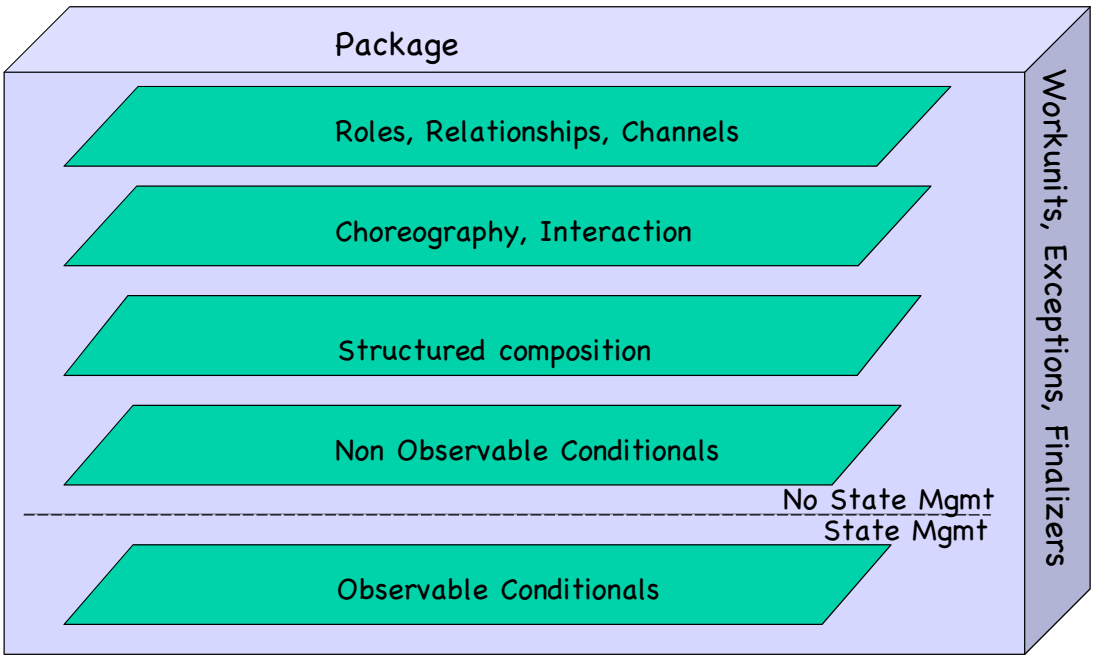


Figure 1: Layered structure of WS-CDL

Adding Non-Observable Conditionals makes it is possible to model branching based on observing changes in the interactions that occur (e.g. one might observe an exchange between a buyer and a seller which is said to be terminated when a "completed" interaction is observed).

If we have no observable conditionals then it is not necessary to perform any explicit state management at the roles that are interacting because we have not needed to express any explicit computation (e.g. totalOrderValue EQUALS expectedOrderValue) required of an observable condition. None of roles used in choreographies of this type have the need for any state variables to control a choreography, rather the progression of a choreography is expressed purely in terms of observable interactions and use observation to determine their state with respect to the other roles.

Some business protocols are defined exposing specific business rules. These constitute shared knowledge between the concerned roles. For example we may terminate an order completion between a buyer and a seller when we calculate that the items delivered match the original order. The business rule in this example is the shared constraint that buyer_quantity equals completion_quantity. At some level the roles must have some shared knowledge of both variables and their values. When business rules of this nature become part of the business protocol such Observable Conditionals can be added into a choreography and which implies state management is needed.

WS-CDL provides some basic interfaces for state management defining it's requirements as a coordination protocol. The specifics of state management is left as an implementation detail for the community.

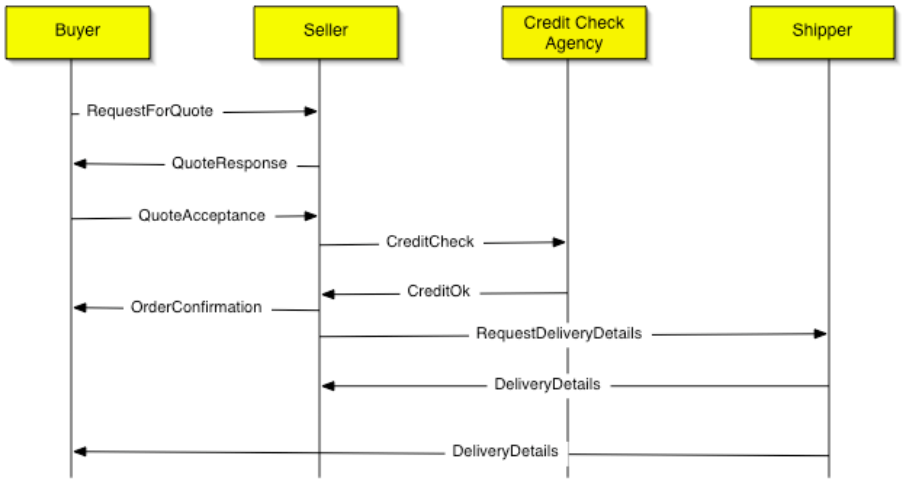
3 Getting Started with WS-CDL

In order to understand WS-CDL is best to illustrate it through the use of an example. In this section we shall introduce an example and use it throughout the rest of document building on it to illustrate different parts of WS-CDL. The Appendicies have the full listing of the various WS-CDL encodings of the example as well as a url to the WS-CDL descriptions. In all cases the WS-CDL descriptions have been tested against at least one implementation of WS-CDL having been constructed in a validating editor.

3.1 An Example

The example that we use concerns the collaborative behavior of a buyer, a seller, a credit check agency and a shipper. The buyer interacts with the seller to determine a price. When a price is acceptable to the buyer the buyer interacts with the seller to order the relevant goods based on this price whereupon the seller checks their credit worthiness by interacting with a credit agent and if this is acceptable requests a delivery date by interacting with the shipper. In our example the shipper interacts directly with the buyer once an agreed delivery date has been achieved and informs the buyer of the delivery details.

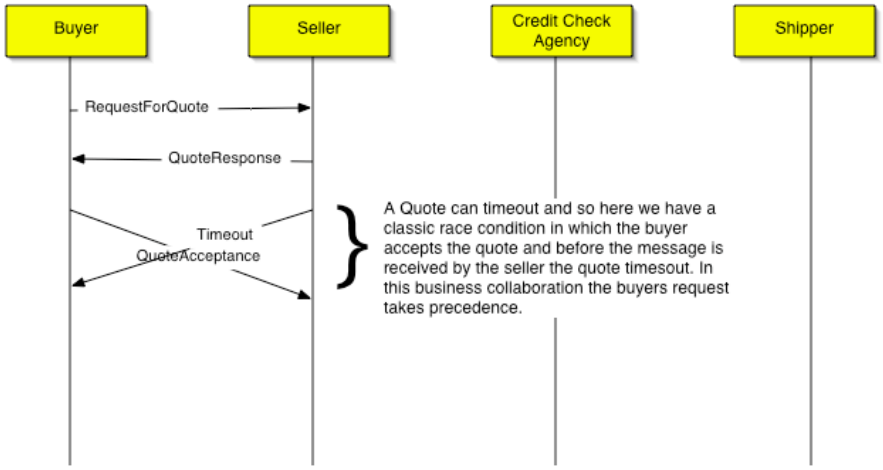
The example is further illustrated by means of a number of sequence diagrams below:



Normal Collaboration

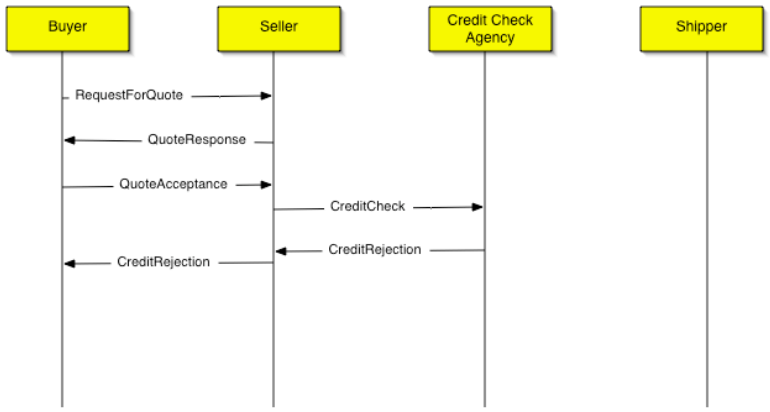
The Normal Collaboration, illustrated in Figure 2, shows the buyer requesting a quote and the seller responding with a quote. The buyer then accepts the quote, which is akin to placing the order. As a result the seller checks the buyers credit rating. As the buyers credit rating is

ok the seller then confirms the order with the buyer and requests from the shipper delivery details which are passed back to the seller by the shipper. The shipper will have picked up all that is necessary from the seller, who received it from the buyer as part of the order placement, all of the details necessary to communicate directly with the buyer so that delivery details can be passed back from the shipper to the buyer.



