<dt>wd</dt><role>editorsdraft</role><W3CDes>Working Draft</W3CDes>

<language>us-en</language>

<t>Web Services Choreography Description Language</t>, <v>Version 1.0</v>

<doct>Editor's Draft</doct>, 21<d></d> September<m>August <y>2004</y>

This version:
TBD
Latest version:
TBD
Previous Version:
Not Applicable
Editors (alphabetically):

<n>Nickolaos Kavantzas</n>, <a>Oracle</a>,
<e><nickolas.kavantzas@oracle.com></e>
<n>David Burdett</n>, <a>Commerce One</a>
<e><david.burdett@commerceone.com></e>
<n>Gregory Ritzinger</n>, <a>Novell</a> <e><gritzinger@novell.com></a>

Abstract

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

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

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

Status of this Document

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

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

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

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

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

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

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

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

Revision Description

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

Table of Contents

1 Introduction

For many years, organizations have being developing solutions for automating 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:

- <emph>*SOAP*</emph>: 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

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

- <emph>*UDDI*</emph>: allows publishing the availability of a Web Service and its discovery from service requesters using sophisticated searching mechanims

- <emph>*Security layer*</emph>: ensures that exchanged information are not modified or forged

- <emph>*Reliable Messaging layer*</emph>: provides exactly-once and guaranteed delivery of information exchanged between parties

- <emph>*Context, Coordination and Transaction layer*</emph>: defines interoperable mechanisms for propagating context of long-lived business transactions and enables parties to meet correctness requirements by following a global agreement protocol
- <emph>*Business Process Languages layer*</emph>: 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
- <emph>*Choreography layer*</emph>: describes peer-to-peer collaborations of parties by defining from a global viewpoint their common and complementary observable behavior, where information exchanges occur, when the jointly agreed ordering rules are satisfied

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

1.1 Notational Conventions

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

The following namespace prefixes are used throughout this document:

Prefix	**Namespace URI**	**Definition**
wsdl	http://schemas.xmlsoap.org/wsdl/	WSDL namespace for WSDL framework.
cdl	http://www.w3.org/ws/choreography/2004/09/WSCDL	WSCDL namespace for Choreography language.
xsi	http://www.w3.org/2001/XMLSchema-instance	Instance namespace as defined by XSD [11].
xsd	http://www.w3.org/2001/XMLSchema	Schema namespace as defined by XSD [12].

tns	(various)	The "this namespace" (tns) prefix is used as a convention to refer to the current document.
(other)	(various)	All other namespace prefixes are samples only. In particular, URIs starting with "http://sample.com" represent some application-dependent or context-dependent URIs [4].

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

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

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

1.2 Purpose of the Choreography Language

Business or other activities that involve multiple different organizations or independent processes 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.

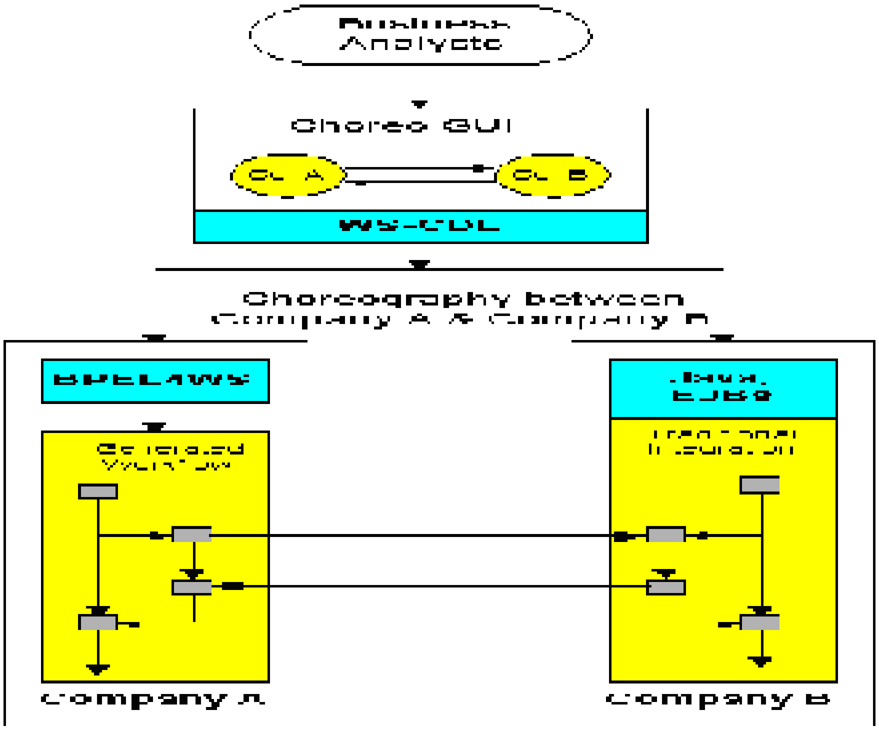
The Web Services Choreography specification is targeted for precisely describing peer-to-peer collaborations between any type of party regardless of the supporting platform or programming model used by the implementation of the hosting environment.

