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 3rd 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. This document represents consensus within
the Working Group about the Web Services Choreography description language.

This document is a chartered deliverable of the Web Services Choreography
Working Group.

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 4th editor's draft of the Web Services Choreography Description
Language document.
# Table of Contents

66  Status of this Document ........................................................................................................ 2
67  Revision Description ............................................................................................................... 2
68  1  Introduction .................................................................................................................. 5
69   1.1  Notational Conventions .......................................................................................... 6
70   1.2  Purpose of the Choreography Description Language ........................................... 8
71   1.3  Goals ....................................................................................................................... 9
72   1.4  Relationship with XML and WSDL ..................................................................... 11
73   1.5  Relationship with Business Process Languages .................................................. 11
74   1.6  Time Assumptions ................................................................................................. 11
75  2  Choreography Description Language Model ............................................................... 11
76   2.1  WS-CDL Model Overview ..................................................................................... 12
77   2.2  WS-CDL Document Structure .............................................................................. 13
78   2.2.1  Choreography Package ................................................................................... 13
79   2.2.2  Including WS-CDL Type Definitions ............................................................... 14
80   2.2.3  WS-CDL document Naming and Linking ......................................................... 15
81   2.2.4  Language Extensibility and Binding ................................................................. 15
82   2.2.5  Semantics ............................................................................................................ 15
83   2.3  Collaborating Parties ............................................................................................. 16
84   2.3.1  Role Types ......................................................................................................... 16
85   2.3.2  Relationship Types ............................................................................................ 17
86   2.3.3  Participant Types ............................................................................................... 18
87   2.3.4  Channel Types .................................................................................................... 19
88   2.4  Information Driven Collaborations ........................................................................ 21
89   2.4.1  Information Types ............................................................................................. 21
90   2.4.2  Variables ............................................................................................................. 23
91   2.4.3  Expressions ........................................................................................................ 25
92   2.4.3.1  WS-CDL Supplied Functions ...................................................................... 25
93   2.4.4  Tokens ................................................................................................................ 27
94   2.4.5  Choreographies ................................................................................................. 29
95   2.4.6  WorkUnits ......................................................................................................... 31
96   2.4.7  Choreography Life-line .................................................................................... 35
97   2.4.8  Choreography Exception Handling .................................................................... 36
98   2.4.9  Choreography Finalization ............................................................................... 38
99   2.4.10  Choreography Coordination ............................................................................ 39
100  2.5  Activities ................................................................................................................ 42
101  2.5.1  Ordering Structures ........................................................................................... 43
102  2.5.1.1  Sequence ....................................................................................................... 43
103  2.5.1.2  Parallel ........................................................................................................... 43
104  2.5.1.3  Choice ............................................................................................................ 43
105  2.5.2  Interacting .......................................................................................................... 44
106  2.5.2.1  Interaction Based Information Alignment .................................................... 45
107  2.5.2.2  Interaction Life-line ....................................................................................... 46
108  2.5.2.3  Interaction Syntax .......................................................................................... 46
109  2.5.3  Composing Choreographies ................................................................................. 53
1 Introduction

For many years, organizations have been developing solutions for automating their 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 interact with a 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 message set 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.

- **Registry**: allows publishing the availability of a Web Service and its discovery from service requesters using sophisticated searching mechanisms.

- **Security layer**: ensures that exchanged information are not modified or forged in a verifiable manner and that parties can be authenticated.

- **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 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 aimed at the composition of interoperable 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:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Namespace URI</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsdl</td>
<td><a href="http://www.w3.org/2004/08/wSDL">http://www.w3.org/2004/08/wSDL</a></td>
<td>WSDL 2.0 namespace for WSDL framework.</td>
</tr>
<tr>
<td>cdl</td>
<td><a href="http://www.w3.org/2004/12/ws-chor/cdl">http://www.w3.org/2004/12/ws-chor/cdl</a></td>
<td>WSCDL namespace for Choreography Description Language.</td>
</tr>
<tr>
<td>xsd</td>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
<td>Schema namespace as defined by XSD [12].</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>tns</th>
<th>(various)</th>
<th>The &quot;this namespace&quot; (tns) prefix is used as a convention to refer to the current document.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(other)</td>
<td>(various)</td>
<td>All other namespace prefixes are samples only. In particular, URIs starting with &quot;<a href="http://sample.com">http://sample.com</a>&quot; represent some application-dependent or context-dependent URIs [4].</td>
</tr>
</tbody>
</table>

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; other examples are fragments and require additional information to be specified in order to conform.

An XSD is provided as a formal definition of WS-CDL grammar (see Section 11).
1.2 Purpose of the Choreography Description Language

Business or other activities that involve different organizations or independent processes are engaged in a collaborative fashion to achieve a common business goal, such as Order Fulfillment.

For the collaboration to work successfully, the rules of engagement between all the interacting parties must be provided. Whereas today these rules are frequently written in English, a standardized way for precisely defining these interactions, leaving unambiguous documentation of the parties and responsibilities of each, is missing.

The Web Services Choreography specification is aimed at being able to precisely describe collaborations between any type of party regardless of the supporting platform or programming model used by the implementation of the hosting environment.

Using the Web Services Choreography specification, a contract containing 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 parties involved. Each party can then use the global definition to build and test solutions that conform to it. The global specification is in turn realized by combination of the resulting local systems, on the basis of appropriate infrastructure support.

The advantage of a contract based on a global viewpoint as opposed to anyone endpoint is that it separates the overall "global" process being followed by an individual business or system within a "domain of control" (an endpoint) from the definition of the sequences in which each business or system exchanges information with others. This means that, as long as the "observable" sequences do not change, the rules and logic followed within a domain of control (endpoint) can change at will and interoperability is therefore guaranteed.

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 or global view. It is the intent of CDL that the conformance of each implementation to the common view expressed in CDL is easy to determine.

The figure below demonstrates a possible usage of the Choreography Description Language.
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. They then generate a Choreography Description Language based representation. In this example, a Choreography specifies the interactions between services across business entities ensuring interoperability, while leaving actual implementation decisions in the hands of each individual company:

- Company “A” relies on a WS-BPEL [18] solution to implement its own part of the Choreography
- 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# to implement its own part of the Choreography

Similarly, a Choreography can specify the interoperability and interactions between services within one business entity.

1.3 Goals

The primary goal of a Choreography Description Language is to specify a declarative, XML based language that defines from a global viewpoint the common and complementary observable behavior specifically, the information...
More specifically, the goals of the Choreography Description 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 "inclusion" facility that allows a Choreography to be created from parts contained in several different Choreographies

- **Information Driven Collaboration.** Choreographies describe how parties make progress within a collaboration, through the recording of exchanged information and changes to observable information that cause ordering constraints to be fulfilled and progress to be made

- **Information Alignment.** Choreographies allow the parties that take part in Choreographies to communicate and synchronize their observable information

- **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 participants, each with their own, non-observable business rules and goals

- **Specification Composability.** 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 (WS-BPEL) [18], etc.
1.4 Relationship with XML and WSDL

The WS-CDL specification depends on the following specifications: XML 1.0 [9], XML-Namespaces [10], XML-Schema 1.0 [11, 12] and XPath 1.0 [13]. Support for including and referencing service definitions given in WSDL 2.0 [7] is a normative part of the WS-CDL specification. In addition, support for including and referencing service definitions given in WSDL 1.1 as constrained by WS-I Basic Profile [Action: add references] is a normative part of the WS-CDL specification.

1.5 Relationship with Business Process Languages

A Choreography Description Language is not an "executable business process description language" or an implementation language. The role of specifying the execution logic of an application will be covered by these specifications [16, 17, 18, 19, 20, 23, 26].

A Choreography Description Language does not depend on a specific business process implementation language. Thus, it can be used to specify truly interoperable, collaborations between any type of party regardless of the supporting platform or programming model used by the implementation of the hosting environment. Each party, adhering to a Choreography Description Language collaboration representation, could be implemented using completely different mechanisms 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

1.6 Time Assumptions

Clock synchronization is unspecified in the WS-CDL technical specification and is considered design-specific. In specific environments between involved parties, it can be assumed that all parties are reasonably well synchronized on second time boundaries. However, finer grained time synchronization within or across parties, or additional support or control are undefined and outside the scope of the WS-CDL specification.

2 Choreography Description Language Model

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

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

The WS-CDL model consists of the following entities:

- **Participant Types, Role Types and Relationship Types** - Within a Choreography, information is always exchanged between parties within or across trust boundaries. A Role Type enumerates the observable behavior a party exhibits in order to collaborate with other parties. A Relationship Type identifies the mutual commitments that must be made between two parties for them to collaborate successfully. A Participant Type is grouping together those parts of the observable behavior that must be implemented by the same logical entity or organization.

- **Information Types, Variables and Tokens** - Variables contain information about commonly observable objects in a collaboration, such as the information exchanged or the observable information 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 contains or the Token references.

- **Choreographies** - Choreographies define collaborations between interacting parties:
  - **Choreography Life-line** – The Choreography Life-line expresses the progression of a collaboration. Initially, the collaboration is established between parties, then work is performed within it and finally it completes either normally or abnormally.
  - **Choreography Exception Blocks** - An Exception Block specifies what additional interactions should occur when a Choreography behaves in an abnormal way.
  - **Choreography Finalizer Blocks** - A Finalizer Block describes how to specify additional interactions that should occur to modify the effect of an earlier successfully completed Choreography, for example to confirm or undo the effect.

- **Channels** - A Channel realizes a point of collaboration between parties by specifying where and how information is exchanged.

- **Work Units** - A Work Unit prescribes the constraints that must be fulfilled for making progress and thus performing actual work within a Choreography.

- **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 information within the Choreography is exchanged

- **Interaction Activity** - An Interaction is the basic building block of a Choreography, which results in an exchange of information between parties and possible synchronization of their observable information changes and the actual values of the exchanged information

- **Semantics** - Semantics allow the creation of descriptions that can record the semantic definitions of every component in the model

### 2.2 WS-CDL 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 type definitions inside.

#### 2.2.1 Choreography Package

A *Choreography Package* aggregates a set of WS-CDL type definitions, provides a namespace for the definitions and through the use of XInclude [27], MAY syntactically include WS-CDL type definitions that are defined in other Choreography Packages.

The syntax of the *package* construct is:

```xml
<package
    name="ncname"
    author="xsd:string"?
    version="xsd:string"?
    targetNamespace="uri"
    xmlns="http://www.w3.org/2004/12/ws-chor/cdl">
    informationType*
    token*
    tokenLocator*
    roleType*
    relationshipType*
    participantType*
    channelType*
    Choreography-Notation*
</package>
```

The Choreography Package contains:

- Zero or more Information Types
- Zero or more Tokens and Token Locators
Zero or more Role Types
Zero or more Relationship Types
Zero or more Participant Types
Zero or more Channel Types
Zero or more Package-level Choreographies

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

The targetNamespace attribute provides the namespace associated with all WS-CDL type definitions contained in this Choreography Package. WS-CDL type definitions included in this Package, using the inclusion mechanism, MAY be associated with other namespaces.

