Web Services Choreography Description Language, Version 1.0

Editor's Draft, 24 July 2004

This version:
TBD
Latest version:
TBD
Previous Version:
Not Applicable

Editors (alphabetically):
Nickolaos Kavantzas, Oracle, <nickolas.kavantzas@oracle.com>
David Burdett, Commerce One <david.burdett@commerceone.com>
Gregory Ritzinger, Novell <gritzinger@novell.com>

Abstract

The Web Services Choreography Description Language (WS-CDL) is an XML-based language that describes peer-to-peer collaborations of parties by defining, from a global viewpoint, their common and complementary observable behavior; where ordered message exchanges result in accomplishing a common business goal.

The Web Services specifications offer a communication bridge between the heterogeneous computational environments used to develop and host applications. The future of E-Business applications requires the ability to perform long-lived, peer-to-peer collaborations between the participating services, within or across the trusted domains of an organization.

The Web Services Choreography specification is targeted for composing interoperable, peer-to-peer collaborations between any type of party regardless of the supporting platform or programming model used by the implementation of the hosting environment.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This is the First Public Working Draft of the Web Services Choreography Description Language document.

It has been produced by the Web Services Choreography Working Group, which is part of the Web Services Activity. Although the Working Group agreed to request publication of this document, this document does not represent consensus within the Working Group about Web Services Choreography description language.

This document is a chartered deliverable of the Web Services Choreography Working Group. It is an early stage document and major changes are expected in the near future.

Comments on this document should be sent to public-ws-chor-comments@w3.org (public archive). It is inappropriate to send discussion emails to this address.

Discussion of this document takes place on the public public-ws-chor@w3.org mailing list (public archive) per the email communication rules in the Web Services Choreography Working Group charter.

This document has been produced under the 24 January 2002 CPP as amended by the W3C Patent Policy Transition Procedure. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) with respect to this specification should disclose the information in accordance with section 6 of the W3C Patent Policy. Patent disclosures relevant to this specification may be found on the Working Group's patent disclosure page.

Publication as a Working Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

Revision Description

This is the second editor's draft of the document.

Table of Contents

1 Introduction

For many years, organizations have being developing solutions for automating peer-to-peer collaborations, within or across their trusted domain, in an effort to improve productivity and reduce operating costs.

The past few years have seen the Extensible Markup Language (XML) and the Web Services framework developing as the de-facto choices for describing interoperable data and platform neutral business interfaces, enabling more open business transactions to be developed.

Web Services are a key component of the emerging, loosely coupled, Web-based computing architecture. A Web Service is an autonomous, standards-based component whose public interfaces are defined and described using XML. Other systems may interaction with the Web Service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.

The Web Services specifications offer a communication bridge between the heterogeneous computational environments used to develop and host applications. The future of E-Business applications requires the ability to perform long-lived, peer-to-peer collaborations between the participating services, within or across the trusted domains of an organization.

The Web Service architecture stack targeted for integrating interacting applications consists of the following components:

- *SOAP*: defines the basic formatting of a message and the basic delivery options independent of programming language, operating system, or platform. A SOAP compliant Web Service knows how to send and receive SOAP-based messages

- *WSDL*: describes the static interface of a Web Service. It defines the protocol and the message characteristics of end points. Data types are defined by XML Schema specification, which supports rich type definitions and allows expressing any kind of XML type requirement for the application data

- *UDDI*: allows publishing the availability of a Web Service and its discovery from service requesters using sophisticated searching mechanims
- *Security layer*: ensures that exchanged information are not modified or forged
- *Reliable Messaging layer*: provides exactly-once and guaranteed delivery of information exchanged between parties
- *Context, Coordination and Transaction layer*: defines interoperable mechanisms for propagating context of long-lived business transactions and enables parties to meet correctness requirements by following a global agreement protocol
- *Business Process Languages layer*: describes the execution logic of Web Services based applications by defining their control flows (such as conditional, sequential, parallel and exceptional execution) and prescribing the rules for consistently managing their non-observable data
- *Choreography layer*: describes peer-to-peer collaborations of parties by defining from a global viewpoint their common and complementary observable behavior, where information exchanges occur, when the jointly agreed ordering rules are satisfied

The Web Services Choreography specification is targeted for composing interoperable, peer-to-peer collaborations between any type of party regardless of the supporting platform or programming model used by the implementation of the hosting environment.

1.1 Notational Conventions

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [2].

The following namespace prefixes are used throughout this document:

Prefix	Namespace URI	Definition
wsdl	http://schemas.xmlsoap.org/wsdl/	WSDL namespace for WSDL framework.
cdl	http://www.w3.org/ws/choreography/2004/02/WSCDL	WSCDL namespace for Choreography language.

xsi	http://www.w3.org/2000/10/XMLSchema-instance	Instance namespace as defined by XSD [10].
xsd	http://www.w3.org/2000/10/XMLSchema	Schema namespace as defined by XSD [10].
tns	(various)	The "this namespace" (tns) prefix is used as a convention to refer to the current document.
(other)	(various)	All other namespace prefixes are samples only. In particular, URIs starting with "http://sample.com" represent some application-dependent or context-dependent URI [4].

This specification uses an *informal syntax* to describe the XML grammar of a WS-CDL document:

- The syntax appears as an XML instance, but the values indicate the data types instead of values.
- Characters are appended to elements and attributes as follows: "?" (0 or 1), "*" (0 or more), "+" (1 or more).
- Elements names ending in "…" (such as <element…/> or <element…>) indicate that elements/attributes irrelevant to the context are being omitted.
- Grammar in bold has not been introduced earlier in the document, or is of particular interest in an example.
- <-- extensibility element --> is a placeholder for elements from some "other" namespace (like ##other in XSD).
- The XML namespace prefixes (defined above) are used to indicate the namespace of the element being defined.

- Examples starting with <?xml contain enough information to conform to this specification; others examples are fragments and require additional information to be specified in order to conform.

XSD schemas are provided as a formal definition of WS-CDL grammar (see Section 9).

1.2 Purpose of the Choreography Language

Business or other activities that involve multiple different organizations or independent processes that collaborate using the Web Services technology can be successful only if they are properly integrated.

To solve this problem, a "global" definition of the common ordering conditions and constraints under which messages are exchanged is produced that describes from a global viewpoint the common and complementary observable behavior of all the partiesinvolved. Each party can then use the global definition to build and test solutions that conform to it.

The main advantage of a global definition approach is that it separates the process being followed by an individual business or system within a "domain of control" from the definition of the sequence in which each business or system exchanges information with others. This means that, as long as the "observable" sequence does not change, the rules and logic followed within the domain of control can change at will.

In real-world scenarios, corporate entities are often unwilling to delegate control of their business processes to their integration partners. Choreography offers a means by which the rules of participation within a collaboration can be clearly defined and agreed to, jointly. Each entity may then implement its portion of the Choreography as determined by the common view.

The figure below demonstrates a possible usage of the Choreography Language.

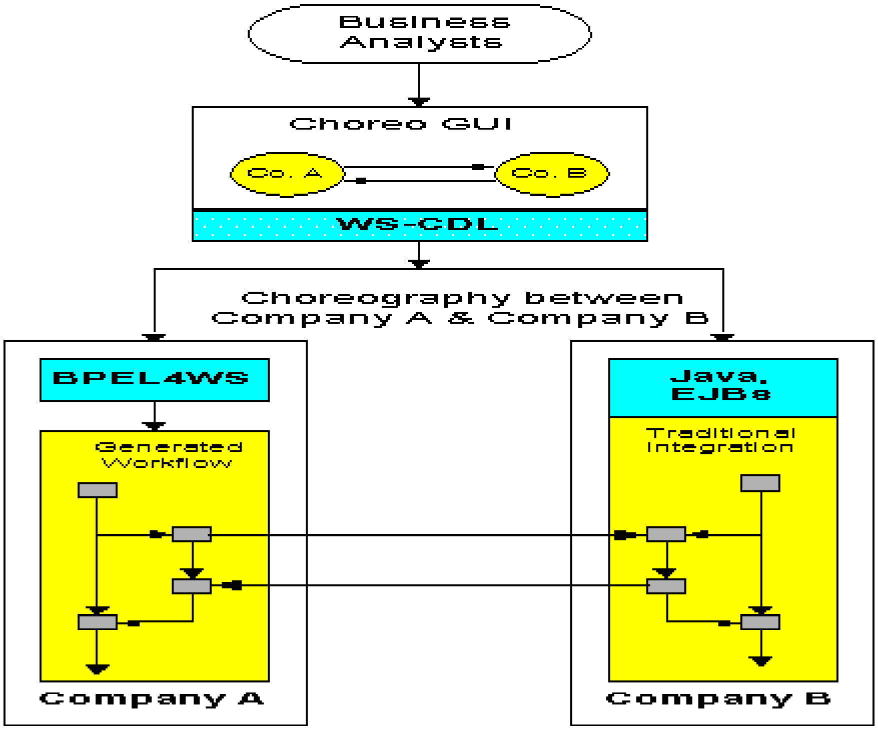

Figure 1: Integrating Web Services based applications using WS-CDL

In Figure 1, Company A and Company B wish to integrate their Web Services based applications. The respective business analysts at both companies agree upon the services involved in the collaboration, their interactions and their common ordering and constraint rules under which the interactions occur and then generate a Choreography Language based representation.

In the case of Company A, relies on a BPEL4WS [18] solution. Company B, having greater legacy driven integration needs, relies on a J2EE [25] solution incorporating Java and Enterprise Java Bean Components or a .NET [26] solution incorporating C#.

In this example, a Choreography specifies the interoperability and interactions between services across business entities, while leaving actual implementation decisions in the hands of each individual company. Similarly, a Choreography can specify the interoperability and interactions required to ensure compatability between services within one business entity.

1.3 Goals

The primary goal of a Choreography Language is to specify a declarative, XML based language that defines from a global viewpoint the common and complementary observable behavior, where message exchanges occur, and when the jointly agreed ordering rules are satisfied.