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

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

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



<file>./images/figure1.gif</file>

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

In Figure 1, Company A and Company B wish to integrate their Web Services based applications. The respective business analysts at both companies agree upon the services involved in the collaboration, their interactions and their common ordering and constraint rules under which the interactions occur and then generate a Choreography Language based representation. In this example, a Choreography specifies the interoperability and interactions between services across business entities, while leaving actual implementation decisions in the hands of each individual company:

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

Some additional goals of this definition language are to permit:

- <emph>*Reusability*</emph>. The same Choreography definition is usable by different parties operating in different contexts (industry, locale, etc.) with different software (e.g. application software)
- <emph>*Cooperation*</emph>. Choreographies define the sequence of exchanging messages between two (or more) independent parties or processes by describing how they should cooperate
- <emph>*Multi-Party Collaboration*</emph>. Choreographies can be defined involving any number of parties or processes
- <emph>*Semantics*</emph>. Choreographies can include human-readable documentation and semantics for all the components in the Choreography
- <emph>*Composability*</emph>. Existing Choreographies can be combined to form new Choreographies that may be reused in different contexts
- <emph>*Modularity.*</emph> Choreographies can be defined using an "inclusion" facility that allows a Choreography to be created from parts contained in several different Choreographies
- <emph>*Information Driven Collaboration*</emph>. Choreographies describe how parties make progress within a collaboration, when recordings of exchanged information and observable information changes cause ordering constraints to be fulfilled
- <emph>*Information Alignment*</emph>. Choreographies allow the parties that take part in Choreographies to communicate and synchronize their observable information changes and the actual values of the exchanged information as well
- <emph>*Exception Handling*</emph>. Choreographies can define how exceptional or unusual conditions that occur while the Choreography is performed are handled
- <emph>*Transactionality*</emph>. The processes or parties that take part in a Choreography can work in a "transactional" way with the ability to coordinate the outcome of the long-lived collaborations, which include multiple, often recursive collaboration units, each with its own business rules and goals
- <emph>*Specification Composability*</emph>. 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

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

1.5 Relationship with Business Process Languages

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

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

2 Choreography Model

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

2.1 Model Overview

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

The Choreography model consists of the following notations:

- <emph>*Participants, Roles and Relationships*</emph> - In a Choreography, information is always exchanged between Participants within the same or across trust boundaries
- <emph>*Types, Variables and Tokens*</emph> - Variables contain information about commonly observable objects in a collaboration, such as the messages 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 or Token contains
- <emph>*Choreographies*</emph> - A Choreography allows defining collaborations between interacting peer-to-peer parties:
- <emph>*Choreography Composition*</emph> allows the creation of new Choreographies by reusing existing Choreography definitions
- <emph>*Choreography Life-line*</emph> expresses the progression of a collaboration. Initially, the collaboration is started at a specific business process, then work is performed by following the Choreography and finally the Choreography completes, either normally or abnormally
- <emph>*Choreography Recovery*</emph> consists of:
  - <emph>*Choreography Exception Block*</emph> - describes how to specify what additional interactions should occur when a Choreography behaves in an abnormal way
  - <emph>*Choreography Finalizer Block*</emph> - describes how to specify what additional interactions should occur to reverse the effect of an earlier successfully completed Choreography
- <emph>*Channels*</emph> - A Channel realizes a point of collaboration between parties by specifying where and how information is exchanged
- <emph>*WorkUnits*</emph> - A Work Unit prescribes the constraints that must be fulfilled for making progress and thus performing actual work within a Choreography
- <emph>
- <emph>*Activities and Ordering Structures*</emph> - Activities are the lowest level components of the Choreography that perform the actual work. Ordering Structures combine activities with other Ordering Structures in a nested structure to express the ordering conditions in which the messages in the Choreography are exchanged
- *Interaction Activity*</emph> - An Interaction is the basic building block of a Choreography, which results in an exchange of messages between parties and possible synchronization of their observable information changes and the actual values of the exchanged information

- <emph>*Semantics* -</emph> Semantics allow the creation of descriptions that can record the semantic definitions of every single component in the model

2.2 Choreography Document Structure

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

2.2.1 Package

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

-

The syntax of the <emph>*package*</emph> construct is:

```
<package
   name="ncname"
   author="xsd:string"?
   version="xsd:string"
   targetNamespace="uri"
   xmlns=”http://www.w3.org/ws/choreography/2004/09/WSCDL/”>


   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 definitions contained in this Package. Choreography definitions included to this Package using the inclusion mechanism, may be associated with other namespaces.

The elements informationType, token, tokenLocator, roleType, relationshipType, participantType and channelType are shared by all the Choreographies defined within this Package.

Within a WS-CDL Package, language constructs that need to be uniquely named MUST use the attribute name for specifying a distinct name.

2.2.2 Choreography document Naming and Linking

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

The targetNamespace attribute of type URI MUST be specified.

The URI MUST NOT be a relative URI.

A reference to a definition is made using a QName.

Each definition type has its own name scope.

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

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

2.2.3 Language Extensibility and Binding

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

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

2.2.4 Semantics

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

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

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

<emph>Descriptions </emph>that are t<emph>ext </emph>or <emph>document references </emph>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 messages in a peer-to-peer environment. In these types of environments a party represents a requester of services provided by another party and is at the same time a provider of services requested from other parties, thus creating mutual multi-party service dependencies.

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

<emph></emph>

The <emph>*Role Types*</emph>, *Participant Types,* <emph>*Relationship Types* </emph>and <emph>*Channel Types* </emph>define the coupling of the collaborating parties.

2.3.1 Role Types

<emph>*Role Type* </emph>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 <emph>*roleType* </emph>TyTyconstruct 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 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 Participant Types

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

The syntax of the <emph>*participantType*</emph> 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.

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:

```
<participantType name="Broker">
   <role type="tns:SellerForBuyer" />
   <role type="tns:SellerForShipper" />
</participantType
```

2.3.3 Relationship Types

A *Relationship Type* identifies the Role Type 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 <emph>*relationshipType*</emph> construct is:

```
<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.

