WS Choreography Description Language, Version 1

Editor's Draft, 19 February 2004

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

This document is available in other format(s):

Copyright

Copyright © 2003 W3C® (MIT, ERCIM, Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply.

Abstract

The Web Services Choreography Description Language (WS-CDL) is an XML-based language that describes cross-enterprise collaborations of Web Services participants by defining their common observable behavior; where ordered and synchronized message exchanges result in alignment of their common information.

The existing Web Services specifications, based on a stateless, connected, client-server model, offer a communication bridge between the heterogeneous computational environments used to develop applications today.

The future of E-Business applications requires the ability to perform long-lived business transactions between autonomous services. Applications, exposed as Web Services, must be able to communicate and synchronize their common business knowledge with other Web Services in a loosely coupled environment. These interactions are long-lived and must avoid resource constraints when accessing state information or relaxing

33 consistency guarantees in the presence of potential error recovery conditions. Business
34 collaborations between autonomous Web Service participants will be stateful and require
35 that all participating services can act as peers while reliably communicating in an
36 asynchronous fashion.
37
38 This specification extends the emerging stack of Web Services standards targeted for
39 integrating applications developed in heterogeneous computation environments.
40

41 Status of this Document

42 This specification is a draft document and may be updated, extended or replaced by
43 other documents, if necessary. It is for review and evaluation only. The authors of this
44 specification provide this document as is and provide no warranty about the use of this
45 document in any case. The authors welcome feedback and contributions to be
46 considered for updates to this document in the near future.

47 # Table of Contents

48 1 Introduction
49 1.1 Notational Conventions
50 1.2 Purpose
51 1.3 Goals
52 1.4 Relationship with WSDL
53 1.5 Relationship with Business Process Languages
54 2 Language Concepts
55 2.1 Collaboration Types
56 2.1.1 Abstract Choreography
57 2.1.2 Portable Choreography
58 2.1.3 Concrete Choreographies
59 2.1.4 Relationship between Collaboration Types
60 2.2 Collaboration Packaging
61 2.3 Coupling Collaborating Agents
62 2.3.1 Roles
63 2.3.2 Participants
64 2.3.3 Relationships
65 2.3.4 Channels
66 2.4 Information Driven Collaborations
67 2.4.1 Declaring Information Types
68 2.4.2 Variables
69 2.4.2.1 Variables and Abstract/Portable/Concrete Choreographies
70 2.4.3 Tokens
71 2.4.4 Choreographies
72 2.4.5 Work Units
73 2.4.6 Reusing existing Choreographies
74 2.4.6.1 Composing Choreographies
75 2.4.6.2 Importing Choreographies
76 2.4.7 Choreography Life-line
77 2.4.8 Choreography Recovery
78 2.4.8.1 Exception Block
79 2.4.8.2 Transaction Block
80 2.5 Activities
81 2.5.1 Control Structures
82 2.5.2 Interacting
83 2.5.3 Performed Choreography
84 2.5.4 Assigning Variables
85 2.5.5 Defining actions with no business effect
86
87 3 Example

4 Language Elements
4.1 Choreography Document Structure
4.1.1 Choreography document Naming and Linking
4.1.2 Language Extensibility and Binding
4.1.3 Semantics
4.2 Choreography Package
4.2.1 Importing definitions
4.3 Roles
4.4 Participants
4.5 Relationships
4.6 Channels
4.7 Information Types
4.8 Tokens and Token Locators
4.8.1 Tokens
4.8.2 Token Locators
4.9 Variables
4.9.1 Expressions
4.10 Choreography Definition
4.10.1 WorkUnit
4.11 Activities Definition
4.11.1 Control Structures
4.11.1.1 Sequence
4.11.1.2 Parallel
4.11.1.3 Choice
4.11.2 Basic Activities
4.11.2.1 Perform activity
4.11.2.2 Interaction activity
4.11.2.2.1 Interaction Roles
4.11.2.2.2 Interaction Message Content
4.11.2.2.3 Interaction Channel Variables
4.11.2.2.4 Interaction Operations
4.11.2.2.5 Interaction State Changes
4.11.2.2.6 Interaction Based Alignment
4.11.2.2.7 Protocol Based Information Exchanges
4.11.2.3 Assign activity
4.11.2.4 NoAction activity
4.11.2.5 Finalize activity

5 WSDL Bindings
6 Relationship with the Security framework
7 Relationship with the Reliable Messaging framework
8 Relationship with the Transaction/Coordination protocol

131 9 Acknowledgements
132 10 References
133 Appendix A – WS-CDL XSD Schemas
134 Appendix B – WS-CDL Supplied Functions
135
136

136 # 1. Introduction

137 For many years, organizations have being developing solutions for automating cross-
138 enterprise, business transactions in an effort to improve productivity and reduce
139 operating costs.
140

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

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

150 An architecture of layered standards is being defined that allows technical
151 interoperability of Web Services. The current Web Service architecture is designed for
152 simple information retrieval in a stateless message exchange is currently defined in the
153 following foundation layers:

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

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

161 ?? UDDI: allows publishing the availability of a Web Service and its discovery from
162 service requesters using sophisticated searching mechanims.

163 ?? Web Services Security: ensures that exchanged messages are not modified or
164 forged.

165 The existing Web Services specifications, based on a stateless, connected, client-server
166 model, provide an interoperable data-bus for bridging heterogeneous computational
167 environments.

168 The future of E-Business applications also requires the ability to perform interoperable,
169 cross-enterprise, long-lived business transactions. That is, applications, exposed as
170 Web Services, exchanging business documents in peer-to-peer environments should be
171 able to communicate and synchronize their common business knowledge with other
172 Web Services in a loosely coupled environment. This interaction can occur over a long
173 period of time, and must do so without resource limitations when accessing state
174 information, or relaxing consistency guarantees.

175 Business collaborations between autonomous Web Service participants will be stateful.
176 The ability to perform certain business operations as well as the way to interpret the
177 content of each business documents may be influenced by the previous exchanges and

require that all participating services act as peers and communicate in a coordinated and reliable fashion.

Extending the current Web Services stack by defining several additional layers, enables autonomous applications to participate in peer-to-peer business transactions.

The following figure shows the emerging stack of standards associated with Web Services for integrating applications developed in heterogeneous computation environments.

The new layers defined at the top of the current Web Service stack (see Fig. 1) are:

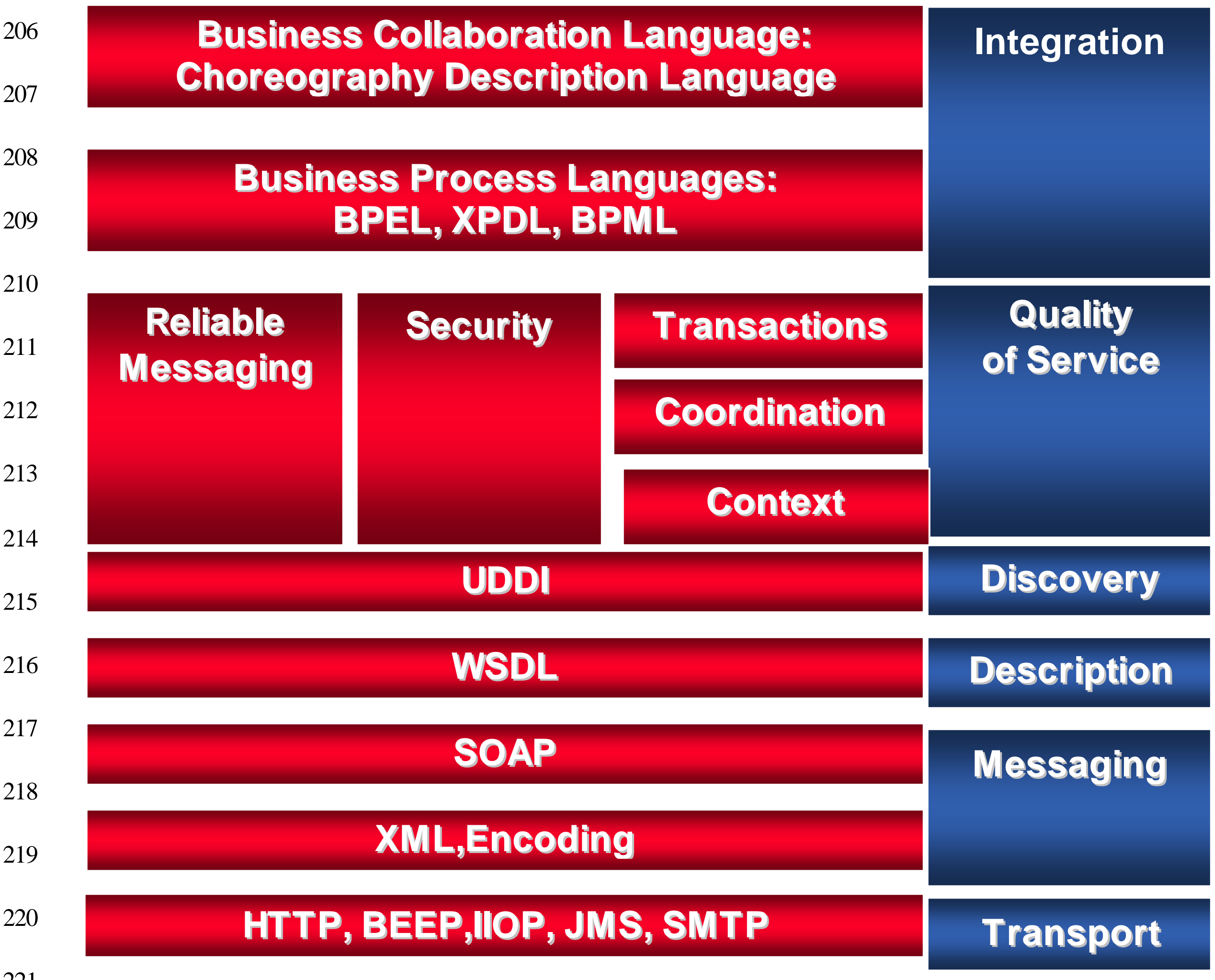
- Business Collaboration Languages layer: describes cross-enterprise collaborations of Web Services participants by defining a global view of their observable behavior, where synchronized information exchanges through their shared collaboration points occur, when the commonly defined ordering rules are satisfied.

- 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 not observable business data.

- Reliable Messaging layer: provides exactly-once and guaranteed delivery of business documents 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.

206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221



222 **Figure 1 Emerging Web Services Framework**

223

223 ## 1.1. Notational Conventions

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

227 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].

228 3. This specification uses an **informal syntax** to describe the XML grammar of a WSDL
229 document:

230 ?? The syntax appears as an XML instance, but the values indicate the data types
231 instead of values.

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

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

236 ?? Grammar in bold has not been introduced earlier in the document, or is of particular
237 interest in an example.