Quote Timeout Collaboration

The Quote Timeout Collaboration, illustrated in Figure 3, illustrated the buyer requesting a quote, the seller sending back a quote response that has a timeout associated with the quote. If the buyer fails to act on the quote in time (before the timeout elapses) the buyer may not honor the quote. In the scenario presented we show the opportunity for the buyer to accept the quote just as the seller decides that the quote has timed-out. This demonstrates a classic race condition between the parties.

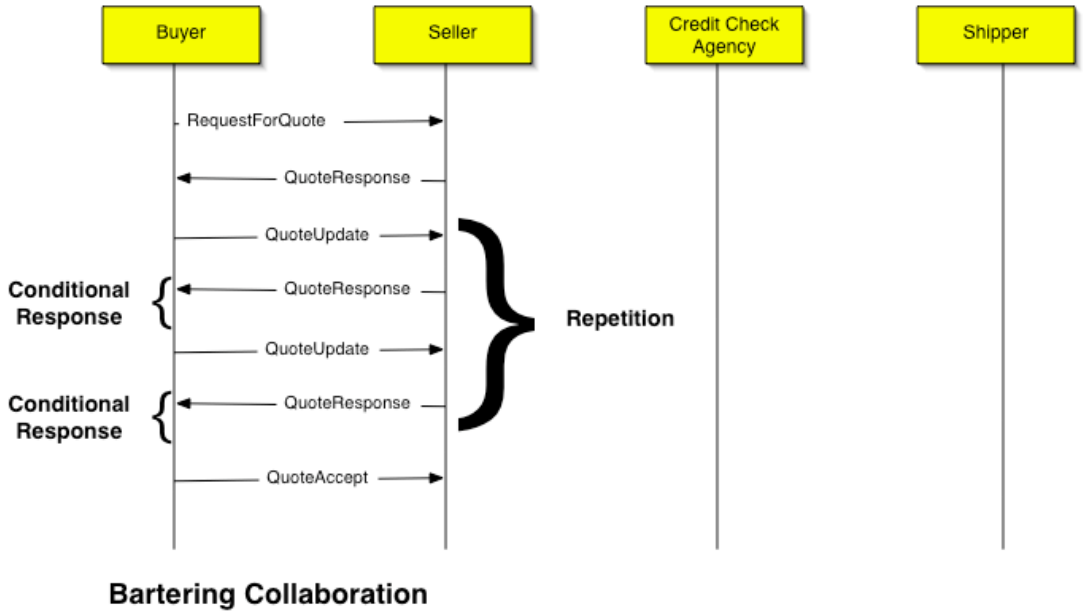


Credit Rejection Collaboration

Figure 4 shows a credit check rejection for the buyer. After the buyer requests the quote, the seller responds, the buyer accepts the it, the seller then checks the credit rating for the buyer. In this case the credit check agency determines that the credit rating is low and returns a

credit rejection to the seller who in turn returns a credit rejection to the buyer, terminating the collaboration.

The final scenario to introduce is that of the bartering collaboration. This is illustrated in Figure 5 below.



Bartering Collaboration

In this collaboration the buyer requests a quote from the seller who responds with a quote. Thereafter the buyer may request an updated quote, filling in a desired price and quantity, from the seller. The seller may respond by accepting the quote, returning a quote response message to the seller. If the seller does not respond then the buyers update is subject to a timeout in the same way that the sellers quote is valid for a specified duration. In receipt of a quote response from the seller the buyer may accept the quote (and by so doing enter into the act of buying with the seller) or may request a further updated quote or simply do nothing at all –allowing the quote to timeout.

We have used a heavily annotated form of sequence diagram to describe the business collaboration protocol necessary for the buyer, seller, credit agency and shipper to go about their business. WS-CDL is very much designed to enable the entire business collaboration protocol to be described in an unambiguous manner. We hope that this becomes self evident to the reader as we walk through constructing the WS-CDL description for this example.

3.2 Interactions Oriented Design

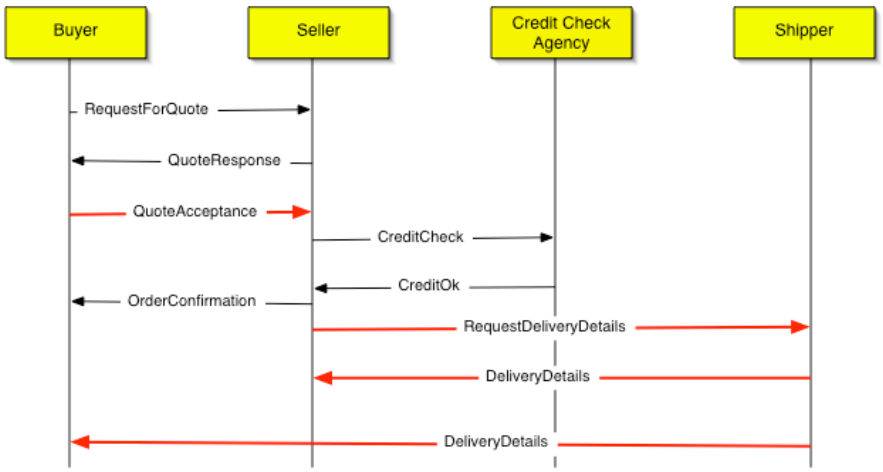
In this section we introduce the fundamental concept of an interaction, which underpins WS-CDL. Thus defining roles, tokens, channels, relationships, participants and variables necessary to properly describe the business protocol in the example.

3.2.1 Interactions

An interaction is the realization of a collaboration between roles or participants. The difference between roles and participants will be detailed later on for now we can consider them as synonymous.

A collaboration is a message exchange between the swim lanes of a sequence diagram. We shall focus on a portion of the normal collaboration between a buyer and seller to illustrate the use of interactions in WS-CDL.

In the diagram below we show the same normal collaboration described earlier but have changed the colours of the relevant arrows to red. These red arrows are what we shall model with our interactions.



Normal Collaboration

In the example when the buyer accepts a quote, it does some extra things so that the seller, on behalf of the buyer, can pass certain contact details to a third party who can contact the buyer to inform the buyer of the delivery details. The buyer sends to the seller a quote acceptance to the seller. This exchange of information is further annotated to ensure that the collaboration includes the call back details for a third party to contact the buyer with the delivery details. When the seller receives the necessary information for quote acceptance the seller passes this information to the shipper, as part of a collaboration, to determine suitable delivery details. The shipper then uses the additional information to respond to the buyer directly.

It is this collaboration between the buyer, seller and shipper than we shall use to illustrate the use of interactions.

