1

Web Services Choreography Description Language 1.0

Editor's Draft, April 3 2004

This version:
   TBD
Latest version:
   TBD
Previous Version:
   Not Applicable
Editors (alphabetically):
   Nickolaos Kavantzas, Oracle <u>mailto:nickolas.kavantzas@oracle.com</u>

Copyright

Abstract

The Web Services Choreography Description Language (WS-CDL) is an XML-based language that describes peer-to-peer collaborations of Web Services participants 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 Web Service participant regardless of

32 the supporting platform or programming model used by the implementation of the
33 hosting environment.

Status of this Document

35 This specification is an editor's draft and will be updated, extended or replaced by other
36 documents, if necessary. It is for review and evaluation only. The authors of this
37 specification provide this document as is and provide no warranty about the use of this
38 document in any case. The authors welcome feedback and contributions to be
39 considered for updates to this document in the near future.

Table of Contents

1. Introduction

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

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

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

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

The Web Service architecture stack targeted for integrating interacting applications consists of the following components:
- SOAP: defines the basic formatting of a message and the basic delivery options independent of programming language, operating system, or platform. A SOAP compliant Web Service knows how to send and receive SOAP-based messages
- WSDL: describes the static interface of a Web Service. It defines the protocol and the message characteristics of end points. Data types are defined by XML Schema specification, which supports rich type definitions and allows expressing any kind of XML type requirement for the application data
- UDDI: allows publishing the availability of a Web Service and its discovery from service requesters using sophisticated searching mechanims
- Security layer: ensures that exchanged information are not modified or forged
- Reliable Messaging layer: provides exactly-once and guaranteed delivery of information exchanged between participants
- Context, Coordination and Transaction layer: defines interoperable mechanisms for propagating context of long-lived business transactions and enables participants to meet correctness requirements by following a global agreement protocol
- Business Process Languages layer: describes the execution logic of Web Services based applications by defining their control flows (such as conditional, sequential, parallel and exceptional execution) and prescribing the rules for consistently managing their non-observable data
- Choreography layer: describes peer-to-peer collaborations of Web Services participants by defining from a global viewpoint their common and complementary observable behavior, where information exchanges occur, when the jointly agreed ordering rules are satisfied

142 The Web Services Choreography specification is targeted for composing interoperable,
143 peer-to-peer collaborations between any type of Web Service participant regardless of
144 the supporting platform or programming model used by the implementation of the
145 hosting environment.
146

1.1. Notational Conventions

146

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

150 2. The following namespace prefixes are used throughout this document:

Prefix	Namespace URI	Definition
wsdl	http://schemas.xmlsoap.org/wsdl/	WSDL namespace for WSDL framework.
cdl	http://www.w3.org/ws/choreography/2004/02/WSCDL	WSCDL namespace for Choreography language.
xsi	http://www.w3.org/2000/10/XMLSchema-instance	Instance namespace as defined by XSD [10].
xsd	http://www.w3.org/2000/10/XMLSchema	Schema namespace as defined by XSD [10].
tns	(various)	The "this namespace" (tns) prefix is used as a convention to refer to the current document.
(other)	(various)	All other namespace prefixes are samples only. In particular, URIs starting with "http://sample.com" represent some application-dependent or context-dependent URI [4].

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

153 • The syntax appears as an XML instance, but the values indicate the data types
154 instead of values.

155 • Characters are appended to elements and attributes as follows: "?" (0 or 1), "*" (0 or
156   more), "+" (1 or more).

157 • Elements names ending in "…" (such as <element…/> or <element…>) indicate that
158   elements/attributes irrelevant to the context are being omitted.

159 • Grammar in bold has not been introduced earlier in the document, or is of particular
160   interest in an example.

161 • <-- extensibility element --> is a placeholder for elements from some "other"
162   namespace (like ##other in XSD).

163 • The XML namespace prefixes (defined above) are used to indicate the namespace
164   of the element being defined.

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

168 XSD schemas are provided as a formal definition of WS-CDL grammar (see Appendix
169 A).

170

1.2. Purpose of the Choreography Language

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

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

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

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

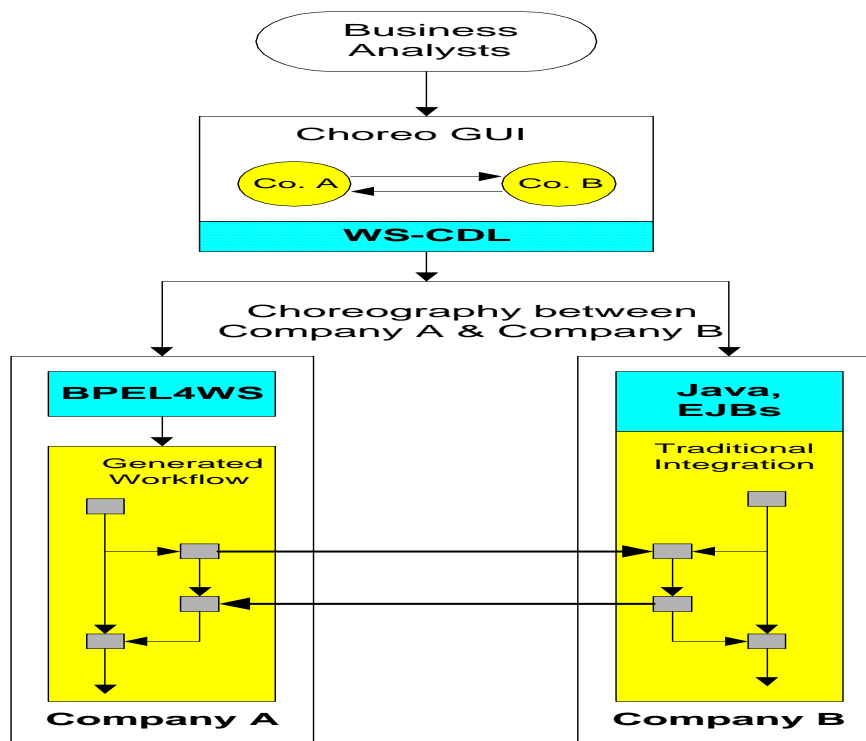The figure below demonstrates a possible usage of the Choreography Language.

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

9

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

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

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

209

1.3. Goals

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

215

216 Some additional goals of this definition language are to permit:

- 217 • *Reusability*. The same choreography definition is usable by different participants
218 operating in different contexts (industry, locale, etc.) with different software (e.g.
219 application software)
- 220 • *Cooperative*. Choreographies define the sequence of exchanging messages
221 between two (or more) independent participants or processes by describing how
222 they should cooperate
- 223 • *Multi-Party*. Choreographies can be defined involving any number of participants
224 or processes
- 225 • *Semantics*. Choreographies can include human-readable documentation and
226 semantics for all the components in the choreography
- 227 • *Composability*. Existing Choreographies can be combined to form new
228 Choreographies that may be reused in different contexts
- 229 • *Modular*. Choreographies can be defined using an "import" facility that allows a
230 choreography to be created from components contained in several different
231 Choreographies
- 232 • *Information Driven*. Choreographies describe how participants that take part in
233 Choreographies maintain where they are in the Choreography by recording their
234 exchanged information and the observable state changes caused by these
235 exchanges of information and also their reactions to them
- 236 • *Information Alignment*. Choreographies allow the participants that take part in
237 Choreographies to communicate and synchronize their observable state changes
238 and the actual values of the exchanged information as well
- 239 • *Exception Handling*. Choreographies can define how exceptional or unusual
240 conditions that occur whilst the choreography is performed are handled
- 241 • *Transactionality*. The processes or participants that take part in a choreography
242 can work in a "transactional" way with the ability to coordinate the outcome of the

243 long-lived collaborations, which include multiple, often recursive collaboration
244 units, each with its own business rules and goals
245 • *Compatibility with other Specifications*. This specification will work alongside and
246 complement other specifications such as the WS-Reliability [22], WS-Composite
247 Application Framework (WS-CAF) [21], WS-Security [24], Business Process
248 Execution Language for WS (BPEL4WS) [18], etc.
249
250

1.4. Relationship with XML and WSDL

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

1.5. Relationship with Business Process Languages

A Choreography Language is not an "executable business process description language" [16, 17, 18, 19, 20] or an implementation language [23]. The role of specifying the execution logic of an application will be covered by these specifications; by enabling the definition of the control flows (such as conditional, sequential, parallel and exceptional execution) and the rules for consistently managing their non-observable business data.

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

- Web Services applications, whose implementation is based on executable business process languages [16, 17, 18, 19, 20]
- Web Services 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 Web Service participants. 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 participants.

The Choreography model consists of the following notations:

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

321   •   *Semantics* - Semantics allow the creation of descriptions that can record the
322   semantic definitions of almost every single component in the model
323

2.2 Choreography Document Structure

325 A WS-CDL document is simply a set of definitions. The WS-CDL definitions are named
326 constructs that can be referenced. There is a *package* element at the root, and individual
327 Choreography definitions inside.
328

2.2.1 Package

330 A WS-CDL package contains a set of one or more Choreographies and a set of one or
331 more collaboration type definitions, allowing the various types whose use may be wider
332 than a single Choreography to be defined once.