238 ?? <-- extensibility element --> is a placeholder for elements from some "other"
239 namespace (like ##other in XSD).

240 ?? The XML namespace prefixes (defined above) are used to indicate the namespace
241 of the element being defined.

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

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

247

1.2. Purpose

Business or other activities that involve multiple different organizations or independent processes that use Web Services technology to exchange information can be successful if they are properly coordinated. This means that the sender and receiver of a message know and agree in advance:

- The format and structure of the messages that are exchanged, and
- The sequence and conditions in which the messages are exchanged.

To solve this problem, a “common” or “global” definition of the sequence and conditions in which messages are exchanged is produced that describes the observable, complementary behavior of all the participants involved. Each participant can then use the definition to build and test solutions that conform to the global definition.

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 example below serves as one example of Choreography in Action:

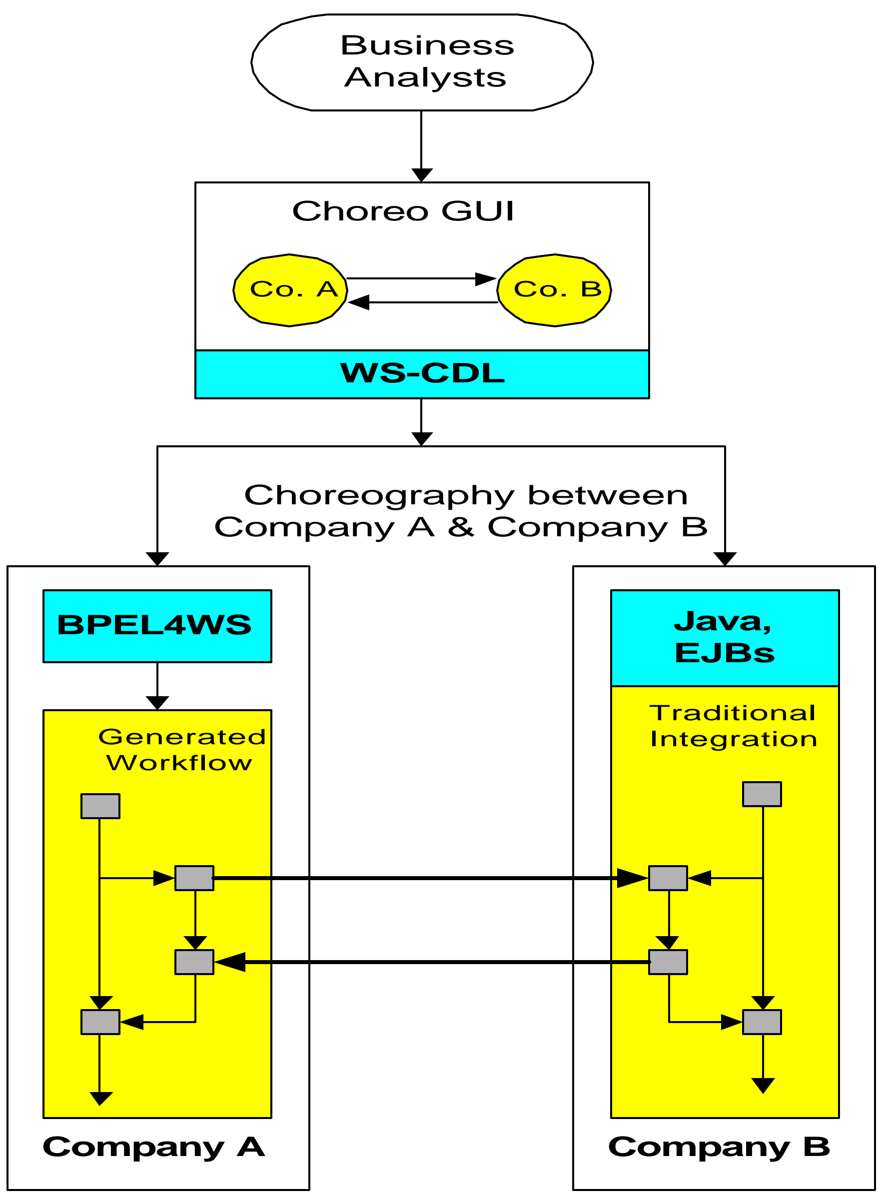


Figure 2

In Figure 2, Company A and Company B wish to integrate their business processes. The respective Business Analysts at both companies agree upon the rules and processes involved for the collaboration. Using a tool that can serve as a basis for the collaboration, Company A and Company B agree upon their interactions and generate a WS-CDL representation.

The WS-CDL representation can then, in the case of Company A, be used to generate a BPEL [18] code template. Company B, having greater legacy driven integration needs, relies on a J2EE [25] solution incorporating Java and EJBs.

In this example, Choreography specifies the interoperability and interactions between business entities, while leaving actual implementation decisions in the hands of each individual company.

1.3. Goals

The primary goal of a Business Collaboration Language for Web Services is to specify a declarative, XML based language that defines a global view of their observable behavior, where synchronized information exchanges occur, when the commonly defined ordering rules are satisfied.

Some additional goals of this definition language are to permit:

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

1.4. Relationship with WSDL

The WS-CDL specification depends on the following specifications: XML 1.0 [9], XML-Namespaces [10], XML-Schema 1.0 [11, 12] and XPath 1.0 [13]. In addition, support for importing and referencing service definitions given in WSDL 1.2 [7] is a normative part of the WS-CDL specification.

1.5. Relationship with Business Process Languages

WS-CDL 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 unobservable business data.

WS-CDL does not depend on a specific business process implementation language. Thus, it can be used to specify trully interoperable collaborations between any type of Web Service participant regardless of the supporting platform or programming model used by the implementation of the hosting environment.

Typically, the respective Business Analysts define in WS-CDL the common observable behavior of all participants engaged in the business collaboration. Each participant could be implemented by completely different languages such as:

- Web Services applications, whose implementation is based on executable business process languages like [16, 17, 18, 19, 20].
- Web Services applications, whose implementation is based on languages like [23].
- Or human controlled software agents.

2. Language Concepts

This section introduces the *Web Services Choreography Description Language (WS-CDL)* definitions to address the Business Collaboration Language requirements as defined in section 1.

2.1 Collaboration Types

One of the key goals of WS-CDL is to enable collaboration types reuse. Global definitions of a Choreography facilitate this especially if Choreographies are defined with varying degrees of abstraction. Although more could be defined, this model identifies and supports three different levels of abstraction in which choreographies can usefully be defined and used.

2.1.1 Abstract Choreography

The first is a highly abstract choreography that defines:

- The types of information that is exchanged, for example an order sent between a buyer and a seller
- The sequence and conditions under which the information is sent.
- When and how information exchanges are coordinated

However, it does not define:

- The physical structure of the information that is exchanged, for example there are no definitions of the XML documents, SOAP messages, WSDL port types and operations, URLs etc that are to be used.
- How the different conditions that are used to control the sequence of exchanging information are determined.
- Where the messages in the choreography should be sent e.g. to a URL
- How the messages are to be secured (if at all) and whether or not the messages are to be sent reliably.

Although abstract, this approach will be useful for defining generally accepted or “canonical” definitions for very common processes, such as placing an order. Definitions of theses types of choreography would best be carried out by international standards organizations that have a cross-industry, multi-geographic responsibility.

2.1.2 Portable Choreography

Clearly, the development of these abstract choreographies will take some time to complete, so the second type of choreography to define is a “portable” choreography. In this type of choreography definition the definitions in an Abstract Choreography are extended with:

- Detailed definitions of the physical structure of the information that is exchanged including the WSDL port types and operations.
- Details of the technology to be used, for example, how to secure the messages and send them reliably.

397 ?? Rules that express, as far as possible, the conditions that are used to control the
398 sequence of exchange of information, in terms of, for example XPath expressions
399 that reference data in the messages.
400
401 However they do not specify the URLs to which the messages are sent nor, for example,
402 the digital certificates used to secure them. This means that an organization should be
403 able to design and build a solution that conforms, in detail, to the rules of the
404 choreography, and only require limited additional information at run time to determine
405 where messages should be sent. As a result realizing interoperability should be much
406 easier.
407
408 This “portable” type of choreography is targeted more at vertical industry organizations,
409 that want to define rules for collaboration between the members of their industry and
410 simplify, as far as possible, the implementation and integration process.
411

412 2.1.3 Concrete Choreographies

413 The final type of choreography, is a Concrete Choreography, where all the details are
414 specified that are required to send a message. This extends the definition in a Portable
415 Choreography to include information about the “endpoints”. This can include information
416 such as:

417 ?? The URLs that are the destinations of the messages that are sent, and
418 ?? Other “endpoint” specific rules such as digital certificates to be used for securing
419 messages.
420
421 These types of choreographies are probably most applicable where two or more
422 participants want to specify how they will cooperate and there is little or no need for
423 other organizations to follow the same process.
424

425 2.1.4 Relationship between Collaboration Types

426 The table below summarizes the three different types of choreographies.
427

	Abstract	**Portable**	**Concrete**
Types of Messages	Identified	Identified	Identified
Message Structure	Not Defined	Defined	Defined
Conditions	Identified	Identified	Identified
Condition evaluation rules	Not defined	Defined as far as possible	Defined as far as possible
Technology used	Not defined	Defined	Defined
Message Endpoint Data	Not defined	Not Defined	Defined

428

An Abstract Choreography to be extended to become a Portable Choreography and a Portable Choreography to be extended to become a Concrete Choreography.

Alternatively each different type of Choreography to be defined directly. This means that:

- A Portable Choreography can be defined without first defining the Abstract Choreography.
- A Concrete Choreography can be defined without defining an Abstract or Portable Choreography.

2.2 Collaboration Packaging

A WS-CDL document describes interoperable, cross-enterprise collaborations that allow participants to perform business transactions. In order to facilitate these collaborations, services commit on mutual responsibilities by establishing business relationships. Their collaboration takes place in a commonly agreed set of constraints, whereby documents are exchanged in a synchronized fashion between the participants.

A collaboration package aggregates the following elements:

- *Participants, Roles and Relationships*. In a Choreography information is always exchanged between Participants, such as a Business or Organization acting in one or more *Roles*, for example Buyer or Seller as part of a Relationship, for example purchasing goods.
- *Choreography*. A Choreography allows constructing compositions of Web Service participants by explicitly asserting their common observable behaviors, where synchronized information exchanges through their shared collaboration points occur, when the commonly defined ordering rules are satisfied.
- *Choreography Composition and Import*. This element enumerates how one Choreography can be created by performing other, pre-existing choreographies and importing content from other choreographies.
- *Types, Variables and Tokens*. *Variables* contain information about objects in the choreography such as the messages exchanged or the state of the *Roles* involved. *Tokens* are aliases that can be used to reference parts of a *Variable*. Both *Variables* and *Tokens* have *Types* that define the structure of what the *Variable* or *Token* contains.
- *Interactions*. These are the basic building blocks of the Choreography which result in the sending of messages between Roles in either a “one-way” or “request-response” message pattern.
- *Activities and Control Structures*. Activities are the lowest level components of the Choreography that do the actual work. Control Structures combine activities with other Control Structures in a nested structure to express the sequence and conditions in which the messages in the choreography are exchanged.
- *Choreography Exceptions and Transactions*. *Choreography Exceptions* describe how to specify what additional Interactions should occur when a Choreography behaves in an abnormal way. *Choreography Transactions* describes how to specify what additional Interactions should occur to reverse the effect of an earlier completed choreography.
- *Semantics*. Semantics allow the creation of descriptions that can record the semantic definitions of almost every single component in the model.