The WS-CDL fragment for these red arrows is illustrated below:

```
<interaction name="Buyer accepts the quote and engages in the act of buying"
        operation="quoteAccept" channelVariable="Buyer2SellerC">
        <description type="description">Quote Accept</description>
        <participate relationshipType="BuyerSeller"  fromRole="BuyerRoleType"  toRole="SellerRoleType" />
        <exchange name="Accept Quote" informationType="QuoteAcceptType" action="request">
        </exchange>
</interaction>

<interaction name="Buyer send channel to seller to enable callback behavior"
        operation="sendChannel" channelVariable="Buyer2SellerC">
        <description type="description">Buyer sends new channel to pass on to shipper</description>
        <participate relationshipType="BuyerSeller" fromRole="BuyerRoleType" toRole="SellerRoleType" />
        <exchange name="sendChannel" channelType="2BuyerChannelType" action="request">
                <send variable="cdl:getVariable('DeliveryDetailsC','','')" />
                <receive variable="cdl:getVariable('DeliveryDetailsC','','')" />
        </exchange>
</interaction>

<interaction name="Seller requests delivery details - passing channel for buyer and shipper to interact"
operation="requestShipping" channelVariable="Seller2ShipperC">
        <description type="description">Request delivery from the shipper</description>
        <participate relationshipType="SellerShipper" fromRole="SellerRoleType" toRole="ShipperRoleType" />
        <exchange name="sellerRequestsDelivery" informationType="RequestDeliveryType" action="request">
        </exchange>
        <exchange name="sellerReturnsDelivery" informationType="DeliveryDetailsType" action="respond">
        </exchange>
</interaction>


<interaction name="Shipper forward channel to shipper" operation="sendChannel"
channelVariable="Seller2ShipperC">
        <description type="description">Pass channel from buyer to shipper</description>
        <participate relationshipType="SellerShipper" fromRole="SellerRoleType" toRole="ShipperRoleType" />
        <exchange name="forwardChannel" channelType="2BuyerChannelType" action="request">
                <send variable="cdl:getVariable('DeliveryDetailsC','','')" />
                <receive variable="cdl:getVariable('DeliveryDetailsC','','')" />
        </exchange>
</interaction>

<interaction name="Shipper sends delivery details to buyer" operation="deliveryDetails"
channelVariable="DeliveryDetailsC">
        <description type="description">Pass back shipping details to the buyer</description>
        <participate relationshipType="ShipperBuyer" fromRole="ShipperRoleType" toRole="BuyerRoleType" />
        <exchange name="sendDeliveryDetails" informationType="DeliveryDetailsType" action="request">
        </exchange>
</interaction>
```

Interactions are descriptions of one or more exchanges between a sender and a receiver. Interactions are labeled with an operation name that could be mapped to a WSDL operation, a topic in a publish-and-subscribe environment or a message queue in a point-to-point messaging environment. Interactions take place on a channel, as indicated by its channelVariable, the variable itself will have been declared to be of a particular channelType. The participating

relationship restricts the interaction to be between roles that are valid for that relationship.

Each exchange names the sender and receivers roles between which exchanges in the interaction take place. Each exchange also names the type of things to be exchanges and the direction of the exchange (e.g. a request or a response).

In order to describe our interactions we shall have to define the channelTypes for the channelVariable in the interaction and for the channelTypes that enables channels to be passes between roles.

We shall also have to define a relationshipTypes for the participate relationshipType in the interaction and in the definition of the channelTypes.

We shall have to define the roleTypes that used in the participate relationshipType as well in the definition of the relationTypes that we define.

We shall have to define some informationTypes for the informationType that is part of the exchange in an interaction.

We shall have to define an XML namespace that we shall call xmlns:bs, that is used as an alias for the full namespace to ground our informationTypes relative to some business focused XML schema.

We shall take the second interaction as our base for describing the main parts of an interaction.

Every interaction has a name which in this case is "**Buyer send channel to seller to enable callback behavior**". This simply allows the interaction to be referenced. We have chosen a long name for the sake of illustration but in practice shorter name would be more likely.

Every interaction has an operation and a channelVariable. The operation denotes the intended service function, implemented by the receiving end of the service that is itself denoted by the channelVariable. In our case the channelVariable is "**Buyer2SellerC**" and the operation is "**sendChannel**". So the buyer requests to call a function called "**sendChannel**" that is implemented by the seller.

At this point we can start to see that channels are an important part of the structure of a choreography. The channels represent the possible communication opportunities that exist between the services. We can also see that the operations in how they are defined and bound to the choreography.

The description element in the interaction is simply a documentation mechanism used to ensure further clarity for our example.

Every interaction will have a participate. This is used to describe the direction of the interaction which is denoted by relationshipType and the fromRole and toRole. Together these components describe that the interaction takes place against a relationship called "**BuyerSeller**" and between the roles "**BuyerRoleType**" and "**SellerRoleType**".

Every interaction has an exchange element that has a name used to identify the element and has a either a channelType or informationType that describe the type of thing being exchanged and an action that denotes the direction of the exchange (i.e. a request or a response). Thus in our exchange we would see that the buyer is sending something of type "**2BuyerChannelType**" which is a "**request**" from the buyer to the seller.

In our example we can go further as the exchange element also describes the variable in which an instance of a "**2BuyerChannelType**" is held at the buyer – namely the '**DeliveryDetailsC**' variable – and where its exchange will be rendered – in this case into a variable of the same type called '**DeliveryDetailsC**' that resides at the seller. In the send variable and receive variable the variables are accessed using a WS-CDL function which names the variable and optionally the role at which the variable resides. In this case the latter parameter is not used and is inferred based on the context. In this case the context is based on the two ends of the channel and direction of the interaction.

What we have described thus far is the fundamental construct in WS-CDL. We use this to drive the description of the other elements. The order in which we shall define these is the order of dependency of the elements. Thus we shall do it in the following order:

1. Define our roleTypes,
2. Define our relationshipTypes,
3. Define our informationTypes,
4. Define our tokenType,
5. Define our channelTypes

3.2.2 Roles

There are 4 roles that are played out in the example. These are the “buyer” the “seller” the “credit agency” and the “shipper”.

A role in WS-CDL is a named behavior and it is clear from the example that the roles we have identified have behavior relative to one another. This is the core of collaboration – identifying common interaction between peers.

In this example we shall assume that none of the roles have any web services defined for them and so we can omit the interface attribute.

Later on we shall recast the example based such that one or more of the roles do have existing web services defined for them.

```
<roleType name="BuyerRoleType">
  <description type="documentation">The Behavior embodied by a buyer</description>
  <behavior name="BuyerBehavior" />
</roleType>

<roleType name="SellerRoleType">
  <description type="documentation">The behavior embodied by a seller</description>
  <behavior name="SellerBehavior" />
</roleType>

<roleType name="CreditCheckerRoleType">
  <description type="documentation">The behavior embodied by a credit checker </description>
  <behavior name="CreditCheckerBehavior" />
</roleType>

<roleType name="ShipperRoleType">
  <description type="documentation">The behavior embodied by a shipper service</description>
  <behavior name="ShipperBehavior" />
</roleType>
```

The roles identified are illustrated above in a WS-CDL fragment. This soley provides a name for the type of role that we are defining. In the case of the buyer we name it "**BuyerRoleType**" and then the "**BuyerBehavior**" is a name used to bind the implementation of an "**BuyerRoleType**" at some service end point.

The abstract syntax for roles is illustrated below:

```
<roleType name="ncname">
        <description type=" documentation" </description>?
        <behavior name="ncname" interface="qname"? />+
</roleType>
```

3.2.3 Relationships

Once we have some roles defined we can define the relationships. In WS-CDL a relationship declares an intention to interact between two roles. In a sequence diagram this is akin to any two of the actors (check the name here) who have a connecting arrow in any direction. In our example we have relationships between the

- buyer and seller
- seller and credit agency
- seller and shipper
- buyer and shipper

Defining these in WS-CDL would look like the following WS-CDL fragment:

```
<relationshipType name="BuyerSeller">
  <role type="BuyerRoleType" />
  <role type="SellerRoleType" />
</relationshipType>

<relationshipType name="SellerCreditCheck">
  <role type="SellerRoleType" />
  <role type="CreditCheckerRoleType" />
</relationshipType>

<relationshipType name="SellerShipper">
  <role type="SellerRoleType" />
  <role type="ShipperRoleType" />
</relationshipType>

<relationshipType name="ShipperBuyer">
  <role type="ShipperRoleType" />
  <role type="BuyerRoleType" />
</relationshipType>
```

A relationship comprises a name and two role types. We use the convention in this document that the first role type defines the "from" role and the second the "to" role though WS-CDL does not require this directionality to be declared in a relationship. The "**ShipperBuyer**" role in the example has the "**ShipperRoleType**" as the first role and the "**BuyerRoleType**" as the second role.

The abstract syntax for relationships is defined as follows:

```
<relationshipType name="ncname">
        <role type="qname" behavior="list of ncname"? />
         <role type="qname" behavior="list of ncname"? />
</relationshipType>
```

3.2.4 Information Types

The informationTypes in a choreography are used to describe the types for many of the variables that we might use in a choreography. They are used to describe the types of messages that we might send between roles in an interaction.