333 The Choreography package contains:
334

335   •   Zero or more Import definitions
336   •   Zero or more Information Types
337   •   Zero or more Token types and Token Locators
338   •   Zero or more Role types
339   •   Zero or more Relationship types
340   •   Zero or more Participants
341   •   Zero or more Channel types
342   •   Zero or more, package-level Choreographies
343

344

345 The syntax of the **package** construct is:
346

```
347    <package
348        name="ncname"
349        author="xsd:string"?
350        version="xsd:string"
351        targetNamespace="uri"
352        xmlns="http://www.w3.org/ws/choreography/2004/02/WSCDL/">
353
354            importDefinitions*
355
356            informationType*
357
358            token*
359
360            tokenLocator*
361
362            role*
363
364            relationship*
365
366            participant*
367
368            channelType*
369
370        Choreography-Notation*
```

```
371        </package>
372
373
```

The package construct allows aggregating a set of Choreography definitions, where the elements informationType, token, tokenLocator, role, relationship, participant and channelType are shared by all the Choreographies defined within this package.

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

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

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

If desired to extend the WS-CDL language, this specification allows inside a WS-CDL document 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 description sub-element is used as a textual description for documentation purposes. This element is allowed inside any WS-CDL language element.

The information provided by the description element will allow for the recording of semantics in any or all of the following ways:
- *Text.* This will be in plain text or possibly HTML and should be brief
- *Document Reference.* This will contain a URL to a document that more fully describes the component. For example on the top level Choreography Definition that might reference a complete paper
- *Structured Attributes.* This will contain machine processable definitions in languages such as RDF or OWL

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

419

2.3 Coupling Web Service participants

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

A WS-CDL document describes how a Web Service participant is capable of engaging in peer-to-peer collaborations with the same participant or with different participants.

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

The *Roles, Relationship* and *Channels* define the coupling of the collaborating Web Services participants.

2.3.1 Roles

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

The syntax of the **role** construct is:

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

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

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

2.3.2 Participants

A *Participant* identifies a set of related Roles. For example a Commercial Organization could take both a Buyer Role when purchasing goods and a Seller Role when selling them.

The syntax of the **participant** construct is:

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

2.3.3 Relationships

A *Relationship* is the association of two Roles for a purpose. A Relationship represents the possible ways in which two Roles can interact. For example the Relationships between a Buyer and a Seller could include:

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

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

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

The syntax of the **relationship** construct is:

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

A relationship MUST have exactly two role types defined.

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

2.3.4 Channels

A *Channel* realizes a point of collaboration between participants by specifying where and how to exchange information. Additionally, Channel information can be passed among participants. This allows modeling how the destination of messages is determined, statically and dynamically, when collaborating within a Choreography. For example, a Buyer could specify Channel information to be used for sending delivery information. The Buyer could then send the Channel information to the Seller who then forwards it to the Shipper. The Shipper could then send delivery information directly to the Buyer using the Channel information originally supplied by the Buyer.

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

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

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

One or more Channel(s) MAY be passed around from one Role to another. A Channel MAY restrict the types of Channel(s) allowed to be exchanged between the Web

510 Services participants, through this Channel. Additionally, a Channel MAY restrict its
511 usage by specifying the number of times a Channel can be used.
512
513 The syntax of the **channelType** construct is:
514

```
515          <channelType  name="ncname"
516              usage="once"|"unlimited"?
517              action="request-respond"|"request"|"respond"? >
518
519              <passing  channel="qname"
520                        action="request-respond"|
521                              "request"|"respond"?
522                        new="true"|"false"? />*
523
524              <role  type="qname"  behavior="ncname"? />
525
526              <reference>
527                 <token  type="qname"/>+
528              </reference>
529              <identity>
530                 <token  type="qname"/>+
531              </identity>*
532          </channelType>
```

533
534 The optional attribute usage is used to restrict the number of times a Channel can be
535 used.

536 The optional element passing restricts the Channel(s) allowed to be exchanged from one
537 Role to another Role, when using this Channel in an Interaction. In the case where the
538 operation used to exchange the Channel is of request-response type, then the attribute
539 action within the passing element defines if the Channel will be exchanged during the
540 request or during the response. The Channels exchanged can be used in subsequent
541 Interaction activities. If the element passing is missing then this Channel can be used for
542 exchanging business documents and all types of Channels without any restrictions.
543
544 The element role is used to identify the Role of a participant, being the target of an
545 Interaction, which is then used for statically determining where and how to send/receive
546 information to/into the participant.

547 The element reference is used for identifying the WSDL service reference of a participant,
548 being the target of an Interaction, which is then used for dynamically determining where
549 and how to send/receive information to/into the participant. The service reference of a
550 participant is distinguished by a set of Token types as specified by the token element
551 within the reference element.

552 The optional element identity MAY be used for identifying an instance of a business
553 process implementing the behavior of a participant and for identifying a logical
554 conversation between participants. The business process identity and the different
555 conversations are distinguished by a set of Token types as specified by the token
556 element within the identity element.
557
558

559 The example below shows the declaration of the Channel type RetailerChannel. The
560 Channel identifies the Role type tns:Retailer. The address of the Channel is specified in
561 reference element, whereas the business process instance can be identified using identity
562 element. The passing element allows ConsumerChannel to be sent over the
563 RetailerChannel.
564
```
565        <channelType name="RetailerChannel">
566            <passing channel="ConsumerChannel" action="request" />
567
568            <role type="tns:Retailer" behavior="retailerForConsumer"/>
569            <reference>
570                <token type="tns:retailerRef"/>
571            </reference>
572            <identity>
573                <token type="tns:purchaseOrderID"/>
574            </identity>
575        </channelType>
```
576
577
578 2.4 Information Driven Collaborations

579 A WS-CDL document allows defining information within a Choreography that can
580 influence the observable behavior of the collaborating participants.
581
582 *Variables* contain information about objects in the Choreography such as the messages
583 exchanged or the state of the Roles involved. *Tokens* are aliases that can be used to
584 reference parts of a *Variable*. Both *Variables* and *Tokens* have *Information Types* that
585 define the data structure of what the *Variable* or *Token* contains.
586
587 2.4.1 Information Types

588 Information types describe the type of information used within a Choreography. By
589 introducing this abstraction, a Choreography definition avoids referencing directly the
590 data types, as defined within a WSDL document or an XML Schema document.
591
592 The syntax of the **informationType** construct is:
593
```
594        <informationType  name="ncname"
595                          type="qname"? |
596                          element="qname"? />
```
597
598 The attributes type, and element describe the document to be an XML Schema type, or an
599 XML Schema element respectively. The document is of one of these types exclusively.
600
601 2.4.2 Variables

602 Variables capture information about objects in a Choreography as defined by the
603 *Variable Usage*:
604 • *Information Exchange Variables* that contain information such as an Order that is
605 used to:
606   • Populate the content of a message to be sent, or
607   • Populated as a result of a message received

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

The value of Variables:
- Is available to all the Roles by initializing them prior to the start of a Choreography
  - Common Variables that contain information that is common knowledge to two or more Roles, e.g. "OrderResponseTime" which is the time in hours in which a response to an Order must be sent
- Can be made available at a Role by populating them as a result of an Interaction
- Can be made available at a Role by assigning data from other information
  - Locally Defined Variables that contain information created and changed locally by a Role. They can be Information Exchange, State or Channel Variables as well as variables of other types. For example "Maximum Order Amount" could be data created by a seller that is used together with an actual order amount from an Order received to control the ordering of the Choreography. In this case how Maximum Order Amount is calculated and its value would not be known by the other Roles
- Can be used to determine the decisions and actions to be taken within a Choreography

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

649 The syntax of the **variableDefinitions** construct is:
650
651   `<variableDefinitions>`
652   `<variable    name="ncname"`
653   `informationType="qname"|channelType="qname"`
654   `mutable="true|false"?`
655   `free="true|false"?`
656   `silent-action="true|false"?`
657   `role="qname"? />+`
658   `</variableDefinitions>`
659
660 The declared variables can be of the following types:
661   • Information Exchange Variables, State Variables. The attribute informationType
662   describes the type of the variable
663   • Channel Variables. The attribute channelType describes the type of the Channel
664
665 The optional attribute mutable, when set to "false" describes that the variable information
666 when initialized, cannot change anymore.