476 ## 2.3 Coupling Collaborating Agents

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

484 A WS-CDL document describes how a Web Service participant is capable of engaging in
485 collaborations with the same participant or with different participants by specifying both
486 their static and dynamic couplings.

487 Within a Choreography, information is always exchanged between *Participants.* The
488 *Roles* and *Relationship types* identify the static coupling of collaborating Web Service
489 participants. The *Channels* identify the dynamic coupling of collaborating Web Service
490 participants.

491 ### 2.3.1 Roles

492 A *Role* identifies a set of related behaviors, for example the Buyer role is associated with
493 purchasing of goods or services and the Supplier role is associated with providing those
494 goods or services for a fee.

495 A Role specifies the observable behavior, in terms of the operations, a participant MUST
496 exhibit in order to collaborate with other participants. Within a Role, a Behavior specifies
497 a subset of the observable behavior a participant MUST exhibit in order to collaborate
498 with other participants under their mutually agreed commitments.

499 ### 2.3.2 Participants

500 A *Participant* identifies a set of related Roles, for example a Commercial Organization
501 could take both a Buyer Role when purchasing goods and a Seller role when selling
502 them.

503 ### 2.3.3 Relationships

504 A *Relationship* is the association of two Roles for a purpose. A relationship represents
505 the possible ways in which two roles can interact. For example the Relationships
506 between a Buyer and a Seller could include:
507 ?? A “Purchasing” Relationship, for the initial procurement of goods or services, and
508 ?? A “Customer Management” Relationship to allow the Supplier to provide service and
509 support after the goods have been purchased or the service provided.
510
511 Although Relationships are always between two Roles, Choreographies involving more
512 than two Roles are possible. For example if the purchase of goods involved a third-party
513 Shipper contracted by the Supplier to deliver the Supplier’s goods, then, in addition to
514 the Purchasing and Customer Management relationships described above, the following
515 relationships might exist:
516 ?? A “Logistics Provider” relationship between the Supplier and the Shipper, and
517 A “Goods Delivery” relationship between the Buyer and the Shipper.
518

519 A Relationship specifies the mutual commitments, based on the Role types and the
520 Behavior type required, that two participants oblige to provide in order to participate in a
521 common business transaction.
522

523 2.3.4 Channels

524
525 A *Channel type* facilitates the collaboration between two Participants by identifying their
526 common collaboration points. A Channel identifies where and how to send/receive
527 information to/into a Role. Additionally, it identifies what is the allowed Channel
528 information that can be passed from a Role to another Role and the usage of a Channel
529 within each participant.
530
531 A Channel MUST describe the Role of a participant sending/receiving information
532 to/from another participant.

533 A Channel MAY describe the service reference type of a participant, used for locating
534 where and how to exchange messages to/into a participant.

535 A Channel MAY describe the business process type implementation within that
536 participant.

537 A Channel MAY describe the identity of an instance of a business process implementing
538 a participant's role.

539 A Channel MAY describe one or more logical conversations between participants, within
540 a collaboration. Each conversation groups a set of related document exchanges.

541 A Channel MAY be passed around from one role to another. Channel MAY prescribe
542 the types of the Channels allowed to be exchanged between the Web Services
543 participants, through this channel. Additionally, a Channel MAY restrict its usage by
544 specifying the number of times a Channel instance can be used.
545

546 2.4 Information Driven Collaborations

547 A WS-CDL document allows defining relevant parts of the exchanged messages that
548 can influence the observable behavior of a collaboration.
549
550 *Variables* contain information about objects in the choreography such as the messages
551 exchanged or the state of the *Roles* involved. *Tokens* are aliases that can be used to
552 reference parts of a *Variable*. Both *Variables* and *Tokens* have *Types* that define the
553 structure of what the *Variable* or *Token* contains.
554

555 2.4.1 Declaring Information Types

556 Information types are used for defining the types of business information used within a
557 Choreography. By introducing this abstraction, a Choreography definition avoids
558 referencing only abstract message types, as described by WSDL, but instead can
559 reference other data types as defined by the XML Schema specification.
560

561 2.4.2 Variables

562 A WS-CDL document allows declaration of the business information shared between
563 participants engaged in stateful, long-lived business transactions, where performing
564 certain collaborations requires the ability to depend upon the exchanges of previous
565 messages.
566

567 Variables capture information about objects in a Choreography as defined by the
568 *Variable Usage*:

569 ?? *Information Exchange Variables* that contain information such as an Order that is
570 used to:
571 ?? Populate the content of a message to be sent, or
572 ?? Populated as a result of a message received
573 ?? *Information Variables* that contain information such as an Order that is used to:
574 ?? Populate the content of a message to be sent, or
575 ?? Populated as a result of a message received
576 ??
577 ?? *State Variables* that contain information about the State of a Role as a result of
578 information exchanged. For example:
579 ?? When a Buyer sends an order to a Seller, the Buyer could have a *State Variable*
580 called “OrderState” set to a value of “OrderSent” and once the message was
581 received by the Seller, the Seller could have an *State Variable* called
582 “OrderState” set to a value of “OrderReceived”. Note that the variable
583 “OrderState” at the Buyer is a different variable to the “OrderState” at the Seller
584 ?? Once an order is received, then it might be validated and checked for
585 acceptability in other ways that affect how the choreography is performed. This
586 could require additional states to be defined for “Order State”, such as:
587 “OrderError”, which means an error was detected that stops processing of the
588 message, “OrderAccepted”, which means that there were no problems with the
589 Order and it can be processed, and “OrderRejected”, which means, although
590 there were no errors, it cannot be processed, e.g. because a credit check failed.
591 ?? *Channel Variables* that contain information that describes how and where a message
592 is sent to a Role. For example, a Channel Variable could contain information such as
593 the URL to which the message should be sent, the policies that are to be applied,
594 such as security, whether or not reliable messaging is to be used, etc.
595 ?? *Other Variables* including
596 ?? *Locally Defined Variables* that contain information created and changed locally
597 by a Role. They can be Information Exchange, State or Channel Variables as
598 well as variables of other types. For example “Maximum Order Amount” could be
599 data created by a seller that is used together with an actual order amount from
600 an Order received to control the flow of the choreography. In this case how
601 Maximum Order Amount is calculated and its value would not be known by the
602 other Roles
603 ?? *Common Variables* that contain information that is common knowledge to two or
604 more Roles, e.g. “OrderResponseTime” which is the time in hours in which a
605 response to an Order must be sent
606

607 The value of Variables can be:
608 ?? Known by all the roles prior to the start of the choreography
609 ?? Assigned by one role and optionally communicated to other roles
610 ?? Assigned as a result of an interaction
611 ?? Assigned by one role from other information

?? Used to determine the decisions and actions to be taken in a Choreography.

Within a WS-CDL document, variable information is made available to a participant, by declaring it. Data access of variable information is done using the features of XML data types, WSDL message types and XPATH expressions.

Within a WS-CDL document variable information is made available to two interacting participants within a Choreography, by performing an interaction through a common Channel. Additionally, the variable information exchanged between the two participants MAY be aligned, so they are the same in both participants.

2.4.2.1 Variables and Abstract/PortableConcrete Choreographies

Defining Variables to hold information about the objects in a Choreography means that:

?? Variables contain all the information about a Choreography that can change from implementation to implementation

?? The definition of the sequence and conditions in which information is exchanged is independent of how those information exchanges are actually implemented

?? As new methods are developed for defining interfaces, messages, as well as other Web Services standards, only the way the variables are defined should need to change. The essence of the choreography, i.e. the basic definition of the sequence and conditions in which information is exchanged, remains the same.

In addition the *Import* statement also allows definitions in one choreography, to be over-ridden by other, replacement definitions. This means that:

?? The same choreography can be reused in different contexts with different interfaces, message types and varying levels of detail as required

?? The *Abstraction Level* of the variables can change as required from abstract through to concrete

?? The definitions of the variables in an “abstract” choreography can be used as a checklist to validate that any replacement definitions at either the Portable or Concrete levels form a complete list.

2.4.3 Tokens

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

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

Tokens types reference a document fragment within a Choreography definition and Token Locators provide a query mechanism to select them. By introducing these abstractions, a Choreography definition avoids depending on specific message parts, as

659 described by WSDL, or a specific query string, as specified by XPATH, but instead the
660 parts or the query string can change without affecting the Choreography definition.

661

662 ### 2.4.4 Choreographies

663 A WS-CDL document explicitly prescribes the rules, agreed between Web Service
664 participants, that govern the ordering of exhanged messages and the provisioning of
665 alternative patterns of behavior. The operational semantics of these rules are based on
666 the data-driven computational model, where availability of shared variable information
667 causes a rule to be triggered and its guarded subsequent actions to be enabled.

668 A *Choreography* allows constructing compositions of Web Service participants by
669 explicitly asserting their common observable behaviors, where synchronized information
670 exchanges through their shared collaboration points occur, when the commonly defined
671 ordering rules are satisfied.

672 A Choreography defined at the Package level is called a base Choreography, and does
673 not share its context with other base choreographies. A Choreography defined within
674 another Choreography is called an enclosed Choreography. A Package MUST contain
675 exactly one base Choreography, that is explicitly marked as the root Choreography. The
676 root Choreography is the only base Choreography that MAY be initiated. The root
677 Choreography is enabled when it is initiated. All non-root base Choreographies MAY be
678 enabled only inside a root Choreography.

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

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

686 A Choreography acts as a name scoping context as it restricts the visibility of variable
687 information. Variable defined in a Choreography is visible in this Choreography and all
688 its enclosed Choreographies, forming a *Choreography Visibility Horizon*. Restricting the
689 visibility of variable information allows aligning activities that are generating, and
690 exchanging variable information between participants to be isolated from activities
691 belonging to different Visibility Horizons.

692 A Choreography MUST contain one or more *Work Units*.

693

694 ### 2.4.5 WorkUnits

695 A *Work Unit* prescribes the explicit rules for enforcing the constraints that preserve the
696 consistency of the business transactions commonly performed by the Web Service
697 participants. Examples of a Work Unit include:
698 ?? A *Send PO* Work Unit that includes Interactions for the Buyer to send an Order, the
699 Supplier to acknowledge the order, and then later accept (or reject) the order. This
700 work unit would probably not have a guard.
701 ?? An *Order Delivery Error* Work Unit that is performed whenever the *Place Order* Work
702 Unit did not reach a “normal” conclusion. This would have a Guard condition that
703 identifies the error – see also Choreography Exceptions and Transactions.