Some additional goals of this definition language are to permit:

- *Reusability*. The same choreography definition is usable by different parties operating in different contexts (industry, locale, etc.) with different software (e.g. application software)
- *Cooperation*. Choreographies define the sequence of exchanging messages between two (or more) independent parties or processes by describing how they should cooperate
- *Multi-Party Collaboration*. Choreographies can be defined involving any number of parties or processes
- *Semantics*. Choreographies can include human-readable documentation and semantics for all the components in the choreography
- *Composability*. Existing Choreographies can be combined to form new Choreographies that may be reused in different contexts
- *Modularity.* Choreographies can be defined using an "import" facility that allows a choreography to be created from parts contained in several different Choreographies
- *Information Driven Collaboration*. Choreographies describe how parties maintain where they are in the choreography, by recording their exchanged information and the observable state changes caused by these exchanges of information, and also their reactions to them
- *Information Alignment*. Choreographies allow the parties that take part in Choreographies to communicate and synchronize their observable state changes and the actual values of the exchanged information as well
- *Exception Handling*. Choreographies can define how exceptional or unusual conditions that occur while the choreography is performed are handled
- *Transactionality*. The processes or parties that take part in a choreography can work in a "transactional" way with the ability to coordinate the outcome of the long-lived collaborations, which include multiple, often recursive collaboration units, each with its own business rules and goals
- *Compatibility with other Specifications*. This specification will work alongside and complement other specifications such as the WS-Reliability [22], WS-Composite Application Framework (WS-CAF) [21], WS-Security [24], Business Process Execution Language for WS (BPEL4WS) [18], etc.

1.4 Relationship with XML and WSDL

This specification depends on the following specifications: XML 1.0 [9], XML-Namespaces [10], XML-Schema 1.0 [11, 12] and XPath 1.0 [13]. In addition, support for importing and referencing service definitions given in WSDL 2.0 [7] is a normative part of this specification.

1.5 Relationship with Business Process Languages

A Choreography Language is not an "executable business process description language" [16, 17, 18, 19, 20] or an implementation language [23]. The role of specifying the execution logic of an application will be covered by these specifications.

A Choreography Language does not depend on a specific business process implementation language. Thus, it can be used to specify truly interoperable, peer-to-peer collaborations between any type of party regardless of the supporting platform or programming model used by the implementation of the hosting environment. Each party could be implemented by completely different languages such as:

- Applications, whose implementation is based on executable business process languages [16, 17, 18, 19, 20]
- Applications, whose implementation is based on general purpose programming languages [23, 26]
- Or human controlled software agents

2 Choreography Model

This section introduces the Web Services Choreography Description Language (WS-CDL) model.

2.1 Model Overview

WS-CDL describes interoperable, peer-to-peer collaborations between parties. In order to facilitate these collaborations, services commit on mutual responsibilities by establishing Relationships. Their collaboration takes place in a jointly agreed set of ordering and constraint rules, whereby messages are exchanged between the parties.

The Choreography model consists of the following notations:

- *Participants, Roles and Relationships* - In a Choreography, information is always exchanged between Participants within the same or across trust boundaries
- *Types, Variables and Tokens* - Variables contain information about commonly observable objects in a collaboration, such as the messages exchanged or the state of the Roles involved. Tokens are aliases that can be used to reference parts of a Variable. Both Variables and Tokens have Types that define the structure of what the Variable or Token contains
- *Choreographies* - A Choreography allows defining collaborations between interacting peer-to-peer processes:

    - *Choreography Composition* allows the creation of new Choreographies by reusing existing Choreography definitions
    - *Choreography Life-line* expresses the progression of a collaboration. Initially, the collaboration is started at a specific business process, then work is performed by following the choreography and finally the choreography completes, either normally or abnormally
    - *Choreography Recovery* consists of:
        - *Choreography Exception Block* - describes how to specify what additional interactions should occur when a Choreography behaves in an abnormal way
        - *Choreography Finalizer Block* - describes how to specify what additional interactions should occur to reverse the effect of an earlier successfully completed choreography
- *Channels* - A Channel realizes a point of collaboration between parties by specifying where and how information is exchanged
- *WorkUnits* - A WorkUnit prescribes constraints that must be fulfilled for making progress within a Choreography
- *Interactions* - An Interaction is the basic building block of a Choreography, which results in an exchange of messages between parties and possible synchronization of their states and the actual values of the exchanged information
- *Activities and Ordering Structures* - Activities are the lowest level components of the Choreography that perform the actual work. Ordering Structures combine activities with other Ordering Structures in a nested structure to express the ordering conditions in which the messages in the choreography are exchanged
- *Semantics* - Semantics allow the creation of descriptions that can record the semantic definitions of every single component in the model

2.2 Choreography Document Structure

A WS-CDL document is simply a set of definitions. Each definition is a named construct that can be referenced. There is a *package* element at the root, and the individual Choreography definitions inside.

2.2.1 Package

The WS-CDL Package aggregates a set of Choreography definitions, provides a namespace for the definitions and through import statements, includes parts of choreography definitions defined in other Packages.

The syntax of the *package* construct is:

```
<package
   name="ncname"
   author="xsd:string"?
   version="xsd:string"
   targetNamespace="uri"
   xmlns="http://www.w3.org/ws/choreography/2004/02/WSCDL/"
   importDefinitions*
   informationType*
   token*
   tokenLocator*
   role*
   relationship*
   participant*
   channelType*
   Choreography-Notation*
</package>
```

The package element contains:

- Zero or more Import definitions
- Zero or more Information Types
- Zero or more Token types and Token Locators
- Zero or more Role types
- Zero or more Relationship types
- Zero or more Participants
- Zero or more Channel types
- Zero or more, package-level Choreographies

The top-level attributes author, and version, define authoring properties of the Choreography document.

The targetNamespace attribute provides the namespace associated with all definitions contained in this package. Choreography definitions imported to this package may be associated with other namespaces.

The elements informationType, token, tokenLocator, role, relationship, participant and channelType are shared by all the Choreographies defined within this package.

The *importDefinitions* construct allows reusing Choreography types defined in another Choreography package such as Token types, Token Locator types, Information Types, Role types, Relationship types, Channel types and Choreographies.

2.2.2 Choreography document Naming and Linking

WS-CDL documents MUST be assigned a name attribute of type NCNAME that serves as a lightweight form of documentation.

The targetNamespace attribute of type URI MUST be specified.

The URI MUST NOT be a relative URI.

A reference to a definition is made using a QName.

Each definition type has its own name scope.

Names within a name scope MUST be unique within a WS-CDL document.

The resolution of QNames in WS-CDL is similar to the resolution of QNames described by the XML Schemas specification [11].

2.2.3 Language Extensibility and Binding

To support extending the WS-CDL language, this specification allows the use of extensibility elements and/or attributes defined in other XML namespaces. Extensibility elements and/or attributes MUST use an XML namespace different from that of WS-CDL. All extension namespaces used in a WS-CDL document MUST be declared.

Extensions MUST NOT change the semantics of any element or attribute from the WS-CDL namespace.

Within a WS-CDL document, the optional attribute id provides a distinct name that can be used to uniquely reference a language construct. This attribute MAY be defined inside any WS-CDL language element.

2.2.4 Semantics

Within a WS-CDL document, descriptions will be required to allow the recording of semantics definitions. The optional *description* sub-element is used as a textual description for documentation purposes. This element is allowed inside any WS-CDL language element.

The information provided by the description element will allow for the recording of semantics in any or all of the following ways:

- *Text.* This will be in plain text or possibly HTML and should be brief
- *Document Reference*. This will contain a URL to a document that more fully describes the component. For example on the top level Choreography Definition that might reference a complete paper
- *Structured Attributes.* This will contain machine processable definitions in languages such as RDF or OWL

Descriptions that are *Text* or *Document References* can be defined in multiple different human readable languages.

2.3 Collaborating Parties

The WSDL specification describes the functionality of a service provided by a party based on a stateless, connected, client-server model. The emerging Web Based applications require the ability to exchange messages in a peer-to-peer environment. In these types of environments a party represents a requester of services provided by another party and is at the same time a provider of services requested from other parties, thus creating mutual multi-party service dependencies.

A WS-CDL document describes how a party is capable of engaging in peer-to-peer collaborations with the same party or with different parties.

Within a Choreography, information is always exchanged between *Participants*.

The *Roles*, *Relationship* and *Channels* define the coupling of the collaborating parties.

2.3.1 Roles

A *Role* enumerates the observable behavior a party exhibits in order to collaborate with other parties. For example the Buyer Role is associated with purchasing of goods or services and the Supplier Role is associated with providing those goods or services for a fee.

The syntax of the *role* construct is:

```
<role name="ncname" >
  <behavior name="ncname"
            interface="qname"? />+
</role>
```

Within the role element, the behavior element specifies a subset of the observable behavior a party exhibits. A Role MUST contain one or more behavior elements.

The behavior element defines an optional interface attribute, which identifies a WSDL interface type. A behavior without an interface describes a Role that is not required to support a specific Web Service interface.

2.3.2 Participants

A *Participant* identifies a set of Roles that MUST be implemented by the same entity or organization. Its purpose is to group together the parts of the observable behavior that MUST be implemented by the same process. For example the Seller Role in a Buyer-Seller Relationship MUST be implemented by the same Participant that is the Seller in a Seller-Shipper Relationship.

The syntax of the *participant* construct is:

```
<participant name="ncname">
  <role type="qname" />+
</participant>
```

2.3.3 Relationships

A *Relationship* identifies the Role/Behavior Types where mutual commitments between two parties MUST be made for them to collaborate successfully. For example the Relationships between a Buyer and a Seller could include:

- A "Purchasing" Relationship, for the initial procurement of goods or services, and
- A "Customer Management" Relationship to allow the Supplier to provide service and support after the goods have been purchased or the service provided

Although Relationships are always between two Roles, Choreographies involving more than two Roles are possible. For example if the purchase of goods involved a third-party Shipper contracted by the Supplier to deliver the Supplier’s goods, then, in addition to the Purchasing and Customer Management Relationships described above, the following Relationships might exist:

- A "Logistics Provider" Relationship between the Supplier and the Shipper, and
- A "Goods Delivery" Relationship between the Buyer and the Shipper

The syntax of the *relationship* construct is:

```
<relationship name="ncname">
  <role type="qname" behavior="ncname" />
  <role type="qname" behavior="ncname" />
</relationship>
```

A relationship MUST have exactly two role types defined.