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

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

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

676 The optional attribute role is used to specify the location at which the variable information
677 will reside.
678
679 The following rules apply to Variable Declarations:
680   • If a variable is declared without a Role, it is implied that it is declared at all the
681   Roles that are part of the Relationships of the Choreography. For example if
682   Choreography C1 has Relationship R that has a tuple (Role1, Role2), then a
683   variable x defined in Chreography C1 without a Role attribute means it is
684   declared at Role1 and Role2
685   • The variable with channelType MUST be declared without a role attribute
686

687 ### 2.4.2.1 Expressions

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

690 Predicate expressions are used in a Work Unit to specify its Guard condition.
691

692 The language used in WS-CDL for specifying expressions and query or conditional
693 predicates is XPath 1.0. Additionally, WS-CDL defines XPath function extensions as
694 described in Appendix B.
695
696 **2.4.3 Tokens**

697 A *Token* is an alias for a piece of data in a variable or message that needs to be used by
698 a Choreography. Tokens differ from Variables in that Variables contain values whereas
699 Tokens contain information that defines the piece of the data that is relevant. For
700 example a Token for "Order Amount" within an Order business could be an alias for an
701 expression that pointed to the Order Amount element within an XML document. This
702 could then be used as part of a condition that controls the ordering of a Choreography,
703 for example "Order Amount > $1000".
704
705 All Tokens MUST have a type, for example, an Order Amount would be of type amount,
706 Order Id could be alphanumeric and counter an integer.
707
708 Tokens types reference a document fragment within a Choreography definition and
709 Token Locators provide a query mechanism to select them. By introducing these
710 abstractions, a Choreography definition avoids depending on specific message parts, as
711 described by WSDL, or a specific query string, as specified by XPATH, but instead the
712 parts or the query string can change without affecting the Choreography definition.
713
714 The syntax of the **token** construct is:
715
```
716         <token  name="ncname"  informationType="qname" />
```
717
718 The attribute informationType identifies the type of the document fragment.
719
720
721 The syntax of the **tokenLocator** construct is:
722
```
723         <tokenLocator  tokenName="qname"
724               informationType="qname"
725               query="XPath-expression"? />
```
726
727 The attribute tokenName identifies the name of the token type that the document
728 fragment locator is associated with.

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

731 The attribute query defines the query string that is used to select a document fragment
732 within a document.
733
734
735 The example below shows that the token purchaseOrderID is of type xsd:int. The two
736 tokenLocators show how to access this token in "purchaseOrder" and
737 "purchaseOrderAck" messages.
738
```
739         <token name="purchaseOrderID" informationType="xsd:int"/>
```

```
740        <tokenLocator tokenName="tns:purchaseOrderID"
741            informationType="purchaseOrder" part="PO"
742            query="/PO/OrderId"/>

743        <tokenLocator tokenName="tns:purchaseOrderID"
744            informationType="purchaseOrderAck" part="POAck"
745            query="/POAck/OrderId"/>
746
```

2.4.4 Choreographies

A WS-CDL document explicitly prescribes the rules, agreed between Web Service participants, that govern the ordering of exhanged messages and the provisioning of alternative patterns of behavior. 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 *Choreography* allows constructing global compositions of Web Service participants by explicitly asserting their common and complementary observable behaviors.

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

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

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

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

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

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

784 The **Choreography-Notation** is used to define a root or a top-level Choreography.
785
786 The syntax is:
787
```
788        <choreography  name="ncname"
789                    complete="xsd:boolean XPath-expression"?
790                    isolation="dirty-write"|
791                            "dirty-read"|"serializable"?
792                    root="true"|"false"? >
793
794        <relationship  type="qname" />+
795
796        variableDefinitions?
797
798        Choreography-Notation*
799
800        Activity-Notation
801
802        <exception  name="ncname">
803            WorkUnit-Notation+
804        </exception>?
805        <finalizer  name="ncname">
806            WorkUnit-Notation
807        </finalizer>?
808     </choreography>
```
809
810
811 The optional complete attribute allows to explicitly complete a Choreography as
812 described below in the Choreography Life-line section.

813 The optional isolation attribute specifies when a variable information that is declared in an
814 enclosing and changed within an enclosed Choreography is visible to its enclosing and
815 sibling Choreographies:
816 • When isolation is set to "dirty-write", the variable information can be immediately
817   overwritten by actions in other Choreographies
818 • When isolation is set to "dirty-read", the variable information is immediately
819   visible to other Choreographies
820 • When isolation is set to "serializable", the variable information is visible to other
821   Choreographies only after this Choreography has ended successfully

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

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

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

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

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

832 The optional finalizer element defines the Finalizer block of a Choreography by specifying
833 one Finalizer Work Unit.
834
835 2.4.5 WorkUnits

836 A *Work Unit* prescribes the constraints that must be fulfilled for making progress within a
837 Choreography. Examples of a Work Unit include:
838 • A *Send PO* Work Unit that includes Interactions for the Buyer to send an Order,
839 the Supplier to acknowledge the order, and then later accept (or reject) the order.
840 This work unit would probably not have a Guard
841 • An *Order Delivery Error* Work Unit that is performed whenever the *Place Order*
842 Work Unit did not reach a "normal" conclusion. This would have a Guard
843 condition that identifies the error – see also Choreography Exceptions and
844 Transactions
845 • A *Change Order* Work Unit that can be performed whenever an order
846 acknowledgement message has been received and an order rejection has not
847 been received
848
849 A Work Unit can prescribe the explicit rules for enforcing the constraints that preserve
850 the consistency of the collaborations commonly performed between the Web Service
851 participants. Using a Work Unit an application can recover from faults that are the result
852 from abnormal actions and also finalize completed actions that need to be logically rolled
853 back.
854
855 A Work Unit specifies the data dependencies that must be satisfied before enabling one
856 or more enclosed actions. These dependencies express interest(s) on the availability of
857 variable information that already exists or will be created in the future.

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

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

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

867 A Work Unit that completes successfully MUST be considered again for matching
868 (based on its Guard condition), if its repetition condition evaluates to "true".
869
870 The **WorkUnit-Notation** is defined as follows:
871
```
872         <workunit  name="ncname"
873              guard="xsd:boolean XPath-expression"?
874              repeat="xsd:boolean XPath-expression"?
```

```
875              block="true|false" >

876
877             Activity-Notation

878        </workunit>

879
```

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

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

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

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

889 The WS-CDL functions, as described in Appendix B, MAY be used within a guard, and a
890 repeat condition.

891

892 **2.4.5.1 Reacting**

893 A Guard describes the Work Unit's interest for reacting on the availability of variable
894 information and on a constraint condition, including these variable information, being
895 satisfied.

896

897 The following rules apply when a Work Unit uses a Guard for reacting:
898 • When a Guard is not specified then the Work Unit always matches
899 • When a Guard is specified then
900   • When the block attribute is set to "false", then the Guard condition
901   assumes that the variable information is currently available. If either the
902   variable information is not available or the Guard condition evaluates to
903   "false", then the Work Unit matching fails and the Activity-Notation
904   enclosed within the Work Unit is skipped
905   • When the block attribute is set to "true" and one or more variable(s) are
906   not available, then the Work Unit MUST block waiting for the variable
907   information to become available. When the variable information specified
908   by the Guard condition become available then the Guard condition is
909   evaluated. If the Guard condition evaluates to "true", then the Work Unit is
910   matched. If the Guard condition evaluates to "false", then the Work Unit
911   matching fails and the Activity-Notation enclosed within the Work Unit is
912   skipped
913 • When the WS-CDL function isAligned() is used in the Guard, it means that the
914   Work Unit that specifies the Guard is waiting for an appropriate alignment
915   Interaction to happen between the two Roles. When the isAligned() WS-CDL
916   function is used in a Guard, then the Relationship within the isAligned() MUST be
917   the subset of the Relationship that the immediate enclosing Choreography
918   defines In the below example, the Guard specifies that the enclosed Work Unit is
919   waiting for an alignment Interaction to happen between the customer Role and
920   the retailer Role:

921

922   guard("cdl:isAligned("PurchaseOrder", PurchaseOrder",
923   "customer-retailer-relationship")")
924
925
926 The examples below demonstrate the possible use of a Work Unit:

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

928 In the following Work Unit, the Guard waits on the availability of POAcknowledgement at
929 customer Role and if it is already available, the activity happens, otherwise, the activity
930 waits until the variable POAcknowledgement is initialized at the customer Role.
931

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

938
939 b. Example of a Work Unit with block equals to "false":

940 In the following Work Unit, the Guard checks if StockQuantity at retailer Role is available
941 and is greater than 10 and if so, the activity happens. If either the Variable is not
942 available or the value is less than 10, the matching condition is "false" and the activity is
943 skipped.
944

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

951
2.4.6 Reusing existing Choreographies

953 Choreographies can be combined and built from other Choreographies.
954
2.4.6.1 Composing Choreographies

956 Choreography Composition is the creation of new Choreographies by reusing existing
957 Choreography definitions. For example if two separate Choreographies were defined as
958 follows:
959   • A Request for Quote (RFQ) Choreography that involved a Buyer Role sending a
960   request for a quotation for goods and services to a Supplier to which the Supplier
961   responding with either a "Quotation" or a "Decline to Quote" message, and
962   • An Order Placement Choreography where the Buyer placed and order for goods
963   or services and the Supplier either accepted the order or rejected it
964
965 You could then create a new "Quote and Order" Choreography by reusing the two where
966 the RFQ Choreography was executed first, and then, depending on the outcome of the
967 RFQ Choreography, the order was placed using the Order Placement Choreography.