The abstract syntax for defining information types is as follows:

```
<informationType name="ncname"
          type="qname"?|element="qname"?
          exceptionType="true"|"false"? />
```

The complete set of information types for our example is illustrated below. They are all message types used as typing information in our interactions except for a couple of general types that we have defined

such as the "**StringType**" – used for the tokens later on – and the "**BooleanType**" used for our control variable it managing the bartering process.

```
<informationType name="BooleanType" type="xsd:boolean" />
<informationType name="StringType" type="xsd:string" />
<informationType name="RequestForQuoteType" type="bs:RequestForQuote">
    <description type="documentation">Request for quote message</description>
</informationType>

<informationType name="QuoteType" type="bs:Quote">
    <description type="documentation">Quote message</description>
</informationType>

<informationType name="QuoteUpdateType" type="bs:QuoteUpdate">
    <description type="documentation">Quote Update Message</description>
</informationType>

<informationType name="QuoteAcceptType" type="bs:QuoteAccept">
    <description type="documentation">Quote Accept Message</description>
</informationType>

<informationType name="CreditCheckType" type="bs:CreditCheckRequest">
    <description type="documentation">Credit Check Message</description>
</informationType>

<informationType name="CreditAcceptType" type="bs:CreditAccept">
    <description type="documentation">Credit Accept Message</description>
</informationType>

<informationType name="CreditRejectType" type="bs:CreditReject">
    <description type="documentation">Credit Reject Message</description>
</informationType>

<informationType name="RequestDeliveryType" type="bs:RequestForDelivery">
    <description type="documentation">Request Delivery Message</description>
</informationType>

<informationType name="DeliveryDetailsType" type="bs:DeliveryDetails">
    <description type="documentation">Delivery Details Message</description>
</informationType>
```

3.2.5 Tokens

A token provides a mechanism for defining an alias for an information type. Token locators can then be defined to locate this particular token from within a message type. We define some tokens here because they are needed when defining channel types.

The tokens we define will refers to a service reference that is a url to the web service. In this context a token defines an alias to the web service so that we can treat refer to it by a shorter name. In our example we will not reference any web service url so we define it as a StringType.

For our example we would define the tokens needed for our channels as follows:

```
<token name="BuyerRef" informationType="StringType" />
<token name="SellerRef" informationType="StringType" />
<token name="CreditCheckRef" informationType="StringType" />
<token name="ShipperRef" informationType="StringType" />
```

The abstract syntax for defining a token is as follows:

```
<token  name="ncname"  informationType="qname" />
```

3.2.6 Channels

Finally, having defined our roles and tokens we are in a position to define our channels. The channels are the principle mechanism used to realize an interaction.

The abstract syntax of a channel definition is provided below and we shall walk through the steps we need to take to fully define the channels for the example presented earlier.

```
<channelType name="Buyer2SellerChannelType">
        <passing channel="2BuyerChannelType" new="true">
                <description
                        type="description">Able to pass channel to enable shipper to talk to
                </description>
        </passing>
        <role type="SellerRoleType" />
        <reference>
                <token name="SellerRef" />
        </reference>
</channelType>
```

```
<channelType  name="ncname"
   usage="once"|"unlimited"?
   action="request-respond"|"request"|"respond"? >

        <passing  channel="qname"
           action="request-respond"|"request"|"respond"?
           new="true"|"false"? />*
        <role  type="qname"  behavior="ncname"? />

        <reference>
                <token name="qname"/>
        </reference>

        <identity>
                <token name="qname"/>+
        </identity>?

</channelType> >
```

The *role type* name declares who is the service provider for this channel. For example we might have a channel between the buyer and the seller

that enables the collaboration between the buyer and seller in the sequence diagrams. In this case the seller is playing the role of service provider and the buyer the role of client and so a channel that we might name "**Buyer2SellerChannelType**" would have a *role type* of "**SellerRoleType**".

The full definition for the "**Buyer2SellerChannelType**" is defined below:

The reference element of the channel type definition is a token that can be used as a place holder for an end point reference. When token name refers to a string, as in this case, the token name is simply a place holder with no specific meaning. In this case the token name refers to "**SellerRef**".

The rest of the Channel Types for the example are defined below:

```
<channelType name="Seller2CreditCheckChannelType">
        <role type="CreditCheckerRoleType" />
        <reference>
                <token name="CreditCheckRef" />
        </reference>
</channelType>

<channelType name="2BuyerChannelType" action="request">
        <role type="BuyerRoleType" />
        <reference>
                <token name="BuyerRef" />
        </reference>
</channelType>

<channelType name="Seller2ShipperChannelType">
        <passing channel="2BuyerChannelType">
                <description
                        type="description">Pass channel through to shipper
                </description>
        </passing>
        <role type="ShipperRoleType" />
        <reference>
                <token name="ShipperRef" />
        </reference>
</channelType>
```

In our example we have two sorts of channel types defined. The "**Buyer2SellerChannelType**" and the "**Seller2ShipperChannelType**" include passing channel details, whereas the "**Seller2CreditCheckChannelType**" and the "**2BuyerChannelType**" do not have this attribute.

In our example when the buyer decides to accept the quote two things happen. Firstly the buyer sends a message to the seller accepting the quote, then sends a further message to the seller with the details of a channel that it passes to the seller. The seller does a similar thing when it requests delivery details from the shipper; sending the request for delivery details and then sending the channel it received from the buyer on to the shipper. This is all done so that the buyer can receive delivery details back a third party that to which it has no knowledge. This is achieved through channel passing, to do it we add details to a

channel allowing the channel to pass other channels of a particular type. In our example the type of channel to be passed is the "**2BuyerChannelType**".

3.3 Choreographies, Sequences, Choices and Workunits

We can now consider describing the choreography itself. We shall do this by taking the normal collaboration and building a choreography based on the first interaction within that collaboration.

3.3.1 Choreographies

Before we can really start describing the interactions between the roles we need to define some basic variables that we shall use. We have already seen that we need to have a channel variable in an interaction. We shall also need some way of controlling the iteration for the bartering part of our choreography.
What we shall need to do is define instances for the

```
<choreography name="Main" root="true">
    <description type="description">Collaboration between buyer, seller, shipper, credit chk</description>

    <relationship type="BuyerSeller" />
    <relationship type="SellerCreditCheck" />
    <relationship type="SellerShipper" />
    <relationship type="ShipperBuyer" />

    <variableDefinitions>
        <variable name="Buyer2SellerC"
            channelType="Buyer2SellerChannelType"
            roleTypes="BuyerRoleType">
            <description type="description">
                Principle channel used to enable interaction between buyer
                and seller for price requests, price confirms and orders
            </description>
        </variable>
        <variable name="Seller2ShipperC"
            channelType="Seller2ShipperChannelType"
            roleTypes="SellerRoleType">
            <description type="description">
                Seller to shipper channel - used to pass a channel to effect
                interaction with the buyer
            </description>
        </variable>
        <variable name="Seller2CreditChkC"
            channelType="Seller2CreditCheckChannelType"
            roleTypes="SellerRoleType">
            <description type="description">
                Seller to Credit Check Channel used to check credit for buyers to
                determine if we do business with them
            </description>
        </variable>
        <variable name="DeliveryDetailsC"
            channelType="2BuyerChannelType"
            roleTypes="BuyerRoleType SellerRoleType ShipperRoleType" />
            <description type="description">
                Channel created by the buyer to pass to third parties so that
                They can communicate with the buyer without have linkage
            </description>
        </variable>
        <variable name="barteringDone"
            informationType="BooleanType"
            roleTypes="BuyerRoleType SellerRoleType">
```

"**Buyer2SellerChannelType**","**Seller2ShipperChannelType**", "**Seller2ShipperChannelType**" and "**2BuyerChannelType**" that we shall name "**Buyer2SellerC**", "**Seller2ShipperC**", "**Seller2ShipperC**" and "**DeliveryDetailsC**" respectively. We shall also define a variable called "**barteringDone**" that will be of type "**BooleanType**" that will be used to control the bartering iteration.

We also need to declare the relationships for which we shall describe behavior in the form of interactions that will be the basis of this choreography. In our case this is the "**BuyerSeller**", "**SellerCreditCheck**", "**SellerCreditCheck**"and "**ShipperBuyer**". Relationships are declared to act as a cross-check against the channel usage within the choreography. This allows implementations of CDL editors to cross-check the relationships that can be inferred from the channels used against that that is explicitly declared.