Within the role element, the behavior attribute points to a behavior type within the role type specified by the type attribute of the role element.

2.3.4 Channels

A *Channel* realizes a point of collaboration between parties by specifying where and how information is exchanged. Additionally, Channel information can be passed among parties. This allows the modeling of both static and dynamic message destinations when collaborating within a Choreography. For example, a Buyer could specify Channel information to be used for sending delivery information. The Buyer could then send the Channel information to the Seller who then forwards it to the Shipper. The Shipper could then send delivery information directly to the Buyer using the Channel information originally supplied by the Buyer.

A Channel MUST describe the Role and the reference type of a party, being the target of an Interaction, which is then used for determining where and how to send/receive information to/into the party.

A Channel MAY specify the instance identity of a process implementing the behavior of a party, being the target of an Interaction.

A Channel MAY describe one or more logical conversations between parties, where each conversation groups a set of related message exchanges.

One or more Channel(s) MAY be passed around from one Role to another. A Channel MAY restrict the types of Channel(s) allowed to be exchanged between the parties, through this Channel. Additionally, a Channel MAY restrict its usage by specifying the number of times a Channel can be used.

The syntax of the *channelType* construct is:

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

  <passing  channel="qname"
        action="request-respond"|"request"|"respond"?
        new="xsd:boolean"? />*

  <role  type="qname"  behavior="ncname"? />

  <reference>
    <token type="qname"/>+
  </reference>
  <identity>
    <token  type="qname"/>+
  </identity>*
</channelType>
```

The optional attribute usage is used to restrict the number of times a Channel can be used.

The optional element passing describes the Channel(s) that are exchanged from one Role to another Role, when using this Channel in an Interaction. In the case where the operation used to exchange the Channel is of request-response type, then the attribute action within the passing element defines if the Channel will be exchanged during the request or during the response. The Channels exchanged can be used in subsequent Interaction activities. If the element passing is missing then this Channel can be used for exchanging business documents and all types of Channels without any restrictions.

The element role is used to identify the Role of a party, being the target of an Interaction, which is then used for statically determining where and how to send/receive information to/into the party.

The element reference is used for describing the reference type of a party, being the target of an Interaction, which is then used for dynamically determining where and how to send/receive information to/into the party. The service reference of a party is distinguished by a set of Token types as specified by the token element within the reference element.

The optional element identity MAY be used for identifying an instance of a process implementing the behavior of a party and for identifying a logical conversation between parties. The process identity and the different conversations are distinguished by a set of Token types as specified by the token element within the identity element.

The example below shows the definition of the Channel type RetailerChannel. The Channel identifies the Role type the tns:Retailer. The address of the Channel is specified in the reference element, whereas the process instance can be identified using the identity element for correlation purposes. The passing element allows an instance of a ConsumerChannel to be sent over the RetailerChannel.

```
<channelType name="RetailerChannel">
  <passing channel="ConsumerChannel" action="request" />
  <role type="tns:Retailer" behavior="retailerForConsumer"/>
  <reference>
    <token type="tns:retailerRef"/>
  </reference>
  <identity>
    <token type="tns:purchaseOrderID"/>
  </identity>
</channelType>
```

2.4 Information Driven Collaborations

A WS-CDL document allows defining information within a Choreography that can influence the observable behavior of the collaborating parties.

Variables contain information about objects in the Choreography such as the messages exchanged or the state of the Roles involved. *Tokens* are aliases that can be used to reference parts of a *Variable*. Both *Variables* and *Tokens* have *Information Types* that define the data structure of what the *Variable* or *Token* contains.

2.4.1 Information Types

Information types describe the type of information used within a Choreography. By introducing this abstraction, a Choreography definition avoids referencing directly the data types, as defined within a WSDL document or an XML Schema document.

The syntax of the *informationType* construct is:

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

The attributes type, and element describe the document to be an XML Schema type, or an XML Schema element respectively. The document is of one of these types exclusively.

2.4.2 Variables

Variables capture information about objects in a Choreography as defined by the *variable usage*:

- *Information Exchange Variables* that contain information such as an Order that is used to:
    - Populate the content of a message to be sent, or
    - Populated as a result of a message received
- *State Variables* that contain observable information about the State of a Role as a result of information exchanged. For example:
    - When a Buyer sends an order to a Seller, the Buyer could have a *State Variable* called "OrderState" set to a value of "OrderSent" and once the message was received by the Seller, the Seller could have an *State Variable* called "OrderState" set to a value of "OrderReceived". Note that the variable "OrderState" at the Buyer is a different variable to the "OrderState" at the Seller
    - Once an order is received, then it might be validated and checked for acceptability in other ways that affect how the Choreography is performed. This could require additional states to be defined for "Order State", such as: "OrderError", which means an error was detected that stops processing of the message, "OrderAccepted", which means that there were no problems with the Order and it can be processed, and "OrderRejected", which means, although there were no errors, it cannot be processed, e.g. because a credit check failed
- *Channel Variables*. For example, a Channel Variable could contain information such as the URL to which the message could be sent, the policies that are to be applied, such as security, whether or not reliable messaging is to be used, etc.

The value of Variables:

- Is available to all the Roles by initializing them prior to the start of a Choreography
- Common Variables that contain information that is common knowledge to two or more Roles, e.g. "OrderResponseTime" which is the time in hours in which a response to an Order must be sent

- Can be made available at a Role by populating them as a result of an Interaction
- Can be made available at a Role by assigning data from other information
    - Locally Defined Variables that contain information created and changed locally by a Role. They can be Information Exchange, State or Channel Variables as well as variables of other types. For example "Maximum Order Amount" could be data created by a seller that is used together with an actual order amount from an Order received to control the ordering of the Choreography. In this case how Maximum Order Amount is calculated and its value would not be known by the other Roles
- Can be used to determine the decisions and actions to be taken within a Choreography

The *variableDefinitions* construct is used for defining one or more variables within a Choreography block.

The syntax of the *variableDefinitions* construct is:

```
<variableDefinitions>
  <variable   name="ncname"
      informationType="qname"|channelType="qname"
      mutable="true|false"?
      free="true|false"?
      silent-action="true|false"?
      role="qname"? />+
</variableDefinitions>
```

The defined variables can be of the following types:

- Information Exchange Variables, State Variables. The attribute informationType describes the type of the variable
- Channel Variables. The attribute channelType describes the type of the Channel

The optional attribute mutable, when set to "false" describes that the variable information when initialized, cannot change anymore.

The optional attribute free, when set to "true" describes that a variable defined in an enclosing Choreography is also used in this Choreography, thus sharing the variable information. When the attribute free is set to "true", the variable type MUST match the type of the variable defined in the enclosing Choreography.

The optional attribute free, when set to "false" describes that a variable is defined in this Choreography. When the attribute free is set to "false", the variable resolves to the closest enclosing Choreography, regardless of the type of the variable.

The optional attribute silent-action, when set to "true" describes that activities used for making this variable available MUST NOT be present in the Choreography.

The optional attribute role is used to specify the location at which the variable information will reside.

The following rules apply to Variable Definitions:

- If a variable is defined without a Role, it is implied that it is defined at all the Roles that are part of the Relationships of the Choreography. For example if Choreography C1 has Relationship R that has a tuple (Role1, Role2), then a variable x defined in Chreography C1 without a Role attribute means it is defined at Role1 and Role2
- The variable with channelType MUST be defined without a role attribute

2.4.2.1 Expressions

Expressions are used in an assign activity to create new variable information by generating it from a constant value.

Predicate expressions are used in a Work Unit to specify its Guard condition.

The language used in WS-CDL for specifying expressions and query or conditional predicates is XPath 1.0. Additionally, WS-CDL defines XPath function extensions as described in Section 10.

2.4.3 Tokens

A *Token* is an alias for a piece of data in a variable or message that needs to be used by a Choreography. Tokens differ from Variables in that Variables contain values whereas Tokens contain information that defines the piece of the data that is relevant. For example a Token for "Order Amount" within an Order business could be an alias for an expression that pointed to the Order Amount element within an XML document. This could then be used as part of a condition that controls the ordering of a Choreography, for example "Order Amount > $1000".

All Tokens MUST have a type, for example, an Order Amount would be of type amount, Order Id could be alphanumeric and counter an integer.

Tokens types reference a document fragment within a Choreography definition and Token Locators provide a query mechanism to select them. By introducing these abstractions, a Choreography definition avoids depending on specific message types, as described by WSDL, or a specific query string, as specified by XPATH, but instead the the query string can change without affecting the Choreography definition.

The syntax of the *token* construct is:

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

The attribute informationType identifies the type of the document fragment.

The syntax of the *tokenLocator* construct is:

```
<tokenLocator  tokenName="qname"
      informationType="qname"
      query="XPath-expression"? />
```

The attribute tokenName identifies the name of the token type that the document fragment locator is associated with.

The attribute informationType identifies the type on which the query is performed to locate the token.

The attribute query defines the query string that is used to select a document fragment within a document.

The example below shows that the token purchaseOrderID is of type xsd:int. The two tokenLocators show how to access this token in "purchaseOrder" and "purchaseOrderAck" messages.