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

970 • *Locally*, i.e. they are included, in the same Choreography definition as the
971   Choreography that performed them, or
972 • *Globally*, i.e. they are specified in a separate Choreography definition that is
973   defined elsewhere and performed in the root Choreography using perform
974   construct
975

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

979
980

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

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

```
986        <choreography name="PurchaseChoreo" root="true">
987           ...

988          <variable name="purchaseOrderAtRetailer"
989             informationType="purchaseOrder" role="Retailer"/>
990          ...

991          <choreography name="CustomerNotifyChoreo">
992          ...

993          </choreography>

994          <workunit name="RetailerNotifyCustomer"
995           guard="cdl:getVariable(PoAckFromWareHouse, tns:WareHouse)">

996               perform choreographyName="CustomerNotifyChoreo"

997          </workunit>
998          ...
999        </choreography> <!--end of root choreography -->
```

1000
1001 2.4.6.2 Importing Choreographies

1002 An *Importing* statement can contain references to a complete Choreography.

1003 Importing statements must be interpreted in the sequence they occur.

1004 When the Import statement contains references to variables or other data that have the
1005 same identity, then the content of the later Import statement replaces the same content
1006 referenced by the earlier Import statement. It also enables one Choreography definition
1007 to effectively be "cloned" by replacing the definitions for some or all of its variables.

1008

1009 The *importDefinitions* construct allows reusing Choreography types defined in another
1010 Choreography document such as Token types, Token Locator types, Information Types,
1011 Role types, Relationship types, Channel types and Choreographies.

1012 In addition, WSDL documents can be imported and their definitions reused.

The syntax of the **importDefinitions** construct is:

```
<importDefinitions>

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

</importDefinitions>
```

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

2.4.7 Choreography Life-line

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

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

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

2.4.8 Choreography Recovery

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

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

2.4.8.1 Exception Block

A Choreography can sometimes fail as a result of an exceptional circumstance or error. Different types of exceptions are possible including this non-exhaustive list:
- *Interaction Failures*, for example the sending of a message did not succeed
- *Protocol Based Exchange failures*, for example no acknowledgement was received as part of a reliable messaging protocol [22]
- *Security failures*, for example a Message was rejected by a recipient because the digital signature was not valid

- *Timeout errors*, for example an Interaction did not complete within a required timescale
- *Validation Errors*, for example an XML order document was not well formed or did not conform to its schema definition
- *Application "failures"*, for example the goods ordered were out of stock

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

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

A fault variable information is a result of:
- A fault occurring while performing an Interaction between participants
- A timeout occuring while an Interaction between participants was not completed within a specified time period

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

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

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

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

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

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

2.4.8.2 Finalizer Block

1107 When a Choreography encounters an exceptional condition it MAY need to revert the
1108 actions it had already completed, by providing finalization actions that semantically
1109 rollback the effects of the completed actions. To handle these a separate Finalizer Work
1110 Unit is defined in the Finalizer Block of a Choreography.

1111 A Choreography MAY define one Finalizer Work Unit.

1112 A Finalizer WorkUnit is enabled only after its enclosing Choreography completes
1113 successfully. The Finalizer Work Unit may be enabled only once for an enclosing
1114 Choreography.

1115 The actions within the Finalizer Work Unit MAY use variable information visible in the
1116 Visibility Horizon of its enclosing Choreography as they were at the time the enclosing
1117 Choreography completed or as they stand at the current time.

1118 The actions of the Finalizer Work Unit MAY fault. The semantics for matching the fault
1119 and acting on it are the same as described in the previous section.
1120

2.5 Activities

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

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

2.5.1 Ordering Structures

An **Ordering Structure** is one of the following:
- *Sequence*
- *Parallel*
- *Choice*

2.5.1.1 Sequence

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

The syntax of this construct is:

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

2.5.1.2 Parallel

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

The syntax of this construct is:

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

2.5.1.3 Choice

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

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

The syntax of this construct is:

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

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

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

2.5.2 Interaction

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

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

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

1211	• A One-Way Interaction that involves the sending of single message
1212	• A Request-Response Interaction when two messages are exchanged
1213	
1214	An Interaction also contains "references" to:
1215	• The From Role and To Role that are involved
1216	• The Message Content Type that is being exchanged
1217	• The Information Exchange Variables at the From Role and To Role that are the
1218	source and destination for the Message Content
1219	• The Channel Variable that specifies the interface and other data that describe
1220	where and how the message is to be sent
1221	• The Operation that specifies what the recipient of the message should do with
1222	the message when it is received
1223	• A list of potential States Changes that can occur and may be aligned at the From
1224	Role and the To Role as a result of carrying out the Interaction
1225	

2.5.2.1 Interaction State Changes

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

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

1. Buyer.OrderState = Sent, Seller.OrderState = Received

2. Buyer.OrderState = SendFailure, Seller.OrderState not set

3. Buyer.OrderState = AckReceived, Seller.OrderState = OrderAckSent

2.5.2.2 Interaction Based Information Alignment

In some Choreographies there may be a requirement that, at the end of an Interaction, the Roles in the Choreography have agreement of the outcome.
- More specifically within an Interaction, a Role may need to have a common understanding of the state creations/changes of one or more *State Variables* that are complementary to one or more *State Variables* of its partner Role
- Additionally within an Interaction, a Role may need to have a common understanding of the values of the *Information Exchange Variables* at the partner Role

With Interaction Alignment both the Buyer and the Seller have a common understanding that:
- State variables such as "Order State" variables at the Buyer and Seller, that have values that are complementary to each other, e.g. Sent at the Buyer and Received at the Seller, and

1255 • Information Exchange Variables that have the same types with the same content,
1256   e.g. The Order variables at the Buyer and Seller have the same Information
1257   Types and hold the same order information
1258
1259 In WS-CDL an alignment Interaction MUST be explicitly used, in the cases where two
1260 interacting participants require the alignment of their States or their exchanged
1261 information between them. After the alignment Interaction completes, both participants
1262 progress at the same time, in a lock-step fashion and the variable information in both
1263 participants is aligned. Their variable alignment comes from the fact that the requesting
1264 participant has to know that the accepting participant has received the message and the
1265 other way around, the accepting participant has to know that the requesting participant
1266 has sent the message before both of them progress. There is no intermediate variable,
1267 where one participant sends a message and then it proceeds independently or the other
1268 participant receives a message and then it proceeds independently.
1269

2.5.2.3 Protocol Based Information Exchanges

1271 The one-Way, request or response messages in an Interaction may also be
1272 implemented using a *Protocol Based Exchange* where a series of messages are
1273 exchanged according to some well-known protocol, such as the reliable messaging
1274 protocols defined in specifications such as WS-Reliability [22].
1275
1276 In both cases, the same or similar Message Content may be exchanged as in a simple
1277 Interaction, for example the sending of an Order between a Buyer and a Seller.
1278 Therefore some of the same State Changes may result.

1279 However when protocols such as the reliable messaging protocols are used, additional
1280 State Changes will occur. For example, if a reliable messaging protocol were being used
1281 then the Buyer, once confirmation of delivery of the message was received, would also
1282 know that the Seller's "Order State" variable was in the state "Received" even though
1283 there was no separate Interaction that described this.
1284

2.5.2.4 Interaction Life-line

1286 The Channel through which an Interaction occurs is used to determine whether to enlist
1287 the Interaction with an already initiated Choreography or to initiate a new Choreography.

1288 Within a Choreography, two or more related Interactions MAY be grouped to form a
1289 logical conversation. The Channel through which an Interaction occurs is used to
1290 determine whether to enlist the Interaction with an already initiated conversation or to
1291 initiate a new conversation.
1292

1293 An Interaction completes normally when the request and the response (if there is one)
1294 complete successfully. In this case the business documents and Channels exchanged
1295 during the request and the response (if there is one) result in the exchanged variable
1296 information being aligned between the two participants.