The elements informationType, token, tokenLocator, roleType, relationshipType, participantType and channelType MAY be used as elements by all the Choreographies defined within this Choreography Package.

### 2.2.2 Including WS-CDL Type Definitions

WS-CDL type definitions or fragments of WS-CDL type definitions can be syntactically reused in any WS-CDL type definition by using XInclude [27]. The assembly of large WS-CDL type definitions from multiple smaller, well-formed WS-CDL type definitions or WS-CDL type definitions fragments is enabled using this mechanism.

Inclusion of fragments of other WS-CDL type definitions SHOULD be done carefully in order to avoid duplicate definitions (Variables, blocks, etc.). A WS-CDL processor MUST ensure that the document is correct before processing it. The correctness may involve XML well-formedness as well as semantic checks, such as unicity of Variable definitions, of a single root Choreography, etc.

The example below shows some possible syntactic reuses of Choreography type definitions.

```xml
<choreography name="newChoreography" root="true">
  ...
  <variable name="newVariable" informationType="someType"
    role="randomRome"/>
  <xinclude href="genericVariableDefinitions.xml"/>
  <xinclude href="otherChoreography.xml"
    xpointer="xpointer(//choreography/variable[1])"/>
  ...
</choreography>
```
2.2.3 WS-CDL 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].

2.2.4 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.

2.2.5 Semantics

Within a WS-CDL document, descriptions allow the recording of semantic
definitions and other documentation. The OPTIONAL description sub-element is
allowed inside any WS-CDL language element. WS-CDL parsers are not
required to parse the contents of the description.
The information provided by the description sub-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 URI to a document that more
  fully describes the component
- **Machine Oriented Semantic Descriptions.** 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 [7] describes the functionality of a service provided by a party based on a stateless, client-server model. The emerging Web Based applications require the ability to exchange information 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 collaborations with the same party or with different parties. The Role Types, Participant Types, Relationship Types and Channel Types define the coupling of the collaborating parties.

2.3.1 Role Types

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

The syntax of the roleType construct is:

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

The attribute name is used for specifying a distinct name for each roleType element declared within a Choreography Package.

Within the roleType element, the behavior element specifies a subset of the observable behavior a party exhibits. A Role Type MUST contain one or more behavior elements. The attribute name within the behavior element is used for specifying a distinct name for each behavior element declared within a roleType element.

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

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

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

Although Relationship Types are always between two Role Types, Choreographies involving more than two Role Types 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” Relationship Types described above, the following Relationship Types might exist:

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

The syntax of the relationshipType construct is:

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

The attribute name is used for specifying a distinct name for each relationshipType element declared within a Choreography Package.

A relationshipType element MUST have exactly two Role Types defined. Each Role Type is specified by the type attribute within the role element. Within each role element, the OPTIONAL attribute behavior identifies the commitment of a party as an XML-Schema list of behavior types belonging to this Role Type. If the behavior attribute is missing then all the behaviors belonging to this Role Type are identified as the commitment of a party.
2.3.3 Participant Types

A Participant Type identifies a set of Role Types that MUST be implemented by the same logical entity or organization. Its purpose is to group together the parts of the observable behavior that MUST be implemented by the same logical entity or organization.

The syntax of the participantType construct is:

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

The attribute name is used for specifying a distinct name for each participantType element declared within a Choreography Package.

Within the participantType element, one or more role elements identify the Role Types that MUST be implemented by this Participant Type. Each Role Type is specified by the type attribute of the role element. A specific Role Type MUST NOT be specified in more than one participantType element.

An example is given below where the “SellerForBuyer” Role Type belonging to a “Buyer-Seller” Relationship Type is implemented by the Participant Type “Broker” which also implements the “SellerForShipper” Role Type belonging to a “Seller-Shipper” Relationship Type:

```
<roleType name="Buyer">
    ...
</roleType>
<roleType name="SellerForBuyer">
    <behavior name="sellerForBuyer" interface="rns:sellerForBuyerPT"/>
</roleType>
<roleType name="SellerForShipper">
    <behavior name="sellerForShipper" interface="rns:sellerForShipperPT"/>
</roleType>
<roleType name="Shipper">
    ...
</roleType>

<relationshipType name="Buyer-Seller">
    <role type="tns:Buyer" />
    <role type="tns:SellerForBuyer" />
</relationshipType>
<relationshipType name="Seller-Shipper">
    <role type="tns:SellerForShipper" />
    <role type="tns:Shipper" />
</relationshipType>

<participantType name="Broker">
    <role type="tns:SellerForBuyer" />
</participantType>
```
2.3.4 Channel Types

A Channel realizes a point of collaboration between parties by specifying where and how information is exchanged between collaborating parties. Additionally, Channel information can be passed among parties in information exchanges. The Channels exchanged MAY be used in subsequent Interaction activities. 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 Type MUST describe the Role Type and the reference type of a party, being the target of an information exchange, which is then used for determining where and how to send or receive information to or into the party.

A Channel Type MAY specify the instance identity of an entity implementing the behavior(s) of a party, being the target of an information exchange.

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

One or more Channel(s) MAY be passed around from one party to another in an information exchange. A Channel Type MAY be used to:

- Restrict the number of times a Channel of this Channel Type can be used
- Restrict the type of information exchange that can be performed when using a Channel of this Channel Type
- Restrict the Channel Type(s) that will be passed through a Channel of this Channel Type
- Enforce that a passed Channel is always distinct

The syntax of the channelType construct is:

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

  <passing channel="qname"
    action="request-respond"|"request"|"respond"?
    new="true"|"false"? />

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

<reference>
```
The attribute name is used for specifying a distinct name for each channelType element declared within a Choreography Package.

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

The OPTIONAL attribute action is used to restrict the type of information exchange that can be performed when using a Channel of this Channel Type.

The type of information exchange performed could either be a request-respond exchange, a request exchange, or a respond exchange. The default for this attribute is set to “request”.

The OPTIONAL element passing describes the Channel Type(s) of the Channel(s) that are passed, from one party to another, when using an information exchange on a Channel of this Channel Type. The OPTIONAL attribute action within the passing element defines if a Channel will be passed during a request exchange, during a response exchange or both. The default for this attribute is set to “request”. The OPTIONAL attribute new within the passing element when set to “true” enforces a passed Channel to be always distinct. If the element passing is missing then this Channel Type MAY be used for exchanging information but MUST NOT be used for passing Channels of any Channel Type.

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

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

The OPTIONAL element identity MAY be used for identifying an instance of an entity implementing the behavior of a party and for identifying a logical conversation between parties. The identity and the different conversations are distinguished by a set of Tokens as specified by the name attribute of the token element within the identity element.

The following rule applies for Channel Type:

- If two or more Channel Types SHOULD point to Role Types that MUST be implemented by the same logical entity or organization, then the specified Role Types MUST belong to the same Participant Type. In addition, the
The example below shows the definition of the Channel Type “RetailerChannel” that realizes a point of collaboration with a Retailer. The Channel Type identifies the Role Type of the Retailer as the “Retailer”. The information for locating the Retailer is specified in the reference element, whereas the instance of a process implementing the Retailer is identified for correlation purposes using the identity element. The element passing allows only a Channel of “ConsumerChannel” Type to be passed in a request information exchange through a Channel of “RetailerChannel” Type.

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

2.4 Information Driven Collaborations

Parties make progress within a collaboration when recordings of exchanged information are made, and changes to observable information occur, that then cause ordering constraints to be fulfilled. A WS-CDL document allows defining information within a Choreography that can influence the observable behavior of the collaborating parties.

Variables capture information about objects in the Choreography, such as the information exchanged or the observable information 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 type of information the Variable contains or the Token references.

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:

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

The attribute name is used for specifying a distinct name for each `informationType` element declared within a Choreography Package.

The OPTIONAL attributes type and element describe the type of information used within a Choreography as a WSDL 1.1 Message Type, an XML Schema type, a WSDL 2.0 Schema element or an XML Schema element. The type of information is exclusively one of the aforementioned.

When the OPTIONAL attribute exceptionType is set to "true", this Information Type is an Exception Type and MAY map to a WSDL fault type. When the attribute exceptionType is set to "false", this information type MUST NOT map to a WSDL fault type. The default for this attribute is set to "false".

In case of WSDL 2.0, the attribute element within the `informationType` refers to a unique WSDL 2.0 faultname when the attribute exceptionType is set to "true".

The examples below show some possible usages of the `informationType` construct.

**Example 1:** The `informationType` "purchaseOrder" refers to the WSDL 1.1 Message type "pns:purchaseOrderMessage"

```xml
<informationType name="purchaseOrder" type="pns:purchaseOrderMessage"/>
```

**Example 2:** The `informationType` "customerAddress" refers to the WSDL 2.0 Schema element "cns:CustomerAddress"

```xml
<informationType name="customerAddress" element="cns:CustomerAddress"/>
```

**Example 3:** The `informationType"intType"` refers to the XML Schema type "xsd:int"

```xml
<informationType name="intType" type="xsd:int"/>
```

**Example 4:** The `informationType"OutOfStockExceptionType"` is of type Exception Type and refers to the WSDL 2.0 fault name "cwns:OutOfStockExceptionType"