Within the role element, the optional attribute behavior identifies the commitment of a party as a list of behavior types belonging to the Role Type specified by the type attribute of the role element. If the behavior attribute is missing then all the behaviors belonging to the Role Type specified by the type attribute of the role element are identified as the commitment of a party.

2.3.4 Channel Types

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

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

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

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

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

The syntax of the <emph>*channelType*</emph> construct is:

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

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

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

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

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

The 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 can be used.

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

The element role is used to identify the Role Type of a party, being the target of an Interaction, 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 Interaction, which is then used for dynamically determining where and how to send or receive information to or into the party. The service reference of a party is distinguished by a Token as specified by 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 process identity and the different conversations are distinguished by a set of Tokens as specified by 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 entity or organization, then the specified Role Types MUST belong to the same Participant Type. In addition the identity elements within the Channel Types MUST have the same number of Tokens with the same informationTypes specified in the same order

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

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

2.4 Information Driven Collaborations

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

<emph>*Variables* capture </emph>capture information about objects in the Choreography, such as the messages exchanged or the observables information of the Roles involved. <emph>*Token* </emph>are aliases that can be used to reference parts of a <emph>Variable</emph>. Both <emph>Variables </emph>and <emph>Tokens </emph>have <emph>*Information Types* </emph>that define the type of information the <emph>Variable </emph>or <emph>Token </emph>contain.

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 <emph>*informationType*</emph> construct is:

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

The attribute name is used for specifying a distinct name for each informationType</emph> element declared within a Choreography Package.

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

2.4.2 Variables

Variables capture information about objects in a Choreography as defined by their <emph>usage</emph>:

- *<emph>Information Exchange Capturing Variables*,</emph> 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
- *<emph>State Capturing Variables*, </emph>whichwhich contain information about the observable changes of 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 <emph>Variable </emph>called "OrderState" set to a value of "OrderSent" and once the message was received by the Seller, the Seller could have a <emph>Variable </emph>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

-

- <emph>*Channel Capturing Variables*</emph>. For example, a Channel Variable could contain information such as the URL to which the message could be sent, the policies that are to be applied, such as security, whether or not reliable messaging is to be used, etc.

The value of Variables:

-

- Is available to 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 in the ends of an Interaction
    - State Capturing Variables, </emph>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

The <emph>*variableDefinitions*</emph> construct is used for defining one or more Variables within a Choreography block.

The syntax of the <emph>*variableDefinitions*</emph> construct is:

```
<variableDefinitions>
   <variable   name="ncname"
       informationType="qname"|channelType="qname"
       mutable="true|false"?
       free="true|false"?
       silentAction="true|false"?
       roleType="qname"? />+
</variableDefinitions>
```

The defined Variables can be of the following types:

- Information Exchange Capturing Variables, State Capturing Variables. The attribute informationType describes the type of the object captured by the Variable
- Channel Capturing Variables. The attribute channelType describes the type of the channel object captured by the Variable

The optional attribute mutable, when set to "false" describes that the Variable information when initialized, cannot change anymore. The default value for this attribute is “true”.

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

- The type of a free Variable MUST match the type of the Variable defined in an enclosing Choreography
- A perform activity MUST bind a free Variable defined in an enclosed Choreography with a Variable defined in an enclosing Choreography when sharing the Variables information

The optional attribute free, when set to "false" describes that a Variable is defined in this Choreography.
The default value for the free attribute is “false”.

The optional attribute silentAction, when set to "true" describes that there SHOULD NOT be any activity used for creating or changing this Variable in the Choreography, if these operations should not be observable to other parties. The default value for this attribute is “false”.

The optional attribute roleType is used to specify the Role Type of a party at which the Variable information will reside.

The following rules apply to Variable Definitions:

- The attribute name is used for specifying a distinct name for each variable element declared within a variableDefinitions element when needed. The Variables with Role Type not specified MUST have distinct names. The Variables with Role Type specified MUST have distinct names, when their Role Type is the same
- 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 a tuple (Role1, Role2), then a Variable x defined in Choreography C1 without a roleType attribute means it is defined at Role1 and Role2

•

2.4.2.1 Expressions

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

Predicate expressions are 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 Section 10. The function extensions are defined in the standard WS-CDL namespace “http://www.w3.org/ws/choreography/2004/09/WSCDL”. The prefix "cdl:" is associated with this namespace.

2.4.3 Tokens

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

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

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

The syntax of the <emph>*token*</emph> construct is:

```
<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 <emph>*tokenLocator*</emph> construct is:

```
<tokenLocator  tokenName="qname"
      informationType="qname"
      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 on which the query is performed to locate the Token.