1297 An Interaction completes abnormally if the following faults occur:
1298 • The time-to-complete timeout identifies the timeframe within which an Interaction
1299   MUST complete. If this timeout occurs, after the Interaction was initiated but
1300   before it completed, then a fault is generated
1301 • A fault signals an exception condition during the management of a request or
1302   within a participant when accepting the request

1303 In these cases the variable information remain the same at the both Roles as if this
1304 Interaction had never occurred.
1305
1306 The syntax of the **interaction** construct is:
1307
```
1308        <interaction  name="ncname"
1309               channelVariable="qname"
1310               operation="ncname"
1311               time-to-complete="xsd:duration"?
1312               align="true"|"false"?
1313               initiateChoreography="true"|"false"? >

1315          <participate  relationship="qname"
1316                        fromRole="qname"
1317                        toRole="qname" />

1319          <exchange
1320             messageContentType="qname"
1321             action="request"|"respond" >

1323             <use        variable="XPath-expression"/>
1324             <populate  variable="XPath-expression"/>
1325          </exchange>*


1327          <record  name="ncname"
1328                  role="qname"
1329                  action="request"|"respond" >
1330             <source    variable="XPath-expression" />
1331             <target    variable="XPath-expression" />
1332          </record>*
1333       </interaction>
```
1334

1335 The channel attribute specifies the Channel variable containing information of a
1336 participant, being the target of an Interaction, which is used for determining where and
1337 how to send/receive information to/into the participant. The Channel variable is used in
1338 an Interaction MUST be available at the two Roles before the Interaction occurs.

1339 At runtime, information about a Channel variable is expanded further. This requires that
1340 the messages in the Choreography also contain correlation information, for example by
1341 including:
1342 • A SOAP header that specifies the correlation data to be used with the Channel,
1343   or
1344 • Using the actual value of data within a message, for example the Order Number
1345   of the Order that is common to all the messages sent over the Channel

1346 In practice, when a Choreography is performed, several different ways of doing
1347 correlation may be employed which vary depending on the Channel Type.
1348
1349 The attribute operation specifies a one-way or a request-response WSDL 2.0 operation
1350 that is the target for the service request/acceptance. The specified operation belongs to
1351 the WSDL interface, identified by the role element of the Channel used in the Interaction
1352 activity.

1353 The optional time-to-complete attribute identifies the timeframe within which an Interaction
1354 MUST complete.

1355 The optional align when set to "true" means that the Interaction results in the common
1356 understanding of the messages exchanged and the resulting complementary state
1357 changes/state creation at both endpoints specified in fromRole and toRole.

1358 An Interaction activity can be marked as a Choreography initiator when the optional
1359 initiateChoreography attribute is set to "true".

1360 Within the participate element, the relationship attribute specifies the Relationship this
1361 Choreography participates in and the fromRole and toRole attributes specify the
1362 requesting and the accepting Roles respectively.
1363
1364 The exchange element allows one or two messages to be exchanged during a one-way
1365 request or a request/response Interaction. When the exchange is missing, it means that
1366 there was no message exchange that populated new variable information at a Role.
1367
1368 The messageContentType attribute of the exchange element identifies the
1369 informationType or the channelType of the information that is exchanged between the
1370 Roles and the *Information Exchange Variables* used as follows:
1371   • *One Way From Message* is the variable that is the source for a One-Way
1372   Message at the *From Role*
1373   • *One Way To Message* is the variable that is the destination for a One-Way
1374   Message at the *To Role*
1375   • *Request From Message* is the variable that is the source for Request Message at
1376   the *From Role*
1377   • *Request To Message* is the variable that is the destination for Request Message
1378   at the *To Role*
1379   • *Response To Message* is the variable that is the source for Response Message
1380   at the *To Role*
1381   • *Response From Message* is the variable that is the destination for Response
1382   Message at the *From Role*
1383
1384 The attribute action specifies the direction of Message Exchange that is performed in the
1385 Interaction. A Request message exchange happens fromRole to toRole and a respond
1386 message exchange happens from toRole to fromRole.
1387
1388 The attributes use and populate describe the message at the source and the destination
1389 respectively:
1390   • Both use and populate MUST be of type as described in the
1391   messageContentType element
1392   • The attribute use MUST be a variable at the fromRole and the attribute populate
1393   MUST be a variable at the toRole, when the action element is set to "request"

1394 • The attribute use MUST be a variable at the toRole and the attribute populate
1395   MUST be a variable at the fromRole, when the action element is set to "respond"
1396
1397 The element record is used to create/change one or more states at both the Roles at the
1398 ends of the Interaction. For example, the PurchaseOrder message contains the Channel
1399 of the Role "customer" when sent to the Role "Retailer". This can be copied into the
1400 appropriate state variable of the "Retailer" within the record element. When align is set to
1401 "true" for the Interaction, it also means that the Customer knows that the Retailer now has
1402 the address of the Customer.

1403 Another usecase of the record element is that it can be used to record the states at each
1404 Role. The Customer sets the state "OrderSent" to "true" and the Retailer sets the state
1405 "OrderReceived" to "true" at the end of the request part of the Interaction. Similarly the
1406 Customer sets "OrderAcknowledged" "true" at the end of the Interaction.
1407
1408 The source and the target elements within record element represent the variable names
1409 at the Role that is specified in the role attribute of the record element.
1410
1411
1412 The example below shows a complete Choreography that involves one interaction. The
1413 interaction happens from Role Consumer to Role Retailer on the Channel "retailer-
1414 channel" as a request/response message exchange.
1415 • The message purchaseOrder is sent from Consumer to Retailer as a request
1416   message
1417 • The message purchaseOrderAck is sent from Retailer to Consumer as a
1418   response message
1419 • The variable consumer-channel is populated at Retailer at the end of the request
1420   using the record element
1421 • The Interaction happens on the retailer-channel which has a token
1422   purchaseOrderID used as an identity of the channel. This identity element is
1423   used to identify the business process of the retailer
1424 • The request message purchaseOrder contains the identity of the retailer
1425   business process as specified in the tokenLocator for purchaseOrder message
1426 • The response message purchaseOrderAck contains the identity of the consumer
1427   business process as specified in the tokenLocator for purchaseOrderAck
1428   message
1429 • The consumer-channel is sent as a part of purchaseOrder message from
1430   consumer to retailer on retailer-channel during the request. The record element
1431   populates the consumer-channel at the retailer role
1432
1433

```
1434     <package name="ConsumerRetailerChoreo" version="1.0"
1435
1436     <informationType name="purchaseOrderType"
1437             type="pons:PurchaseOrderMsg"/>
1438      <informationType name="purchaseOrderAckType"
1439             type="pons:PurchaseOrderAckMsg"/>
1440
1441     <token name="purchaseOrderID" informationType="tns:intType"/>
1442     <token name="retailerRef" informationType="tns:uriType"/>
1443
```

```
1444          <tokenLocator tokenName="tns:purchaseOrderID"
1445          informationType="tns:purchaseOrderType" query="/PO/orderId"/>
1446          <tokenLocator tokenName="tns:purchaseOrderID"
1447          informationType="tns:purchaseOrderAckType" query="/PO/orderId"/>
1448
1449          <role name="Consumer">
1450              <behavior name="consumerForRetailer"
1451                  interface="cns:ConsumerRetailerPT"/>
1452            <behavior name="consumerForWarehouse"
1453                  interface="cns:ConsumerWarehousePT"/>
1454          </role>
1455
1456          <role name="Retailer">
1457              <behavior name="retailerForConsumer"
1458                  interface="rns:RetailerConsumerPT"/>
1459          </role>
1460
1461          <relationship name="ConsumerRetailerRelationship">
1462              <role type="tns:Consumer" behavior="consumerForRetailer"/>
1463              <role type="tns:Retailer" behavior="retailerForConsumer"/>
1464          </relationship>
1465
1466          <channelType name="ConsumerChannel">
1467              <role type="tns:Consumer"/>
1468              <reference>
1469                  <token type="tns:consumerRef"/>
1470              </reference>
1471              <identity>
1472                  <token type="tns:purchaseOrderID"/>
1473              </identity>
1474          </channelType>
1475
1476          <channelType name="RetailerChannel">
1477              <passing channel="ConsumerChannel" action="request" />
1478
1479              <role type="tns:Retailer" behavior="retailerForConsumer"/>
1480              <reference>
1481                  <token type="tns:retailerRef"/>
1482              </reference>
1483              <identity>
1484                  <token type="tns:purchaseOrderID"/>
1485              </identity>
1486          </channelType>
1487
1488          <choreography name="ConsumerRetailerChoreo" root="true">
1489              <relationship type="tns:ConsumerRetailerRelationship"/>
1490
1491            <variableDefinitions>
1492                <variable name="purchaseOrder"
1493                    informationType="tns:purchaseOrderType" silent-
1494                    action="true" />
1495                <variable name="purchaseOrderAck"
1496                    informationType="tns:purchaseOrderAckType" />
1497                <variable name="retailer-channel"
1498                    channelType="tns:RetailerChannel"/>
1499                 <variable name="consumer-channel"
1500                    channelType="tns:ConsumerChannel"/>
```

```
1501
1502
1503            <interaction channelVariable="tns:retailer-channel "
1504               operation="handlePurchaseOrder" align="true"
1505               initiateChoreography="true">
1506             <participate
1507              relationship="tns:ConsumerRetailerRelationship"
1508              fromRole="tns:Consumer" toRole="tns:Retailer"/>
1509             <exchange messageContentType="tns:purchaseOrderType"
1510             action="request">
1511            <use variable="cdl:getVariable(tns:purchaseOrder,
1512               tns:Consumer)"/>
1513            <populate variable="cdl:getVariable(tns:purchaseOrder,
1514               tns:Retailer)"/>
1515            </exchange>
1516            <exchange messageContentType="purchaseOrderAckType"
1517               action="respond">
1518            <use variable="cdl:getVariable(tns:purchaseOrderAck,
1519               tns:Retailer)"/>
1520            <populate variable="cdl:getVariable(tns:purchaseOrderAck,
1521               tns:Consumer)"/>
1522              </exchange>
1523            <record role="tns:Retailer" action="request">
1524              <source variable="cdl:getVariable(tns:purchaseOrder,
1525                  PO/CustomerRef, tns:Retailer)"/>
1526            <target variable="cdl:getVariable(tns:consumer-channel,
1527                  tns:Retailer)"/>
1528            </record>
1529          </interaction>
1530        </choreography>
1531      </package>
1532
1533
```

2.5.3 Performed Choreography

The Performed Choreography activity enables a Choreography to define that a
separately defined Choreography is to be performed as an enclosed Choreography. The
Choreography that is performed can be defined either within the same Choreography
Definition or separately.