This first part of the actual choreography is illsuratedabove. We have named our choreography name="**Main**", we have marked the choreography as the root, root="**true**" so that we know that this initiates the whole thing and we have declared our relationship types and defined our variables in the variableDefinitions section.

Taking just one of the variables defined, "**DeliveryDetailsC**" ,we can see that variables have a variable name and a **type** which may be a channelType or an informationType. Variables also have a roleTypes that determines where the variable resides. We can have as many roleTypes as we like so that the same named variable resides at all of the roles. What this means is that each role has a variable of the same name. It does not mean that the variable is the same. In our case the list of roleTypes is "**BuyerRoleType SellerRoleType ShipperRoleType**". We have seen these variables in the different roles in the interaction described above. To make the variable

```
<choreography name="ncname"
    complete="xsd:boolean XPath-expression"?
    isolation="true"|"false"?
    root="true"|"false"?
    coordination="true"|"false"? >
     <relationship type="qname" />+
     variableDefinitions?
     Choreography-Notation*
    Activity-Notation
     <exceptionBlock name="ncname">
      WorkUnit-Notation+
     </exceptionBlock>?
     <finalizerBlock name="ncname">
    WorkUnit-Notation
     </finalizerBlock>*
</choreography>

<variableDefinitions>
  <variable name="ncname"
     informationType="qname"?|channelType="qname"?
     mutable="true|false"?
     free="true|false"?
     silent="true|false"?
     roleTypes="list of qname"? />+
</variableDefinitions>
```

have the same information we ensure that we align them in some way

which can only be done through some interaction and exchange. This is exactly what is done in the interaction already described.

3.3.2 Sequences

We start off by defining a sequence. The sequence encapsulates the overall choreography that we are modeling and in it we shall place our first interaction. The first interaction starts the choreography and it is the one between the buyer and the seller in which the buyer requests a quote and the seller responds with a quote. Clearly we can model this as a single interaction with a request/response exchange. The buyer initiates the exchange of information that defines the request and the seller responds with an exchange that defines the response. The interaction is part of a sequence illsuratedbelow.

The first element of this sequence is the interaction that has the name "**Buyer requests a Quote - this is the initiator**" and the operation "**requestForQuote**". The "**requestForQuote**" operation is something implemented by the seller as part of its service description (i.e. WSDL). It will use the channel that we have defined called "**Buyer2SellerC**" and it has initiate set to "**true**". This means that the channel will use "**Buyer2SellerC**" to realize the first interaction in the choreography.

The interaction declares the relationship type that is participating in the interaction and in the exchanges the roles involved. These act as cross-check to ensure that the channel is the correct channel for the named roles and that the roles have a pre-declared relationship. In this case the participate relationshipType is "**BuyerSeller**" and the roles are "**BuyerRoleType**" and "**SellerRoleType**" respectively.

```
<?xml version="1.0" encoding="UTF-8" ?>
<package name="BuyerSellerCDL" author="Steve Ross-Talbot"
    version="1.0" targetNamespace="www.pi4tech.com/cdl/BuyerSeller"
    xmlns="http://www.w3.org/2004/12/ws-chor/cdl"
    xmlns:bs="http://www.pi4tech.com/cdl/BuyerSellerExample-1">
    <description type="description">This is the basic BuyerSeller Choreography Description</description>

    .....

    <choreography name="Main" root="true">
        <description type="description">Collaboration between buyer, seller, shipper, credit chk</description>

        .....

      <sequence>
        <interaction name="Buyer requests a Quote - this is the initiator"
            operation="requestForQuote" channelVariable="Buyer2SellerC" initiate="true">
            <description type="description">Request for Quote</description>

            <participate relationshipType="BuyerSeller" fromRole="BuyerRoleType" toRole="SellerRoleType" />
            <exchange name="request" informationType="RequestForQuoteType" action="request">
                <description type="description">Requesting Quote</description>
            </exchange>
            <exchange name="response" informationType="QuoteType" action="respond">
                <description type="description">Quote returned</description>
            </exchange>
        </interaction>

        .....

      </sequence>
    </choreography>
</package>
```

Finally we have the exchanges that make up this interaction. Interactions can have one or more exchanges. This one only requires two because it is a request/response pattern. Thus the first exchange that we have named as "**request**" described an exchange from the buyer to the seller in which a "**RequestForQuoteType**" message is exchanged as a "**request**". The second named "**response**" exchanges is the other direction from seller to buyer and uses a "**QuoteType**" message that is marked as a "**respond**".

3.3.3 Repeating Workunits

Once we initiate this first interaction we need to use some form of repetition to describe the collaboration pattern for the bartering-process. In this process the buyer may request an update to a quote, perhaps putting in a target price, and the seller may either accept or reject the updated quote from the buyer. The buyer may decide that the initial quote is acceptable and take action accordingly or the quote itself may timeout because in our example the price/quote is valid only for a specified period.

We need to model repetition to control the bartering process itself and we need some form of choice in which the choices made are the buyer deciding to update quote, the buyer deciding to accept the quote or the quote timing-out. Such a workunit is illustrated below. For now ignore the choice construct as we shall deal with it and it's elements in the next section.

```
<workunit name="Repeat until bartering has been completed" repeat="barteringDone = false">
    <choice>
        <silentAction roleType="BuyerRoleType">
            <description type="description">Do nothing - let the quote timeout</description>
        </silentAction>

        <sequence>
            <interaction name="Buyer accepts the quote and engages in the act of buying"
                operation="quoteAccept" channelVariable="Buyer2SellerC">
                <description type="description">Quote Accept</description>

                <participate relationshipType="BuyerSeller"
                    fromRole="BuyerRoleType" toRole="SellerRoleType" />
                <exchange name="Accept Quote" informationType="QuoteAcceptType"
                     action="request">
                </exchange>
            </interaction>
            <interaction name="Buyer send channel to seller to enable callback behavior"
                operation="sendChannel" channelVariable="Buyer2SellerC">
                <description type="description">Buyer sends channel to pass to shipper</description>
                <participate relationshipType="BuyerSeller"
                    fromRole="BuyerRoleType" toRole="SellerRoleType" />
                <exchange name="sendChannel" channelType="2BuyerChannelType" action="request">
                    <send variable="cdl:getVariable('DeliveryDetailsC','','')" />
                    <receive variable="cdl:getVariable('DeliveryDetailsC','','')" />
                </exchange>
            </interaction>
            <assign roleType="BuyerRoleType">
                <copy name="copy">
                    <source expression="true" />
                    <target variable="cdl:getVariable('barteringDone','','')" />
                </copy>
            </assign>
        </sequence>

        <sequence>
            <interaction name="Buyer updates the Quote - in effect requesting a new price"
                operation="quoteUpdate" channelVariable="Buyer2SellerC">
                <description type="documentation">Quot Update</description>
                <participate relationshipType="BuyerSeller"
                    fromRole="BuyerRoleType" toRole="SellerRoleType" />
                <exchange name="updateQuote"
                    informationType="QuoteUpdateType" action="request">
                </exchange>
                <exchange name="acceptUpdatedQuote"
                    informationType="QuoteAcceptType" action="respond">
                    <description type="documentation">Accept Updated Quote</description>
                </exchange>

            </interaction>
        </sequence>
    </choice>
</workunit>
```

In this workunit, that we have named "**Repeat until bartering has been completed**", we have a repetition condition "**barteringDone = false**". This condition evaluates the declared variable "**barteringDone**" to see if it is false. If false the workunit proceeds and repeats until such time as the condition ("**barteringDone = false**") evaluates to true.