?? A *Change Order* Work Unit that can be performed whenever an order acknowledgement message has been received and an order rejection has not been received.

A Work Unit specifies the data dependencies that must be satisfied before enabling one or more business actions, by expressing interest on the availability of variable information that already exists or will be created in the future.

A Work Unit is enabled when the Choreography it belongs to is enabled. Enabled Work Units are matched when the required, zero, one or more variable information become available. Availability of some variable information does not mean that a Work Unit matches immediately. Only when all variable information required by a Work Unit become available in the appropriate Visibility Horizon does matching succeed. Variable information available in a Choreography MAY be matched with a Work Unit that will be registered in the future. The matching of variable information with Work Units MAY enable one, or more concurrent Choreographies.

A Work Unit MUST contain one *Control Structures or Activities*.

A *Control Structure* prescribes the sequencing and conditional rules for enabling one or more business actions.

An *Activity* describes the detailed actions of a business transaction, commonly performed by the Web Service participants.

A Work Unit allows an application to recover from faults that are the result from abnormal business actions and also to finalize for completed business actions that need to be logically rolled back.

A Choreography MAY contain more than one Work Unit, thus allowing selecting between alternative collaboration paths only one.

A Work Unit MAY belong to a Select group. One or more Work Units MAY belong to the same Select group. Within a Choreography, one or more Select groups MAY be specified. When two or more Work Units are matched within the same Select group then only one Work Unit is allowed to make progress by enabling its own actions. The lexicographical order of the Work Units belonging to the same Select group, determines which Work Unit makes progress. All matched Work Units that belong to different Select groups are allowed to make progress by enabling their own actions.

A Work Unit completes successfully when all its enclosed business actions complete successfully. A Work Unit that completes successfully MUST be considered again for matching (based on its guard condition) if its repetition condition (optionally defined in the Work Unit) evaluates to true.

2.4.6 Reusing existing Choreographies

Choreographies can be combined and built from other Choreographies.

2.4.6.1 Composing Choreographies

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

754 ?? A Request for Quote (RFQ) Choreography that involved a Buyer role sending a
755 request for a quotation for goods and services to a Supplier to which the Supplier
756 responding with either a “Quotation” or a “Decline to Quote” message, and
757 ?? An Order Placement Choreography where the Buyer placed and order for goods or
758 services and the Supplier either accepted the order or rejected it.
759
760 You could then create a new “Quote and Order” Choreography by reusing the two where
761 the RFQ choreography was executed first, and then, depending on the outcome of the
762 RFQ Choreography, the order was placed using the Order Placement Choreography.
763 In this case the new choreography is “composed” out of the two previously defined
764 choreographies. These choreographies may be specified either:

765 ?? *Locally*, i.e. they are included, as a *Enclosed Choreography,* in the same
766 choreography definition as the choreography that performed them, or
767 ?? *Globally,* i.e. they are specified in a separate choreography definition that is defined
768 elsewhere.
769
770 Using this approach, Choreographies can be recursively combined to support
771 choreographies of any required complexity allowing more flexibility as Choreographies
772 defined elsewhere can be reused.
773
774 **2.4.6.2 Importing Choreographies**
775
776 An *Import* statement can contain references to a complete Choreography or part of a
777 Choreography.
778
779 Import statements must be interpreted in the sequence they occur.
780
781 When the Import statement contains references to variables or other data that have the
782 same identity, then the content of the later Import statement replaces the same content
783 referenced by the earlier Import statement.
784
785 This means, for example, that if an initial Choreography definition referenced by an
786 Import statement contained variables, etc, that were defined in an Abstract way, then the
787 replacement definition could either be Portable or Concrete.
788
789 It also enables one Choreography definition to effectively be “cloned” by replacing the
790 definitions for some or all of its variables.
791
792 **2.4.7 Choreography Life-line**

793 A Choreography expresses the progression of a business transaction. Initially, the
794 business transaction MUST be started, then work MAY be performed within it and finally
795 it MUST complete. These different phases are designated by explicitly marked business
796 operations within the Choreography and its semantics.

797 A root Choreography is initiated when the first interaction, marked as the Choreography
798 initiator, is performed. Two or more interactions MAY be marked as initiators, indicating
799 alternative initiation actions. In this case, the first action will initiate the Choreography

800 and the other actions will enlist with the already initiated Choreography. An interaction
801 designated as a Choreography initiator MUST be the first action performed in a
802 Choreography. If a Choreography has two or more Work Units with interactions marked
803 as initiators, then these are mutually exclusive and the Choreography will be initiated
804 when the first interaction occurs and the remaining Work Units will be disabled. All the
805 interactions not marked as initiators indicate that they will enlist with an already initiated
806 Choreography.

807 A Choreography completes successfully when there no more enabled Work Unit(s)
808 within it. Alternatively, a Choreography completes successfully if its complete condition
809 (optionally defined in the Choreography) evaluates to true even if one or more enabled
810 Work Units are still unmatched.
811

812 2.4.8 Choreography Recovery

813 An Exception WorkUnit MAY be defined as part of an enclosing Choreography to
814 recover from exceptional conditions that may occur in that Choreography.
815
816 A Finalization WorkUnit MAY be defined as part of an enclosing Choreography to
817 provide the finalization actions that semantically rollback the completed enclosing
818 Choreography.
819

820 2.4.8.1 Exception Block

821 A Choreography can sometimes fail as a result of an exceptional circumstance or error.
822 Different types of exceptions are possible including this non-exhaustive list:
823 ?? *Interaction Failures*, for example the sending of a message did not occur
824 ?? *Protocol Based Exchange failures*, for example no acknowledgement was received
825 as part of a reliable messaging protocol
826 ?? *Security failures*, for example a Message was rejected by a recipient because the
827 digital signature was not valid
828 ?? *Choreography Sequence Failures*, for example a Message was received that was
829 not in the sequence as defined by the Choreography
830 ?? *Timeout errors*, for example an Interaction did not complete within a required
831 timescale
832 ?? *Validation Errors*, for example an XML order document was not well formed or did
833 not conform to its schema definition
834 ?? *Business Process "failures"*, for example the goods ordered were out of stock.
835
836 To handle these and other "errors" separate Work Units are defined in the *Exception*
837 *Block* for each "exception" condition (as identified by its guards) that needs to be
838 handled. Only one Work Unit per exception should be performed.
839
840 When a Choreography encounters an exceptional condition it MAY need to act on it.
841 An Exception WorkUnit MAY be defined as part of an enclosing Choreography for the
842 purpose of handling the exceptional conditions occurring on that Choreography. A
843 Choreography MAY define one or more Exception WorkUnits. An Exception Work Unit
844 expresses interest on fault variable information that MAY become available.

845 The fault variable information is a result of:

846 ?? A fault occurring while performing an interaction between collaborating participants.

847 ?? A timeout occuring while an interaction between collaborating participants was not
848 completed within a specified time period.

849 Exception Work Units are enabled when the enclosing Choregraphy is enabled. An
850 Exception WorkUnit MAY be enabled only once for an enclosing Choreography.
851 Exception Work Units enabled in an enclosing Choreography MAY behave as the default
852 mechanism to recover from faults for all its enclosed Choreographies. Exception Work
853 Units enabled in an enclosed Choreography MAY behave as a refined mechanism to
854 recover from faults for any of its enclosing Choreographies.

855 If a fault occurs, then the faulted Choreography completes unsuccessfully and its
856 Finalization WorkUnit is not enabled. The actions, including enclosed Choreographies,
857 enabled within the faulted Choreography are completed abnormally before an Exception
858 Work Unit can be matched.

859 Within a Choreography only one Exception Work Unit MAY be matched. When an
860 Exception Work Unit matches, it enables the appropriate activities for recovering from
861 the fault.
862
863 Matching a fault with an Exception Work Unit is done as follows:

864 If a fault is matched by an Exception Work Unit then the actions of the matched Work
865 Unit are enabled. If a fault is not matched by an Exception Work Unit defined within the
866 Choreography in which the fault occurs, then the fault will be recursively propagated to
867 the parent Exception WorkUnit until a match is successful.

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

870 The actions of an Exception WorkUnit MAY also fault. The semantics for matching the
871 fault and acting on it are the same as described in this section.
872

873 **2.4.8.2 Transaction Block**

874 When a Choreography encounters an exceptional condition it MAY need to revert the
875 actions it had already completed, by providing finalization actions that semantically
876 rollback the effects of the completed actions.

877 A Finalization WorkUnit is a WorkUnit defined as part of an enclosing Choreography for
878 the purpose of reverting the effects of that Choreography. A Choreography MAY define
879 exactly one Finalization WorkUnit.

880 A Finalization Work Unit expresses interest on variable information that MAY become
881 available by a finalization command targetted to the enclosing Choreography. The
882 finalization command is issued by the Exception or the Finalization WorkUnit of the
883 parent of the enclosing Choreography. When a Finalization Work Unit matches, it
884 enables the appropriate finalization actions.

885 A Finalization WorkUnit is enabled only after its enclosing Choreography completes
886 successfully. The Finalization WorkUnit may be enabled only once for an enclosing
887 Choreography.

888 The actions within the Finalization WorkUnit MAY use variable information visible in the
889 Visibility Horizon of its enclosing Choreography as they were at the time the enclosing
890 Choreography completed or as they stand at the current time.

891 The actions of the Finalization WorkUnit MAY fault. The semantics for matching the fault
892 and acting on it are the same as described in the previous section.
893

2.5 Activities

Activities are the lowest level components of the Choreography, which do the actual work.

Control Structures combine these *Activities* with other *Control Structures* in a nested way to specify the sequence and flow of the exchange of information within the Choreography.

However at the highest level, the Choreographies consist of *Work Units*, that each contains a single Activity that is performed whenever an optional enabling condition on the *Work Unit*, called a *guard*, is true.

Each Activity within a Work Unit is then either:

- A *Basic Activity* that does the actual work. These are:
  - An *Interaction*, i.e. the Work Unit consists of a single Interaction
  - 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

- *Control Structures* that group Basic Activities and Control Structures together in a nested structure to express the logic and decision flow involved in the Choreography. The Control Structures are:
  - *Sequence*, which specifies a list of Activities that are performed in sequence
  - *Parallel*, which means that all the Activities are performed at the same time
  - *Choice*, which specifies that just one of two or more Activities are performed depending on the predicate associated with each Activity

2.5.1 Control Structures

The Sequence Control Structure enables a Work Unit to define that one or more Activities must be performed in sequence. Activities must be performed in the same sequence that they are defined.