```xml
<informationType name="OutOfStockExceptionType" type="cwns:OutOfStockExceptionType" exceptionType="true"/>
```
2.4.2 Variables

Variables capture information about objects in a Choreography as defined by their usage:

- **Information Exchange Capturing Variables**, which 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 Capturing Variables**, which contain information about the observable changes at a Role as a result of information being exchanged. For example when a Buyer sends an “Order” to a Seller, the Buyer could have a Variable called “OrderState” set to a value of “OrderSent” and once the message was received by the Seller, the Seller could have a 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

- **Channel Capturing 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 (e.g. security), whether or not reliable messaging is to be used, etc.

The value of Variables:

- Are available to Roles within a Choreography, when the Variables contain information that is common knowledge. For example the Variable "OrderResponseTime" which is the time in hours in which a response to an Order must be sent is initialized prior to the initiation of a Choreography and can be used by all Roles within the Choreography

- Can be made available as a result of an Interaction
  - Information Exchange Capturing Variables are populated and become available at the Roles at the ends of an Interaction
  - State Capturing Variables, that contain information about the observable information changes of a Role as a result of information being exchanged, are recorded and become available

- Can be created or changed and made available locally at a Role by assigning data from other information. They can be Information Exchange, State or Channel Capturing Variables. 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

• Can be used to cause Exceptions at one or more parties in a Choreography

• Defined at different Roles that are part of the same Participant is shared between these Roles when the Variables have the same name

The variableDefinitions construct is used for defining one or more Variables within a Choreography.

The syntax of the variableDefinitions construct is:

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

A Variable defined using the attribute informationType specifies either Information Exchange Capturing Variables or State Capturing Variables. A Variable defined using the attribute informationType specifies Exception Capturing Variables when the referenced information type has the attribute exceptionType set to "true". A Variable defined using the attribute channelType specifies Channel Capturing Variables. The attributes informationType and channelType are mutually exclusive.

The OPTIONAL attribute mutable, when set to "false", specifies that the Variable information cannot change once initialized. The default value for this attribute is "true".

The OPTIONAL attribute silent, when set to "true" specifies that there SHOULD NOT be any activity used for creating or changing this Variable in the Choreography. A silent Variable is used to represent the result of actions within a party that are either not observable or are of no interest from the WS-CDL perspective. The default value for this attribute is "false".

The OPTIONAL attribute free, when set to "true" specifies that a Variable defined in an enclosing Choreography is also used in this Choreography, thus sharing the Variables information. The following rules apply in this case:

• The type (as specified by the informationType or the channelType attributes) of a free Variable MUST match the type of the Variable defined in an enclosing Choreography

• The attributes silent and mutable of a free Variable MUST match the attributes silent and mutable of the Variable defined in an enclosing Choreography
• A perform activity MUST bind a free Variable defined in an performed Choreography with a Variable defined in a performing Choreography.

The OPTIONAL attribute free, when set to "false" specifies that a Variable is defined in this Choreography.

The default value for the free attribute is “false”.

The OPTIONAL attribute roleTypes is used to specify an XML-Schema list of one or more Role Type(s) of a party at which the Variable information will reside. A Variable defined without a Role Type is equivalent to a Variable that is defined at all the Role Types that are part of the Relationship Types of the Choreography where the Variable is defined. For example if Choreography “C1” has Relationship Type “R” that has Roles “Role1”, “Role2”, then a Variable “var” defined in Choreography “C1” without a roleTypes attribute means it is defined at both “Role1” and “Role2”.

The attribute name is used for specifying a distinct name for each Variable declared within the variableDefinitions element. In those cases where the visibility of a Variable is wholly within a single Role then that Role needs to be named in the definition of the Variable as the Role Type using the attribute roleTypes. In those cases where the Variable is shared amongst a subset of Roles within a Choreography those Roles need to be listed within the definition of the Variable as the Role Types using the attribute roleTypes.

### 2.4.3 Expressions

Expressions can be used within WS-CDL to obtain existing information and to create new or change existing information.

Generic expressions and literals can be used for populating a Variable. Predicate expressions can be used within WS-CDL to specify conditions. Query expressions are used within WS-CDL to specify query strings.

The language used in WS-CDL for specifying expressions and query or conditional predicates is XPath 1.0.

WS-CDL defines XPath function extensions as described in the following section. The function extensions are defined in the standard WS-CDL namespace “http://www.w3.org/2004/12/ws-chor/cdl”. The prefix "cdl:" is associated with this namespace.

### 2.4.3.1 WS-CDL Supplied Functions

There are several functions that the WS-CDL specification supplies as XPATH 1.0 extension functions. These functions can be used in any XPath expression as long as the types are compatible:

```
xsd:time getCurrentTime(xsd:QName  roleName)
```
895 Returns the current time at the caller for the Role specified by \textit{roleName} (for 
exanple a Role can ask only about it’s own time).

898 \texttt{xsd:date getCurrentDate(xsd:QName \ \textit{rolenName})}

899 Returns the current date at the caller for the Role specified by \textit{roleName} (for 
exanple a Role can ask only about it’s own date).

902 \texttt{xsd:dateTime getCurrentDateTime(xsd:QName \ \textit{rolenName})}

903 Returns the current date and time at the caller for the Role specified by \textit{roleName} 
(for example a Role can ask only about it’s own date/time).

906 \texttt{xsd:boolean hasDurationPassed(xsd:duration \ \textit{elapsedTime}, xsd:QName 
\ \textit{rolenName})}

907 Returns “true” if, (a) used in a guard or repetition condition of a Work Unit with 
the block attribute set to “true” or in a complete condition of a Choreography and 
(b) the duration specified by \textit{elapsedTime} at the caller for the Role specified by 
\textit{rolenName} has elapsed from the time either the guard or the repetition condition 
were enabled for matching or the Choreography was enabled. Otherwise it 
returns “false”.

915 \texttt{xsd:boolean hasDeadlinePassed(xsd:dateTime \ \textit{deadlineTime}, xsd:QName 
\ \textit{rolenName})}

917 Returns "true" if, (a) used in a guard or repetition condition of a Work Unit with 
the block attribute set to "true" or in a complete condition of a Choreography and 
(b) the time specified by \textit{deadlineTime} at the Role specified by \textit{rolenName} has 
elapsed given that either the guard or the repetition condition were enabled for 
matching or the Choreography was enabled. Otherwise it returns “false”.

923 \texttt{xsd:string createNewID()}

924 Returns a new globally unique value of XML-Schema ‘string’ type.

926 \texttt{xsd:any getVariable(xsd:string \ \textit{varName}, xsd:string \ \textit{part}, xsd:string 
\ \textit{documentPath}, xsd:QName \ \textit{rolenName}?)}

928 Returns the information of the Variable with name \textit{varName} as a node set 
containing a single node. The second parameter, \textit{part}, specifies the message 
part of a WSDL1.1 document. For a WSDL 2.0 document it MUST be empty. 
When the third parameter \textit{documentPath} is empty, then this function retrieves the 
entire document from the Variable information. When it is non-empty, then this 
function retrieves from the Variable information, the fragment of the document at 
the provided absolute location path. The fourth parameter is OPTIONAL. When
the fourth parameter is used, the Variable information MUST be available at the Role specified by roleName. If this parameter is not used then the Role is inferred from the context that this function is used.

```xml
xsd:boolean isVariableAvailable(xsd:string varName, xsd:QName roleName)
```

Returns "true" if the information of the Variable with name varName is available at the Role specified by roleName. Returns "false" otherwise.

```xml
xsd:boolean variablesAligned(xsd:string varName, xsd:string withVarName, xsd:QName relationshipName)
```

Returns "true" if within a Relationship specified by relationshipName the Variable with name varName residing at the first Role of the Relationship has aligned its information with the Variable named withVarName residing at the second Role of the Relationship.

```xml
xsd:any getChannelReference(xsd:string varName)
```

Returns the reference information of the Variable with name varName. The Variable MUST be of Channel Type.

```xml
xsd:any getChannelIdentity(xsd:string varName)
```

Returns the identity information of the Variable with name varName. The Variable MUST be of Channel Type.

```xml
xsd:boolean globalizedTrigger(xsd:string expression, xsd:string roleName, xsd:string expression2, xsd:string roleName2, …)
```

Combines expressions that include Variables that are defined at different Roles. Only one expression MUST be defined per Role name.

```xml
xsd:boolean cdl:hasExceptionOccurred(xsd:string exceptionType)
```

Returns "true" if an Exception of Exception Type identified by the parameter exceptionType has occurred. Otherwise it returns "false".

### 2.4.4 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 define the piece of the data that is relevant.
All Tokens MUST have an informationType, for example, an “Order Id” could be ‘alphanumeric’ and a “counter” an ‘integer’.

Tokens 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. Instead the document part and the query string can change without affecting the Choreography definition.

The syntax of the token construct is:

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

The attribute name is used for specifying a distinct name for each token element declared within a Choreography Package.

The attribute informationType identifies the type of the document fragment.

The syntax of the tokenLocator construct is:

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

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

The attribute informationType identifies the type of the document on which the query is performed to locate the Token.

The OPTIONAL attribute part defines the document part on which the query is performed to locate the Token. This attribute SHOULD NOT be defined for a WSDL 2.0 document.

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

The example below shows that the Token “purchaseOrderID” is of XML-Schema type ‘int’. The two Token Locators show how to access this Token in the “purchaseOrder” and “purchaseOrderAck” messages.