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

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

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

2.4.4 Choreographies

. The operational semantics of these rules are based on the information-driven computational model, where availability of variable information causes a guarded unit-of-work and its enclosed actions to be enabled.

A<emph> *Choreography* defines re-usable the common rules, that govern the ordering of exhanged messages and the provisioning patterns of collaborative behavior, agreed between two or more interacting peer-to-peer parties.</emph>

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

A Choreography defines the re-usable the common rules, that govern the ordering of exhanged messages and the provisioning patterns of behavior, </emph> action(s) performing the actual work, such as exchange of messages, when the specified ordering constraints are satisfied.

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

The Choreography that is performed MAY be defined either:

- <emph>*Locally*</emph>- - they are contained, in the same Choreography definition as the Choreography that performed them
- <emph>*Globally* - </emph> they are specified in a separate top-level Choreography definition that is defined in the same or in a different Choreography Package and can be used by other Choreographies and hence the contract is reusable

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 <emph>*Choreography Visibility Horizon* for this Variable</emph>.

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

A Choreography MUST contain an <emph>*Activity-Notation*</emph>. The Activity-Notation specifies the enclosed actions of the Choreography that perform the actual work.

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

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

The <emph>*Choreography-Notation*</emph> is used to define a Choreography

. The syntax is:

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

```
   <relationship  type="qname" />+


   variableDefinitions?

   Choreography-Notation*

      Activity-Notation

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

The 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 Variable information that is defined in an enclosing, and changed within an enclosed Choreography is available to its sibling Choreographies:

- When isolation is set to "dirty-write", the Variable information MAY be immediately overwritten by actions in other Choreographies
- When isolation is set to "dirty-read", the Variable information MAY be immediately visible for read but not for write to other Choreographies
- When isolation is set to "serializable", the Variable information MUST be visible for read or for write to other Choreographies only after this Choreography has ended successfully

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

The optional variableDefinitions element 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 Choreographies that MAY be performed only within this Choreography.

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

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

2.4.5 WorkUnits

A<emph> *Work Unit* </emph>prescribes the constraints that must be fulfilled for making progress and thus performing actual work within a Choreography. Examples of a Work Unit include:

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

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

When enabled, a Work Unit expresses interest(s) on the availability of one or more Variable information that already exist or will be created in the future.

The Work Unit's interest(s) are matched when all the required Variable information are or become available and the specified matching condition on the Variable information is met. Availability of some Variable information does not mean that a Work Unit matches immediately. Only when all Variable information required by a Work Unit become available, in the appropriate Visibility Horizon, does matching succeed. Variable information available within a Choreography MAY be matched with a Work Unit that will be enabled in the future. One or more Work Units MAY be matched concurrently if their respective interests are matched. When a Work Unit matching succeeds then its enclosed actions are enabled.

A Work Unit MUST contain an <emph>*Activity-Notation*</emph> that performs the actual work.

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<emph> *WorkUnit-Notation*</emph> is defined as follows:

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

      Activity-Notation
</workunit>
```

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

The guard condition of a Work Unit, specified by the optional guard attribute, describes the interest on the availability of one or more, existing or future Variable information.

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

The optional attribute block specifies whether the matching condition relies on the Variable that is currently available, or whether the Work Unit has to block waiting for the Variable to become available in the future if it is not currently available. The default is set to “false”.

2.4.5.1

The following rules apply:

- When a guard condition is not specified then the Work Unit always matches
- When a repetition condition is not specified then the Work Unit is not considered again for matching after the Work Unit got matched once
- When a guard condition or repetition condition is specified then:
    - 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 10.
    - 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 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. 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.
- If Variables are defined at different Roles, then they can be combined together in a guard condition or repetition condition using only the globalizedTrigger() WS-CDL function
- If Variables are defined at the same Role, then they can be combined together in a guard condition or repetition condition using all available XPATH operators and the WS-CDL functions except the globalizedTrigger() WS-CDL function
- When the bttribute block is set to "true" and one or more variable(s) are not available, then the Work Unit MUST block waiting for the variable information to become available. This attribute MUST not be used in Exception Work Units. hen the Variable information specified by the Guard guard condition conditionbecome available then the Guard guard condition iscondition is evaluated:
  - If 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 even if specified is not evaluated
  - If the guard condition evaluates to "true", then the Work Unit is matched and then the repetition condition, if specified, is evaluated when the Variable information specified by the repetition condition become available
    - If the repetition condition evaluates to "true", then the Work Unit is considered again for matching
    - If the repetition condition evaluates to "false", then the Work Unit is not considered again for matching
- When 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 even if specified is not evaluated
  - If matching succeeds, then the repetition condition, if specified, is evaluated immediately

- If either the Variable information is not currently available or the repetition condition evaluates to "false", then the Work Unit is not considered again for matching
- Othewise, then the Work Unit is considered again for matching

1.

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

<emph>*a. Example of a Work Unit with block equals to "true"*</emph>:

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.