```
<token name="purchaseOrderID" informationType="xsd:int"/>
<tokenLocator tokenName="tns:purchaseOrderID" informationType="purchaseOrder"
query="/PO/OrderId"/>
<tokenLocator tokenName="tns:purchaseOrderID" informationType="purchaseOrderAck"
query="/POAck/OrderId"/>
```

2.4.4 Choreographies

A WS-CDL document defines agreed between parties, of alternative patterns of behaviorA *Choreography* allows constructing global compositions of parties by explicitly asserting their common and complementary observable behaviors.

A Choreography defined at the package level is called a top-level Choreography, and does not share its context with other top-level Choreographies. A Choreography performed within another Choreography is called an enclosed Choreography. A Package MAY contain exactly one top-level Choreography, that is explicitly marked as the root Choreography. The root Choreography is the only top-level Choreography that MAY be initiated. The root Choreography is enabled when it is initiated. All non-root, top-level Choreographies MAY be enabled when performed.

A Choreography facilitates recursive composition, where combining two or more Choreographies can form a new enclosing Choreography that may be re-used in different contexts.

A Choreography MUST contain at least one Relationship type, enumerating the observable behavior this Choreography requires its parties to exhibit. One or more Relationships MAY be defined within a Choreography, modeling multi-party collaborations.

A Choreography acts as a name scoping context as it restricts the visibility of variable information. A variable defined in a Choreography is visible in this Choreography and all its enclosed Choreographies, forming a *Choreography Visibility Horizon*.

A Choreography MUST contains one *Activity-Notation*. The Activity-Notation specifies the enclosed actions of the Choreography that perform the actual work.

A Choreography can recover from exceptional conditions and provide finalization actions by defining:

- One *Exception block*, which MAY be defined as part of the Choreography to recover from exceptional conditions that can occur in that enclosing Choreography
- One *Finalizer block*, which MAY be defined as part of the Choreography to provide the finalization actions for that enclosing Choreography

The *Choreography-Notation* is used to define a root or a top-level Choreography.

The syntax is:

```
<choreography  name="ncname"
      complete="xsd:boolean XPath-expression"?
      isolation="dirty-write"|
      "dirty-read"|"serializable"?
      root="true"|"false"? >

        <relationship  type="qname" />+

        variableDefinitions?

        Choreography-Notation*

        Activity-Notation

   <exception  name="ncname">
        WorkUnit-Notation+
   </exception>?
   <finalizer  name="ncname">
        WorkUnit-Notation
   </finalizer>?
</choreography>
```

The optional complete attribute allows to explicitly complete a Choreography as described below in the Choreography Life-line section.

The optional isolation attribute specifies when a variable information that is defined in an enclosing and changed within an enclosed Choreography is visible to its enclosing and sibling Choreographies:

- When isolation is set to "dirty-write", the variable information can be immediately overwritten by actions in other Choreographies
- When isolation is set to "dirty-read", the variable information is immediately visible to other Choreographies
- When isolation is set to "serializable", the variable information is visible to other Choreographies only after this Choreography has ended successfully

The relationship element within the choreography element enumerates the Relationships this Choreography MAY participate in.

The optional variableDefinitions element defines the variables that are visible in this Choreography and all its enclosed Choreographies and activities.

The optional root element marks a top-level Choreography as the root Choreography of a package.

The optional Choreography-Notation within the choreography element defines the Choreographies that MAY be performed only within this Choreography.

The optional exception element defines the Exception block of a Choreography by specifying one or more Exception Work Unit(s).

The optional finalizer element defines the Finalizer block of a Choreography by specifying one Finalizer Work Unit.

2.4.5 WorkUnits

A *Work Unit* prescribes the constraints that must be fulfilled for making progress within a Choreography. Examples of a Work Unit include:

- A *Send PO* Work Unit that includes Interactions for the Buyer to send an Order, the Supplier to acknowledge the order, and then later accept (or reject) the order. This work unit would probably not have a Guard
- An *Order Delivery Error* Work Unit that is performed whenever the *Place Order* Work Unit did not reach a "normal" conclusion. This would have a Guard condition that identifies the error – see also Choreography Exceptions and Transactions
- A *Change Order* Work Unit that can be performed whenever an order acknowledgement message has been received and an order rejection has not been received

A Work Unit can prescribe explicit enforcing the constraints that preserve the consistency of the collaborations commonly performed between the parties. Using a Work Unit an application can recover from faults that are the result from abnormal actions and also finalize completed actions that need to be logically rolled back.

A Work Unit specifies the data dependencies that must be satisfied before enabling one or more enclosed actions. These dependencies express interest(s) on the availability of variable information that already exists or will be created in the future.

Work Units interest(s) are matched when the required, one or more variable information become available. Availability of some variable information does not mean that a Work Unit matches immediately. Only when all variable information required by a Work Unit become available, in the appropriate Visibility Horizon, does matching succeed. Variable information available within a Choreography MAY be matched with a Work Unit that will be enabled in the future. When the matching succeeds the Work Unit is enabled.

A Work Unit MUST contain an *Activity-Notation*, which is enabled when its enclosing Work Unit is enabled.

A Work Unit completes successfully when all its enclosed actions complete successfully.

A Work Unit that completes successfully MUST be considered again for matching (based on its Guard condition), if its repetition condition evaluates to "true".

The *WorkUnit-Notation* is defined as follows:

```
<workunit  name="ncname"
      guard="xsd:boolean XPath-expression"?
      repeat="xsd:boolean XPath-expression"?
      block="true|false" >

      Activity-Notation
</workunit>
```

The Activity-Notation specifies the enclosed actions of a Work Unit.

The optional guard attribute describes the reactive interest on the availability of one or more, existing or future variable information and its usage is explained in section 2.4.5.1.

The optional repeat attribute allows, when the condition it specifies evaluates to "true", to make the current Work Unit considered again for matching (based on the guard condition attribute).

The block attribute specifies whether the matching condition relies on the variable that is currently available, or whether the Work Unit has to block for the variable to be available and its usage is explained in section 2.4.5.1.

The WS-CDL functions, as described in Section 10, MAY be used within a guard, and a repeat condition.

2.4.5.1 Reacting

A Reaction Guard describes a Work Unit’s interest for reacting on the availability of variable information when a constraint condition, which based on the variable information, is being satisfied.

The following rules apply when a Work Unit uses a Guard for reacting:

- When a Guard is not specified then the Work Unit always matches
- When a Guard is specified then:
  - One or more variables can be specified in a Guard, using the WS-CDL functions, as described in Section 10. Variables defined at different Roles can be combined together in a Guard using only an “and” logical operator.
  - When the block attribute is set to "false", then the Guard condition assumes that the variable information is currently available. If either the variable information is not available or the Guard condition evaluates to "false", then the Work Unit matching fails and the Activity-Notation enclosed within the Work Unit is skipped.
  - When the block attribute is set to "true" and one or more variable(s) are not available, then the Work Unit MUST block waiting for the variable

information to become available. When the variable information specified by the Guard condition become available then the Guard condition is evaluated. If the Guard condition evaluates to "true", then the Work Unit is matched. If the Guard condition evaluates to "false", then the Work Unit matching fails and the Activity-Notation enclosed within the Work Unit is skipped.

- When the WS-CDL function isAligned() is used in the Guard, it means that the Work Unit that specifies the Guard is waiting for an appropriate alignment Interaction to happen between the two Roles. When the isAligned() WS-CDL function is used in a Guard, then the Relationship within the isAligned() MUST be the subset of the Relationship that the immediate enclosing Choreography defined in the example below, the Guard specifies that the enclosed Work Unit is waiting for an alignment Interaction to happen between the customer Role and the retailer Role:

```
guard("cdl:isAligned("PurchaseOrder", "PurchaseOrder",
                     "customer-retailer-relationship")")
```

The examples below demonstrate the possible use of a Work Unit:

a. Example of a Work Unit with block equals to "true":

In the following Work Unit, the Guard waits on the availability of POAcknowledgement at customer Role and if it is already available, the activity happens, otherwise, the activity waits until the variable POAcknowledgement is initialized at the customer Role.

```
<workunit  name="POProcess"
      guard="cdl:getVariable("POAcknowledgement",
                              "tns:customer")"
      block="true"
... <!--some activity -->
</workunit>
```

b. Example of a Work Unit with block equals to "false":

In the following Work Unit, the Guard checks if StockQuantity at retailer Role is available and is greater than 10 and if so, the activity happens. If either the Variable is not available or the value is less than 10, the matching condition is "false" and the activity is skipped.

```
<workunit  name="Stockcheck"
           guard="cdl:getVariable("StockQuantity", "/Product/Qty",
                                  "retailer") > 10)"
           block="false" >
... <!--some activity -->
</workunit>
```

2.4.6 Reusing existing Choreographies

Choreographies can be combined and built from other Choreographies.

2.4.6.1 Composing Choreographies

Choreography Composition is the creation of new Choreographies by reusing existing Choreography definitions. For example if two separate Choreographies were defined as follows:

- A Request for Quote (RFQ) Choreography that involves a Buyer Role sending a request for a quotation for goods and services to a Supplier to which the Supplier responds with either a "Quotation" or a "Decline to Quote" message, and
- An Order Placement Choreography where the Buyer places and order for goods or services and the Supplier either accepts the order or rejects it

You could then create a new "Quote and Order" Choreography by reusing the two where the RFQ Choreography was executed first, and then, depending on the outcome of the RFQ Choreography, the order was placed using the Order Placement Choreography.

In this case the new Choreography is "composed" out of the two previously defined Choreographies. These Choreographies may be specified either:

- *Locally*, i.e. they are included, in the same Choreography definition as the Choreography that performed them, or
- *Globally,* i.e. they are specified in a separate Choreography definition that is defined elsewhere and performed in the root Choreography using perform construct

Using this approach, Choreographies can be recursively combined to support Choreographies of any required complexity allowing more flexibility as Choreographies defined elsewhere can be reused.

The example below shows a Choreography composition using an enclosed Choreography:

The root Choreography "PurchaseChoreo" has an enclosed Choreography "CustomerNotifyChoreo". The variable RetailerNotifyCustomer is visible to the enclosed Choreography.