```xml
<token name="purchaseOrderID" informationType="xsd:int"/>
```
2.4.5 Choreographies

A Choreography defines re-usable common rules, that govern the ordering of exchanged messages and the provisioning patterns of collaborative behavior, agreed between two or more interacting parties.

A Choreography defined at the Choreography Package level is called a top-level Choreography, and does not share its context with other top-level Choreographies. A Choreography Package MAY contain exactly one top-level Choreography, marked explicitly as the root Choreography. The root Choreography is the only top-level Choreography that is enabled by default.

The re-usable behavior encapsulated within a Choreography MAY be performed within an enclosing Choreography, thus facilitating composition. The performed Choreography is then called an enclosed Choreography and MAY be defined:

- **Locally** - its definition is contained within the enclosing Choreography
- **Globally** - a separate top-level, non-root Choreography definition is specified in the same or in a different Choreography Package that can be used by other Choreographies and hence the contract described becomes reusable

A non-root Choreography is enabled when performed.

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

A Choreography acts as a lexical name scoping context for Variables. A Variable defined in a Choreography is visible for use in this Choreography and all its enclosed Choreographies up-to the point that the Variable is re-defined as an non-free Variable, thus forming a Choreography Visibility Horizon for this Variable.

A Choreography MAY contain one or more Choreography definitions that MAY be performed only locally within this Choreography.

A Choreography MUST contain an Activity-Notation. The Activity-Notation specifies the actions of the Choreography that perform the actual work. These actions are enabled when the Choreography they belong to is enabled.

A Choreography can recover from exceptional conditions by defining one Exception Block, which MAY be defined as part of the Choreography to recover from exceptional conditions that can occur.
An enclosed Choreography that has successfully completed MAY need to provide finalization actions that confirm, cancel or otherwise modify the effects of its completed actions. To handle these modifications, one or more separate Finalizer Block(s) MAY be defined for an enclosed Choreography.

A Choreography can also be coordinated. Choreography Coordination guarantees that all involved Roles agree on how the Choreography ended. That is, if the Choreography completed successfully or suffered an Exception, and if the Choreography completed successfully and Finalizer Block(s) were installed, all Roles have the same Finalizer Block enabled.

The Choreography-Notation is used to define a Choreography as follows:

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

The attribute name is used for specifying a distinct name for each choreography element declared within a Choreography Package.

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 defined in an enclosing Choreography, and changed within an enclosed Choreography is available to its sibling Choreographies. The default for this attribute is set to "false". The following rules apply:

- When isolation is set to "false", the Variable information MAY be immediately overwritten by actions in its sibling Choreographies
• When isolation is set to "true", changes to the Variable information MUST be visible for read or for write to its sibling Choreographies only after this Choreography has completed.

The OPTIONAL coordination attribute specifies whether Choreography Coordination is required. The default for this attribute is set to "false". The following rules apply:

• When the coordination attribute is set to "true", Choreography Coordination is required and a Coordination protocol MUST ensure that all the Roles agree on how the Choreography ended.

• When the coordination attribute is set to "false", the Choreography is not bound to a Coordination protocol, and since none of the above guarantees of agreement on the outcome apply any required coordination SHOULD be performed using explicitly modeled Interactions.

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

The OPTIONAL variableDefinitions element enumerates the Variables defined in this Choreography.

The OPTIONAL root element marks a top-level Choreography as the root Choreography of a Choreography Package.

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

The OPTIONAL exceptionBlock element defines the Exception Block of a Choreography by specifying one or more Exception Work Unit(s) using a WorkUnit-Notation. Within this element, the attribute name is used for specifying a name for this Exception Block element.

The OPTIONAL finalizerBlock element defines a Finalizer Block for a Choreography. A Choreography MAY have more than one Finalizer Blocks. Each Finalizer Block specifies one Finalizer Work Unit using a WorkUnit-Notation. If a Choreography defines more than one Finalizer Blocks, then each MUST be differentiated by a distinct name as specified with the name attribute within the finalizerBlock element.

2.4.6 WorkUnits

A Work Unit can prescribe the constraints that have to be fulfilled for making progress and thus performing actual work within a Choreography. A Work Unit can also prescribe the constraints that preserve the consistency of the collaborations commonly performed between the parties. Using a Work Unit an application can recover from errors that are the result of abnormal actions and can also finalize successfully completed Choreographies that need further action, for example to confirm or logically roll back effects, or to close the Choreography.
so that any defined “rollback” Work Unit will not be enabled. Examples of a Work Unit include:

- 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
- 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 constraint that identifies the error

The guard condition of a Work Unit, if specified, expresses the interest on one or more Variable information (that already exist or will become available in the future) being available under certain prescribed constraints. The Work Unit’s expressed interest MUST be matched for its enclosed actions to be 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 used to define a Work Unit as follows:

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

The attribute name is used for specifying a name for each Work Unit element declared within a Choreography Package.

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

The OPTIONAL attribute guard specifies the guard condition of a Work Unit.

The OPTIONAL attribute repeat specifies the repetition condition of a Work Unit.

The OPTIONAL attribute block specifies whether the Work Unit has to block waiting for referenced Variables within the guard condition to become available (if they are not already) and the guard condition to evaluate to “true”. This attribute MUST always be set to “false” in Exception Work Units. The default for this attribute is set to “false”.

The following rules apply:
• When a guard condition is not specified then the Work Unit always matches.

• One or more Work Units MAY be matched concurrently if their respective expressed interests are matched.

• When a repetition condition is not specified then the Work Unit is not considered again for matching after the Work Unit was matched once.

• One or more Variables can be specified in a guard condition or repetition condition, using XPATH and the WS-CDL functions, as described in Section 2.4.3.1.

• The WS-CDL function `getVariable` is used in the guard or repetition condition to obtain the information of a Variable.

• When the WS-CDL function `isVariableAvailable` is used in the guard or repetition condition, it means that the Work Unit that specifies the guard or repetition condition is checking if a Variable is already available at a specific Role or is waiting for a Variable to become available at a specific Role, based on the block attribute being “false” or “true” respectively.

• When the WS-CDL function `variablesAligned` is used in the guard or repetition condition, it means that the Work Unit that specifies the guard or repetition condition is checking or waiting for an appropriate alignment Interaction to happen between the two Roles, based on the block attribute being “false” or “true” respectively. The Variables checked or waited for alignment are the sending and receiving ones in an alignment Interaction or the ones used in the recordings at the two Roles at the ends of an alignment Interaction. When the `variablesAligned` WS-CDL function is used in a guard or repetition condition, then the Relationship Type within the `variablesAligned` MUST be the subset of the Relationship Type that the immediate enclosing Choreography defines.

• Variables defined at different Roles MAY be used in a guard condition or repetition condition to form a globalized view, thus combining constraints prescribed for each Role but without requiring that all these constraints have to be fulfilled for progress to be made. The `globalizedTrigger` WS-CDL function MUST be used in a guard condition or repetition condition in this case. Variables defined at the same Role MAY be combined together in a guard condition or repetition condition using all available XPATH operators and all the WS-CDL functions.

• If the attribute block is set to “true” and one or more required Variable(s) are not available or the guard condition evaluates to "false", then the Work Unit MUST block. When the required Variable information specified by the guard condition become available and the guard condition evaluates to "true", then the Work Unit is matched. If the repetition condition is specified, then it is evaluated when the Work Unit completes successfully. Then, if the required Variable information specified by the repetition condition is available and the repetition condition evaluates to "true", the
Work Unit is considered again for matching. Otherwise, the Work Unit is not considered again for matching.

- If the attribute block is set to "false", then the guard condition or repetition 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 and the repetition condition is not evaluated even if specified. Otherwise, if the Work Unit matching succeeds, then the repetition condition, if specified, is evaluated when the Work Unit completes successfully. Then, if the required Variable information specified by the repetition condition is available and the repetition condition evaluates to "true", the Work Unit is considered again for matching. Otherwise, the Work Unit is not considered again for matching.

The examples below demonstrate some usages of a Work Unit:

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

In the following Work Unit, the guard condition 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" become available at the "Customer" Role.

```xml
<workunit name="POProcess"
    guard="cdl:isVariableAvailable(
      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 condition checks if the Variable "StockQuantity" at the "Retailer" Role is available and is greater than 10 and if so, the activity happens. If either the Variable is not available or its value is less than '10', then the matching condition is "false" and the activity is skipped.

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

c. Example of a Work Unit waiting for alignment to happen:
In the following Work Unit, the guard condition waits for an alignment Interaction to happen between the “Customer” Role and the “Retailer” Role:

```xml
<roleType name="Customer">
</roleType>

<roleType name="Retailer">
</roleType>

<relationshipType name="Customer-Retailer-Relationship">
  <role type="tns:Customer" />
  <role type="tns:Retailer" />
</relationshipType>

<workunit name="WaitForAlignment" guard="cdl:variablesAligned(
  "PurchaseOrderAtBuyer", "PurchaseOrderAtSeller",
  "tns:Customer-Retailer-Relationship")"
  block="true" >
  ...
</workunit>
```

### 2.4.7 Choreography Life-line

A Choreography life-line expresses the progression of a collaboration through enabled activities and enclosed Choreographies. Initially, the collaboration is established between parties, then work is performed within it and finally it ends.

A Choreography is initiated, establishing a collaboration when an Interaction, explicitly marked as an **Choreography Initiator**, is performed. This causes the Exception Block to be installed and the Choreography enters the **Enabled State**. Before this point there is no observable association between any of the parties.

Two or more Interactions MAY be marked as Choreography Initiators, indicating alternatives for establishing a collaboration. In this case, the first performed Interaction will establish the collaboration and the other Interactions will enlist with the already established collaboration.

A Choreography Initiator Interaction MAY be defined within a root Choreography or within an enclosed Choreography. In either case the collaboration is established when the first Choreography Initiator Interaction is performed.

A Choreography in an Enabled State, completes unsuccessfully, when an Exception is caused in the Choreography and its Exception Block is enabled, if present. This causes the Choreography to enter the **Unsuccessfully Completed State**.

The unsuccessfully completed Choreography, enters the **Closed State** once the Exception Block, if present, is completed. If the Exception Block is not present, the Choreography implicitly enters the Closed State and the Exception occurred is propagated to the enclosing Choreography.
A Choreography in an Enabled State, completes successfully when there are no more enabled activities within its body. This causes its Exception Block to be deinstalled, Finalizer Blocks to be installed if specified, and the Choreography enters the *Successfully Completed State*. Alternatively, a Choreography completes successfully if its complete condition, is matched by evaluating to "true". A complete condition is considered for matching while the Choreography is in Enabled State. The complete condition MUST be possible to be matched in all Roles that participate in the Choreography. When the complete condition of a Choreography is matched then all activities in the Choreography are disabled, and the Choreography completes as if there were no more enabled activities within it. When a Choreography completes, all uncompleted enclosed Choreographies will automatically become completed. Messages that were sent as part of a Choreography that has since completed MUST be ignored.

A Choreography, in a Successfully Completed State, enters the Closed State if no Finalizer Blocks were specified in that Choreography. A Choreography, in a Successfully Completed State with Finalizer Block(s) specified enters the Closed State when one of its installed Finalizer Block(s) is enabled and completed. The Finalizer Block of a Choreography is enabled by a finalize activity in the immediately enclosing Choreography. Alternatively, a Choreography in Successfully Completed State with Finalizer Block(s) specified implicitly enters the Closed State when its enclosing Choreography enters the Closed State without enabling the Finalizer Block(s) of its enclosed Choreography. In other words, when a Choreography enters the Closed State, all its enclosed successfully completed Choreographies are implicitly entering the Closed State even if none of their Finalizer Blocks has been enabled.

### 2.4.8 Choreography Exception Handling

A Choreography can sometimes fail as a result of an exceptional circumstance or an “error” that occurred during its performance. An *Exception* is caused in the Choreography when an Exception Variable is populated in an Interaction activity with the attribute `causeException` set to “true”. An Exception MUST be propagated to all parties in the Choreography using explicitly modeled, *Exception Causing Interactions* when the Choreography is not coordinated. This causes the Choreography to enter the Exception state and its Exception Block to be enabled, if specified.

Different types of errors 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

• 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 XML-Schema definition

• Application "failures", for example the “goods ordered” were ‘out of stock’

To handle these and other "errors" separate Exception Work Units MAY be defined in the Exception Block of a Choreography, for each Exception that needs to be handled.

One or more Exception Work Unit(s) MAY be defined within the Exception Block of a Choreography. At least one Exception Work Unit MUST be defined as part of the Exception Block of a Choreography. An Exception Work Unit MAY express interest on Exception information using its guard condition on Exception Types or Exception Variables. If no guard condition is specified, then the Exception Work Unit is called the Default Exception Work Unit and expresses interest on any type of Exception. Within the Exception Block of a Choreography there MUST NOT be more than one Default Exception Work Unit. An Exception Work Unit MUST always set its block attribute to “false” and MUST NOT define a repetition condition.

Exception Work Units are enabled when the Exception Block of the Choreography they belong to is enabled. Enabled Exception Work Units in a Choreography MAY behave as the mechanism to recover from Exceptions occurring in this and its enclosed Choreographies.

Within the Exception Block of a Choreography only one Exception Work Unit MAY be matched.

The rules for matching an Exception are:

• When an Exception Work Unit has a guard condition using the hasExceptionOccurred(exceptionType) WS-CDL function, then it is matched when an Exception Variable with Exception Type that matches the parameter exceptionType is populated using an Exception Causing Interaction activity

• If an Exception is matched by the guard condition of an Exception Work Unit, then the actions of the matched Work Unit are enabled. When two or more Exception Work Units are defined then the order of evaluating their guard conditions is based on the order that the Work Units have been defined within the Exception Block
• If none of the guard condition(s) match, then if there is a Default Exception Work Unit without a guard condition defined then its actions are enabled

• If an Exception is not matched by an Exception Work Unit defined within the Choreography in which the Exception occurs, the Exception will be recursively propagated to the Exception Work Unit of the immediate enclosing Choreography until a match is successful

• If an Exception occurs within a Choreography, then the Choreography completes unsuccessfully. In this case its Finalizer Block(s) MUST NOT be installed. The actions, including enclosed Choreographies, within this Choreography are completed abnormally before an Exception Work Unit can be matched

The actions within the Exception Work Unit MAY use Variable information visible in the Visibility Horizon of the Choreography it belongs to as they stand at the current time.