The syntax of the **perform** construct is:

```
         <perform  choreographyName="qname">

             <alias  name="ncname">

                  <this  variable="XPath-expression" role="qname" />

                  <free  variable="XPath-expression" role="qname" />

             </alias>+

         </perform>
```

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

1552 performed Choreography defined in a different package is conceptually treated as an
1553 enclosed Choreography.

1554 The alias element within the perform helps in aliasing the variables from the performing
1555 Choreography to the performed Choreography. The role attribute aliases the Roles from
1556 the performing Choreography to the performed Choreography.
1557
1558 The following rules apply on the performed Choreography:
1559 • The Choreography to be performed MUST NOT be a root Choreography
1560 • The Choreography to be performed MUST be declared either using a
1561 Choreography-Notation in the same Choreography or it MUST be a top-level
1562 Choreography with root attribute set to "false" in the same or different
1563 Choreography package
1564 • If the performed Choreography is defined within the performing Choreography,
1565 the variables that are in the visibility horizon are visible to the performed
1566 Choreography also
1567 • Performed Choreography, if not defined within the enclosing Choreography, can
1568 be used by other Choreographies and hence the contract is reusable
1569 • There should not be a cyclic dependency on the Choreographies performed. For
1570 example Choreography C1 is performing Choreography C2 which is performing
1571 Choreography C1 again
1572
1573
1574 The example below shows a Choreography performing another Choreography:
1575
1576 The root Choreography "PurchaseChoreo" performs the Choreography
1577 "RetailerWarehouseChoreo" and aliases the variable "purchaseOrderAtRetailer" defined
1578 in the enclosing Choreography to "purchaseOrder" defined at the performed enclosed
1579 Choreography "RetailerWarehouseChoreo". Once aliased, the visibility horizon of the
1580 variable purchaseOrderAtRetailer is the same as it would be for the enclosed
1581 Choreography.
1582

```
1583      <choreography name="PurchaseChoreo" root="true">
1584        ...
1585        <variable name="purchaseOrderAtRetailer"
1586          informationType="purchaseOrder" role="Retailer"/>
1587        ...
1588        <perform choreographyName="RetailerWarehouseChoreo">
1589        <alias>
1590          <this variable "cdl:getVariable(tns:purchaseOrder,
1591                     tns:Retailer)"/>
1592          <free variable="cdl:getVariable(tns:purchaseOrder,
1593                        rwns:Retailer)"/>
1594        </alias>
1595        ...
1596     </choreography>
1597
```

2.5.4. Assigning Variables

Assign makes available, within one Role, the value of one Variable using the value of a Variable or Token.

The assignments may include:

- Assigning one *Information Exchange Variable* to another or a part of the *Information Exchange Variable* to another variable so that a message received can be used to trigger/constrain, using a Work Unit Guard, other Interactions
- Assigning a *Locally Defined Variable* to part of the data contained in an Information Exchange Variable

The syntax of the **assign** construct is:

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

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

The following rules apply to assignment:

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

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

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

2.5.5 Actions with non-observable effects

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

1642   The syntax of the **noaction** construct is:

1643

1644   `<noaction/>`

1645

3 Example

4. Relationship with the Security framework

Because messages can have consequences in the real world, the collaboration participants 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 participants. 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 Web Service participants make progress by interacting. In the cases where two interacting participants require the alignment of their States or their exchanged information between them, an alignment Interaction is modeled in a Choreography. After the alignment Interaction completes, both participants progress at the same time, in a lock-step fashion. The variable information alignment comes from the fact that the requesting participant has to know that the accepting participant has received the message and the other way around, the accepting participant has to know that the requesting participant has sent the message before both of them progress. There is no intermediate variable, where one participant sends a message and then it proceeds independently or the other participant 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 participants can be reached even in the case of failures and loss of messages.

1681 **7. Acknowledgments**

1682

8. References

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

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

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

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

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

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

[7] Web Services Definition Language (WSDL) 2.0

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

[9] W3C Recommendation "The XML Specification"