3.3.4 Choices

The body of the above workunit is a choice. The choice element is used to declare possible alternative paths in a choreography. In our example there are only three choices that can be made at this point. The quote could timeout, the buyer decides to accept the quote or the buyer requests an update to the existing quote. We model these as a silentAction for the timeout of the quote – we shall change this later on –, as a simple sequence for the buyer accepting the quote and as a complex sequence to handle the bartering process itself. After adding the basic components the workunit is illsuratedbelow:

```
<workunit name="Repeat until bartering has been completed" repeat="barteringDone = false">
    <choice>
        <silentAction roleType="BuyerRoleType">
            <description type="description">Do nothing - let the quote timeout</description>
        </silentAction>

        <sequence>
            .....
        </sequence>

        <sequence>
            .....
        </sequence>
    </choice>
</workunit>
```

We shall start to elaborate the choice by defining the complex sequence that will control the bartering collaboration. What we shall describe are the interactions needed for the bartering process. We start by defining the interaction from buyer to seller to update the price, interaction name="**Buyer updates the Quote - in effect requesting a new price**", and the exchanges that comprise this interaction. We model this as an interaction within a sequence such that the exchanges are the outbound "**updateQuote**" and the inbound "**acceptUpdatedQuote**". This illsuratedbelow:

```
<workunit name="Repeat until bartering has been completed" repeat="barteringDone = false">
    <choice>
        <silentAction roleType="BuyerRoleType">
            <description type="description">Do nothing - let the quote timeout</description>
        </silentAction>

        <sequence>
            .....
        </sequence>

        <sequence>
            <interaction name="Buyer updates the Quote - in effect requesting a new price"
                operation="quoteUpdate" channelVariable="Buyer2SellerC">
                <description type="documentation">Quot Update</description>
                <participate relationshipType="BuyerSeller"
                    fromRole="BuyerRoleType" toRole="SellerRoleType" />
                <exchange name="updateQuote"
                    informationType="QuoteUpdateType" action="request">
                </exchange>
                <exchange name="acceptUpdatedQuote"
                    informationType="QuoteAcceptType" action="respond">
                    <description type="documentation">Accept Updated Quote</description>
                </exchange>
            </interaction>
        </sequence>
    </choice>
</workunit>
```

The final choice element in this workunit is the element that manages the repeat variable "**barteringDone**". This sequence has two interactions, named "**Buyer accepts the quote and engages in the act of buying**" and "**Buyer send channel to seller to enable callback behavior**". The first describes the interaction between buyer and seller to accept the quote – this has an exchange called "**Accept Quote**" – and thus place an order. The second describes the additional information passed to the seller by the buyer – this has an exchange called "**sendChannel**" – so that a third party, in our case the shipper, may send back delivery details to the buyer without knowing the buyer before hand. To effect the exchange we need to make sure that the channel variable NAME that resides at both the buyer and the seller independently of each other is used as the output variable at the buyer and the input variable at the seller. To do this we use the WS-CDL function that gets a variable at a specified role. This is why we see "**cdl:getVariable('DeliveryDetailsC','','')**" and "**cdl:getVariable('DeliveryDetailsC','','')**" in the exchange. The role is omitted because it can be inferred through the channel used for interaction. The final part of the sequence is to change the value in the variable, "**barteringDone**", to "**true**"so that the workunit repeat condition evaluates to false and the workunit terminates. To do this we use an assign element and indicate the actual variable and where it resides my using the "**cdl:getVariable('barteringDone','','')**" WS-CDL function. This part of the workunit and choice is illsuratedbelow:

```
<workunit name="Repeat until bartering has been completed" repeat="barteringDone = false">
    <choice>
        <silentAction roleType="BuyerRoleType">
            <description type="description">Do nothing - let the quote timeout</description>
        </silentAction>

        <sequence>
            <interaction name="Buyer accepts the quote and engages in the act of buying"
                operation="quoteAccept" channelVariable="Buyer2SellerC">
                <description type="description">Quote Accept</description>
                <participate relationshipType="BuyerSeller"
                    fromRole="BuyerRoleType" toRole="SellerRoleType" />
                <exchange name="Accept Quote" informationType="QuoteAcceptType"
                     action="request">
                </exchange>
            </interaction>
            <interaction name="Buyer send channel to seller to enable callback behavior"
                operation="sendChannel" channelVariable="Buyer2SellerC">
                <description type="description">Buyer sends channel to pass to shipper</description>
                <participate relationshipType="BuyerSeller"
                    fromRole="BuyerRoleType" toRole="SellerRoleType" />
                <exchange name="sendChannel" channelType="2BuyerChannelType" action="request">
                    <send variable="cdl:getVariable('DeliveryDetailsC','','')" />
                    <receive variable="cdl:getVariable('DeliveryDetailsC','','')" />
                </exchange>
            </interaction>

            <assign roleType="BuyerRoleType">
                <copy name="copy">
                    <source expression="true" />
                    <target variable="cdl:getVariable('barteringDone','','')" />
                </copy>
            </assign>
        </sequence>

        <sequence>
            .....
        </sequence>
    </choice>
</workunit>
```

3.3.5 Complete Example

The rest of the example is all about describing the interactions and choices needed by the seller to check credit and if successful to request delivery. This is listed in schematic form below. We have not filled all of the details because it is illustrated fully in Appendix I.

```
        <interaction name="Seller check credit with CreditChecker"
            operation="creditCheck" channelVariable="Seller2CreditChkC">
            ........
        </interaction>
        <choice>
            <interaction name="Credit Checker fails credit check"
                operation="creditFailed" channelVariable="Seller2CreditChkC">
                .......
            </interaction>
            <sequence>
                <interaction name="Credit Checker passes credit"
                    operation="creditOk" channelVariable="Seller2CreditChkC">
                    .........
                </interaction>
                <interaction name="Seller requests delivery details"
                    operation="requestShipping" channelVariable="Seller2ShipperC">
                    ........
                </interaction>
                <interaction name="Shipper forward channel to shipper"
                    operation="sendChannel" channelVariable="Seller2ShipperC">
                    <description type="description">Pass channel from buyer to shipper</description>
                    <participate relationshipType="SellerShipper"
                        fromRole="SellerRoleType" toRole="ShipperRoleType" />
                    <exchange name="forwardChannel" channelType="2BuyerChannelType" action="request">
                        <send variable="cdl:getVariable('DeliveryDetailsC','','')" />
                        <receive variable="cdl:getVariable('DeliveryDetailsC','','')" />
                    </exchange>
                </interaction>
                <interaction name="Shipper sends delivery details to buyer"
                    operation="deliveryDetails" channelVariable="DeliveryDetailsC">
                    <description type="description">Pass back shipping details to the buyer</description>
                    <participate relationshipType="ShipperBuyer"
                        fromRole="ShipperRoleType" toRole="BuyerRoleType" />
                    <exchange name="sendDeliveryDetails"
                        informationType="DeliveryDetailsType" action="request">
                    </exchange>
                </interaction>
            </sequence>
        </choice>
      </sequence>
    </choreography>
</package>
```

In this outline we can see a choice made after a credit check has been done. If the credit check fails we do very little. If it succeeds we "requestShipping" from seller to shipper and pass the buyer details that we got previously onto the shipper. The shipper then responds back to the buyer using the necessary channel details that were passed ("DeliveryDetailsC") to affect the interaction.

4 Advanced topics

4.1 Dependent Workunits

We can change our example and make it somewhat more interesting by having two workunits. The first is unchanged and the second incorporates all of the previous choreography notation that follows the completion of the first workunit. What we shall do to model this is to make the second workunit dependent on the availability and value of **'barteringDone'**. To do this we shall introduce a guard condition into our workunit and make a blocking workunit. Such dependent workunits represent a structural dependence that might exist in real systems and so provides the choreography designer with an elegant way of expressing the dependencies directly as opposed to adding further conditional and state to achieve much the same thing.