```
<choreography name="PurchaseChoreo" root="true">
...
  <variable name="purchaseOrderAtRetailer" informationType="purchaseOrder"
role="Retailer"/>
...
  <choreography name="CustomerNotifyChoreo">
...
  </choreography>
  <workunit name="RetailerNotifyCustomer"
guard="cdl:getVariable(PoAckFromWareHouse, tns:WareHouse)">
    perform choreographyName="CustomerNotifyChoreo"
```

```
</workunit>
...
</choreography> <!--end of root choreography -->
```

2.4.6.2 Importing Choreographies

An *Importing* statement can contain references to a complete Choreography.

Importing statements must be interpreted in the sequence they occur.

When the Import statement contains references to variables or other data that have the same identity, then the content of the later Import statement replaces the same content referenced by the earlier Import statement. It also enables one Choreography definition to effectively be "cloned" by replacing the definitions for some or all of its variables.

The *importDefinitions* construct allows reusing Choreography types defined in another Choreography package such as Token types, Token Locator types, Information Types, Role types, Relationship types, Channel types and Choreographies.

In addition, WSDL documents can be imported and their definitions reused.

The syntax of the *importDefinitions* construct is:

```
<importDefinitions>
  <import  namespace="uri"  location="uri" />+
</importDefinitions>
```

The namespace and location attributes provide the namespace names and document location that contain additional Choreography and WSDL definitions that MUST be imported into this package.

2.4.7 Choreography Life-line

A Choreography life-line expresses the progression of a collaboration. Initially, the collaboration MUST be started, then work MAY be performed within it and finally it MAY complete. These different phases are designated by explicitly marked actions within the Choreography.

A root Choreography is initiated when the first Interaction, marked as the Choreography initiator, is performed. Two or more interactions MAY be marked as initiators, indicating alternative initiation actions. In this case, the first action will initiate the Choreography and the other actions will enlist with the already initiated Choreography. An Interaction designated as a Choreography initiator MUST be the first action performed in a Choreography. If a Choreography has two or more Work Units with interactions marked as initiators, then these are mutually exclusive and the Choreography will be initiated when the first Interaction occurs and the remaining Work Units will be disabled. All the interactions not marked as initiators indicate that they will enlist with an already initiated Choreography.

A Choreography completes successfully when there are no more enabled Work Unit(s) within it. Alternatively, a Choreography completes successfully if its complete condition, defined by the optional complete attribute within the choreography element, evaluates to "true" and there MUST NOT be any enabled Work Unit(s) within it but there MAY be one or more Work Units still unmatched.

2.4.8 Choreography Recovery

One or more Exception WorkUnit(s) MAY be defined as part of an enclosing Choreography to recover from exceptional conditions that may occur in that Choreography.

A Finalizer WorkUnit MAY be defined as part of an enclosing Choreography to provide the finalization actions that semantically rollback the completed enclosing Choreography.

2.4.8.1 Exception Block

A Choreography can sometimes fail as a result of an exceptional circumstance or error. Different types of exceptions are possible including this non-exhaustive list:

- *Interaction Failures*, for example the sending of a message did not succeed
- *Protocol Based Exchange failures*, for example no acknowledgement was received as part of a reliable messaging protocol [22]
- *Security failures*, for example a Message was rejected by a recipient because the digital signature was not valid
- *Timeout errors*, for example an Interaction did not complete within the required time
- *Validation Errors*, for example an XML order document was not well formed or did not conform to its schema definition
- *Application "failures"*, for example the goods ordered were out of stock

To handle these and other "errors" separate Work Units are defined in the Exception Block of a Choreography for each "exception" condition (as identified by its Guards) that needs to be handled. Only one Work Unit per exception SHOULD be performed.

When a Choreography encounters an exceptional condition it MAY need to act on it.

One or more Exception WorkUnit(s) MAY be defined as part of the Exception block of an enclosing Choreography for the purpose of handling the exceptional conditions occurring on that Choreography. To handle these an Exception Work Unit expresses interest on fault variable information that MAY become available.

A fault variable information is a result of:

- A fault occurring while performing an Interaction between parties

- A timeout occuring while an Interaction between parties was not completed within a specified time period

Exception Work Units are enabled when the enclosing Choregraphy is enabled. An Exception Work Unit MAY be enabled only once for an enclosing Choreography. Exception Work Units enabled in an enclosing Choreography MAY behave as the default mechanism to recover from faults for all its enclosed Choreographies. Exception Work Units enabled in an enclosed Choreography MAY behave as a mechanism to recover from faults for any of its enclosing Choreographies.

If a fault occurs within the top-level Choreography, then the faulted Choreography completes unsuccessfully and its Finalizer WorkUnit is not enabled. The actions, including enclosed Choreographies, enabled within the faulted Choreography are completed abnormally before an Exception Work Unit can be matched.

Within a Choreography only one Exception Work Unit MAY be matched. When an Exception Work Unit matches, it enables its appropriate activities for recovering from the fault.

Matching a fault with an Exception Work Unit is done as follows:

- If a fault is matched by an Exception Work Unit then the actions of the matched Work Unit are enabled
- If a fault is not matched by an Exception Work Unit defined within the Choreography in which the fault occurs, then the fault will be recursively propagated to the enclosing Exception Work Unit until a match is successful

The actions within the Exception Work Unit MAY use variable information visible in the Visibility Horizon of its enclosing Choreography as they stand at the current time.

The actions of an Exception Work Unit MAY also fault. The semantics for matching the fault and acting on it are the same as described in this section.

2.4.8.2 Finalizer Block

When a Choreography encounters an exceptional condition it MAY need to revert the actions it had already completed, by providing finalization actions that semantically rollback the effects of the completed actions. To handle these a separate Finalizer Work Unit is defined in the Finalizer Block of a Choreography.

A Choreography MAY define one Finalizer Work Unit.

A Finalizer WorkUnit is enabled only after its enclosing Choreography completes successfully. The Finalizer Work Unit may be enabled only once for an enclosing Choreography.

The actions within the Finalizer Work Unit MAY use variable information visible in the Visibility Horizon of its enclosing Choreography as they were at the time the enclosing Choreography completed or as they stand at the current time.

The actions of the Finalizer Work Unit MAY fault. The semantics for matching the fault and acting on it are the same as described in the previous section.

2.5 Activities

Activities are the lowest level components of the Choreography, used to describe the actual work.

An Activity-Notation is then either:

- A *Ordering Structure* – which combines Activities with other Ordering Structures in a nested way to specify the ordering rules of activities within the Choreography
- A *WorkUnit-Notation*
- A *Basic Activity* that performs the actual work. These are:
    - *Interaction*, which results in an exchange of messages between parties and possible synchronization of their states and the actual values of the exchanged information
    - A *Perform,* which means that a complete, separately defined Choreography is performed
    - An *Assign*, which assigns, within one Role, the value of one Variable to the value of a Variable
    - *No Action*, which means that the Choreography should take no particular action at that point

2.5.1 Ordering Structures

An *Ordering Structure* is one of the following:

- Sequence
- Parallel
- Choice

2.5.1.1 Sequence

The *sequence* ordering structure contains one or more Activity-Notations. When the sequence activity is enabled, the sequence element restricts the series of enclosed Activity-Notations to be enabled sequentially, in the same order that they are defined.

The syntax of this construct is:

```
<sequence>
    Activity-Notation+
</sequence>
```

2.5.1.2 Parallel

The *parallel* ordering structure contains one or more Activity-Notations that are enabled concurrently when the parallel activity is enabled.

The syntax of this construct is:

```
<parallel>
    Activity-Notation+
</parallel>
```

2.5.1.3 Choice

The *choice* ordering structure enables a Work Unit to define that only one of two or more Activity-Notations should be performed.

When two or more activities are specified in a choice element, only one activity is selected and the other activities are disabled. If the choice has Work Units with Guards, the first Work Unit that matches the Guard condition is selected and the other Work Units are disabled. If the choice has other activities, it is assumed that the selection criteria for the activities are non-observable.

The syntax of this construct is:

```
<choice>
    Activity-Notation+
</choice>
```

In the example below, choice element has two Interactions, processGoodCredit and processBadCredit. The Interactions have the same directionality, participate within the same Relationship and have the same fromRoles and toRoles names. If one Interaction happens, then the other one is disabled.