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

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

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

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

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

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

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

[16] XLANG: Web Services for Business Process Design

[17] WSFL: Web Service Flow Language 1.0

[18] BPEL: Business Process Execution Language 1.1

[19] BPML: Business Process Modeling Language 1.0

[20] XPDL: XML Processing Description Language 1.0

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

[22] Web Services Reliability 1.0

[23] The Java Language Specification

[24] Web Services Security

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

1713   [26] ECMA. 2001. Standard ECMA-334: C# Language Specification

1714

A. WS-CDL XSD Schemas

```
1716  <?xml version="1.0" encoding="UTF-8"?>
1717  <schema
1718       targetNamespace=http://www.w3.org/ws/choreography/2004/02/WSCDL/
1719       xmlns=http://www.w3.org/2001/XMLSchema
1720       xmlns:cdl=http://www.w3.org/ws/choreography/2004/02/WSCDL/
1721       elementFormDefault="qualified">
1722
1723     <complexType name="tExtensibleElements">
1724        <annotation>
1725           <documentation>
1726              This type is extended by other CDL component types to allow
1727              elements and attributes from other namespaces to be added.
1728              This type also contains the optional description element that
1729              is applied to all CDL constructs.
1730           </documentation>
1731        </annotation>
1732        <sequence>
1733           <element name="description" minOccurs="0">
1734              <complexType mixed="true">
1735                 <sequence minOccurs="0" maxOccurs="unbounded">
1736                    <any processContents="lax"/>
1737                 </sequence>
1738              </complexType>
1739           </element>
1740           <any namespace="##other" processContents="lax"
1741                minOccurs="0" maxOccurs="unbounded"/>
1742        </sequence>
1743        <anyAttribute namespace="##other" processContents="lax"/>
1744     </complexType>
1745
1746     <element name="package" type="cdl:tPackage"/>
1747
1748     <complexType name="tPackage">
1749        <complexContent>
1750           <extension base="cdl:tExtensibleElements">
1751              <sequence>
1752                 <element name="importDefinitions"
1753                          type="cdl:tImportDefinitions" minOccurs="0"
1754                          maxOccurs="unbounded"/>
1755                 <element name="informationType" type="cdl:tInformationType"
1756                          minOccurs="0" maxOccurs="unbounded"/>
1757                 <element name="token" type="cdl:tToken" minOccurs="0"
1758                          maxOccurs="unbounded"/>
1759                 <element name="tokenLocator" type="cdl:tTokenLocator"
1760                          minOccurs="0" maxOccurs="unbounded"/>
1761                 <element name="role" type="cdl:tRole" minOccurs="0"
1762                          maxOccurs="unbounded"/>
1763                 <element name="relationship" type="cdl:tRelationship"
1764                          minOccurs="0" maxOccurs="unbounded"/>
1765                 <element name="participant" type="cdl:tParticipant"
1766                          minOccurs="0" maxOccurs="unbounded"/>
1767                 <element name="channelType" type="cdl:tChannelType"
```

```
1768                              minOccurs="0" maxOccurs="unbounded"/>
1769                 <element name="choreography" type="cdl:tChoreography"
1770                              minOccurs="0" maxOccurs="unbounded"/>
1771            </sequence>
1772            <attribute name="name" type="NCName" use="required"/>
1773            <attribute name="author" type="string" use="optional"/>
1774            <attribute name="version" type="string" use="required"/>
1775            <attribute name="targetNamespace" type="anyURI"
1776                      use="required"/>
1777         </extension>
1778       </complexContent>
1779    </complexType>
1780
1781    <complexType name="tImportDefinitions">
1782       <complexContent>
1783          <extension base="cdl:tExtensibleElements">
1784             <sequence>
1785                <element name="import" type="cdl:tImport"
1786                             maxOccurs="unbounded"/>
1787             </sequence>
1788          </extension>
1789       </complexContent>
1790    </complexType>
1791
1792    <complexType name="tImport">
1793       <complexContent>
1794          <extension base="cdl:tExtensibleElements">
1795             <attribute name="namespace" type="anyURI" use="required"/>
1796             <attribute name="location" type="anyURI" use="required"/>
1797          </extension>
1798       </complexContent>
1799    </complexType>
1800
1801    <complexType name="tInformationType">
1802       <complexContent>
1803          <extension base="cdl:tExtensibleElements">
1804             <attribute name="name" type="NCName" use="required"/>
1805             <attribute name="type" type="QName" use="optional"/>
1806             <attribute name="element" type="QName" use="optional"/>
1807          </extension>
1808       </complexContent>
1809    </complexType>
1810
1811    <complexType name="tToken">
1812       <complexContent>
1813          <extension base="cdl:tExtensibleElements">
1814             <attribute name="name" type="NCName" use="required"/>
1815             <attribute name="informationType" type="QName"
1816                      use="required"/>
1817          </extension>
1818       </complexContent>
1819    </complexType>
1820
1821    <complexType name="tTokenLocator">
1822       <complexContent>
1823          <extension base="cdl:tExtensibleElements">
1824             <attribute name="tokenName" type="QName" use="required"/>
```

```
1825              <attribute name="informationType" type="QName"
1826                          use="required"/>
1827              <attribute name="query" type="cdl:tXPath-expr"
1828                          use="optional"/>
1829          </extension>
1830        </complexContent>
1831    </complexType>
1832
1833    <complexType name="tRole">
1834        <complexContent>
1835          <extension base="cdl:tExtensibleElements">
1836              <sequence>
1837                 <element name="behavior" type="cdl:tBehavior"
1838                          maxOccurs="unbounded"/>
1839              </sequence>
1840              <attribute name="name" type="NCName" use="required"/>
1841          </extension>
1842        </complexContent>
1843    </complexType>
1844
1845    <complexType name="tBehavior">
1846        <complexContent>
1847          <extension base="cdl:tExtensibleElements">
1848             <attribute name="name" type="NCName" use="required"/>
1849             <attribute name="interface" type="QName" use="optional"/>
1850          </extension>
1851        </complexContent>
1852    </complexType>
1853
1854    <complexType name="tRelationship">
1855        <complexContent>
1856          <extension base="cdl:tExtensibleElements">
1857              <sequence>
1858                 <element name="role" type="cdl:tRoleRef" minOccurs="2"
1859                          maxOccurs="2"/>
1860              </sequence>
1861              <attribute name="name" type="NCName" use="required"/>
1862          </extension>
1863        </complexContent>
1864    </complexType>
1865
1866    <complexType name="tRoleRef">
1867        <complexContent>
1868          <extension base="cdl:tExtensibleElements">
1869             <attribute name="type" type="QName" use="required"/>
1870             <attribute name="behavior" type="NCName" use="required"/>
1871          </extension>
1872        </complexContent>
1873    </complexType>
1874
1875    <complexType name="tParticipant">
1876        <complexContent>
1877          <extension base="cdl:tExtensibleElements">
1878              <sequence>
1879                 <element name="role" type="cdl:tRoleRef2"
1880                          maxOccurs="unbounded"/>
1881              </sequence>
```

```
1882              <attribute name="name" type="NCName" use="required"/>
1883            </extension>
1884          </complexContent>
1885      </complexType>
1886
1887      <complexType name="tRoleRef2">
1888          <complexContent>
1889            <extension base="cdl:tExtensibleElements">
1890              <attribute name="type" type="QName" use="required"/>
1891            </extension>
1892          </complexContent>
1893      </complexType>
1894
1895      <complexType name="tChannelType">
1896          <complexContent>
1897            <extension base="cdl:tExtensibleElements">
1898              <sequence>
1899                <element name="passing" type="cdl:tPassing" minOccurs="0"
1900                          maxOccurs="unbounded"/>
1901                <element name="role" type="cdl:tRoleRef3"/>
1902                <element name="reference" type="cdl:tReference"/>
1903                <element name="identity" type="cdl:tIdentity" minOccurs="0"
1904                          maxOccurs="unbounded"/>
1905              </sequence>
1906              <attribute name="name" type="NCName" use="required"/>
1907              <attribute name="usage" type="cdl:tUsage" use="optional"
1908                          default="unlimited"/>
1909              <attribute name="action" type="cdl:tAction" use="optional"
1910                          default="request-respond"/>
1911            </extension>
1912          </complexContent>
1913      </complexType>
1914
1915      <complexType name="tRoleRef3">
1916          <complexContent>
1917            <extension base="cdl:tExtensibleElements">
1918              <attribute name="type" type="QName" use="required"/>
1919              <attribute name="behavior" type="NCName" use="optional"/>
1920            </extension>
1921          </complexContent>
1922      </complexType>
1923
1924      <complexType name="tPassing">
1925          <complexContent>
1926            <extension base="cdl:tExtensibleElements">
1927              <attribute name="channel" type="QName" use="required"/>
1928              <attribute name="action" type="cdl:tAction" use="optional"
1929                          default="request-respond"/>
1930              <attribute name="new" type="boolean" use="optional"
1931                          default="true"/>
1932            </extension>
1933          </complexContent>
1934      </complexType>
1935
1936      <complexType name="tReference">
1937          <complexContent>
1938            <extension base="cdl:tExtensibleElements">
```

```
1939            <sequence>
1940               <element name="token" type="cdl:tTokenReference"
1941                          maxOccurs="unbounded"/>
1942            </sequence>
1943         </extension>
1944      </complexContent>
1945   </complexType>
1946
1947   <complexType name="tTokenReference">
1948      <complexContent>
1949         <extension base="cdl:tExtensibleElements">
1950            <attribute name="name" type="QName" use="required"/>
1951         </extension>
1952      </complexContent>
1953   </complexType>
1954
1955   <complexType name="tIdentity">
1956      <complexContent>
1957         <extension base="cdl:tExtensibleElements">
1958            <sequence>
1959               <element name="token" type="cdl:tTokenReference"
1960                          maxOccurs="unbounded"/>
1961            </sequence>
1962         </extension>
1963      </complexContent>
1964   </complexType>
1965
1966   <complexType name="tChoreography">
1967      <complexContent>
1968         <extension base="cdl:tExtensibleElements">
1969            <sequence>
1970               <element name="relationship" type="cdl:tRelationshipRef"
1971                          maxOccurs="unbounded"/>
1972               <element name="variableDefinitions"
1973                          type="cdl:tVariableDefinitions" minOccurs="0"/>
1974               <element name="choreography" type="cdl:tChoreography"
1975                          minOccurs="0" maxOccurs="unbounded"/>
1976               <group ref="cdl:activity"/>
1977               <element name="exception" type="cdl:tException"
1978                          minOccurs="0"/>
1979               <element name="finalizer" type="cdl:tFinalizer"
1980                          minOccurs="0"/>
1981            </sequence>
1982            <attribute name="name" type="NCName" use="required"/>
1983            <attribute name="complete" type="cdl:tBoolean-expr"
1984                          use="optional"/>
1985            <attribute name="isolation" type="cdl:tIsolation"
1986                          use="optional" default="dirty-write"/>