Our workunit sketch looks like the following:

```
<parallel>
    <workunit name="Repeat until bartering has been completed" repeat="barteringDone = false">
        ......
    </workunit>
    <workunit name="Process Order" guard="barteringDone = true" blocking="true">
        ......
    </workunit>
</parallel>
```

In this example the guard condition is "**barteringDone = true**" and the blocking is set to "**true**". This second workunit waits until "**barteringDone**" is available and is set to true before enacting whatever is described inside of it. We place the two workunits inside a parallel construct which means that the two workunits operate concurrently. The second being dependent on the first waits until its preconditions are met before proceeding.

```
<parallel>
    <workunit name="Repeat until bartering has been completed" repeat="barteringDone = false">
        ......
    </workunit>
    <workunit name="Process Order" guard="barteringDone = true" blocking="true">
        ......
    </workunit>
</parallel>
```

We can use the same data-driven collaboration technique to rewrite how we handle the credit checking response, by introducing another variable, **'creditRatingOk'**, at the seller role, that records, as a Boolean, the

response from the credit check. The second, blocking, workunit is made dependent on the outcome of the first by using a guard that looks a little like the following:

```
<parallel>
    <workunit name="Check Credit Rating">
        <sequence>
            <interaction name="Seller check credit with CreditChecker"
                operation="creditCheck" channelVariable="Seller2CreditChkC">
                <description type="description">
                    Check the credit for this buyer with the credit check agency
                </description>
                <participate relationshipType="SellerCreditCheck"
                    fromRole="SellerRoleType" toRole="CreditCheckerRoleType" />
                <exchange name="checkCredit" informationType="CreditCheckType" action="request">
                </exchange>
            </interaction>
            <choice>
                <sequence>
                    <interaction name="Credit Checker fails credit check"
                        operation="creditFailed" channelVariable="Seller2CreditChkC">
                        <description type="description">
                            Credit response from the credit checking agency
                        </description>
                        <participate relationshipType="SellerCreditCheck"
                            fromRole="SellerRoleType" toRole="CreditCheckerRoleType" />
                        <exchange name="creditCheckFails"
                            informationType="CreditRejectType" action="respond">
                        </exchange>
                    </interaction>
                    <assign roleType="SellerRoleType">
                        <copy name="copy">
                            <source expression="false" />
                            <target variable="cdl:getVariable('creditRatingOk','','')" />
                        </copy>
                    </assign>
                </sequence>

                <sequence>
                    <interaction name="Credit Checker passes credit"
                        operation="creditOk" channelVariable="Seller2CreditChkC">
                        <description type="description">
                            Credit response from the credit checking agency
                        </description>
                        <participate relationshipType="SellerCreditCheck" fromRole="BuyerRoleType"
                            toRole="CreditCheckerRoleType" />
                        <exchange name="creditCheckPasses"
                            informationType="CreditAcceptType" action="respond">
                        </exchange>
                    </interaction>
                    <assign roleType="SellerRoleType">
                        <copy name="copy">
                            <source expression="true" />
                            <target variable="cdl:getVariable('creditRatingOk','','')" />
                        </copy>
                    </assign>
                </sequence>
            </choice>
        </sequence>
    </workunit>

    <workunit name="Request Delivery" guard="creditRatingOk = true" blocking="true">
        ......
    </workunit>
</parallel>
```

The operational semantic of the workunit in WS-CDL can be described as follows:

Blocking

Workunit (G) (R) (B is True)
Body

Where
G => guard condition
R => repeat condition
B => blocking attribute
Body => CDL activities within the work unit

A typical order of evaluation is as follows:

(G) Body (R G) Body (R G) Body

With respect to a G then the G is only evaluated when the variables are available and evaluate to True and otherwise we wait at the guard condition. Thus the Body after the first G only gets executed when G is True. Or put another way Body is primed ready for action and then is executed when G evaluates to True.

IF G is unavailable or evaluates to False THEN it equates to:

```
when (G) {
        Body
} until (!R)
```

IF G is always True THEN it equates to:

```
repeat {
        Body
} until (!R)
```

IF R is always False THEN it equates to:

```
when (G) {
        Body
}
```

Non-blocking

Workunit (G) (R) (B is False)
Body

A typical order of evaluation is as follows:

(G) Body (R G) Body (R G) Body

Which equates to (in pseudo code):

```
while (G) {
        Body
} until (!R)
```

IF G is always True THEN it equates to:

```
repeat {
        Body
} until (!R)
```

IF R is always False THEN it equates to:

```
if (G) {
        Body
}
```

4.2 Advanced Channels

4.2.1 Usage

4.2.2 Channel Passing

4.3 Business exceptions

4.3.1 Exceptions

4.3.2 Exceptions as messages

4.4 Compensations

4.4.1 Finalizers and Finalization

4.5 Modularization

4.5.1 Choreographies and sub-choreographies

4.5.2 Performing a choroeography

4.6 Parallel and concurrent

4.6.1 Managing join conditions

4.7 Silent Actions and Conditions

4.8 NoActions

4.9 Time

4.10 Isolation levels

5 An EAI example

Up to now we have only used cross-domain examples. That is a situation in which buyer, seller, credit check agency and shipper can be considered to be different entities. What we shall do now is describe a simple business scenario which applied inside the firewall of an organization in order to show how WS-CDL can be applied in this context.