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

<emph>*b. Example of a Work Unit with block equals to "false"*</emph>:

In the following Work Unit, the guard condition checks if 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 the value is less than 10, the matching condition is "false" and the activity is skipped.

```
<workunit  name="Stockcheck"
           guard="cdl:getVariable("StockQuantity", "/Product/Qty",
                                  "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:

```
<workunit  name="WaitForAlignment"
           guard="cdl:variablesAligned(
                    "PurchaseOrderAtBuyer", "PurchaseOrderAtSeller",
                    "customer-retailer-relationship")”
       block="true" >
```

```
   ... <!--some activity -->
</workunit>
```

2.4.6 Including Choreographies

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

The example below shows a possible syntactic reuse of a Choreography definition:

```
<choreography name="newChoreography” root="true">
...
   <variable name="newVariable" informationType="someType"
             role="randomRome"/>
   <xi:include href="genericVariableDefinitions.xml" />
   <xi:include href="otherChoreography.xml"
               xpointer="xpointer(//choreography/variable[1]) />
...
</choreography>
```

The first xi:include element includes an entire WS-CDL document. The second xi:include element includes a portion of a WS-CDL document.

All XInclude declarations in the Choreography definition MUST be resolved before any WS-CDL related processing can occur.

2.4.7 Choreography Life-line

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

The root Choreography is the only top-level Choreography that MAY be initiated. The root Choreography is enabled when it is initiated. All non-root, top-level Choreographies MAY be enabled when performed.

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

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

2.4.8 Choreography Recovery

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

A Finalizer WorkUnit MAY be defined as part of a Choreography to provide the finalization actions that semantically “undo” that completed Choreography.

2.4.8.1 Exception Block

A Choreography can sometimes fail as a result of an exceptional circumstance or error.

Different types of exceptions are possible including this non-exhaustive list:

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

At least one

Exception Work Unit MUST be defined as part of the Exception block of a Choreography for the purpose of handling the exceptional conditions occurring on that Choreography. To handle these, an Exception Work Unit MAY express interest on fault information using its guard condition. If no guard condition is specified, then the *default* Exception Work Unit expresses interest on any type of fault. Within the Exception Block of a Choreography there MUST NOT be more than one Exception Work Units without guard condition defined. An Exception Work Unit MUST set its attribute block always to "false" and MUST NOT define a repetition condition.

-
-

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

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

The rules for matching a fault are as follows:

- If a fault is matched by the guard condition of an Exception Work Unit, then the actions of the matched Work Unit are enabled for recovering from the fault. 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 being 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 for recovering from the fault
-
- Otherwise, the fault is not matched by an Exception Work Unit defined within the Choreography in which the fault occurs, and in this case the fault will be recursively propagated to the Exception Work Unit of the immediate enclosing Choreography until a match is successful

If a fault occurs within a Choreography, then the faulted Choreography completes unsuccessfully and this causes its Finalizer WorkUnit to disabled. The actions, including enclosed Choreographies, enabled within the faulted 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 fault. The semantics for matching the fault and acting on it are the same as described in this section.

2.4.8.2 Finalizer Block

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

A Choreography MAY define one Finalizer Work Unit.

A Finalizer WorkUnit is enabled only after the Choreography it belongs to completes successfully. The Finalizer Work Unit may be enabled only once.

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 or as they stand at the current time.

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

2.5 Activities

<emph>*Activities* </emph>are the lowest level components of the Choreography, used to describe the actual work performed when the specified ordering constraints are satisfied.

An Activity-Notation is then either:

- An <emph>*Ordering Structure* </emph> – which combines Activities with other Ordering Structures in a nested way to specify the ordering rules of activities within the Choreography
- A <emph>WorkUnit-Notation</emph>
- A <emph>*Basic Activity* </emph>that performs the actual work. A Basic Activity is then either:
    - <emph>*Interaction*</emph>, which results in an exchange of messages between parties and possible synchronization of their observable information changes and the actual values of the exchanged information
    - A <emph>*Perform,* </emph>which means that a complete, separately defined Choreography is performed
    - An <emph>*Assign*</emph>, which assigns, within one Role, the value of one Variable to the value of another Variable

- <emph>AAa*Silent Action*</emph>, which provides an explicit designator used for specifying the point where party specific operation (s) with non-observable operational details MUST be performed
- A *No Action*</emph>, which provides an explicit designator used for specifying the point where a party does not perform any action

2.5.1 Ordering Structures

An <emph>*Ordering Structure*</emph> is one of the following:

- *Sequence*
- *Parallel*
- *Choice*

2.5.1.1 Sequence

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

The syntax of this construct is:

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

2.5.1.2 Parallel

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

The syntax of this construct is:

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

2.5.1.3 Choice

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

When two or more activities are specified in a choice element, only one activity is selected and the other activities are disabled. If the choice has Work Units with 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  channelVariable="doGoodCredit-channel" operation="doCredit">
      ...
  </interaction>

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

2.5.2 Interacting

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

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

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

- A <emph>*One-Way Interaction* </emph>that involves the exchanging of a single message
- A <emph>*Request-Response Interaction* </emph>when two messages are exchanged

An Interaction also contains "references" to:

- The <emph>*Channel Capturing Variable* </emph>that specifies the interface and other data that describe where and how the message is to be sent

- The <emph>*Operation* </emph>that specifies what the recipient of the message should do with the message when it is received
- The <emph>*From Role* </emph>and <emph>*To Role* </emph>that are involved
- The <emph>Information informType or Channel Type </emph>that is being exchanged
- The <emph>*Information Exchange Capturing Variables* </emph>at the From Role and To Role that are the source and destination for the Message Content
- A list of potential observable information changes </emph>that MAY occur and MAY need to be aligned at the <emph>*From Role*</emph> and the <emph>*To Role,*</emph> 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 <emph>Capturing Variables <emph>that are complementary to one or more State Capturing <emph>Variables </emph>of its partner Role
- Additionally, within an Interaction a Role MAY need to have a common understanding of the values of the I<emph>nformation Exchange Capturing Variables </emph>at the partner Role

With Interaction Alignment both the Buyer and the Seller 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 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 Protocol Based Information Exchanges

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

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

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

2.5.2.3 Interaction Life-line

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

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

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

An Interaction completes abnormally if the following faults occur:

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

2.5.2.4 Interaction Syntax

The syntax of the <emph>*interaction*</emph> construct is:

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

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

   <exchange  name="ncname"
              informationType="qname"? channelType="qname"?
              action="request"|"respond" >
     <send    variable="XPath-expression"? recordReference=”list of ncname”? />

     <receive variable="XPath-expression"? recordReference=”list of ncname”? />
   </exchange>*

   <record  name="ncname"
            when="before"|"after" >
     <source  variable="XPath-expression" />
     <target  variable="XPath-expression" />
   </record>*
</interaction>
```