```
<choice>
  <interaction  channelVariable="doGoodCredit-channel" operation="doCredit">
...
  </interaction>
  <interaction channelVariable="badCredit-channel" operation="doBadCredit">
  ...
  </interaction>
<choice>
```

2.5.2 Interaction

An Interaction is the basic building block of a Choreography, which results in the exchange of information between parties and possibly the synchronization of their states and the values of the exchanged information.

An Interaction forms the base atom of the recursive Choreography composition, where multiple Interactions are combined to form a Choreography, which can then be used in different business contexts.

An Interaction is initiated when a party playing the requesting Role sends a request message, through a common Channel, to a party playing the accepting Role. The Interaction is continued when the accepting party, sends zero or one response message back to the requesting party. This means an Interaction can be one of two types:

- A *One-Way Interaction* that involves the sending of a single message
- A *Request-Response Interaction* when two messages are exchanged

An Interaction also contains "references" to:

- The *From Role* and *To Role* that are involved
- The *Message Content Type* that is being exchanged
- The *Information Exchange Variables* at the From Role and To Role that are the source and destination for the Message Content
- The *Channel Variable* that specifies the interface and other data that describe where and how the message is to be sent
- The *Operation* that specifies what the recipient of the message should do with the message when it is received
- A list of potential *State Changes* that can occur and may be aligned at the *From Role* and the *To Role* as a result of carrying out the Interaction

2.5.2.1 Interaction State Changes

State variables contain information about the state of a Role as a result of information exchanged in the form of an Interaction. For example after an Interaction where an order is sent by a Buyer to a Seller, the Buyer could create the *state variable* "Order State" and assign the value "Sent" when the message was sent, and when the Seller received the order, the Seller could also create its own version of the "Order State" *state variable* and assign it the value "Received".

As a result of a state change, several different state outcomes are possible, which can only be determined at run time. The Interaction MAY result in each of these allowed *state changes*, for example when an order is sent from a Buyer to a Seller the outcomes could be one of the following *state changes*:

1) Buyer.OrderState = Sent, Seller.OrderState = Received
2) Buyer.OrderState = SendFailure, Seller.OrderState not set
3) Buyer.OrderState = AckReceived, Seller.OrderState = OrderAckSent

2.5.2.2 Interaction Based Information Alignment

In some Choreographies there may be a requirement that, when the Interaction is performed, the Roles in the Choreography have agreement on the outcome.

- More specifically within an Interaction, a Role may need to have a common understanding of the state creations/changes of one or more *state variables* that are complementary to one or more *state variables* of its partner Role
- Additionally within an Interaction, a Role may need to have a common understanding of the values of the *information exchange variables* at the partner Role

With Interaction Alignment both the Buyer and the Seller have a common understanding that:

- State variables such as "Order State" variables at the Buyer and Seller, that have values that are complementary to each other, e.g. Sent at the Buyer and Received at the Seller, and
- Information exchange variables that have the same types with the same content, e.g. The Order variables at the Buyer and Seller have the same Information Types and hold the same order information

In WS-CDL an alignment Interaction MUST be explicitly used, in the cases where two interacting parties require the alignment of their states or their exchanged information between them. After the alignment Interaction completes, both parties progress at the same time, in a lock-step fashion and the variable information in both parties is aligned. Their variable alignment comes from the fact that the requesting party has to know that the accepting party has received the message and the other way around, the accepting party has to know that the requesting party has sent the message before both of them progress. There is no intermediate variable, where one party sends a message and then it proceeds independently or the other party receives a message and then it proceeds independently.

2.5.2.3 Protocol Based Information Exchanges

The one-way, request or response messages in an Interaction may also be implemented using a *Protocol Based Exchange* where a series of messages are exchanged according to some well-known protocol, such as the reliable messaging protocols defined in specifications such as WS-Reliability [22].

In both cases, the same or similar message content may be exchanged as in a simple Interaction, for example the sending of an Order between a Buyer and a Seller. Therefore some of the same state changes may result.

However when protocols such as the reliable messaging protocols are used, additional state changes will occur. For example, if a Reliable Messaging protocol were being used then the Buyer, once confirmation of delivery of the message was received, would also know that the Seller's "Order State" variable was in the state "Received" even though there was no separate Interaction that described this.

2.5.2.4 Interaction Life-line

The Channel through which an Interaction occurs is used to determine whether to enlist the Interaction with an already initiated Choreography or to initiate a new Choreography.

Within a Choreography, two or more related Interactions MAY be grouped to form a logical conversation. The Channel through which an Interaction occurs is used to determine whether to enlist the Interaction with an already initiated conversation or to initiate a new conversation.

An Interaction completes normally when the request and the response (if there is one) complete successfully. In this case the business documents and Channels exchanged during the request and the response (if there is one) result in the exchanged variable information being aligned between the two parties.

An Interaction completes abnormally if the following faults occur:

- The time-to-complete timeout identifies the timeframe within which an Interaction MUST complete. If this timeout occurs, after the Interaction was initiated but before it completed, then a fault is generated
- A fault signals an exception condition during the management of a request or within a party when accepting the request

In these cases the variable information remain the same at the both Roles as if this Interaction had never occurred.

The syntax of the *interaction* construct is:

```
<interaction  name="ncname"
              channelVariable="qname"
              operation="ncname"
              time-to-complete="xsd:duration"?
              align="true"|"false"?
              initiateChoreography="true"|"false"? >

   <participate  relationship="qname"
                 fromRole="qname" toRole="qname" />

   <exchange  messageContentType="qname"
              action="request"|"respond" >
     <send    variable="XPath-expression"? />

     <receive variable="XPath-expression"? />
   </exchange>*

   <record  name="ncname"
            role="qname" action="request"|"respond" >
     <source  variable="XPath-expression" />
     <target  variable="XPath-expression" />
   </record>*
</interaction>
```

The channel attribute specifies the Channel variable containing information of a party, being the target of an Interaction, which is used for determining where and how to send/receive information to/into the party. The Channel variable used in an Interaction MUST be available at the two Roles before the Interaction occurs.

At runtime, information about a Channel variable is expanded further. This requires that the messages in the Choreography also contain correlation information, for example by including:

- A SOAP header that specifies the correlation data to be used with the Channel, or
- Using the actual value of data within a message, for example the Order Number of the Order that is common to all the messages sent over the Channel

In practice, when a Choreography is performed, several different ways of doing correlation may be employed which vary depending on the Channel Type.

The attribute operation specifies a one-way or a request-response operation. The specified operation belongs to the interface, as identified by the role and behavior elements of the Channel used in the interaction activity.

The optional time-to-complete attribute identifies the timeframe within which an Interaction MUST complete.

The optional align attribute when set to "true" means that the Interaction results in the common understanding of both the information exchanged and the resulting state creations or changes at the ends of the Interaction as specified in the fromRole and the toRole. The default for this attribute is "false".

An Interaction activity can be marked as a Choreography initiator when the optional initiateChoreography attribute is set to "true". The default for this attribute is "false".

Within the participate element, the relationship attribute specifies the Relationship this Choreography participates in and the fromRole and toRole attributes specify the requesting and the accepting Roles respectively.

The optional exchange element allows information to be exchanged during a one-way request or a request/response Interaction.

The messageContentType attribute, within the exchange element, identifies the informationType or the channelType of the information that is exchanged between the two Roles in an Interaction.

The attribute action, within the exchange element, specifies the direction of the information exchanged in the Interaction:

- When the action attribute is set to “request”, then the message exchange happens fromRole to toRole
- When the action attribute is set to ”respond”, then the message exchange happens from toRole to fromRole

Within the exchange element, the send element shows that information is sent from a Role and the receive element shows that information is received at a Role respectively in the Interaction:

- The optional variables specified within the send and receive elements MUST be of type as described in the messageContentType element

- When the action element is set to "request", then the variable specified within the send element using the variable attribute MUST be defined at the fromRole and the variable specified within the receive element using the variable attribute MUST be defined at the toRole

- When the action element is set to "respond", then the variable specified within the send element using the variable attribute MUST be defined at the toRole and the variable specified within the receive element using the variable attribute MUST be defined at fromRole

The optional element record is used to create or change one or more variables at the ends of the Interaction, either at one or at both Roles. For example, the PurchaseOrder message contains the Channel of the Role "Customer" when sent to the Role "Retailer". This can be copied into the appropriate variable of the "Retailer" within the record element. When the align attribute is set to "true" for the Interaction, it also means that the Customer knows that the Retailer now has the contact information of the Customer. In another example, the Customer sets its state "OrderSent" to "true" and the Retailer sets its state "OrderReceived" to "true". Similarly the Customer sets "OrderAcknowledged" "true".

The source and the target elements within the record element represent the variable names at the Role that is specified in the role attribute within the record element.

The following rules apply for record:

- One or more records MAY be defined at only one or both the Roles in an Interaction

- A record MAY be defined before or after a request exchange or a response exchange. In addition a record MAY be defined even in the absence of an exchange

The example below shows a complete Choreography that involves one Interaction. The Interaction happens from Role “Consumer” to Role “Retailer” on the Channel "retailer-channel" as a request/response message exchange.

- The message purchaseOrder is sent from Consumer to Retailer as a request message

- The message purchaseOrderAck is sent from Retailer to Consumer as a response message

- The variable consumer-channel is populated at Retailer using the record element

- The Interaction happens on the retailer-channel which has a token purchaseOrderID used as an identity of the channel. This identity element is used to identify the business process of the retailer

- The request message purchaseOrder contains the identity of the retailer business process as specified in the tokenLocator for purchaseOrder message
- The response message purchaseOrderAck contains the identity of the consumer business process as specified in the tokenLocator for purchaseOrderAck message
- The consumer-channel is sent as a part of purchaseOrder message from consumer to retailer on retailer-channel during the request. The record element populates the consumer-channel at the retailer role

```
<package name="ConsumerRetailerChoreo" version="1.0"
  <informationType name="purchaseOrderType" type="pons:PurchaseOrderMsg"/>
  <informationType name="purchaseOrderAckType" type="pons:PurchaseOrderAckMsg"/>
  <token name="purchaseOrderID" informationType="tns:intType"/>
  <token name="retailerRef" informationType="tns:uriType"/>
  <tokenLocator tokenName="tns:purchaseOrderID"
                informationType="tns:purchaseOrderType" query="/PO/orderId"/>
  <tokenLocator tokenName="tns:purchaseOrderID"
                informationType="tns:purchaseOrderAckType" query="/PO/orderId"/>
  <role name="Consumer">
    <behavior name="consumerForRetailer" interface="cns:ConsumerRetailerPT"/>
    <behavior name="consumerForWarehouse" interface="cns:ConsumerWarehousePT"/>
  </role>
  <role name="Retailer">
    <behavior name="retailerForConsumer" interface="rns:RetailerConsumerPT"/>
  </role>
  <relationship name="ConsumerRetailerRelationship">
    <role type="tns:Consumer" behavior="consumerForRetailer"/>
    <role type="tns:Retailer" behavior="retailerForConsumer"/>
  </relationship>
  <channelType name="ConsumerChannel">
    <role type="tns:Consumer"/>
    <reference>
      <token type="tns:consumerRef"/>
    </reference>
    <identity>
      <token type="tns:purchaseOrderID"/>
    </identity>
  </channelType>
  <channelType name="RetailerChannel">
    <passing channel="ConsumerChannel" action="request" />
    <role type="tns:Retailer" behavior="retailerForConsumer"/>
    <reference>
      <token type="tns:retailerRef"/>
    </reference>
    <identity>
      <token type="tns:purchaseOrderID"/>
    </identity>
  </channelType>
  <choreography name="ConsumerRetailerChoreo" root="true">
    <relationship type="tns:ConsumerRetailerRelationship"/>
    <variableDefinitions>
    <variable name="purchaseOrder" informationType="tns:purchaseOrderType"
              silent-action="true" />
    <variable name="purchaseOrderAck" informationType="tns:purchaseOrderAckType" />
    <variable name="retailer-channel" channelType="tns:RetailerChannel"/>
    <variable name="consumer-channel" channelType="tns:ConsumerChannel"/>
    <interaction channelVariable="tns:retailer-channel "
```