6 Implementation considerations

6.1 End point projection

6.1.1 Java

6.1.2 WS-BPEL

6.1.3 Runtime Monitoring

6.2 WSDL

6.2.1 WSDL 1.1

6.2.2 WSDL 1.2

6.3 WS-Addressing

6.3.1 Channel representation

Appendix 1 – Simple WS-CDL encoding of the example.

```
<?xml version="1.0" encoding="UTF-8" ?>
<package name="BuyerSellerCDL" author="Steve Ross-Talbot"
         version="1.0"
         targetNamespace=www.pi4tech.com/cdl/BuyerSeller
         xmlns="http://www.w3.org/2004/12/ws-chor/cdl"
         xmlns:bs="http://www.pi4tech.com/cdl/BuyerSellerExample-1">
    <description type="description">This is the basic BuyerSeller Choreography Description</description>

    <informationType name="BooleanType" type="xs:boolean" />
    <informationType name="StringType" type="xsd:string" />
    <informationType name="RequestForQuoteType" type="bs:RequestForQuote">
        <description type="documentation">Request for quote message</description>
    </informationType>

    <informationType name="QuoteType" type="bs:Quote">
        <description type="documentation">Quote message</description>
    </informationType>

    <informationType name="QuoteUpdateType" type="bs:QuoteUpdate">
        <description type="documentation">Quote Update Message</description>
    </informationType>

    <informationType name="QuoteAcceptType" type="bs:QuoteAccept">
        <description type="documentation">Quote Accept Message</description>
    </informationType>

    <informationType name="CreditCheckType" type="bs:CreditCheckRequest">
        <description type="documentation">Credit Check Message</description>
    </informationType>

    <informationType name="CreditAcceptType" type="bs:CreditAccept">
        <description type="documentation">Credit Accept Message</description>
    </informationType>

    <informationType name="CreditRejectType" type="bs:CreditReject">
        <description type="documentation">Credit Reject Message</description>
    </informationType>

    <informationType name="RequestDeliveryType" type="bs:RequestForDelivery">
        <description type="documentation">Request Delivery Message</description>
    </informationType>

    <informationType name="DeliveryDetailsType" type="bs:DeliveryDetails">
        <description type="documentation">Delivery Details Message</description>
    </informationType>

    <token name="BuyerRef" informationType="StringType" />
    <token name="SellerRef" informationType="StringType" />
    <token name="CreditCheckRef" informationType="StringType" />
    <token name="ShipperRef" informationType="StringType" />
```

```
<roleType name="BuyerRoleType">
    <description type="description">The Behavior embodied by a buyer</description>
    <behavior name="BuyerBehavior" />
</roleType>
<roleType name="SellerRoleType">
    <description type="description">The behavior embodied by a seller</description>
    <behavior name="SellerBehavior" />
</roleType>
<roleType name="CreditCheckerRoleType">
    <description type="description">The behavior embodied by a credit checker service</description>
    <behavior name="CreditCheckerBehavior" />
</roleType>
<roleType name="ShipperRoleType">
    <description type="description">The behavior embodied by a shipper service</description>
    <behavior name="ShipperBehavior" />
</roleType>

<relationshipType name="BuyerSeller">
    <role type="BuyerRoleType" />
    <role type="SellerRoleType" />
</relationshipType>
<relationshipType name="SellerCreditCheck">
    <role type="SellerRoleType" />
    <role type="CreditCheckerRoleType" />
</relationshipType>
<relationshipType name="SellerShipper">
    <role type="SellerRoleType" />
    <role type="ShipperRoleType" />
</relationshipType>
<relationshipType name="ShipperBuyer">
    <role type="ShipperRoleType" />
    <role type="BuyerRoleType" />
</relationshipType>

<channelType name="Buyer2SellerChannelType">
    <passing channel="2BuyerChannelType" new="true">
        <description type="description">Pass channel to enable shipper to talk to buyer</description>
    </passing>
    <role type="SellerRoleType" />
    <reference>
        <token name="SellerRef" />
    </reference>
</channelType>
<channelType name="Seller2CreditCheckChannelType">
    <role type="CreditCheckerRoleType" />
    <reference>
        <token name="CreditCheckRef" />
    </reference>
</channelType>
<channelType name="2BuyerChannelType" action="request">
    <role type="BuyerRoleType" />
    <reference>
        <token name="BuyerRef" />
    </reference>
</channelType>
<channelType name="Seller2ShipperChannelType">
    <passing channel="2BuyerChannelType">
        <description type="description">Pass channel through to shipper</description>
    </passing>
    <role type="ShipperRoleType" />
    <reference>
        <token name="ShipperRef" />
    </reference>
</channelType>
```

```
<choreography name="Main" root="true">
    <description type="description">Collaboration between buyer, seller, shipper, credit chk</description>

    <relationship type="BuyerSeller" />
    <relationship type="SellerCreditCheck" />
    <relationship type="SellerShipper" />
    <relationship type="ShipperBuyer" />

    <variableDefinitions>
        <variable name="Buyer2SellerC"
            channelType="Buyer2SellerChannelType"
            roleTypes="BuyerRoleType">
            <description type="description">
                Principle channel used to enable interaction between buyer
                and seller for price requests, price confirms and orders
            </description>
        </variable>
        <variable name="Seller2ShipperC"
            channelType="Seller2ShipperChannelType"
            roleTypes="SellerRoleType">
            <description type="description">
                Seller to shipper channel - used to pass a channel to effect
                interaction with the buyer
            </description>
        </variable>
        <variable name="Seller2CreditChkC"
            channelType="Seller2CreditCheckChannelType"
            roleTypes="SellerRoleType">
            <description type="description">
                Seller to Credit Check Channel used to check credit for buyers to
                determine if we do business with them
            </description>
        </variable>
        <variable name="DeliveryDetailsC"
            channelType="2BuyerChannelType"
            roleTypes="BuyerRoleType SellerRoleType ShipperRoleType" />
            <description type="description">
                Channel created by the buyer to pass to third parties so that
                They can communicate with the buyer without have linkage
            </description>
        </variable>
        <variable name="barteringDone"
            informationType="BooleanType"
            roleTypes="BuyerRoleType SellerRoleType">
            <description type="description">Has Bartering Finished flag</description>
        </variable>
    </variableDefinitions>
```

```
<sequence>
  <interaction name="Buyer requests a Quote - this is the initiator"
      operation="requestForQuote" channelVariable="Buyer2SellerC" initiate="true">
      <description type="description">Request for Quote</description>

      <participate relationshipType="BuyerSeller" fromRole="BuyerRoleType" toRole="SellerRoleType" />
      <exchange name="request" informationType="RequestForQuoteType" action="request">
          <description type="description">Requesting Quote</description>
      </exchange>
      <exchange name="response" informationType="QuoteType" action="respond">
          <description type="description">Quote returned</description>
      </exchange>
  </interaction>

  <workunit name="Repeat until bartering has been completed" repeat="barteringDone = false">
      <choice>
          <silentAction roleType="BuyerRoleType">
              <description type="description">Do nothing - let the quote timeout</description>
          </silentAction>

          <sequence>
              <interaction name="Buyer accepts the quote and engages in the act of buying"
                  operation="quoteAccept" channelVariable="Buyer2SellerC">
                  <description type="description">Quote Accept</description>

                  <participate relationshipType="BuyerSeller"
                      fromRole="BuyerRoleType" toRole="SellerRoleType" />
                  <exchange name="Accept Quote" informationType="QuoteAcceptType"
                       action="request">
                  </exchange>
              </interaction>
              <interaction name="Buyer send channel to seller to enable callback behavior"
                  operation="sendChannel" channelVariable="Buyer2SellerC">
                  <description type="description">Buyer sends channel to pass to shipper</description>
                  <participate relationshipType="BuyerSeller"
                      fromRole="BuyerRoleType" toRole="SellerRoleType" />
                  <exchange name="sendChannel" channelType="2BuyerChannelType" action="request">
                      <send variable="cdl:getVariable('DeliveryDetailsC','','')" />
                      <receive variable="cdl:getVariable('DeliveryDetailsC','','')" />
                  </exchange>
              </interaction>

              <assign roleType="BuyerRoleType">
                  <copy name="copy">
                      <source expression="true" />
                      <target variable="cdl:getVariable('barteringDone','','')" />
                  </copy>
              </assign>
          </sequence>
          <sequence>
              <interaction name="Buyer updates the Quote - in effect requesting a new price"
                  operation="quoteUpdate" channelVariable="Buyer2SellerC">
                  <description type="documentation">Quot Update</description>
                  <participate relationshipType="BuyerSeller"
                      fromRole="BuyerRoleType" toRole="SellerRoleType" />
                  <exchange name="updateQuote"
                      informationType="QuoteUpdateType" action="request">
                  </exchange>
                  <exchange name="acceptUpdatedQuote"
                      informationType="QuoteAcceptType" action="respond">
                      <description type="documentation">Accept Updated Quote</description>
                  </exchange>
              </interaction>
          </sequence>
      </choice>
  </workunit>
```

```
        <interaction name="Seller check credit with CreditChecker"
            operation="creditCheck" channelVariable="Seller2CreditChkC">
            <description type="description">
                Check the credit for this buyer with the credit check agency
            </description>
            <participate relationshipType="SellerCreditCheck"
                fromRole="SellerRoleType" toRole="CreditCheckerRoleType" />
            <exchange name="checkCredit" informationType="CreditCheckType" action="request">
            </exchange>
        </interaction>
        <choice>
            <interaction name="Credit Checker fails credit check"
                operation="creditFailed" channelVariable="Seller2CreditChkC">
                <description type="description">
                    Credit response from the credit checking agency
                </description>
                <participate relationshipType="SellerCreditCheck"
                    fromRole="SellerRoleType" toRole="CreditCheckerRoleType" />
                <exchange name="creditCheckFails" informationType="CreditRejectType" action="respond">
                </exchange>
            </interaction>
            <sequence>
                <interaction name="Credit Checker passes credit"
                    operation="creditOk" channelVariable="Seller2CreditChkC">
                    <description type="description">
                        Credit response from the credit checking agency
                    </description>
                    <participate relationshipType="SellerCreditCheck" fromRole="BuyerRoleType"
                        toRole="CreditCheckerRoleType" />
                    <exchange name="creditCheckPasses"
                        informationType="CreditAcceptType" action="respond">
                    </exchange>
                </interaction>
                <interaction name="Seller requests delivery details"
                    operation="requestShipping" channelVariable="Seller2ShipperC">
                    <description type="description">Request delivery from the shipper</description>
                    <participate relationshipType="SellerShipper"
                        fromRole="SellerRoleType" toRole="ShipperRoleType" />
                    <exchange name="sellerRequestsDelivery"
                        informationType="RequestDeliveryType" action="request">
                    </exchange>

                    <exchange name="sellerReturnsDelivery"
                        informationType="DeliveryDetailsType" action="respond">
                    </exchange>
                </interaction>
                <interaction name="Shipper forward channel to shipper"
                    operation="sendChannel" channelVariable="Seller2ShipperC">
                    <description type="description">Pass channel from buyer to shipper</description>
                    <participate relationshipType="SellerShipper"
                        fromRole="SellerRoleType" toRole="ShipperRoleType" />
                    <exchange name="forwardChannel" channelType="2BuyerChannelType" action="request">
                        <send variable="cdl:getVariable('DeliveryDetailsC','','')" />
                        <receive variable="cdl:getVariable('DeliveryDetailsC','','')" />
                    </exchange>
                </interaction>
                <interaction name="Shipper sends delivery details to buyer"
                    operation="deliveryDetails" channelVariable="DeliveryDetailsC">
                    <description type="description">Pass back shipping details to the buyer</description>
                    <participate relationshipType="ShipperBuyer"
                        fromRole="ShipperRoleType" toRole="BuyerRoleType" />
                    <exchange name="sendDeliveryDetails"
                        informationType="DeliveryDetailsType" action="request">
                    </exchange>
                </interaction>
            </sequence>
        </choice>
    </sequence>
</choreography>
</package>
```