1987            <attribute name="root" type="boolean" use="optional"
1988                          default="false"/>
1989         </extension>
1990      </complexContent>
1991   </complexType>
1992
1993   <complexType name="tRelationshipRef">
1994      <complexContent>
1995         <extension base="cdl:tExtensibleElements">
```

```
1996             <attribute name="type" type="QName" use="required"/>
1997           </extension>
1998         </complexContent>
1999      </complexType>
2000
2001      <complexType name="tVariableDefinitions">
2002         <complexContent>
2003            <extension base="cdl:tExtensibleElements">
2004               <sequence>
2005                  <element name="variable" type="cdl:tVariable"
2006                           maxOccurs="unbounded"/>
2007               </sequence>
2008            </extension>
2009         </complexContent>
2010      </complexType>
2011
2012      <complexType name="tVariable">
2013         <complexContent>
2014            <extension base="cdl:tExtensibleElements">
2015               <attribute name="name" type="NCName" use="required"/>
2016               <attribute name="informationType" type="QName"
2017                           use="optional"/>
2018               <attribute name="channelType" type="QName" use="optional"/>
2019               <attribute name="mutable" type="boolean" use="optional"
2020                           default="true"/>
2021               <attribute name="free" type="boolean" use="optional"
2022                           default="false"/>
2023               <attribute name="silent-action" type="boolean" use="optional"
2024                           default="false"/>
2025               <attribute name="role" type="QName" use="optional"/>
2026            </extension>
2027         </complexContent>
2028      </complexType>
2029
2030      <group name="activity">
2031        <choice>
2032          <element name="sequence" type="cdl:tSequence"/>
2033          <element name="parallel" type="cdl:tParallel"/>
2034          <element name="choice" type="cdl:tChoice"/>
2035          <element name="workunit" type="cdl:tWorkunit"/>
2036          <element name="interaction" type="cdl:tInteraction"/>
2037          <element name="perform" type="cdl:tPerform"/>
2038          <element name="assign" type="cdl:tAssign"/>
2039          <element name="noaction" type="cdl:tNoaction"/>
2040        </choice>
2041      </group>
2042
2043      <complexType name="tSequence">
2044         <complexContent>
2045            <extension base="cdl:tExtensibleElements">
2046               <sequence>
2047                  <group ref="cdl:activity" maxOccurs="unbounded"/>
2048               </sequence>
2049            </extension>
2050         </complexContent>
2051      </complexType>
2052
```

```
2053        <complexType name="tParallel">
2054          <complexContent>
2055            <extension base="cdl:tExtensibleElements">
2056              <sequence>
2057                <group ref="cdl:activity" maxOccurs="unbounded"/>
2058              </sequence>
2059            </extension>
2060          </complexContent>
2061        </complexType>
2062
2063        <complexType name="tChoice">
2064          <complexContent>
2065            <extension base="cdl:tExtensibleElements">
2066              <sequence>
2067                <group ref="cdl:activity" maxOccurs="unbounded"/>
2068              </sequence>
2069            </extension>
2070          </complexContent>
2071        </complexType>
2072
2073        <complexType name="tWorkunit">
2074          <complexContent>
2075            <extension base="cdl:tExtensibleElements">
2076              <sequence>
2077                <group ref="cdl:activity"/>
2078              </sequence>
2079              <attribute name="name" type="NCName" use="required"/>
2080              <attribute name="guard" type="cdl:tBoolean-expr"
2081                         use="optional"/>
2082              <attribute name="repeat" type="cdl:tBoolean-expr"
2083                         use="optional"/>
2084              <attribute name="block" type="boolean" use="required"/>
2085            </extension>
2086          </complexContent>
2087        </complexType>
2088
2089        <complexType name="tPerform">
2090          <complexContent>
2091            <extension base="cdl:tExtensibleElements">
2092              <sequence>
2093                <element name="alias" type="cdl:tAlias"
2094                         maxOccurs="unbounded"/>
2095              </sequence>
2096              <attribute name="choreographyName" type="QName"
2097                         use="required"/>
2098            </extension>
2099          </complexContent>
2100        </complexType>
2101
2102        <complexType name="tAlias">
2103          <complexContent>
2104            <extension base="cdl:tExtensibleElements">
2105              <sequence>
2106                <element name="this" type="cdl:tAliasVariable"/>
2107                <element name="free" type="cdl:tAliasVariable"/>
2108              </sequence>
2109            </extension>
```

```
2110          </complexContent>
2111      </complexType>
2112
2113      <complexType name="tAliasVariable">
2114         <complexContent>
2115            <extension base="cdl:tExtensibleElements">
2116               <attribute name="variable" type="cdl:tXPath-expr"
2117                          use="required"/>
2118               <attribute name="role" type="QName" use="required"/>
2119            </extension>
2120         </complexContent>
2121      </complexType>
2122
2123      <complexType name="tInteraction">
2124         <complexContent>
2125            <extension base="cdl:tExtensibleElements">
2126               <sequence>
2127                  <element name="participate" type="cdl:tParticipate"/>
2128                  <element name="exchange" type="cdl:tExchange" minOccurs="0"
2129                           maxOccurs="unbounded"/>
2130                  <element name="record" type="cdl:tRecord" minOccurs="0"
2131                           maxOccurs="unbounded"/>
2132               </sequence>
2133               <attribute name="name" type="NCName" use="required"/>
2134               <attribute name="channelVariable" type="QName"
2135                          use="required"/>
2136               <attribute name="operation" type="NCName" use="required"/>
2137               <attribute name="time-to-complete" type="duration"
2138                          use="optional"/>
2139               <attribute name="align" type="boolean" use="optional"
2140                          default="false"/>
2141               <attribute name="initiateChoreography" type="boolean"
2142                          use="optional" default="false"/>
2143            </extension>
2144         </complexContent>
2145      </complexType>
2146
2147      <complexType name="tParticipate">
2148         <complexContent>
2149            <extension base="cdl:tExtensibleElements">
2150               <attribute name="relationship" type="QName" use="required"/>
2151               <attribute name="fromRole" type="QName" use="required"/>
2152               <attribute name="toRole" type="QName" use="required"/>
2153            </extension>
2154         </complexContent>
2155      </complexType>
2156
2157      <complexType name="tExchange">
2158         <complexContent>
2159            <extension base="cdl:tExtensibleElements">
2160               <sequence>
2161                  <element name="use" type="cdl:tVariableRef"/>
2162                  <element name="populate" type="cdl:tVariableRef"/>
2163               </sequence>
2164               <attribute name="messageContentType" type="QName"
2165                          use="required"/>
2166               <attribute name="action" type="cdl:tAction2" use="required"/>
```

```
2167            </extension>
2168         </complexContent>
2169      </complexType>
2170
2171      <complexType name="tVariableRef">
2172         <complexContent>
2173            <extension base="cdl:tExtensibleElements">
2174               <attribute name="variable" type="cdl:tXPath-expr"
2175                          use="required"/>
2176            </extension>
2177         </complexContent>
2178      </complexType>
2179
2180      <complexType name="tRecord">
2181         <complexContent>
2182            <extension base="cdl:tExtensibleElements">
2183               <sequence>
2184                  <element name="source" type="cdl:tVariableRef"/>
2185                  <element name="target" type="cdl:tVariableRef"/>
2186               </sequence>
2187               <attribute name="name" type="string" use="required"/>
2188               <attribute name="role" type="QName" use="required"/>
2189               <attribute name="action" type="cdl:tAction2" use="required"/>
2190            </extension>
2191         </complexContent>
2192      </complexType>
2193
2194      <complexType name="tAssign">
2195         <complexContent>
2196            <extension base="cdl:tExtensibleElements">
2197               <sequence>
2198                  <element name="copy" type="cdl:tCopy"
2199                           maxOccurs="unbounded"/>
2200               </sequence>
2201               <attribute name="role" type="QName" use="required"/>
2202            </extension>
2203         </complexContent>
2204      </complexType>
2205
2206      <complexType name="tCopy">
2207         <complexContent>
2208            <extension base="cdl:tExtensibleElements">
2209               <sequence>
2210                  <element name="source" type="cdl:tVariableRef"/>
2211                  <element name="target" type="cdl:tVariableRef"/>
2212               </sequence>
2213               <attribute name="name" type="NCName" use="required"/>
2214            </extension>
2215         </complexContent>
2216      </complexType>
2217
2218      <complexType name="tNoaction">
2219         <complexContent>
2220            <extension base="cdl:tExtensibleElements"/>
2221         </complexContent>
2222      </complexType>
2223
```

```
2224        <complexType name="tException">
2225           <complexContent>
2226              <extension base="cdl:tExtensibleElements">
2227                 <sequence>
2228                    <element name="workunit" type="cdl:tWorkunit"
2229                            maxOccurs="unbounded"/>
2230                 </sequence>
2231                 <attribute name="name" type="NCName" use="required"/>
2232              </extension>
2233           </complexContent>
2234        </complexType>
2235
2236        <complexType name="tFinalizer">
2237           <complexContent>
2238              <extension base="cdl:tExtensibleElements">
2239                 <sequence>
2240                    <element name="workunit" type="cdl:tWorkunit"/>
2241                 </sequence>
2242                 <attribute name="name" type="NCName" use="required"/>
2243              </extension>
2244           </complexContent>
2245        </complexType>
2246
2247        <simpleType name="tAction">
2248           <restriction base="string">
2249              <enumeration value="request-respond"/>
2250              <enumeration value="request"/>
2251              <enumeration value="respond"/>
2252           </restriction>
2253        </simpleType>
2254
2255        <simpleType name="tAction2">
2256           <restriction base="string">
2257              <enumeration value="request"/>
2258              <enumeration value="respond"/>
2259           </restriction>
2260        </simpleType>
2261
2262        <simpleType name="tUsage">
2263           <restriction base="string">
2264              <enumeration value="once"/>
2265              <enumeration value="unlimited"/>
2266           </restriction>
2267        </simpleType>
2268
2269        <simpleType name="tBoolean-expr">
2270           <restriction base="string"/>
2271        </simpleType>
2272
2273        <simpleType name="tXPath-expr">
2274           <restriction base="string"/>
2275        </simpleType>
2276
2277        <simpleType name="tIsolation">
2278           <restriction base="string">
2279              <enumeration value="dirty-write"/>
2280              <enumeration value="dirty-read"/>
```

```
2281            <enumeration value="serializable"/>
2282        </restriction>
2283    </simpleType>
2284
2285  </schema>

2286
```

B. WS-CDL Supplied Functions

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

xsd:dateTime getCurrentTime()

xsd:dateTime getCurrentDate()

xsd:dateTime getCurrentDateTime()

Returns the current date/time.

xsd:string createNewID()

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

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

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

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

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