```
                 operation="handlePurchaseOrder" align="true"
                 initiateChoreography="true">
      <participate relationship="tns:ConsumerRetailerRelationship"
                   fromRole="tns:Consumer" toRole="tns:Retailer"/>
      <exchange messageContentType="tns:purchaseOrderType" action="request">
        <send variable="cdl:getVariable(tns:purchaseOrder, tns:Consumer)"/>
        <receive variable="cdl:getVariable(tns:purchaseOrder, tns:Retailer)"/>
      </exchange>
      <exchange messageContentType="purchaseOrderAckType" action="respond">
        <send variable="cdl:getVariable(tns:purchaseOrderAck, tns:Retailer)"/>
        <receive variable="cdl:getVariable(tns:purchaseOrderAck, tns:Consumer)"/>
      </exchange>
      <record role="tns:Retailer" action="request">
        <source variable="cdl:getVariable(tns:purchaseOrder, PO/CustomerRef,
tns:Retailer)"/>
        <target variable="cdl:getVariable(tns:consumer-channel, tns:Retailer)"/>
      </record>
    </interaction>
  </choreography>
</package>
```

2.5.3 Performed Choreography

The perform activity enables a Choreography to specify that another Choreography is performed at this point in its definition, as an enclosed Choreography. The Choreography that is performed can be defined either within the same Choreography Definition or separately.

The syntax of the *perform* construct is:

```
<perform  choreographyName="qname">
   <alias  name="ncname">
     <this variable="XPath-expression" role="qname"/>
     <free variable="XPath-expression" role="qname"/>
   </alias>*
</perform>
```

Within the perform element the choreographyName attribute references a non-root Choreography defined in the same or in a different Choreography package that is to be performed. The performed Choreography can be defined locally within the same Choreography or globally, in the same or different Choreography package. The performed Choreography defined in a different package is conceptually treated as an enclosed Choreography.

The optional alias element within the perform element enables information in the performing Choreography to be shared with the performed Choreography and vice versa. The role attribute aliases the Roles from the performing Choreography to the performed Choreography.

The variable within the this element identifies a variable in the performing choreography that replaces the variable identified by the free element in the performed choreography.

The following rules applywhen a Choreography is performed:

- The Choreography to be performed MUST NOT be a root Choreography
- The Choreography to be performed MUST be defined either using a Choreography-Notation in the same Choreography or it MUST be a top-level Choreography with root attribute set to "false" in the same or different Choreography package
- The roles within a single alias element must be carried out by the same participant
- If the performed Choreography is defined within the performing Choreography, the variables that are in the visibility horizon are visible to the performed Choreography also
- Performed Choreography, if not defined within the enclosing Choreography, can be used by other Choreographies and hence the contract is reusable
- There should not be a cyclic dependency on the Choreographies performed. For example Choreography C1 is performing Choreography C2 which is performing Choreography C1 again

The example below shows a Choreography performing another Choreography:

The root Choreography "PurchaseChoreo" performs the Choreography "RetailerWarehouseChoreo" and aliases the variable "purchaseOrderAtRetailer" defined in the enclosing Choreography to "purchaseOrder" defined at the performed enclosed Choreography "RetailerWarehouseChoreo". Once aliased, the visibility horizon of the variable purchaseOrderAtRetailer is the same as it would be for the enclosed Choreography.

```
<choreography name="PurchaseChoreo" root="true">
...
  <variable name="purchaseOrderAtRetailer"
                  informationType="purchaseOrder" role="Retailer"/>
...
  <perform choreographyName="RetailerWarehouseChoreo">
    <alias name="aliasRetailer">
      <this variable="cdl:getVariable(tns:purchaseOrder, tns:Retailer)"
            role="tns:Retailer"/>
      <free variable="cdl:getVariable(tns:purchaseOrder, rwns:Retailer)"
            role="rwns:Retailer"/>
    </alias>
    ...
</choreography>
```

2.5.4 Assigning Variables

Assign is used to create or change and then make available within one Role, the value of one Variable using the value of another Variable.

The assignments may include:

- Assigning one variable to another or a part of the variable to another variable so that a message received can be used to trigger/constrain, using a Work Unit Guard, or other Interactions

- Assigning a locally defined variable to part of the data contained in an information exchange variable

The syntax of the *assign* construct is:

```
<assign  role="qname">
   <copy  name="ncname">
      <source  variable="XPath-expression" />
      <target  variable="XPath-expression" />
   </copy>+
</assign>
```

The assign construct makes available at a Role the variable defined by the target element using the variable defined by the source element at the same Role.

The following rules apply to assignment:

- The source and the target variable MUST be of same type

- The source and the target variable MUST be defined at the same Role

The following example assigns the customer address part from PurchaseOrderMsg to CustomerAddress variable.

```
<assign role="tns:retailer">
  <copy name="copyChannel">
    <source variable="cdl:getVariable("PurchaseOrderMsg", "/PO/CustomerAddress",
            tns:retailer)" />
    <target variable="cdl:getVariable("CustomerAddress", tns:retailer)" />
  </copy>
</assign>
```

2.5.5 Actions with non-observable effects

The *Noaction* activity models the performance of a silent action that has non-observable effects on any of the collaborating parties.

The syntax of the *noaction* construct is:

```
<noaction/>
```

3 Example

To be completed

4 Relationship with the Security framework

Because messages can have consequences in the real world, the collaboration parties will impose security requirements on the message exchanges. Many of these requirements can be satisfied by the use of WS-Security [24].

5 Relationship with the Reliable Messaging framework

The WS-Reliability specification [22] provides a reliable mechanism to exchange business documents among collaborating parties. The WS-Reliability specification prescribes the formats for all messages exchanged without placing any restrictions on the content of the encapsulated business documents. The WS-Reliability specification supports one-way and request/response message exchange patterns, over various transport protocols (examples are HTTP/S, FTP, SMTP, etc.). The WS-Reliability specification supports sequencing of messages and guaranteed, exactly once delivery.

A violation of any of these consistency guarantees results in an error condition, reflected in the Choreography as an Interaction fault.

6 Relationship with the Transaction/Coordination framework

In WS-CDL, two parties make progress by interacting. In the cases where two interacting parties require the alignment of their States or their exchanged information between them, an alignment Interaction is modeled in a Choreography. After the alignment Interaction completes, both parties progress at the same time, in a lock-step fashion. The variable information alignment comes from the fact that the requesting party has to know that the accepting party has received the message and the other way around, the accepting party has to know that the requesting party has sent the message before both of them progress. There is no intermediate variable, where one party sends a message and then it proceeds independently or the other party receives a message and then it proceeds independently.

Implementing this type of handshaking in a distributed system requires support from a Transaction/Coordination protocol, where agreement of the outcome among parties can be reached even in the case of failures and loss of messages.

7 Acknowledgments

To be completed

8 References

[1] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, Harvard University, March 1997

[2] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, MIT/LCS, U.C. Irvine, Xerox Corporation, August 1998.

[3] http://www.w3.org/TR/html401/interaction/forms.html#submit-format

[4] http://www.w3.org/TR/html401/appendix/notes.html#ampersands-in-uris

[5] http://www.w3.org/TR/html401/interaction/forms.html#h-17.13.4

[6] Simple Object Access Protocol (SOAP) 1.1 "http://www.w3.org/TR/2000/NOTE-SOAP-20000508/"

[7] Web Services Definition Language (WSDL) 2.0

[8] Industry Initiative "Universal Description, Discovery and Integration"

[9] W3C Recommendation "The XML Specification"

[10] XML-Namespaces " Namespaces in XML, Tim Bray et al., eds., W3C, January 1999"

http://www.w3.org/TR/REC-xml-names

[11] W3C Working Draft "XML Schema Part 1: Structures". This is work in progress.

[12] W3C Working Draft "XML Schema Part 2: Datatypes". This is work in progress.

[13] W3C Recommendation "XML Path Language (XPath) Version 1.0"

[14] "Uniform Resource Identifiers (URI): Generic Syntax," RFC 2396, T. Berners-Lee, R. Fielding, L. Masinter, MIT/LCS, U.C. Irvine, Xerox Corporation, August 1998.

[15] WSCI: Web Services Choreography Interface 1.0, A.Arkin et.al

[16] XLANG: Web Services for Business Process Design

[17] WSFL: Web Service Flow Language 1.0

[18] BPEL: Business Process Execution Language 1.1

[19] BPML: Business Process Modeling Language 1.0

[20] XPDL: XML Processing Description Language 1.0

[21] WS-CAF: Web Services Context, Coordination and Transaction Framework 1.0

[22] Web Services Reliability 1.0

[23] The Java Language Specification

[24] Web Services Security

[25] J2EE: Java 2 Platform, Enterprise Edition, Sun Microsystems

[26] ECMA. 2001. Standard ECMA-334: C# Language Specification

9 WS-CDL XSD Schemas

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema
     targetNamespace=http://www.w3.org/ws/choreography/2004/02/WSCDL/
     xmlns=http://www.w3.org/2001/XMLSchema
     xmlns:cdl=http://www.w3.org/ws/choreography/2004/02/WSCDL/
     elementFormDefault="qualified">

  <complexType name="tExtensibleElements">
    <annotation>
      <documentation>
        This type is extended by other CDL component types to allow
          elements and attributes from other namespaces to be added.
        This type also contains the optional description element that
        is applied to all CDL constructs.
      </documentation>
    </annotation>
    <sequence>
      <element name="description" minOccurs="0">
        <complexType mixed="true">
          <sequence minOccurs="0" maxOccurs="unbounded">
            <any processContents="lax"/>
          </sequence>
        </complexType>
      </element>
      <any namespace="##other" processContents="lax"
          minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <anyAttribute namespace="##other" processContents="lax"/>

  </complexType>
  <element name="package" type="cdl:tPackage"/>
  <complexType name="tPackage">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="importDefinitions"
                  type="cdl:tImportDefinitions" minOccurs="0"
                  maxOccurs="unbounded"/>
          <element name="informationType" type="cdl:tInformationType"
                  minOccurs="0" maxOccurs="unbounded"/>
          <element name="token" type="cdl:tToken" minOccurs="0"
                  maxOccurs="unbounded"/>
          <element name="tokenLocator" type="cdl:tTokenLocator"
                  minOccurs="0" maxOccurs="unbounded"/>
          <element name="role" type="cdl:tRole" minOccurs="0"
                  maxOccurs="unbounded"/>
          <element name="relationship" type="cdl:tRelationship"
                  minOccurs="0" maxOccurs="unbounded"/>
          <element name="participant" type="cdl:tParticipant"
                  minOccurs="0" maxOccurs="unbounded"/>
          <element name="channelType" type="cdl:tChannelType"
                  minOccurs="0" maxOccurs="unbounded"/>
          <element name="choreography" type="cdl:tChoreography"
                  minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="author" type="string" use="optional"/>
        <attribute name="version" type="string" use="required"/>
        <attribute name="targetNamespace" type="anyURI"
```

```
                 use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tImportDefinitions">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="import" type="cdl:tImport"
                  maxOccurs="unbounded"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tImport">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="namespace" type="anyURI" use="required"/>
        <attribute name="location" type="anyURI" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tInformationType">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="type" type="QName" use="optional"/>
        <attribute name="element" type="QName" use="optional"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tToken">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="informationType" type="QName"
                 use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tTokenLocator">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="tokenName" type="QName" use="required"/>
        <attribute name="informationType" type="QName"
                 use="required"/>
        <attribute name="query" type="cdl:tXPath-expr"
                 use="optional"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tRole">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="behavior" type="cdl:tBehavior"
                  maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tBehavior">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="interface" type="QName" use="optional"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tRelationship">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="role" type="cdl:tRoleRef" minOccurs="2"
                  maxOccurs="2"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tRoleRef">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="type" type="QName" use="required"/>
        <attribute name="behavior" type="NCName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tParticipant">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="role" type="cdl:tRoleRef2"
                  maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tRoleRef2">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="type" type="QName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tChannelType">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="passing" type="cdl:tPassing" minOccurs="0"
                  maxOccurs="unbounded"/>
          <element name="role" type="cdl:tRoleRef3"/>
          <element name="reference" type="cdl:tReference"/>
          <element name="identity" type="cdl:tIdentity" minOccurs="0"
                  maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="usage" type="cdl:tUsage" use="optional"
                     default="unlimited"/>
        <attribute name="action" type="cdl:tAction" use="optional"
                     default="request-respond"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tRoleRef3">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="type" type="QName" use="required"/>
        <attribute name="behavior" type="NCName" use="optional"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tPassing">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="channel" type="QName" use="required"/>
        <attribute name="action" type="cdl:tAction" use="optional"
                 default="request-respond"/>
        <attribute name="new" type="boolean" use="optional"
                 default="true"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tReference">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="token" type="cdl:tTokenReference"
                      maxOccurs="unbounded"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tTokenReference">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="name" type="QName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tIdentity">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="token" type="cdl:tTokenReference"
                  maxOccurs="unbounded"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
```

```
<complexType name="tChoreography">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <sequence>
        <element name="relationship" type="cdl:tRelationshipRef"
                maxOccurs="unbounded"/>
        <element name="variableDefinitions"
                type="cdl:tVariableDefinitions" minOccurs="0"/>
        <element name="choreography" type="cdl:tChoreography"
                 minOccurs="0" maxOccurs="unbounded"/>
        <group ref="cdl:activity"/>
        <element name="exception" type="cdl:tException"
                minOccurs="0"/>
        <element name="finalizer" type="cdl:tFinalizer"
                    minOccurs="0"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="complete" type="cdl:tBoolean-expr"
                   use="optional"/>
      <attribute name="isolation" type="cdl:tIsolation"
                   use="optional" default="dirty-write"/>
      <attribute name="root" type="boolean" use="optional"
                   default="false"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tRelationshipRef">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <attribute name="type" type="QName" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tVariableDefinitions">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <sequence>
        <element name="variable" type="cdl:tVariable"
                maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tVariable">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="informationType" type="QName"
               use="optional"/>
      <attribute name="channelType" type="QName" use="optional"/>
      <attribute name="mutable" type="boolean" use="optional"
               default="true"/>
      <attribute name="free" type="boolean" use="optional"
               default="false"/>
      <attribute name="silent-action" type="boolean" use="optional"
               default="false"/>
      <attribute name="role" type="QName" use="optional"/>
    </extension>
  </complexContent>
</complexType>

<group name="activity">
  <choice>
    <element name="sequence" type="cdl:tSequence"/>
    <element name="parallel" type="cdl:tParallel"/>
    <element name="choice" type="cdl:tChoice"/>
    <element name="workunit" type="cdl:tWorkunit"/>
    <element name="interaction" type="cdl:tInteraction"/>
    <element name="perform" type="cdl:tPerform"/>
    <element name="assign" type="cdl:tAssign"/>
    <element name="noaction" type="cdl:tNoaction"/>
  </choice>
</group>

<complexType name="tSequence">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <sequence>
        <group ref="cdl:activity" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tParallel">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <sequence>
        <group ref="cdl:activity" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="tChoice">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <sequence>
        <group ref="cdl:activity" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tWorkunit">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <sequence>
        <group ref="cdl:activity"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="guard" type="cdl:tBoolean-expr"
               use="optional"/>
      <attribute name="repeat" type="cdl:tBoolean-expr"
               use="optional"/>
      <attribute name="block" type="boolean" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tPerform">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
```

```
        <sequence>
          <element name="alias" type="cdl:tAlias"
                  maxOccurs="unbounded"/>
        </sequence>
        <attribute name="choreographyName" type="QName"
                 use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tAlias">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="this" type="cdl:tAliasVariable"/>
          <element name="free" type="cdl:tAliasVariable"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tAliasVariable">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="variable" type="cdl:tXPath-expr"
                 use="required"/>
        <attribute name="role" type="QName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tInteraction">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="participate" type="cdl:tParticipate"/>
          <element name="exchange" type="cdl:tExchange" minOccurs="0"
                  maxOccurs="unbounded"/>
          <element name="record" type="cdl:tRecord" minOccurs="0"
                  maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="channelVariable" type="QName"
                 use="required"/>
        <attribute name="operation" type="NCName" use="required"/>
        <attribute name="time-to-complete" type="duration"
                 use="optional"/>
        <attribute name="align" type="boolean" use="optional"
                 default="false"/>
        <attribute name="initiateChoreography" type="boolean"
                 use="optional" default="false"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tParticipate">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="relationship" type="QName" use="required"/>
        <attribute name="fromRole" type="QName" use="required"/>
        <attribute name="toRole" type="QName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tExchange">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="send" type="cdl:tVariableRef"/>
          <element name="receive" type="cdl:tVariableRef"/>
        </sequence>
        <attribute name="messageContentType" type="QName"
                 use="required"/>
        <attribute name="action" type="cdl:tAction2" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tVariableRef">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="variable" type="cdl:tXPath-expr"
                use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tRecord">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="source" type="cdl:tVariableRef"/>
          <element name="target" type="cdl:tVariableRef"/>
        </sequence>
        <attribute name="name" type="string" use="required"/>
        <attribute name="role" type="QName" use="required"/>
        <attribute name="action" type="cdl:tAction2" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tAssign">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
      <element name="copy" type="cdl:tCopy"
               maxOccurs="unbounded"/>
        </sequence>
        <attribute name="role" type="QName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tCopy">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="source" type="cdl:tVariableRef"/>
          <element name="target" type="cdl:tVariableRef"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
      </extension>
    </complexContent>
  </complexType>
  <complexType name="tNoaction">
    <complexContent>
      <extension base="cdl:tExtensibleElements"/>
    </complexContent>
  </complexType>

  <complexType name="tException">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="workunit" type="cdl:tWorkunit"
                  maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tFinalizer">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="workunit" type="cdl:tWorkunit"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <simpleType name="tAction">
    <restriction base="string">
      <enumeration value="request-respond"/>
      <enumeration value="request"/>
      <enumeration value="respond"/>
    </restriction>
  </simpleType>

  <simpleType name="tAction2">
    <restriction base="string">
      <enumeration value="request"/>
      <enumeration value="respond"/>
    </restriction>
  </simpleType>

  <simpleType name="tUsage">
    <restriction base="string">
      <enumeration value="once"/>
      <enumeration value="unlimited"/>
    </restriction>
  </simpleType>

  <simpleType name="tBoolean-expr">
    <restriction base="string"/>
  </simpleType>

  <simpleType name="tXPath-expr">
    <restriction base="string"/>
  </simpleType>

  <simpleType name="tIsolation">
    <restriction base="string">
      <enumeration value="dirty-write"/>
      <enumeration value="dirty-read"/>
      <enumeration value="serializable"/>
    </restriction>
  </simpleType>
</schema>
```

10 WS-CDL Supplied Functions

There are several functions that the WS-CDL specification supplies as XPATH extension functions. These functions can be used in any XPath expression as long as the types are compatible.

xsd:dateTime getCurrentTime()

xsd:dateTime getCurrentDate()

xsd:dateTime getCurrentDateTime()

Returns the current date/time.

xsd:string createNewID()

Returns a new globally unique string value for use as an identifier.

xsd:any getVariable(xsd:string varName, xsd:string documentPath?, xsd:string roleName)*

Returns the information of the variable with name varName at a Role as a node set containing a single node. The second parameter is optional. When the second parameter is not used, this function retrieves from the variable information the entire document. When the second parameter is used, this function retrieves from the variable information, the fragment of the document at the provide absolute location path.

xsd:boolean isAligned(xsd:string varName, xsd:string withVarName, xsd:string relationshipName)

Returns "true" if the variable with name varName has aligned its information (states or values) with the variable named withVarName, within a Relationship as specified by the relationshipName.