The Parallel Control Structure enables a Work Unit to define that Activities are performed in parallel.

The Choice Control structure enables a Work Unit to define that only one of two or more Activities should be performed. It works by adding a Guard statement to each individual Activity within the Choice. An Activity should only occur if the Guard on the Activity evaluates to true. Once one of the Activities in the Choice has been performed, then no other Activities in the Choice must be performed.

2.5.2 Interacting

An Interaction represents the unit of work for communicating and synchronizing two participants. Two participants make progress by interacting and can possibly align their

shared information by performing synchronized information exchanges through a common Channel.

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(s) back to the requesting participant.

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 outcomes are possible which can only be determined at run time. The *Interaction* lists 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

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 needs to have a common understanding of the state changes of one or more *State Variables* that are complimentary to one or more *State Variables* of its partner Role.

Additionally within an Interaction, a Role needs to have a common understanding of the values of the *Information Exchange Variables* at the partner Role. Without alignment the Buyer knows that her “OrderState” is set to “Sent”, but does not know the value of the OrderState at the Seller. Once the Seller receives the Order then the Seller knows that his “OrderState” variable is set to “Received”. He also knows the Buyers “OrderState” was set to “Sent”, as the Choreography defines that the Buyer’s Order State variable is set in this way when an Order is sent.

With Choreography Alignment the difference is that both the Buyer and the Seller have:

?? 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

?? Knowledge of the values of each others States Variables, i.e. the Buyer knows that the Seller’s “OrderState” variable has the value “Received” and the Seller knows that the Buyer’s “OrderState” variable is set to “Sent”

?? Information Exchange Variables that have the same types with the same content, e.g. The Order variables at the Buyer and Seller have the same Information Types and hold the same order

This assurance of the outcome with respect to States is achieved by an ‘agreement’ protocol that is used in conjunction with the Choreography such as the Web Services and other specifications designed to coordinate long-running transactions.

987 The variable information that need to be aligned and made available to the two
988 interacting participants MUST be explicitly modeled in WS-CDL as a variable alignment
989 interaction between them. After the alignment interaction completes, both participants
990 progress at the same time, in a lock-step fashion and the variable information in both
991 participants is aligned. Their variable alignment comes from the fact that the requesting
992 participant has to know that the accepting participant has received the message and the
993 other way around, the accepting participant has to know that the requesting participant
994 has sent the message before both of them progress. The mechanism of how this is
995 accomplished is implementation specific.
996
997 The *One-Way*, *Request* or *Response* messages in an Interaction may also be
998 implemented using a *Protocol Based Exchange* where a series of messages are
999 exchanged according to some well-known protocol, such as the reliable messaging
1000 protocols defined in specifications such as WS Reliability.
1001 In both cases, the same or similar *Message Content* may be exchanged as in a simple
1002 Interaction, for example the sending of an Order between a Buyer and a Seller.
1003 Therefore some of the same *State Changes* may result.
1004 However when protocols such as the reliable messaging protocols are used, additional
1005 *State Changes* will occur. For example, if a reliable messaging protocol were being used
1006 then the Buyer, once confirmation of delivery of the message was received, would also
1007 know that the Seller's "Order State" variable was in the state “Received” even though
1008 there was no separate Interaction that described this.
1009
1010 An interaction activity forms the atom of composition, where multiple interacts are
1011 combined to form a Choreography, which can be used in different business contexts. A
1012 business collaboration that does not have atomic semantics, SHOULD be modeled as
1013 several interactions between participants instead of one interaction.
1014
1015 The Channel through which an interaction occurs is used to determine whether to enlist
1016 the interaction with an already initiated Choreography or to initiate a new Choreography.

1017 Within a Choreography, two or more related interactions MAY be grouped to form a
1018 logical conversation. The Channel through which an interaction occurs is used to
1019 determine whether to enlist the interaction with an already initiated conversation or to
1020 initiate a new conversation.
1021
1022 An interaction completes normally when the request and the response (if there is one)
1023 complete successfully. In this case the business documents and Channels exchanged
1024 during the request and the response (if there is one) result in the shared variable being
1025 aligned between the two participants.

1026 An interaction completes abnormally if faults occur:

1027 ?? The time-to-complete timeout identifies the time an interaction takes to complete. If
1028 this timeout occurs, after the interaction was initiated but before it completed, then a
1029 fault is generated.
1030 ?? A fault signals an exceptional condition during the management of a request or within
1031 a participant when accepting the request.

1032 In this case the business documents and channels exchanged do not result in any new
1033 variable alignment between these participants, rather the shared variable remains the
1034 same as if this interaction had never occurred:

1035 ?? A requesting participant completes the current interaction. The requesting participant
1036 may create a new interaction to recover from this failure as part of the same
1037 Choreography or of a completely new Choreography.
1038 ?? An accepting participant completes the current interaction.
1039

1040 ### 2.5.3 Performed Choreography

1041 The Performed Choreography Structure enables a Choreography to define that a
1042 separately defined Choreography is to be performed. The Choreography that is
1043 performed can be defined either within the same Choreography Definition or separately.
1044

1045 ### 2.5.4. Assigning Variables

1046 *Assign* populates, within one Role, the value of one Variable using the value of a
1047 Variable or Token, or makes a Token reference a Variable or another Token.
1048

1049 The assignments may include:
1050 ?? Assigning one *Information Exchange Variable* to another, for example so that a
1051 Choreography can define that a message received by one role is forwarded to
1052 another.
1053 ?? Assigning a *Locally Defined Variable* to part of the data contained in an Information
1054 Exchange Variable.
1055

1056 ### 2.5.5 Defining actions with no business effect

1057 The Noaction activity models the performance of a business action that has no business
1058 effect to any of the collaborating participants. Examples of its use include:
1059 ?? In a *Work Unit*, when there is a need to wait until the Guard condition on the *Work*
1060 *Unit* is true, for example you need to wait until say three separate Interactions are
1061 complete before progressing to the next step in the Choreography
1062 ?? In a *Choice* so that you can enumerate all the possible choices even if some of the
1063 choices involve no Interactions.
1064

1065 ### 2.5.6 Finalization Command

1066 The Finalize activity models issuing a finalization command from a parent Choreography
1067 to the Finalization WorkUnit of an enclosed Choreography.
1068

1068 # 3 Example

1069 This example depicts a multi-party choreography illustrating a simple purchase
1070 sequence. The choreography involves four parties; a Buyer, Seller, Credit Checking
1071 Service and Inventory Service.
1072
1073 The Seller, upon receiving the Purchase Order from the buyer, initiates 3 interactions:
1074 ?? one with the Buyer to acknowledge the receipt of the Purchase Order
1075 ?? one with the Credit Checking Service to check the credit of the Buyer
1076 ?? one with the Inventory Service to check the inventory of the product ordered by the
1077 Buyer.
1078
1079 The Seller waits for the response from both the Credit Checking Service and Inventory
1080 Service. If both responses are positive, the order is processed and the Purchase Order
1081 Response is sent.
1082
1083 If either of the responses from Credit Checking Service or Inventory Service are
1084 negative, then a Purchase Order Reject message is sent to the Buyer.
1085
1086 In the example below, the main choreography involves 3 relationships:
1087 ?? Buyer-Seller
1088 ?? Seller-Credit Checking Service
1089 ?? Seller-Inventory
1090
1091 Within this main choreography, we have 3 sub-Choreographies, one for each
1092 relationship. The various Work Units with guards on them reflect the ordering of the
1093 activities.
1094 The Work Unit 'purchaseDocAvailibility' has a guard on 'purchaseOrderDocAtSeller' that
1095 triggers when a Purchase Order document is received by the Seller.
1096
1097 Similarly, the Work Unit with the guard 'creditApprovalInventoryApproval' is triggered
1098 when the query on 'CreditCheckResponseDocAtSeller' and
1099 'InventoryResponseDocAtSeller' result in positive answers from the Credit Checking
1100 Service and Inventory Service.
1101

1101 **WSDL Definitions**

```
1102
1103 <definitions name="PurchaseOrderDefs"
1104     targetNamespace="urn:purchaseOrder:purchaseOrderDefs"
1105     xmlns:tns="urn:purchaseOrder:purchaseOrderDefs"
1106     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
1107     xmlns="http://schemas.xmlsoap.org/wsdl/">
1108
1109     <types>
1110        <xsd:complexType name="purchaseOrderType">
1111            <xsd:element name="CID" type="xsd:string"/>
1112            <xsd:element name="Order" type="xsd:int"/>
1113            <!-- more stuff-->
1114        </xsd:complexType>
1115     </types>
1116        <types>
1117        <xsd:complexType name="purchaseOrderAckType">
1118            <xsd:element name="CID" type="xsd:string"/>
1119            <xsd:element name="Order" type="xsd:int"/>
1120              <!-- more stuff-->
1121        </xsd:complexType>
1122     </types>
1123     <types>
1124        <xsd:complexType name="purchaseOrderResponseType">
1125            <xsd:element name="CID" type="xsd:string"/>
1126            <xsd:element name="Order" type="xsd:int"/>
1127              <!-- more stuff-->
1128        </xsd:complexType>
1129     </types>
1130     <types>
1131        <xsd:complexType name="purchaseOrderRejectType">
1132            <xsd:element name="CID" type="xsd:string"/>
1133            <xsd:element name="Order" type="xsd:int"/>
1134            <xsd:element name="reason" type="xsd:string"/>
1135              <!-- more stuff-->
1136        </xsd:complexType>
1137     </types>
1138     <types>
1139        <xsd:complexType name="inventoryCheckRequestType">
1140            <xsd:element name="productID" type="xsd:int"/>
1141            <xsd:element name="productQty" type="xsd:int"/>
1142              <!-- more stuff-->
1143        </xsd:complexType>
1144     </types>
1145     <types>
1146        <xsd:complexType name="inventoryCheckResponseType">
1147            <xsd:element name="productID" type="xsd:int"/>
1148            <xsd:element name="productQty" type="xsd:int"/>
1149              <!-- more stuff-->
1150        </xsd:complexType>
1151     </types>
1152     <types>
1153        <xsd:complexType name="creditCheckRequestType">
1154            <xsd:element name="CustomerID" type="xsd:int"/>
1155            <xsd:element name="SSN" type="xsd:string"/>
1156              <!-- more stuff-->
```

```
1157          </xsd:complexType>
1158       </types>
1159       <types>
1160          <xsd:complexType name="creditCheckResponseType">
1161              <xsd:element name="CustomerID" type="xsd:int"/>
1162              <xsd:element name="SSN" type="xsd:string"/>
1163                <!-- more stuff-->
1164          </xsd:complexType>
1165       </types>
1166       <message name="purchaseOrderMsg">
1167           <part name="PO" type="purchaseOrderType"/>
1168       </message>
1169       <message name="purchaseOrderAckMsg">
1170           <part name="PO" type="purchaseOrderAckType"/>
1171       </message>
1172       <message name="purchaseOrderResponseMsg">
1173           <part name="PO" type="purchaseOrderResponseType"/>
1174       </message>
1175       <message name="purchaseOrderRejectMsg">
1176           <part name="PO" type="purchaseOrderRejectType"/>
1177       </message>
1178       <message name="creditRequestMsg">
1179           <part name="Credit" type="creditRequestType"/>
1180       </message>
1181       <message name="creditResponseMsg">
1182           <part name="Credit" type="creditResponseType"/>
1183       </message>
1184       <message name="inventoryRequestMsg">
1185           <part name="Inventory" type="inventoryRequestType"/>
1186       </message>
1187       <message name="inventoryResponseMsg">
1188           <part name="Inventory" type="inventoryResponseType"/>
1189       </message>
1190      <!-- more stuff-->
1191 </definitions>
1192
1193
```