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

### 2.4.9 Choreography Finalization

After a Choreography instance has successfully completed, it MAY need to provide finalization actions that confirm, cancel or otherwise modify the effects of its completed actions. To handle these modifications, one or more separate Finalizer Block(s) MAY be defined for an enclosed Choreography. When its Choreography body completes successfully, any Finalizer Blocks specified in the Choreography are installed.

If more than one Finalizer Blocks are defined for the same Choreography, each of them MUST be differentiated by their name attributes. However, at most one Finalizer Block MAY be enabled for any given Choreography instance during the subsequent progress, including Exception handling and finalization, of the enclosing Choreography.

Finalizer Block(s) MAY implement whatever actions are appropriate for the particular Choreography. Common patterns might include:

- A single Finalizer Block to semantically "rollback" the Choreography
- Two Finalizer Blocks, for example one with name "confirm" and one with name "cancel", to implement a two-phase outcome protocol
- One "undo" Finalizer Block along with a "close" Finalizer Block to signal that the "undo" Finalizer Block is no longer able to be enabled, that is, the Choreography is now closed

The actions within the Finalizer Work Unit MAY use Variable information visible in the Visibility Horizon of the Choreography it belongs to as they were at the time the Choreography completed for the Variables belonging to this
Choreography and as they stand at the current time for the Variables belonging to the enclosing Choreography.

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

### 2.4.10 Choreography Coordination

Choreography Coordination guarantees that all involved Roles will agree on how the Choreography ended. That is, all Roles will agree on whether the Choreography completed successfully or suffered an Exception, and if the Choreography completed successfully and Finalizer Blocks were installed, all Roles will agree on which Finalizer Block was enabled. Such agreement differs from Interaction based alignment in that the Choreography as a whole is aligned, regardless of whether each Interaction in the Coordinated Choreography is aligned. In contrast to Alignment Interactions, a Coordinated Choreography provides a larger unit of coordination - a set of Interactions that end with shared knowledge among all the parties that their Relationship is in a defined state.

Such a unit need not be aligned at each step - it is only required that clear alignment points are made to guarantee that all involved Roles will agree on how the Choreography ended.

Choreographies defined as requiring coordination must be bound to a Coordination protocol. When Choreography Coordination is not required, then the Choreography is not bound to a Coordination protocol, and since none of the above guarantees of agreement on the outcome apply any required coordination should be performed using explicitly modeled Interactions.

The implications of Choreography Coordination differ for root Choreographies versus enclosed Choreographies:

- An enclosed Choreography MAY have one or more Finalizer Block(s). In this case, coordination means that all Roles agree on whether the Choreography completed successfully or suffered an Exception, and if the Choreography completed successfully and Finalizer Block(s) were installed, all Roles agree on which Finalizer Block was enabled.

- A root Choreography can also be coordinated, but it cannot have any Finalizer Block(s). In this case, coordination means that all Roles agree on whether the Choreography completed successfully or suffered an Exception.

- In both cases, all Roles MUST agree on whether the Choreography completed successfully, or if an Exception occurs, all Roles MUST experience an Exception rather than successful completion. When an Exception occurs within a Choreography, the Coordination protocol will throw an Exception to Roles which have not otherwise detected the Exception that occurred.
The two examples below show two usages of Coordinated Choreographies.

**Example 1: Coordinated credit authorization without Finalizer Block(s):**

```xml
<informationType name="creditDeniedType" exceptionType="true"/>

<!-- Coordinated CreditAuthorization choreography without Finalizer Block(s) -->
<choreography name="CreditAuthorization" root="false" coordination="true">
  <variableDefinitions>
    <variable name="CreditExtended" informationType="xsd:int" silent="true"
      roleTypes="tns:CreditResponder"/>
    <variable name="creditRequest"/>
    <variable name="creditAuthorized"/>
    <variable name="creditDenied" informationType = "creditDeniedType"/>
  </variableDefinitions>

  <!-- the normal work - receive the request and decide whether to approve -->
  <interaction name="creditAuthorization" channelVariable="tns:CreditRequestor"
    operation="authorize">
    <participate relationshipType="SuperiorInferior" fromRole="tns:Superior"
      toRole="Inferior"/>
    <exchange name="creditRequest" informationType="creditRequest"
      action="request">
      <send variable="tns:creditRequest"/>
      <receive variable="tns:creditRequest"/>
    </exchange>
    <exchange name="creditAuthorized" informationType="creditDenied"
      action="respond">
      <send variable="tns:creditAuthorized"/>
      <receive variable="tns:creditAuthorized"/>
    </exchange>
    <exchange name="creditDenied" informationType="refusal" action="respond">
      <send variable="tns:creditDenied" causeException="true"/>
      <receive variable="tns:creditDenied" causeException="true"/>
    </exchange>
  </interaction>

  <!-- catch the (application) exception - as an exception it will abort the choreography -->
  <exceptionBlock name="handleBadCreditException">
    <workunit name="handleBadCredit">
      <interaction name="badCreditInteraction"
        channelVariable="tns:CreditResponder"
        operation="creditDenied">
        <participate relationshipType="CreditReqCreditResp"
          fromRole="tns:Responder" toRole="CreditRequestor"/>
      </interaction>
    </workunit>
  </exceptionBlock>
</choreography>
```

**Example 2: Coordinated credit authorization with Finalizer Block(s):**

```xml
<informationType name="creditDeniedType" exceptionType="true"/>

<!-- Coordinated CreditAuthorization choreography with Finalizer Block(s) -->
<choreography name="CreditAuthorization" root="false" coordination="true">
```

---
<variableDefinitions>
  <variable name="CreditExtended" informationType="xsd:int" silent="true" roleTypes="tns:CreditResponder"/>
  <variable name="creditRequest"/>
  <variable name="creditAuthorized"/>
  <variable name="creditDenied" informationType = "creditDeniedType"/>
</variableDefinitions>

<!-- the normal work - receive the request and decide whether to approve -->
<interaction name="creditAuthorization" channelVariable="tns:CreditRequestor" operation="authorize">
  <participate relationshipType="SuperiorInferior" fromRole="tns:Superior" toRole="Inferior"/>
  <exchange name="creditRequest" informationType="creditRequest" action="request">
    <send variable="tns:creditRequest"/>
    <receive variable="tns:creditRequest"/>
  </exchange>
  <exchange name="creditAuthorized" informationType="creditDenied" action="respond">
    <send variable="tns:creditAuthorized"/>
    <receive variable="tns:creditAuthorized"/>
  </exchange>
  <exchange name="creditDenied" informationType="refusal" action="respond">
    <send variable="tns:creditDenied" causeException="true"/>
    <receive variable="tns:creditDenied" causeException="true"/>
  </exchange>
</interaction>

<!-- catch the (application) exception - as an exception it will abort the choreography and the Finalizer Block(s) are not accessible -->
<exceptionBlock name="handleBadCreditException">
  <workunit name="handleBadCredit">
    <interaction name="badCreditInteraction" channelVariable="tns:CreditResponder" operation="creditDenied">
      <participate relationshipType="CreditReqCreditResp" fromRole="tns:Responder" toRole="CreditRequestor"/>
      <exchange name="dummy" action="request">
        <send></send>
        <receive recordReference="drawdownRecord"/>
      </exchange>
      <record name="drawdownRecord" when="before"><source expression="drawnDown"/> <target variable="CreditExtended"/></record>
    </interaction>
  </workunit>
</exceptionBlock>

<!-- Finalizer Block(s) -->
<!-- what to do if the credit is drawn down -->
<finalizerBlock name="drawDown">
  <!-- if there is no application content to send, this could just be an assignment to the state capture variable creditExtended -->
  <workunit name="drawdown">
    <interaction name="drawdownInteraction" channelVariable="tns:CreditRequestor" operation="drawDown">
      <participate relationshipType="CreditReqCreditResp" fromRole="tns:CreditRequestor" toRole="CreditResponder"/>
      <exchange name="dummy" action="request">
        <send></send>
        <receive recordReference="drawdownRecord"/>
      </exchange>
      <record name="drawdownRecord" when="before"> <source expression="drawnDown"/> <target variable="CreditExtended"/></record>
    </interaction>
  </workunit>
</finalizerBlock>
Activities are the lowest level components of the Choreography, used to describe the actual work performed.

The Activity-Notation is used to define activities as either:

- An 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. A Basic Activity is then either:
  - An Interaction Activity, which results in an exchange of information between parties and possible synchronization of their observable information changes and the actual values of the exchanged information
  - A Perform Activity, which means that a complete, separately defined Choreography is performed
  - An Assign Activity, which assigns, within one Role, the value of one Variable to another Variable
  - A Silent Action Activity, which provides an explicit designator used for specifying the point where party specific action(s) with non-observable operational details MUST be performed
A No Action Activity, which provides an explicit designator used for specifying the point where a party does not perform any action.

A Finalize Activity, which enables a particular Finalizer Block in a particular instance of an immediately enclosed Choreography and thus brings that Choreography to a defined conclusion.

### 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 activities (as defined by one or more Activity-Notations) to be enabled sequentially, in the same order that they are defined.

The syntax of this construct is:

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

#### 2.5.1.2 Parallel

The parallel ordering structure contains one or more Activity-Notation that are enabled concurrently when the parallel activity is enabled. The parallel activity completes successfully when all activities (as defined by one or more Activity-Notations) performing work within it complete successfully.