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 further. This requires that the messages in the Choreography also contain correlation information, for example by including:

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

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

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

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

The optional align attribute when set to "true" means that the Interaction results in the common understanding of both the information exchanged and the resulting 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 relationship 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 a one-way request or a request/response Interaction.

Within the exchange element, the optional informationType and channelType attributes, identify the Information Type or the Channel Type of the information that is exchanged between the two Roles in an Interaction.

- If none of these attributes are specified, then it is assumed that either no actual data is exchanged or the type of data 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 message exchange happens fromRole to toRole
- When the action attribute is set to ”respond”, then the message exchange happens from toRole to fromRole

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

- The optional Variables specified using the WS-CDL function getVariable() 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 silentAction set to “true”
- Within the send or the receive element(s) of an exchange element, the recordReference attribute contains a list of references to record element(s) in the same Interaction
- 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 the Request and the Response exchanges complete successfully and all referenced records complete successfully
    - 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
    - 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

The optional element record is used to create or change one or more Variables at the send and receive ends of the Interaction. Within the record element, the source and target elements specify using the WS-CDL function getVariable() the Variables whose information is recorded:

- When the action element is set to "request", then the recording(s) of the Variables 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) of the Variables 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 a receive of a message at a Role in a Request or Response exchange

.

The following rules apply for record:

- One or more record elements MAY be specified and performed at one or both the Roles within an Interaction
    - The source and the target Variable MUST be of compatible type
    - The source and the target Variable MUST be defined at the same Role
    - The target Variable MUST not be defined with the attribute silentAction set to “true”
    - A record element MUST NOT be specified in the absence of an exchange element within an Interaction
- 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 Variable 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 Variables 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. The Interaction happens from Role Type “Consumer” to Role Type “Retailer” on the Channel "retailer-channel" as a request/response message exchange.

- The message purchaseOrder is sent from Consumer to Retailer as a request message
- The message purchaseOrderAck is sent from Retailer to Consumer as a response message
- The Variable consumer-channel is populated at Retailer using the record element
- The Interaction happens on the retailer-channel which has a Token Type purchaseOrderID used as an identity of the channel. This identity element is used to identify the business process of the retailer
- The request message purchaseOrder contains the identity of the retailer business process as specified in the tokenLocator for purchaseOrder message
- The response message purchaseOrderAck contains the identity of the consumer business process as specified in the tokenLocator for purchaseOrderAck message
- The consumer-channel is sent as a part of purchaseOrder Interaction from the consumer to the retailer on retailer-channel during the request. Here the record element populates 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.

```
<package name="ConsumerRetailerChoreography” version="1.0"
  <informationType name="purchaseOrderType" type="pons:PurchaseOrderMsg"/>
  <informationType name="purchaseOrderAckType" type="pons:PurchaseOrderAckMsg"/>
  <token name="purchaseOrderID" informationType="tns:intType"/>
  <token name="retailerRef" informationType="tns:uriType"/>
  <tokenLocator tokenName="tns:purchaseOrderID"
                informationType="tns:purchaseOrderType" query="/PO/orderId"/>
  <tokenLocator tokenName="tns:purchaseOrderID"
                informationType="tns:purchaseOrderAckType" query="/PO/orderId"/>

  <roleType name="Consumer">
    <behavior name="consumerForRetailer" interface="cns:ConsumerRetailerPT"/>
    <behavior name="consumerForWarehouse" interface="cns:ConsumerWarehousePT"/>
  </roleType>
  < roleType name="Retailer">
    <behavior name="retailerForConsumer" interface="rns: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 type="tns:consumerRef"/>
    </reference>
    <identity>
      <token type="tns:purchaseOrderID"/>
    </identity>
  </channelType>

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

  <choreography name="ConsumerRetailerChoreography” root="true">
    <relationship type="tns:ConsumerRetailerRelationship"/>
    <variableDefinitions>
    <variable name="purchaseOrder" informationType="tns:purchaseOrderType"
              silentAction="true" />
    <variable name="purchaseOrderAck" informationType="tns:purchaseOrderAckType" /
>
    <variable name="retailer-channel" channelType="tns:RetailerChannel"/>
    <variable name="consumer-channel" channelType="tns:ConsumerChannel"/>

    <interaction channelVariable="tns:retailer-channel "
                 operation="handlePurchaseOrder" align="true"
                 initiate="true">
      <participate relationship="tns:ConsumerRetailerRelationship"
                   fromRole="tns:Consumer" toRole="tns:Retailer"/>

      <exchange informationType="tns:purchaseOrderType" action="request">
        <send variable="cdl:getVariable(“tns:purchaseOrder”)" />
        <receive variable="cdl:getVariable(“tns:purchaseOrder”)"
                 recordReference="populateChannel" />
      </exchange>

      <exchange informationType="purchaseOrderAckType" action="respond">
        <send variable="cdl:getVariable(“tns:purchaseOrderAck”)" />
        <receive variable="cdl:getVariable(“tns:purchaseOrderAck”)" />
      </exchange>

      <record name="populateChannel" when="after">
        <source variable="cdl:getVariable(“tns:purchaseOrder, “PO/CustomerRef”)"/>
        <target variable="cdl:getVariable(“tns:consumer-channel”)"/>
      </record>
    </interaction>
  </choreography>
</package>
```