1194 **Buyer Service**

```
1195
1196 <definitions name="BuyerService"
1197    targetNamespace="urn:purchaseOrder:buyerService"
1198    xmlns:pons="urn:purchaseOrder:purchaseOrderDefs"
1199     xmlns="http://schemas.xmlsoap.org/wsdl/">
1200
1201   <interface name="BuyerPT">
1202      <operation name="createOrderAck">
1203         <input message="pons:purchaseOrderAckMsg"/>
1204      </operation>
1205
1206      <operation name="createOrderResponse">
1207      <input message="pons:purchaseOrderResponseMsg"/>
1208      </operation>
1209      <operation name="createOrderReject">
1210      <input message="pons:purchaseOrderRejectMsg"/>
1211      </operation>
1212   </interface>
```

1213
```
</definitions>
```

Seller Service

```
<definitions name="SellerService"
   targetNamespace="urn:purchaseOrder:sellerService"
   xmlns:pons="urn:purchaseOrder:purchaseOrderDefs"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

  <interface name="SellerPOPT">
    <operation name="createOrder">
      <input message="pons:purchaseOrderMsg"/>
    </operation>
  </interface>

  <interface name="SellerCreditPT">
    <operation name="creditResponse">
      <input message="pons:creditResponseMsg"/>
    </operation>
  </interface>

  <interface name="SellerInventoryPT">
    <operation name="inventoryResponse">
      <input message="pons:inventoryResponseMsg"/>
    </operation>
  </interface>
</definitions>
```

Credit Service

```
<definitions name="CreditService"
   targetNamespace="urn:purchaseOrder:creditService"
   xmlns:pons="urn:purchaseOrder:purchaseOrderDefs"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

  <interface name="CreditPT">
    <operation name="creditCheck">
      <input message="pons:creditRequestMsg"/>
    </operation>
  </interface>
 </definitions>
```

Inventory Service

```
<definitions name="InventoryService"
   targetNamespace="urn:purchaseOrder:inventoryService"
   xmlns:pons="urn:purchaseOrder:purchaseOrderDefs"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

  <interface name="InventoryPT">
    <operation name="inventoryCheck">
      <input message="pons:inventoryRequestMsg"/>
    </operation>
```

```
  </interface>
 </definitions>
```