The syntax of this construct is:

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

#### 2.5.1.3 Choice

The choice ordering structure enables specifying that only one of two or more activities (as defined by 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 guard conditions, 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 name=""processGoodCredit"
        channelVariable="goodCredit-channel" operation="doCredit">
        ...
    </interaction>
</choice>

<choice>
    <interaction name=""processBadCredit"
        channelVariable="badCredit-channel" operation="doBadCredit">
        ...
    </interaction>
</choice>
```

### 2.5.2 Interacting

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

An Interaction forms the base atom of the 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 one of the Roles participating in the Interaction sends a message, through a common Channel, to another Role that is participating in the Interaction, that receives the message. If the initial message is a request, then the accepting Role can optionally respond with a normal response message or a fault message, which will be received by the initiating Role.

An Interaction also contains "references" to:

- The Channel Capturing Variable that specifies the interface and other data that describe where and how the message is to be sent to and received into the accepting Role
• The *Operation* that specifies what the recipient of the message should do with the message when it is received

• The *From Role* and *To Role* that are involved

• The *Information Type or Channel Type* that is being exchanged

• The *Information Exchange Capturing Variables* at the From Role and To Role that are the source and destination for the message content

• A list of potential observable information changes that can occur and may need to be aligned at the From Role and the To Role, as a result of carrying out the Interaction

### 2.5.2.1 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 observable information creations or changes of one or more *State Capturing Variables* that are complementary to one or more *State Capturing 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 Capturing Variables at the partner Role.

For example, after an Interaction happens, both the Buyer and the Seller want to have a common understanding that:

• State Capturing Variables, such as "Order State", that contain observable information at the Buyer and Seller, have values that are complementary to each other, e.g. “Sent” at the Buyer and “Received” at the Seller, and

• Information Exchange Capturing Variables 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 observable information changes and the values of their exchanged information. 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 state, where one party sends a message and then it proceeds independently or the other party receives a message and then it proceeds independently.

### 2.5.2.2 Interaction Life-line

An Interaction completes normally when its message exchange(s) complete successfully.
An Interaction completes abnormally when:

- An application signals an error condition during the management of a request or within a party when processing the request
- The *time-to-complete* timeout, identifying the timeframe within which an Interaction MUST complete, occurs after the Interaction was initiated but before it completed
- Other types of errors, such as Protocol Based Exchange failures, Security failures, Document Validation errors, etc.

### 2.5.2.3 Interaction Syntax

The syntax of the *interaction* construct is:

```xml
<interaction name="ncname"
    channelVariable="qname"
    operation="ncname"
    align="true"|"false"?
    initiate="true"|"false"? >

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

  <exchange name="ncname"
      informationType="qname"|channelType="qname"?
      action="request"|"respond" >

     <send variable="XPath-expression"?
        recordReference="list of ncname"?
        causeException="true"|"false"? />

     <receive variable="XPath-expression"?
        recordReference="list of ncname"?
        causeException="true"|"false"? />

  </exchange>*

  <timeout time-to-complete="XPath-expression"
    fromRoleRecordReference="list of ncname"?
    toRoleRecordReference="list of ncname"? />

  <record name="ncname"
    when="before"|"after"|"timeout"
    causeException="true"|"false"? >

     <source variable="XPath-expression"? | expression="Xpath-expression"? />

     <target variable="XPath-expression" />

  </record>*

</interaction>
```

The attribute *name* is used for specifying a name for each Interaction element declared within a Choreography.

The *channelVariable* attribute specifies the Channel Variable containing information of a party, being the target of the Interaction, which is used for determining where and how to send and receive information to and 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.
This requires that the messages exchanged in the Choreography also contain reference and correlation information, for example by:

- Including a protocol header, such as a SOAP header 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

The operation attribute specifies the name of the operation that is associated with this Interaction. The specified operation belongs to the interface, as identified by the role and behavior elements of the Channel Type of the Channel Variable used in this Interaction.

The OPTIONAL align attribute when set to "true" means that this Alignment Interaction results in the common understanding of both the information exchanged and the resulting observable information 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 initiate attribute is set to "true". The default for this attribute is "false".

Within the participate element, the relationshipType attribute specifies the Relationship Type this Interaction participates in and the fromRole and toRole attributes specify the requesting and the accepting Role Types respectively. The Role Type identified by the toRole attribute MUST be the same as the Role Type identified by the role element of the Channel Type of the Channel Variable used in the interaction activity.

The OPTIONAL exchange element allows information to be exchanged during an Interaction. The attribute name is used for specifying a name for this exchange element.

Within the exchange element, the OPTIONAL attributes informationType and channelType identify the Information Type or the Channel Type of the information that is exchanged between the two Roles in an Interaction. The attributes informationType and channelType are mutually exclusive. If none of these attributes are specified, then it is assumed that either no actual information is exchanged or the type of information being exchanged is of no interest to the Choreography definition.

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

- When the action attribute is set to "request", then the information exchange happens fromRole to toRole
- When the action attribute is set to "respond", then the information 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 send and the receive elements MUST only use the WS-CDL function getVariable within the variable attribute.
- The OPTIONAL Variables specified within the send and receive elements MUST be of type as described in the informationType or channelType attributes.
- 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 Variable specified within the receive element MUST not be defined with the attribute silent set to “true”.
- Within the send or the receive element(s) of an exchange element, the recordReference attribute contains an XML-Schema list of references to record element(s) in the same Interaction. The same record element MAY be referenced from different send or the receive element(s) within the same Interaction thus enabling re-use.
- Within the send or the receive element(s) of an exchange element, the causeException attribute when set to “true”, specifies that an Exception MAY be caused at the respective Roles. In this case, the informationType of the exchange element MUST be of Exception Type. The default for this attribute is "false".
- The request exchange MUST NOT have causeException attribute set to “true”.
- When two or more respond exchanges are specified, one respond exchange MAY be of normal informationType and all others MUST be of Exception Type. There is an implicit choice between two or more respond exchanges.
- If the align attribute is set to "false" for the Interaction, then it means that the:
  - Request exchange completes successfully for the requesting Role once it has successfully sent the information of the Variable specified within the send element and the Request exchange completes successfully for the accepting Role once it has successfully received the information of the Variable specified within the receive element.
Response exchange completes successfully for the accepting Role once it has successfully sent the information of the Variable specified within the send element and the Response exchange completes successfully for the requesting Role once it has successfully received the information of the Variable specified within the receive element.

- If the align attribute is set to "true" for the Interaction, then it means that the Interaction completes successfully if its Request and Response exchanges complete successfully and all referenced records complete successfully:
  - A Request exchange completes successfully once both the requesting Role has successfully sent the information of the Variable specified within the send element and the accepting Role has successfully received the information of the Variable specified within the receive element.
  - A Response exchange completes successfully once both the accepting Role has successfully sent the information of the Variable specified within the send element and the requesting Role has successfully received the information of the Variable specified within the receive element.

Within the OPTIONAL timeout element, the time-to-complete attribute identifies the timeframe within which an Interaction MUST complete after it was initiated or the deadline before an Interaction MUST complete. The time-to-complete SHOULD be of XML-Schema duration type when conveying the timeframe and SHOULD be of XML-Schema dateTime type when conveying the deadline. The OPTIONAL fromRoleRecordReference attribute contains an XML-Schema list of references to record element(s) in the same Interaction that will take effect at the fromRole when a timeout occurs. The OPTIONAL toRoleRecordReference attribute contains an XML-Schema list of references to record element(s) in the same Interaction that will take effect at the toRole when a timeout occurs.

The OPTIONAL element record is used to create or change and then make available within one Role, the value of one or more Variables using another Variable or an expression. The attribute name is used for specifying a distinct name for a record element within an Interaction. Within the record element, the source and target elements specify these recordings of information happening at the send and receive ends of the Interaction:

- When the action element is set to "request", then the recording(s) specified within the source and the target elements occur at the fromRole for the send and at the toRole for the receive.
- When the action element is set to "response", then the recording(s) specified within the source and the target elements occur at the toRole for the send and at the fromRole for the receive.
Within the record element, the when attribute specifies if a recording happens before or after a send or “before” or “after” a receive of a message at a Role in a Request or a Response exchange or when a timeout has expired. When the when attribute is set to “timeout”, the record element specifies the recording to be performed when a timeout occurs. If two or more record elements have the same value in their when attribute and are referenced within the recordReference attribute of a send or a receive element, then they are performed in the order in which they are specified.

The following rules apply for the information recordings when using the record element:

- The source MUST define either a variable attribute or an expression attribute:
  - When the source defines an expression attribute, it MUST contain expressions, as defined in Section 2.4.3. The resulting type of the defined expression MUST be compatible with the target Variable type
  - When the source defines a Variable, then the source and the target Variable MUST be of compatible type
  - When the source defines a Variable, then the source and the target Variable MUST be defined at the same Role
- When the attribute variable is defined it MUST use only the WS-CDL function getVariable
- The target Variable MUST NOT be defined with the attribute silent set to “true”
- One or more record elements MAY be specified and performed at one or both the Roles within an Interaction
- A record element MUST NOT be specified in the absence of an exchange element or a timeout element that reference it
- The attribute causeException MAY be set to "true" in a record element if the target Variable is an Exception Variable
- When the attribute causeException is set to "true" in a record element, the corresponding Role gets into the Exception state
- When two or more record elements are specified for the same Role in an Interaction with target Variables of Exception Type, one of the Exception recordings MAY occur. An Exception recording has a non-observable predicate condition, associated implicitly with it, that decides if an Exception occurs
- If the align attribute is set to "false" for the Interaction, then it means that the Role specified within the record element makes available the creation or change of the information specified within the record element immediately after the successful completion of each record
- If the align attribute is set to "true" for the Interaction, then it means that
Both Roles know the availability of the creation or change of the information specified within the record element only at the successful completion of the Interaction.

If there are two or more record elements specified within an Interaction, then all record operations MUST complete successfully for the Interact to complete successfully. Otherwise, none of the Variables specified in the target attribute will be affected.

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

- The message “purchaseOrder” is sent from the “Consumer” to the “Retailer” as a request message.
- The message “purchaseOrderAck” is sent from the “Retailer” to the “Consumer” as a response message.
- The Variable “consumer-channel” is made available at the “Retailer” using the record element.
- The Interaction happens on the “retailer-channel”, which has a Token “purchaseOrderID” used within the identity element 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” Interaction from the “Consumer” to the “Retailer” on “retailer-channel” during the request. Here the record element makes available the “Consumer-channel” at the “Retailer” Role. If the align attribute was set to "true" for this Interaction, then it also means that the “Consumer” knows that the “Retailer” now has the contact information of the “Consumer”. In another example, the “Consumer” could set its Variable "OrderSent" to "true" and the “Retailer” would set its Variable "OrderReceived" to "true" using the record element.
- The exchange “badPurchaseOrderAckException” specifies that an Exception of “badPOAckType” Exception Type could occur at both parties.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<package xmlns="http://www.w3.org/2004/12/ws-chor/cdl"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.oracle.com/ashwini/sample"/>
```
<informationType name="purchaseOrderType" type="tns:PurchaseOrderMsg"/>
<informationType name="purchaseOrderAckType" type="tns:PurchaseOrderAckMsg"/>
<informationType name="badPOAckType" type="xsd:string" exceptionType="true"/>
	<token name="purchaseOrderID" informationType="tns:intType"/>
	<tokenLocator tokenName="tns:purchaseOrderID" informationType="tns:purchaseOrderType" query="/PO/orderId"/>
	<tokenLocator tokenName="tns:purchaseOrderID" informationType="tns:purchaseOrderAckType" query="/PO/orderId"/>
	<roleType name="Consumer">
		<behavior name="consumerForRetailer" interface="tns:ConsumerRetailerPT"/>
		<behavior name="consumerForWarehouse" interface="tns:ConsumerWarehousePT"/>
	</roleType>
	<roleType name="Retailer">
		<behavior name="retailerForConsumer" interface="tns:RetailerConsumerPT"/>
	</roleType>
	<relationshipType name="ConsumerRetailerRelationship">
		<role type="tns:Consumer" behavior="consumerForRetailer"/>
		<role type="tns:Retailer" behavior="retailerForConsumer"/>
	</relationshipType>
	<channelType name="ConsumerChannel">
		<role type="tns:Consumer"/>
		<reference>
			<token name="tns:consumerRef"/>
		</reference>
		<identity>
			<token name="tns:purchaseOrderID"/>
		</identity>
	</channelType>
	<channelType name="RetailerChannel">
		<role type="tns:Retailer" behavior="retailerForConsumer"/>
		<reference>
			<token name="tns:retailerRef"/>
		</reference>
		<identity>
			<token name="tns:purchaseOrderID"/>
		</identity>
	</channelType>
</choreography>

<choreography name="ConsumerRetailerChoreography" root="true">

</choreography>
2.5.3 Composing Choreographies

The perform activity realizes the “composition of Choreographies”, whereas combining existing Choreographies results in the creation of new Choreographies. 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” Role to which the “Supplier” Role responds with either a “Quotation” or a “Decline to Quote” message, and

- An “Order Placement” Choreography, where the “Buyer” Role places and order for goods or services and the “Supplier” Role either accepts the order or rejects it

One could then create a new "Quote and Order" Choreography by reusing the two, where the “RFQ” Choreography was performed first, and then, depending on the outcome of the “RFQ” Choreography, the order is placed using the “Order Placement” Choreography. In this case the new Choreography is "composed" out of the two previously defined Choreographies. Using this approach, Choreographies can be combined to support Choreographies of any required complexity, allowing more flexibility as Choreographies defined elsewhere can be reused.
The perform activity enables a Choreography to specify that another Choreography is performed at this point in its definition, as an enclosed Choreography. The performed Choreography, even when defined in a different Choreography Package, is conceptually treated as an enclosed Choreography.