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

You 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 was 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 recursively 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 syntax of the <emph>*perform*</emph> construct is:

```
<perform  choreographyName="qname">
   <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 a Locally or Globally defined Choreography to be performed. The performed Choreography even when defined in a different Choreography Package is conceptually treated as an enclosed Choreography. An enclosing Choreography MAY perform only an immediately contained Choreography that is Locally defined.

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

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

The following rules apply when a Choreography is performed:

- The Choreography to be performed MUST NOT be a root Choreography
- The Choreography to be performed MUST be defined either using a Choreography-Notation immediately contained in the same Choreography or it MUST be a top-level Choreography with root attribute set to "false" in the same or different Choreography Package.
- Performed Choreographies that are declared in a different Choreography Package MUST be included first
- 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 free Variables within the bind element MUST have the attribute free set to "true" in their definition
-
-
- 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

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.

```
<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

<emph>*Assign*</emph> activity is used to create or change and then make available within one Role, the value of one Variable using the value of another Variable.

The assignments may include:

- Assigning an Information Capturing *<emph>*Variable*</emph>* to another or a part of that *<emph>*Variable*</emph>* to a State Capturing Variable or another Information Capturing *<emph>*Variable*</emph>* so that a message received can be used to trigger/constrain, using a Work Unit guard condition, or other Interactions
- Assigning a State Capturing Variable to another State Capturing Variable or to an Information Capturing *<emph>*Variable*</emph>* locally at a Role
- </emph>tion ariable

The syntax of the <emph>*assign*</emph> construct is:

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

The assign construct creates or changes at a Role the Variable defined by the target element using the Variable defined by the source element at the same Role.

The following rules apply to assignment:

- The source and the target Variable MUST be of compatible type
- The source and the target Variable MUST be defined at the same Role
- 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 following example assigns the customer address part from Variable “PurchaseOrderMsg” to Variable “CustomerAddress”.

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

2.5.5 Marking Silent Actions

Silent actions are explicit designators used for marking the points where party specific operations with non-observable operational details MAY 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*<emph>siss construct is:

```
<silentAction role="qname? />
```

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

2.5.6 Marking the Absence of Actions

No actions are explicit designators used for marking the points where a party does not perform any action.

The syntax of the *no action*<emph>siss construct is:

```
<noAction role="qname? />
```

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

3 Example

To be completed

4 Relationship with the Security framework

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

5 Relationship with the Reliable Messaging framework

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

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

6 Relationship with the Transaction/Coordination framework

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

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

7 Acknowledgments

To be completed

8 References

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

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

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

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

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

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

[7] Web Services Definition Language (WSDL) 2.0

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

[9] W3C Recommendation "The XML Specification"

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

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

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

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

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

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

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

[16] XLANG: Web Services for Business Process Design, S. Thatte, 2001 Microsoft Corporation

[17] WSFL: Web Service Flow Language 1.0, F. Leymann, 2001 IBM Corporation

[18] OASIS Working Draft “BPEL: Business Process Execution Language 2.0”. This is work in progress.

[19] BPMI.org “BPML: Business Process Modeling Language 1.0”

[20] Workflow Management Coalition “XPDL: XML Processing Description Language 1.0”, M. Marin, R. Norin R. Shapiro

[21] OASIS Working Draft “WS-CAF: Web Services Context, Coordination and Transaction Framework 1.0”. This is work in progress.

[22] OASIS Working Draft “Web Services Reliability 1.0”. This is work in progress.