Choreography Example

```
<package
   name="purchaseOrderChoreography"
   version="1.0"
   targetNamespace="urn:purchaseOrder:purchaseOrderChoreography"
   xmlns:pons="urn:purchaseOrder:purchaseOrderDefs"
   xmlns:bns="urn:purchaseOrder:buyerService"
   xmlns:sns="urn:purchaseOrder:sellerService"
   xmlns:cns="urn:credit:creditService"
   xmlns:ins="urn:inventory:inventoryService"
   xmlns:wsrf="http://www.wsref.com"
   xmlns="http://www.w3.org/ws/choreography/2004/02/CDL"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl">

  <importDefinitions>
       <import namespace=”http://otn.oracle.com/”
               location=”http://otn.oracle.com/ws/choreo/sample”/>
  </importDefinitions>

   <!-- Information Type definitions -->
   <informationType  name="purchaseOrderDocType"
             messageType="pons:purchaseOrderMsg"/>
   <informationType  name="purchaseOrderAckDocType"
             messageType="pons:purchaseOrderAckMsg"/>
   <informationType  name="purchaseOrderRejectDocType"
             messageType="pons:purchaseOrderRejectMsg"/>
   <informationType  name="purchaseOrderResponseDocType"
             messageType="pons:purchaseOrderResponseMsg"/>
   <informationType  name="creditCheckRequestDocType"
             messageType="pons:creditCheckRequestMsg"/>

   <informationType  name="creditCheckResponseDocType"
             messageType="pons:creditCheckResponseMsg"/>
   <informationType  name="inventoryCheckRequestDocType"
             messageType="pons:inventoryCheckRequestMsg"/>

   <informationType  name="inventoryCheckResponseDocType"
             messageType="pons:inventoryCheckResponseMsg"/>

   <!-- Token and token locators -->
   <token
      name="purchaseOrderID" type="xsd:int"/>
   <tokenLocator tokenName="sns:purchaseOrderID"
           informationType ="purchaseOrderDocType"
           part="PO"
           query="/PO/Order"/>
   <tokenLocator tokenName="sns:purchaseOrderID"
           informationType ="purchaseOrderResponseDocType"
           part="PO"
           query="/PO/Order"/>
```

```
1325      <token
1326         name="customerID" type="xsd:int"/>
1327      <tokenLocator tokenName="sns:customerID"
1328               informationType ="creditRequestDocType"
1329               part="Credit"
1330               query="/CreditCheck/CustomerID"/>
1331      <tokenLocator tokenName="sns:customerID"
1332               informationType ="creditResponseDocType"
1333               part="Credit"
1334               query="/CreditCheckResponse/CustomerID"/>
1335
1336      <token
1337         name="productID" type="xsd:int"/>
1338      <tokenLocator tokenName="sns:productID"
1339               informationType ="inventoryRequestDocType"
1340               part="Inventory"
1341               query="/InventoryCheck/ProductID"/>
1342      <tokenLocator tokenName="sns:productID"
1343               informationType ="inventoryResponseDocType"
1344               part="Inventory"
1345               query="/InventoryCheckResponse/ProductID"/>
1346
1347      <token
1348         name="buyerRef" type="anyURI"/>
1349      <token
1350         name="warehouseRef" type="anyURI"/>
1351      <token
1352         name="creditRef" type="anyURI"/>
1353      <token
1354         name="inventoryRef" type="anyURI"/>
1355
1356
1357      <!-- Role definitions -->
1358      <role name="Buyer">
1359           <behavior name="buyer" interface="bns:BuyerPT"/>
1360      </role>
1361
1362      <!-- The role seller has 3 interfaces one for buyer, one for
1363            credit and one for inventory-->
1364      <role name="Seller">
1365           <behavior name="sellerForBuyer"
1366                         interface="sns:SellerPOPT"/>
1367           <behavior name="sellerForCredit"
1368                         interface="sns:SellerCreditPT"/>
1369           <behavior name="sellerForInventory"
1370                         interface="sns:SellerInventoryPT"/>
1371      </role>
1372
1373      <role name="Credit">
1374           <behavior name="buyer" interface="cns:CreditPT"/>
1375      </role>
1376
1377      <role name="Inventory">
1378           <behavior name="buyer" interface="ins:InventoryPT"/>
1379      </role>
1380
1381      <!-- Relationship definitions -->
```

```
1382     <relationship name="BuyerSellerBinding">
1383          <role type="Buyer"/>
1384          <role type="Seller"/>
1385     </relationship>
1386
1387     <relationship name="SellerCreditBinding">
1388          <role type="Buyer"/>
1389          <role type="Seller"/>
1390     </relationship>
1391
1392     <relationship name="SellerInventoryBinding">
1393          <role type="Buyer"/>
1394          <role type="Seller"/>
1395     </relationship>
1396
1397     <!-- Channel Type definitions -->
1398     <channelType  name="purchaseOrderCHT">
1399          <role type="Seller" behavior="sellerForBuyer"/>
1400          <reference>
1401               <token name="warehouseRef"/>
1402          </reference>
1403          <identity>
1404               <token name="purchaseOrderID"/>
1405          </identity>
1406     </channelType>
1407
1408     <channelType  name="purchaseOrderResponseCHT">
1409          <role type="Buyer"/>
1410          <reference>
1411               <token name="buyerRef"/>
1412          </reference>
1413          <identity>
1414               <token name="purchaseOrderID"/>
1415          </identity>
1416     </channelType>
1417
1418      <channelType  name="creditCheckCHT">
1419          <role type="Credit"/>
1420          <reference>
1421               <token name="creditRef"/>
1422          </reference>
1423          <identity>
1424               <token name="customerID"/>
1425          </identity>
1426     </channelType>
1427
1428      <channelType  name="creditCheckResponseCHT">
1429          <role type="Seller" behavior="sellerForCredit"/>
1430          <reference>
1431               <token name="warehouseRef"/>
1432          </reference>
1433          <identity>
1434               <token name="customerID"/>
1435          </identity>
1436     </channelType>
1437
1438
```

```
1439      <channelType  name="inventoryCheckCHT">
1440           <role type="Credit"/>
1441           <reference>
1442                <token name="inventoryRef"/>
1443           </reference>
1444           <identity>
1445                <token name="productID"/>
1446           </identity>
1447      </channelType>
1448
1449      <channelType  name="inventoryCheckResponseCHT">
1450           <role type="Seller" behavior="sellerForInventory"/>
1451           <reference>
1452                <token name="warehouseRef"/>
1453           </reference>
1454           <identity>
1455                <token name="productID"/>
1456           </identity>
1457      </channelType>
1458
1459
1460      <choreography name="purchaseOrderChoreograpy" root="true">
1461         <relationship type="BuyerSellerBinding"/>
1462         <relationship type="SellerCreditBinding"/>
1463         <relationship type="SellerInventoryBinding"/>
1464
1465         <variableDefinitions  name="purchaseOrderChoreographyVariable">
1466              <variable  name="purchaseOrderDocAtSeller"
1467                          informationType="purchaseOrderDocType"
1468                          role="Seller"/>
1469              <variable  name="creditCheckResponseDocAtSeller"
1470                          informationType="creditCheckResponseDocType"
1471                          role="Seller"/>
1472              <variable  name="inventoryCheckResponseDocAtSeller"
1473                          informationType="inventoryCheckResponseDocType"
1474                          role="Seller"/>
1475              <variable  name="seller-channel-for-buyer"
1476                          channelType="purchaseOrderCHT"/>
1477         </variableDefinitions>
1478
1479
1480         <workunit name="programPurchaseOrder">
1481            <!-- A sub choreography for buyer/seller interactions -->
1482            <choreography
1483                   name="buyerSellerChoreography">
1484              <relationship="BuyerSellerBinding"/>
1485
1486              <variableDefinitions
1487                   name="purchaseOrderChoreographyVariable">
1488
1489                 <variable  name="purchaseOrderAckDoc"
1490                          informationType="purchaseOrderAckDocType"/>
1491                 <variable  name="purchaseOrderDocAtBuyer"
1492                          informationType="purchaseOrderDocType"
1493                          role="Buyer"/>
1494                 <variable  name="purchaseOrderResponseDoc"
```

```
1495                           informationType="purchaseOrderResponseDocType"/>
1496                 <variable  name="purchaseOrderRejectDoc"
1497                           informationType="purchaseOrderRejectDocType"/>
1498                 <variable  name="buyer-channel"
1499                           channelType="purchaseOrderResponseCHT"/>
1500               </variableDefinitions>
1501               <workunit name="purchaseOrder">
1502                 <!-- This is the first interaction, hence we assume that
1503                      the buyer and seller
1504                 have negotiated and know the seller-channel-for-buyer -->
1505
1506                 <interaction name="createOrderInteract"
1507                     channel="seller-channel-for-buyer"
1508                     operation="createOrder"
1509                     messageContentType="purchaseOrderDocType"
1510                     initiateChoreography="true">
1511
1512                     <participate relationship="BuyerSellerBinding"
1513                        fromRole="Buyer"
1514                        toRole="Seller"/>
1515                     <align variable="purchaseOrderDocAtBuyer"
1516                            with-variable="purchaseOrderDocAtSeller"/>
1517                     <align variable="buyer-channel"
1518                            with-variable="buyer-channel"/>
1519                 </interaction>
1520
1521                 <!-- This is not a response, but just an acknowledgement
1522                    -->
1523                 <interaction name="createOrderAckInteract"
1524                     channel="buyer-channel"
1525                     operation="createOrderAck"
1526                     messageContentType="purchaseOrderDocType">
1527                     <participate relationship="BuyerSellerBinding"
1528                       fromRole="Seller"
1529                       toRole="Buyer"/>
1530                     <align variable="purchaseOrderAckDoc"
1531                            with-variable="purchaseOrderAckDoc"/>
1532                 </interaction>
1533
1534                 <!-- this is a sub choreography of buyerSellerChoreography
1535                      because we want to share the scope of buyer-channel
1536                     between the two -->
1537                 <choreography
1538                    name="sellerResponseChoreography">
1539                    <relationship=”BuyerSellerBinding"/>
1540
1541                    <!-- The group name ApproveOrReject indicates that
1542                         all the Work Units that belong to this group are
1543                          mutually exclusive. When one is fired, the
1544                          others are disabled.-->
1545                    <workunit name="creditApprovalInventoryApproval"
1546                           guard=
1547 "cdl:getVariable(CreditCheckResponseDocAtSeller, \“/Customer/Credit\”)
1548 [text() = 'success'] && cdl:getVariable(InventoryResponseDocAtSeller,
1549 \“/Order/Inventory\”) [text()='available']">
1550
1551                      <interaction name="createOrderResponse"
```

```
1552                        channel="buyer-channel"
1553                        operation="createOrderResponse"
1554                        messageContentType="purchaseOrderDocType">
1555                        <participate relationship="BuyerSellerBinding"
1556                            fromRole="Seller"
1557                            toRole="Buyer"/>
1558                        <align variable="purchaseOrderResponseDoc"
1559                               with-variable="purchaseOrderResponseDoc"/>
1560                    </interaction>
1561                 </workunit>
1562              </choreography>
1563
1564
1565              <!-- this is a sub choreography of buyerSellerChoreography
1566                     because we want to share the scope of buyer-channel
1567                 between the two. -->
1568              <choreography
1569                      name="sellerRejectChoreography">
1570                 <relationship=”BuyerSellerBinding"/>
1571
1572                 <!--workunit belongs to group ApproveOrReject.-->
1573                 <workunit name="creditInventoryRejection"
1574                        guard=
1575  "cdl:getVariable(CreditCheckResponseDocAtSeller,
1576  \“/Customer/Credit\”)[text='failedToApprove'] ||
1577  cdl:getVariable(InventoryResponseDocAtSeller,
1578  \“/Order/Inventory\”)[text= 'notAvailable']">
1579
1580                    <interaction name="createOrderReject"
1581                         channel="buyer-channel"
1582                         operation="createOrderReject"
1583                         messageContentType="purchaseOrderDocType">
1584                         <participate relationship="BuyerSellerBinding"
1585                            fromRole="Seller"
1586                            toRole="Buyer"/>
1587                         <align variable="purchaseOrderRejectDoc"
1588                                with-variable="purchaseOrderRejectDoc"/>
1589                    </interaction>
1590                 </workunit>
1591              </choreography>
1592           </workunit>
1593        </choreography>
1594
1595
1596        <!-- Choreography to check credit of the buyer -->
1597        <choreography name="creditCheckChoreography">
1598           <relationship type="SellerCreditBinding"/>
1599           <variableDefinitions  name=”creditCheckChoreographyVariable”>
1600              <variable  name="creditCheckRequestDoc"
1601                         informationType="creditCheckRequestDocType"/>
1602              <variable  name="creditCheckResponseDocAtCredit"
1603                         informationType="creditCheckResponseDocType"
1604                         role="Credit"/>
1605              <variable  name="seller-channel-for-credit"
1606                         channelType="tns:creditCheckResponseCHT"/>
1607           </variableDefinitions>
1608
```

```
1609            <!-- When purchaseOrderDocAtSeller is available at seller,
1610                 this workunit is enabled -->
1611            <workunit name="purcaseDocAvailibility"
1612                      guard="purchaseOrderDocAtSeller">
1613
1614               <interaction name="creditCheck"
1615                     channel="credit-channel"
1616                     operation="checkCredit"
1617                     messageContentType="purchaseOrderDocType"
1618                     initiateChoreography="true">
1619                     <participate relationship="SellerCreditBinding"
1620                         fromRole="Seller"
1621                         toRole="Credit"/>
1622                     <align variable="creditCheckRequestDoc"
1623                            with-variable="creditCheckRequestDoc"/>
1624                     <align variable="seller-channel-for-credit"
1625                           with-variable="seller-channel-for-credit"/>
1626               </interaction>
1627
1628               <interaction name="creditCheckResponse"
1629                    channel="seller-channel-for-credit"
1630                    operation="creditResponse"
1631                    messageContentType="purchaseOrderDocType">
1632                    <participate relationship="SellerCreditBinding"
1633                           fromRole="Credit"
1634                           toRole="Seller"/>
1635                   <align variable="creditCheckResponseDocAtCredit"
1636                           with-variable="
1637 creditCheckResponseDocAtSeller"/>
1638               </interaction>
1639            </workunit>
1640         </choreography>
1641
1642         <!-- Choreography to check inventory of the products ordered
1643               by the buyer -->
1644         <choreography name="inventoryCheckChoreography">
1645            <relationship type="SellerInventoryBinding"/>
1646            <variableDefinitions
1647 name=”inventoryCheckChoreographyVariable”>
1648               <variable  name="inventoryCheckRequestDoc"
1649                  informationType="inventoryCheckRequestDocType"/>
1650               <variable  name="inventoryCheckResponseDocAtInventory"
1651                  informationType="inventoryCheckResponseDocType"
1652                  role="Inventory"/>
1653                <variable  name="seller-channel-for-inventory"
1654                  channelType="inventoryCheckResponseCHT"/>
1655            </variableDefinitions>
1656
1657         <!-- When purchaseOrderDocAtSeller is available at seller,
1658            this workunit is enabled. -->
1659            <workunit name="purcaseDocAvailibility"
1660                    guard="purchaseOrderDocAtSeller">
1661               <interaction name="inventoryCheck"
1662                   channel="inventory-channel"
1663                   operation="inventoryCredit"
1664                   messageContentType="purchaseOrderDocType"
1665                   initiateChoreography="true">
```

```
1666                    <participate relationship="SellerInventoryBinding"
1667                             fromRole="Seller"
1668                             toRole="Inventory"/>
1669                    <align variable="inventoryRequestDoc"
1670                           with-variable="inventoryRequestDoc"/>
1671                </interaction>
1672
1673                <interaction name="inventoryResponse"
1674                    channel="seller-channel-for-inventory"
1675                    operation="inventoryResponse"
1676                    messageContentType="purchaseOrderDocType">
1677                    <participate relationship="SellerInventoryBinding"
1678                               fromRole="Inventory"
1679                               toRole="Seller"/>
1680                    <align variable="inventoryResponseDocAtInventory"
1681                           with-variable="inventoryResponseDocAtSeller"/>
1682                </interaction>
1683             </workunit>
1684          </choreography>
1685       </workunit>
1686    </choreography>
1687 </package>
1688
```

4 Language Elements

This section describes the WS-CDL language constructs in detail. We first introduce some principles and notions that apply to all or most of the language elements.

4.1 Choreography Document Structure

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

4.1.1 Choreography document Naming and Linking

WS-CDL documents can be assigned an optional name attribute of type NCNAME that serves as a lightweight form of documentation. Optionally, a targetNamespace attribute of type URI may 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].

4.1.2 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 MUST use an XML namespace different from that of WS-CDL. All extension namespaces used in a WS-CDL document MUST be declared. An extension namespace is declared by using the following syntax:

```
<extension namespace="anyURI"/>*
```

Extensions MUST NOT change the semantics of any element or attribute from the WSCDL namespace.

4.1.3 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 attribute is allowed inside any WS-CDL language element.

The information provided by the description attribute 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.

1734 Descriptions that are *Text* or *Document References* can be defined in multiple different
1735 human readable languages.
1736
1737 WS-CDL uses the optional name attribute for providing a name that can be used to
1738 reference some language constructs. This attribute is used inside some WS-CDL
1739 language elements. Language construct definitions MUST have distinct names.
1740

4.2 Choreography Package

The **package** construct allows aggregating a set of business related collaborations, where the elements informationType, tokenType, tokenLocator, roleType, relationshipType and channelType are shared by all choreographies.

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 name, author, version, and description define authoring properties of the Choreography document.

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

?? One or more Information Types

?? One or more Token types and Token Locators

?? One or more Role types

?? One or more Relationship types

?? One or more Channel types

?? One or more Interaction types

?? Zero or more, package-level Choreographies

The syntax of this construct is:

```
<package
     name="ncname"
     author="xsd:string"?
     version="xsd:string"
     description="xsd:string"?
     targetNamespace="uri"
     xmlns="http://www.w3.org/ws/choreography/2004/02/WSCDL/">

        importDefinitions+

        informationType+

        token+

        tokenLocator+

        role+

        relationship+

        participant+
```

```
            channelType+

         ChoreographyNotation*

     </package>
```

4.2.1 Importing definitions

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

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

The syntax of this 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 definitions that MUST be imported into this Package.

4.3 Roles

A Role identifies a set of related behaviors.