The syntax of the `perform` construct is:

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

Within the `perform` element, the `choreographyName` attribute references the name of the Choreography to be performed.

The `choreographyInstanceId` attribute defines an identifier for this performance of the Choreography identified by the `choreographyName` attribute. If the performed Choreography can only be performed once within the enclosing Choreography, the `choreographyInstanceId` attribute is OPTIONAL. Otherwise it MUST be specified and the value MUST be different for each performance. This is a dynamic requirement. For example, if a single `perform` element appears in a Work Unit that can repeat, each use of `perform` must assign a different `ChoreographyInstanceId` identifier.

The `Choreography-Notation` within the `perform` element defines a Locally defined Choreography that is performed only by this perform activity. If specified, the `choreographyName` attribute within the `perform` element MUST match the attribute name within the choreography element of the `Choreography-Notation`.

The `bind` element within the `perform` element enables information in the performing Choreography to be shared with the performed Choreography and vice versa. Within the `bind` element, the attribute name is used for specifying a name for each `bind` element declared within this perform activity. Within the `bind` element, the `role` attribute aliases the Roles from the performing Choreography to the performed Choreography.

The `variable` attribute within this element specifies that a Variable in the performing Choreography is bound with the Variable identified by the `variable` attribute within the `free` element in the performed Choreography.

The following rules apply:

- The Choreography to be performed MUST be either a Locally defined Choreography that is immediately contained within the performing Choreography or a Globally defined Choreography. Performed
Choreographies that are declared in a different Choreography Package MUST be included first before they can be performed.

- The Role Types within a single bind element MUST be carried out by the same party, hence they MUST belong to the same Participant Type.

- The variable attribute within this element and free element MUST define only the WS-CDL function getVariable.

- The free Variables specified within the free element MUST have the attribute free set to "true" in their definition within the performed Choreography.

- There MUST not be a cyclic dependency on the Choreographies performed. For example, Choreography “C1” is performing Choreography “C2” which is performing Choreography “C1” again is disallowed.

The example below shows a Choreography composition, where a Choreography "PurchaseChoreography" is performing the Globally defined Choreography "RetailerWarehouseChoreography" and aliases the Variable "purchaseOrderAtRetailer" to the Variable "purchaseOrder" defined at the performed Choreography "RetailerWarehouseChoreography". Once aliased, the Variable “purchaseOrderAtRetailer” extends to the enclosed Choreography and thus these Variables can be used interchangeably for sharing their information.

```xml
<choreography name="PurchaseChoreography">
  ...
  <variableDefinitions>
    <variable name="purchaseOrderAtRetailer" informationType="purchaseOrder" role="tns:Retailer"/>
  </variableDefinitions>
  ...
  <perform choreographyName="RetailerWarehouseChoreography">
    <bind name="aliasRetailer">
      <this variable="cdl:getVariable("tns:purchaseOrderAtRetailer", "", ")"
            role="tns:Retailer"/>
      <free variable="cdl:getVariable("tns:purchaseOrder", "", ")"
             role="tns:Retailer"/>
    </bind>
  </perform>
  ...
</choreography>

<choreography name="RetailerWarehouseChoreography">
  <variableDefinitions>
    <variable name="purchaseOrder" informationType="purchaseOrder" role="tns:Retailer" free="true"/>
  </variableDefinitions>
  ...
</choreography>
```
2.5.4 Assigning Variables

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

The assign activity MAY also be used to cause an Exception at a Role.

The syntax of the assign construct is:

```xml
<assign roleType="qname">
  <copy name="ncname" causeException="true"|"false"? >
    <source variable="XPath-expression"|expression="Xpath-expression"? />
    <target variable="XPath-expression" />
  </copy>+</assign>
```

The copy element within the assign element creates or changes, at the Role specified by the roleType attribute, the Variable defined by the target element using the Variable or expression defined by the source element at the same Role. Within the copy element, the attribute name is used for specifying a name for each copy element declared within this assign activity.

The following rules apply to assignment:

- The source MUST define either a variable attribute or an expression attribute:
  - When the source defines an expression attribute, it MUST contain expressions, as defined in Section 2.4.3. The resulting type of the defined expression MUST be compatible with the target Variable type
  - When the source defines a Variable, then the source and the target Variable MUST be of compatible type
  - When the source defines a Variable, then the source and the target Variable MUST be defined at the same Role

- When the attribute variable is defined it MUST use only the WS-CDL function getVariable
- The target Variable MUST NOT be defined with the attribute silent set to “true”
- When two or more copy elements belong to the same assign element, then they are performed in the order in which they are defined
- If there are two or more copy elements specified within an assign, then all copy operations MUST complete successfully for the assign to complete successfully. Otherwise, none of the Variables specified in the target attribute will be affected
• The OPTIONAL attribute causeException MAY be set to "true" in a copy element if the target Variable is an Exception Variable. The default for this attribute is "false"

• At most one copy element MAY have the attribute causeException set to "true"

• When the attribute causeException is set to "true" in a copy element, the Role specified by the attribute roleType gets into the Exception state after the assign activity has completed

The examples below show some possible usages of assign.

Example 1:
```xml
<assign roleType="tns:Retailer">
  <copy name="copyAddressInfo">
    <source variable="cdl:getVariable("PurchaseOrderMsg", ",", "/PO/CustomerAddress")" />
    <target variable="cdl:getVariable("CustomerAddress", ",", ",")" />
  </copy>
</assign>
```

Example 2:
```xml
<assign roleType="tns:Retailer">
  <copy name="copyPriceInfo">
    <source expression="(10+237)/34" />
    <target variable="cdl:getVariable("ProductPrice", ",", ",", "tns:Retailer")" />
  </copy>
</assign>
```

Example 3:
```xml
<assign roleType="tns:Customer">
  <copy name="copyLiteral">
    <source expression="Hello World" />
    <target variable="cdl:getVariable("VarName", ",", ",", "tns:Customer")" />
  </copy>
</assign>
```

2.5.5 Marking Silent Actions

The Silent Action activity is an explicit designator used for marking the point where party specific actions with non-observable operational details MUST be performed. For example, the mechanism for checking the inventory of a warehouse should not be observable to other parties, but the fact that the inventory level does influence the global observable behavior with a buyer party needs to be specified in the Choreography definition.
The syntax of the *silent action* construct is:

```xml
<silentAction roleType="qname? />
```

The OPTIONAL attribute `roleType` is used to specify the party at which the silent action will be performed. If a silent action is defined without a Role Type, it is implied that the action is performed at all the Role Types that are part of the Relationships of the Choreography this activity is enclosed within.

### 2.5.6 Marking the Absence of Actions

The *No Action* activity is an explicit designator used for marking the point where a party does not perform any action.

The syntax of the *no action* construct is:

```xml
<noAction roleType="qname? />
```

The OPTIONAL attribute `roleType` is used to specify the party at which no action will be performed. If a `noAction` is defined without a Role Type, it is implied that no action will be performed at any of the Role Types that are part of the Relationships of the Choreography this activity is enclosed within.

### 2.5.7 Finalizing a Choreography

The *finalize* activity is used to enable a specific Finalizer Block in successfully completed instances of immediately enclosed Choreographies, and thus bring those Choreographies to defined conclusions.

A Choreography that does not perform any Choreographies that have Finalizer Block(s) defined MUST NOT have any finalize activities specified within it. A finalize activity MAY be present within a Choreography that has performed a Choreography with one or more defined Finalizer Block(s) - that is a finalize activity can be specified within the Choreography body, within an Exception Block and within Finalizer Blocks.

For a single performed Choreography instance, at most one of its Finalizer Block(s) SHOULD be enabled by a finalize activity during the subsequent progress, including Exception handling and finalization, of the enclosing Choreography.
The syntax of the `finalize` construct is:

```
<finalize name="ncname"? >
  <finalizerReference
    choreographyName="ncname"
    choreographyInstanceId="XPath-expression"?
    finalizerName="ncname"? />
</finalizerReference>+
</finalize>
```

The OPTIONAL attribute `name` is used for specifying a distinct name for each finalize element declared within a Choreography Package.

Each `finalizerReference` element enables a Finalizer Block in a performed instance of an immediately enclosed Choreography. Within a finalize element, each `finalizerReference` MUST refer to a different performed Choreography instance.

Within the `finalizerReference` element, the `choreographyName` attribute identifies the Choreography referenced by the `choreographyName` attribute of the `perform` construct.

Within the `finalizerReference` element, the OPTIONAL `choreographyInstanceId` attribute identifies the performed Choreography instance to be finalized, using the value defined by the `choreographyInstanceId` attribute of the `perform` construct. The `choreographyInstanceId` attribute MAY be omitted if the contract logic of the performing Choreography is such that only one instance of the Choreography identified by the `choreographyName` attribute could have been performed when the finalize activity is enabled. If more than one instance of the Choreography identified by the `choreographyName` attribute could have been performed, the `choreographyInstanceId` attribute MUST be present.

Within the `finalizerReference` element, the attribute `finalizerName` indicates which Finalizer Block is to be enabled in the performed instance. If the targeted, immediately enclosed, Choreography has only one defined Finalizer Block, then the `finalizerName` attribute is OPTIONAL.