[23] The Java Language Specification

[24] OASIS “Web Services Security”

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

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

[27] “XML Inclusions Version 1.0” http://www.w3.org/TR/xinclude/

9 WS-CDL XSD Schemas

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
     targetNamespace=”http://www.w3.org/ws/choreography/2004/09/WSCDL/”
     xmlns=”http://www.w3.org/2001/XMLSchema”
     xmlns:cdl=”http://www.w3.org/ws/choreography/2004/09/WSCDL/”
     elementFormDefault="qualified">

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

  </complexType>

  <element name="package" type="cdl:tPackage"/>

  <complexType name="tPackage">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>

          <element name="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:tRole" minOccurs="0"
                  maxOccurs="unbounded"/>
          <element name="relationshipType" type="cdl:tRelationship"
                  minOccurs="0" maxOccurs="unbounded"/>
          <element name="participantType" type="cdl:tParticipant"
                  minOccurs="0" maxOccurs="unbounded"/>
          <element name="channelType" type="cdl:tChannelType"
                  minOccurs="0" maxOccurs="unbounded"/>
          <element name="choreography" type="cdl:tChoreography"
                  minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="author" type="string" use="optional"/>
        <attribute name="version" type="string" use="required"/>
```

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

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

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

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

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

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

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

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

  <complexType name="tReference">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="token" type="cdl:tTokenReference"
                      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="exception" type="cdl:tException"
                minOccurs="0"/>
        <element name="finalizer" type="cdl:tFinalizer"
                    minOccurs="0"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="complete" type="cdl:tBoolean-expr"
                   use="optional"/>
      <attribute name="isolation" type="cdl:tIsolation"
                   use="optional" default="dirty-write"/>
      <attribute name="root" type="boolean" use="optional"
                   default="false"/>
    </extension>
  </complexContent>
</complexType>

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

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

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

<group name="activity">
```

```
    <choice>
      <element name="sequence" type="cdl:tSequence"/>
      <element name="parallel" type="cdl:tParallel"/>
      <element name="choice" type="cdl:tChoice"/>
      <element name="workunit" type="cdl:tWorkunit"/>
      <element name="interaction" type="cdl:tInteraction"/>
      <element name="perform" type="cdl:tPerform"/>
      <element name="assign" type="cdl:tAssign"/>
      <element name="silentAction" type="cdl:tSilentAction"/>
      <element name="noAction" type="cdl:tNoAction"/>

    </choice>
  </group>

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

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

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

  <complexType name="tPerform">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="bind" type="cdl:tBind"
                  minOccurs=”0” maxOccurs="unbounded"/>
        </sequence>
        <attribute name="choreographyName" type="QName"
                 use="required"/>
      </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>
      </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="record" type="cdl:tRecord" minOccurs="0"
                  maxOccurs="unbounded"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
        <attribute name="channelVariable" type="QName"
                 use="required"/>
        <attribute name="operation" type="NCName" use="required"/>
        <attribute name="time-to-complete" type="duration"
                 use="optional"/>
        <attribute name="align" type="boolean" use="optional"
                 default="false"/>
        <attribute name="initiate" type="boolean"
                 use="optional" default="false"/>
      </extension>
    </complexContent>
  </complexType>

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

  <complexType name="tExchange">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <sequence>
          <element name="send" type="cdl:tVariableRecordRef"/>
          <element name="receive" type="cdl:tVariableRecordRef"/>
        </sequence>
        <attribute name="name" type="string" use="optional"/>
        <attribute name="informationType" type="QName"
                 use="optional"/>
        <attribute name="channelType" type="QName"
                 use="optional"/>
        <attribute name="action" type="cdl:tAction2" use="required"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="tVariableRecordRef">
    <complexContent>
      <extension base="cdl:tExtensibleElements">
        <attribute name="variable" type="cdl:tXPath-expr"
                use="optional"/>
        <attribute name="recordReference" use="optional">
          <simpleType>
             <list itemType=”NCName”/>
          </simpleType>
        </attribute>
      </extension>
    </complexContent>
  </complexType>

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

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

      </extension>
    </complexContent>
  </complexType>

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

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

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

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

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

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

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

```
      <enumeration value="request"/>
      <enumeration value="respond"/>
    </restriction>
  </simpleType>

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

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

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

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

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

10 WS-CDL Supplied Functions

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

<emph>*xsd:time getCurrentTime(xsd:QName roleName)*

Returns the current time at the Role specified by *roleName.*

</emph>

<emph>*xsd:date getCurrentDate(xsd:QName roleName)*

Returns the current date at the Role specified by *roleName.*

</emph>

<emph>*xsd:dateTime getCurrentDateTime(xsd:QName roleName)*

</emph>

Returns the current date and time at the Role specified by *roleName.*

<emph>*xsd:boolean hasTimeElapsed(xsd:duration elapsedTime, xsd:QName roleName)*

Returns “true” if used in a guard or repetition condition of a Work Unit with the block attribute set to “true” and the time specified by elapsedTime at the Role specified by *roleName* has elapsed from the time the either the guard or the repetition condition were enabled for matching. Otherwise it returns “false”.

<emph>*xsd:string createNewID()*

</emph>

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

<emph>*xsd:any getVariable(xsd:string varName, xsd:string documentPath?, xsd:QName roleName?)*

</emph>

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

<emph>*xsd:boolean isVariableAvailable(xsd:string varName, xsd:QName roleName)*

</emph>

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

<emph>*xsd:boolean variablesAligned(xsd:string varName, xsd:string withVarName,*

xsd:QName relationshipName)

</emph>

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.

<emph>*xsd:any getChannelReference(xsd:string varName)*

</emph>

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

<emph>*xsd:any getChannelIdentity(xsd:string varName)*

</emph>

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

<emph>*xsd:boolean globalizedTrigger(xsd:string expression, xsd:string roleName, xsd:string expression2, xsd:string roleName2, …)*

</emph>

Combines expressions that include Variables that are defined at different Roles.