In the example below, Choreography “CreditDecider” gets credit authorizations for two bidders, “A” and “B”, at most one of which can be selected. The “CreditDecider” performs a “CoordinatedCreditAuthorization” Choreography for each bidder, and then finalizes each performed Choreography depending on whether “A”, ”B” or neither was selected.

```
<choreography name="CreditDecider">
  <!-- only a snippet is shown here -->
  <parallel>
    <perform name="creditForA"
      choreographyName="CoordinatedCreditAuthorization"
    >
    <perform name="creditForB"
      choreographyName="CoordinatedCreditAuthorization"
    >
    <perform name="creditForC"
      choreographyName="CoordinatedCreditAuthorization"
    >
  </parallel>
</choreography>
```
<perform>
  <perform name="creditForB"
    choreographyName="CoordinatedCreditAuthorization"
    choreographyInstanceId="creditForB">
    <!-- bind such that this does the business for A -->
  </perform>
</perform>
</parallel>

3 Example

To be completed
4 Relationship with the Security framework

The WS-Security specification [24] provides enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication, including a general-purpose mechanism for associating security tokens with messages, and a description of how to encode binary security tokens. As messages can have consequences in the real world, collaboration parties will impose security requirements on their information exchanges. WS-Security can be used to satisfy many of these requirements.

5 Relationship with the Reliable Messaging framework

The WS-Reliability specification [22] provides a reliable mechanism to exchange information among collaborating parties. The WS-Reliability specification prescribes the formats for all information exchanged without placing any restrictions on the content of the encapsulated business documents. The WS-Reliability specification supports 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”, which MAY be reflected in the Choreography with an Exception.

6 Relationship with the Coordination framework

In WS-CDL, Alignment Interactions and Coordinated Choreographies require support from a Coordination protocol, where agreement on the outcome among parties can be reached even in the case of failures and loss of messages. In this case, the Alignment Interactions and the Coordinated Choreographies MUST be bound to a Coordination protocol.

7 Relationship with the Addressing framework

The WS-Addressing specification [28] provides transport-neutral mechanisms to address Web services and messages, specifically, XML [9, 10] elements to identify Web service endpoints and to secure end-to-end endpoint identification in messages. WS-Addressing enables messaging systems to support message transmission through networks that include processing nodes such as endpoint...
managers, firewalls, and gateways in a transport-neutral manner.

WS-Addressing can be used to convey the reference and correlation information for normalizing expanded Channel Variable information into an uniform format that can be processed independently of transport or application.

The WS-Addressing specification is in progress and the WS-Choreography Working Group will review and comment on developments in this effort on an ongoing basis.

8 Conformance

To be completed

9 Acknowledgments

This document has been produced by the members of the Web Services Choreography Working Group. The chairs of this Working Group are Martin Chapman (Oracle Corporation) and Steve Ross-Talbot (Enigmatec Corporation).

The editors would like to thank the Working Group members for their contributions. Members of the Working Group are (at the time of writing):

Abbie Barbir (Nortel Networks), Charlton Barreto (webMethods, Inc.), Carine Bournez (W3C), Gary Brown (Enigmatec Corporation), Anthony Fletcher (Choreology Ltd), Peter Furniss (Choreology Ltd), Jim Hendler (University of Maryland (Mind Lab)), Kohei Honda (Queen Mary and Westerfield College), Nickolas Kavantzas (Oracle Corporation), Yutaka Kudou (Hitachi, Ltd.), Yves Lafon (W3C), Monica Martin (Sun Microsystems, Inc.), Robin Milner (Cambridge University), Jeff Mischkinsky (Oracle Corporation), Bijan Parsia (University of Maryland (Mind Lab)), Greg Ritzinger (Novell), Yoko Seki (Hitachi, Ltd.), Prasad Yendluri (webMethods, Inc.), Nobuko Yoshida (Imperial College London).

Previous members of the Working Group were: Assaf Arkin (Intalio Inc.), Daniel Austin (Sun Microsystems, Inc.), Alistair Barros (DSTC Pty Ltd (CITC)), Richard Bonneau (IONA), Allen Brown (Microsoft Corporation), Mike Brumbelow (Apple), David Burdett (Commerce One), Ravi Byakod (Intalio Inc.), Michael Champion (Software AG), David Chapell (Sonic Software), Ugo Corda (SeeBeyond Technology Corporation), Fred Cummins (EDS), Jon Dart (TIBCO Software), Jean-Jacques Dubray (Attachmate), William Eidson (TIBCO Software), Colleen Evans (Sonic Software), Keith Evans (Hewlett-Packard), Yaron Goland (BEA Systems), Leonard Greski (W. W. Grainger, Inc.), Ricky Ho (Cisco Systems Inc.), Andre Huertas (Uniform Code Council), Duncan Johnston-Watt (Enigmatec Corporation), Eunju Kim (National Computerization Agency), Mayilraj Krishnan (Cisco Systems Inc.), Melanie Kudela (Uniform Code Council), Bruno Kurtic.
(webMethods, Inc.), Paul Lipton (Computer Associates), Kevin Liu (SAP AG),
Francis McCabe (Fujitsu Ltd.), Carol McDonald (Sun Microsystems, Inc.), Greg
Meredith (Microsoft Corporation), Eric Newcomer (IONA), Sanjay Patil (IONA),
Ed Peters (webMethods, Inc.), Steve Pruitt (Novell), Dinesh Shahane (TIBCO
Software), Evren Sirin (University of Maryland (Mind Lab)), Ivana Trickovic (SAP
AG), William Vambenepe (Hewlett-Packard), Jim Webber (Arjuna Technologies
Ltd.), Stuart Wheater (Arjuna Technologies Ltd.), Steven White (SeeBeyond
Technology Corporation), Hadrian Zbarcea (IONA).

10 References

University, March 1997
20000508/"
[7] Web Services Definition Language (WSDL) 2.0
[8] OASIS Committee Draft "Universal Description, Discovery and Integration" version 3.0.2
[10] XML-Namespace " Namespaces in XML, Tim Bray et al., eds., W3C, January 1999"
http://www.w3.org/TR/REC-xml-names
[15] WSCI: Web Services Choreography Interface 1.0, A. Arkin et.al
[18] OASIS Working Draft "WS-BPEL: Business Process Execution Language 2.0". This is work
in progress.
Marin, R. Norin R. Shapiro
Framework 1.0”. This is work in progress.
[22] OASIS Working Draft “Web Services Reliability 1.0”. This is work in progress.
[27] ”XML Inclusions Version 1.0” http://www.w3.org/TR/xinclude/
[28] Web Services Addressing (WS-Addressing) - W3C Member Submission
10 August 2004
11 Last Call Issues

11.1 Issue 1

Due to a lack of clarity in existing XML specifications, the WS-Choreography Working Group is unable at this time to recommend an approach for accessing and modifying members of lists and arrays.

11.2 Issue 2

The WS-Choreography Working Group is working on a proposal for extending Choreographies (that is specifying a choreography by defining how it is based on another choreography). This work is not finalized as of yet, but we do not believe it will have a major impact on the architecture.
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:cdl="http://www.w3.org/2004/12/ws-chor/cdl"
targetNamespace="http://www.w3.org/2004/12/ws-chor/cdl"
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>
        <attribute name="type" type="cdl:tDescriptionType" use="optional"
                    default="documentation"/>
      </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="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="roleType" type="cdl:tRoleType" minOccurs="0"
                 maxOccurs="un bounded"/>
        <element name="relationshipType" type="cdl:tRelationshipType"
                 minOccurs="0" maxOccurs="un bounded"/>
        <element name="participantType" type="cdl:tParticipantType"
                 minOccurs="0" maxOccurs="un bounded"/>
        <element name="channelType" type="cdl:tChannelType"
                 minOccurs="0" maxOccurs="un bounded"/>
        <element name="choreography" type="cdl:tChoreography"
                 minOccurs="0" maxOccurs="un bounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="author" type="string" use="optional"/>
      <attribute name="version" type="string" use="optional"/>
      <attribute name="targetNamespace" 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"/>
      <attribute name="exceptionType" type="boolean" use="optional" default="false"/>
    </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="part" type="NCName" use="optional"/>
      <attribute name="query" type="cdl:tXPath-expr" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tRoleType">
  <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="tRelationshipType">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <sequence>
      </sequence>
    </extension>
  </complexContent>
</complexType>
<element name="role" type="cdl:tRoleRef" minOccurs="2" maxOccurs="2"/>
</sequence>

<attribute name="name" type="NCName" use="required"/>
</extension>
</complexType>

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

<complexType name="tParticipantType">
<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="1"/>
</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"/>
</extension>
</complexContent>
</complexType>

<complexType name="tRoleRef3">
<complexContent>
<extension base="cdl:tExtensibleElements">
<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"/>
</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"/>
      <attribute name="new" type="boolean" use="optional"
                 default="false"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tReference">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <sequence>
        <element name="token" type="cdl:tTokenReference"
                 minOccurs="1" maxOccurs="1"/>
      </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"
                 minOccurs="1" 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="exceptionBlock" type="cdl:tException"
                 minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="tBind">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <sequence>
        <element name="this" type="cdl:tBindVariable"/>
        <element name="free" type="cdl:tBindVariable"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tBindVariable">
  <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="timeout" type="cdl:tTimeout" minOccurs="0"
          maxOccurs="1"/>
        <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="align" type="boolean" use="optional" default="false"/>
      <attribute name="initiate" type="boolean" use="optional" default="false"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tTimeout">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <attribute name="time-to-complete" type="cdl:tXPath-expr" use="required"/>
      <attribute name="fromRoleRecordReference" use="optional">
        <simpleType>
          <list itemType="NCName"/>
        </simpleType>
      </attribute>
    </extension>
  </complexContent>
</complexType>
<complexType name="tSourceVariableRef">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <attribute name="variable" type="cdl:tXPath-expr" use="optional"/>
      <attribute name="expression" type="cdl:tXPath-expr" use="optional"/>
    </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="tAssign">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <sequence>
        <element name="copy" type="cdl:tCopy" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="roleType" type="QName" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tCopy">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <sequence>
        <element name="source" type="cdl:tSourceVariableRef"/>
        <element name="target" type="cdl:tVariableRef"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="causeException" type="boolean" use="optional" default="false"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tSilentAction">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <attribute name="roleType" type="QName" use="optional"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tNoAction">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <attribute name="roleType" type="QName" use="optional"/>
    </extension>
  </complexContent>
</complexType>
<complexType name="tFinalize">
  <complexContent>
    <extension base="cdl:tExtensibleElements">
      <element name="finalizerReference" type="cdl:tFinalizerReference"
        maxOccurs="unbounded"/>
      <attribute name="name" type="NCName" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tFinalizerReference">
  <extension base="cdl:tExtensibleElements">
    <attribute name="choreographyName" type="NCName" use="required"/>
    <attribute name="choreographyInstanceId" type="cdl:tXPath-expr"
      use="optional"/>
    <attribute name="finalizerName" type="NCName" use="optional"/>
  </extension>
</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="tWhenType">
    <restriction base="string">
        <enumeration value="before"/>
        <enumeration value="after"/>
        <enumeration value="timeout"/>
    </restriction>
</simpleType>

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

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

<simpleType name="tDescriptionType">
    <restriction base="string">
        <enumeration value="documentation"/>
        <enumeration value="reference"/>
        <enumeration value="semantics"/>
    </restriction>
</simpleType>