The **role** construct allows the definition of the observable behavior, in terms of the observable operations a participant MUST provide.

A role is associated with one process type and one or more Web Service interface types.

The syntax of this construct is:

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

The role element defines an optional processType attribute, which identifies the process type a participant implements. The behavior element defines an optional interface attribute, which identifies a WSDL interface type the process exhibits.

A role that defines a behavior without an interface describes a simple client role without any Web Service interface, i.e. it never acts as an accepting role in a relationship.

4.4 Participants

A Participant identifies a set of related Roles.

The syntax of this construct is:

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

4.5 Relationships

A Relationship is the association of two Roles.

The **relationship** construct allows the definition of the observable behavior that two collaborating participants are mutually agreeing to exhibit.

The syntax of this construct is:

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

   <role  type="ncname"  behavior="ncname"? />
</relationship>
```

A relationship has two role types defined.

The optional behavior attribute points to a behavior type within the role type specified by the type attribute of the role element.

4.6 Channels

A Channel Type facilitates the collaboration between two Participants. A Channel identifies where and how to send/receive information to/into a Role.

Additionally, it identifies what is the allowed Channel information that can be passed from a Role to another Role and the usage of a Channel within each participant.

The **channelType** construct defines a collaboration point, used for communication and synchronization between participants.

The syntax of this construct is:

```
<channelType  name="ncname"

    usage="once"|"unlimited"

    viewpoint="common"|"participant"?
    action="request"|"receive"|"respond" >

    <channelDefinitions>?
        <channel  type="ncname"
                  action="request"|"receive"|"respond"
                  new="true"|"false" >+
```

```
1878            </channelDefinitions>
1879
1880            <role  type="ncname"  behavior="ncname"? />
1881
1882            <reference>
1883               <token  name="ncname"/>+
1884            </reference>
1885
1886            <identity>
1887               <token  name="ncname"/>+
1888            </identity>*
1889        </channelType>
1890
1891
```

1891 The content varies depending on the type of the choreography:
1892

Choreography Type	**Channel**
Abstract	In an Abstract Choreography, the Channel Type is described by: ✍ A unique identifier, e.g. a URI that identifies the Channel Type within the Role ✍ A semantic definition, that describes the type of channel information that the Channel can accept. Including: o What channel information can be passed using this channel type o How a channel should be used
Portable	In a Portable Choreography, the abstract Channel Type is extended by identifying: ✍ One or more WSDL Service Interfaces that collectively implement the channel type. ✍ How Correlation of the messages sent using the Channel Type is to be done
Concrete	Channel Types in a Concrete Choreography are defined in the same way as for a Portable Choreography.

1893

1894 The element usage is used to restrict the number of times a channel can be used within
1895 a Choreography.

1896 The optional element channelDefinitions prescribes the channel(s) allowed to be
1897 exchanged from one Role to another Role, when using this Channel in an interaction. In
1898 the case where the operation used to exchange the Channel is of request-response
1899 type, then the attribute direction within the channelDefinitions element defines if the
1900 Channel will be exchanged during the request or during the response. The Channels
1901 exchanged can be used in subsequent interaction activities. If the element
1902 channelDefinitions is missing then this Channel can only be used for exchanging business
1903 documents.

1904 The element role is used to statically identify the collaboration point.

1905 The element reference MAY be used for identifying the WSDL service reference type of a
1906 participant. The service type of a participant is distinguished by a set of Token types as
1907 specified by the token element within the reference element.

1908 The optional element identity MAY be used for identifying an instance of a business
1909 process implementing a participant's role and for identifying a logical conversation
1910 (group of related message exchanges) between participants within a Choreography. The
1911 business process type within a participant and the different conversations between
1912 participants are distinguished by a set of Token types as specified by the token element
1913 within the identity element.
1914

4.7 Information Types

Information Types describe the type of information that is being captured within a Variable at a Role.

The syntax of this construct is:

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

The type of information that is referenced will vary depending on the type of the Choreography and the type of information that the variable contains.

Choreography Type	Information Type
Abstract	In an Abstract Choreography, the Information Type is described by: ✍ A unique identifier, e.g. a URI, that identifies the information Type and ✍ A semantic definition that explains the purpose of the information Type and outlines its content. No detail is provided of the actual type, e.g. XSD definitions
Portable	In a Portable Choreography the Information Type extends the Abstract Information Type by defining its XML Schema Type. Note that this may be simple or complex depending on the need.
Concrete	In a Concrete Choreography, Information Type is defined in the same way as for a Portable Choreography

The **informationType** construct specifies the type of information used within a Choreography. The attributes messageType, type, and element describe the document to be a WSDL message type, an XML Schema simple type, or an XML Schema element respectively. The document is of one of these types exclusively.

4.8 Tokens and Token Locators

4.8.1 Tokens

A Token is an alias for a piece of data in a variable or message that needs to be used by a Choreography.

The syntax of this construct is:

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

The **token** construct describes the naming and typing of document fragments. The named fragments are called Token types.

The way these tokens are defined will vary depending on the type of choreography.

Choreography Type	Tokens
Abstract	In an abstract choreography, tokens are described by: ✍ A unique identifier, e.g. a URI that identifies the token ✍ A semantic definition, that describes what the token means However Abstract tokens do not have a type.
Portable	In a portable choreography, a token extends an Abstract definition of a token by defining: ✍ Its type, e.g. by giving it an XML Schema type ✍ A reference to the location of the item, for example using an XML Path expression
Concrete	Tokens in a Concrete Choreography are defined in the same way as for a Portable Choreography.

The attribute informationType identifies the type of the document fragment.

4.8.2 Token Locators

The **tokenLocator** construct defines a locator for selecting a document fragment within a document as a token using the part and the query attributes.

The syntax of this construct is:

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

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

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

The optional attribute part identifies the part of the document, if any, on which the query is performed to locate the token.

1967 The attribute query defines the query string that is used to select a document fragment
1968 within a document.
1969

1970 4.9 Variables

1971 Variables capture information about objects in a Choreography.
1972

1973 The **variableDefinitions** is used for declaring one or more variables within a
1974 Choreography block. The location of the variableDefinitions within a Choreography defines
1975 the visibility of the variable information.

1976 The syntax is as follows:
1977

```
1978         <variableDefinitions>
1979             <variable    name="ncname"
1980                          informationType="ncname"?
1981                          channelType="ncname"?
1982                          mutable="true|false"
1983                          free="true|false"
1984                          observable="true|false"
1985                          role="ncname"? />+
1986         </variableDefinitions>
```

1987
1988
1989 The declared variables can be of the following types:
1990 ?? Information Exchange Variables, State Variables. The attribute informationType
1991 describes the type of the variable.
1992 ?? Channel Variables: The attribute channelType describes the type of the Channel.
1993
1994 The way Variables are declared will vary depending on the type of choreography.

Choreography Type	Variables
Abstract	In an abstract choreography, variables are described by: ✍ An Role name that identifies the role within which the variable is known ✍ A name that identifies the variable, that is unique within the Role within the Choreography Definition ✍ A semantic definition, that describes what the variable means
Portable	In a portable choreography, the abstract definition of the Variables is extended to include a Information Type, which define what type of information the variable contains
Concrete	Variables in a Concrete Choreography are defined in the same way as for a Portable Choreography.

1995

The attribute mutable, when set to “false” describes that the variable information when initialized, cannot change anymore. The attribute observable, when set to “false” describes that the method of the variable information initialization with an actual value is not of relevance to other participants. The element role is used to specify where variable information resides.

4.9.1 Expressions

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

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

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

4.10 Choreography Definition

The **choreography** construct allows specifying compositions of Web Service participants by asserting their common observable behaviors where computation progress occurs by exchanging messages in a predetermined order.

The **ChoreographyNotation** is as follows:

Define a root or a base Choreography. A base Choreography MAY be enclosed within another Choreography using the other type of ChoreographyNotation.

The syntax of this construct is:

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

   ChoreographyNotation*

   <relationship  type="ncname">+

    variableDefinitions?

    <select>
        WorkUnitNotation+
    </select>*
    WorkUnitNotation+

    <exception  name="ncname">
        WorkUnitNotation+
    </exception>?

    <finalization  name="ncname">
```

2039 WorkUnitNotation

2040 </finalization>?

2041 </choreography>

2042

2043 The complete attribute allows to explicitly complete a Choreography.

2044 The ChoreographyNotation within the choreography element declares the Choreographies
2045 that MAY be used through a perform activity within this Choreography.

2046 The relationship element within the choreography element enumerates the relationships
2047 this Choreography MAY participate in.

2048 The variableDefinitions element enumerates the variable information holders, shared
2049 between roles within this Choreography and all its enclosed activities.

2050 The root element marks a base Choreography as the root Choreography of a Package.

2051 A Choreography MUST have one or more Work Unit Notations.

2052 A Choreography MAY have one or more workunit elements within a Select group. The
2053 element select allows specifying Work Units belonging to a Select group. When a
2054 workunit matches in a Select group then all other Work Units belonging to the same
2055 Select group are disabled.

2056

2057 A Choreography MAY define a recovery block. One or more Exception WorkUnits MAY
2058 be defined as part of the Choreography to recover from exceptional conditions that may
2059 occur in that enclosing Choreography. Additionally one Finalization WorkUnit MAY be
2060 defined as part of the Choreography to provide the finalization activities for that
2061 enclosing Choreography.

2062

2063 **4.10.1 WorkUnit**

2064 Within a Choreography, a workunit expresses interest on variable information before
2065 enabling its enclosed activities identified by the Activity notation.

2066

2067 The **WorkUnitNotation** is defined as follows:

2068

```
2069            <workunit  name="ncname"
2070                  guard="xsd:boolean XPath-expression”?
2071                  repeat="xsd:boolean XPath-expression”? >
2072                ActivityNotation
2073            </workunit>+
```

2074

2075 The guard attribute describes the interest on the availability of one or more, existing or
2076 future variable information. If a guard is not specified then the workunit always matches
2077 and the activities enclosed within the workunit are enabled. When the variable
2078 information specified by a guard become available and the guard condition (which
2079 references the matched variable) evaluates to true, then the workunit matches and the
2080 activities enclosed within the workunit are enabled.

2081 The repeat attribute allows, when the condition it specifies evaluates to true, to make the
2082 current Work Unit considered again for matching (based on the guard condition
2083 attribute).
2084

2085 ## 4.11 Activities Definition

2086 An **ActivityNotation** is a ChoreographyNotation, a ControlNotation or a Basic activity.
2087

2088 ### 4.11.1 Control Structure

2089 A Control structure contains one or more activities that are all enabled in parallel, in
2090 sequence or when a condition evaluates to true.

2091 The **ControlNotation** is as follows:

2092 #### 4.11.1.1 Sequence

2093 A Control structure contains two or more activities that are enabled when the sequence
2094 activity is enabled. The sequence element restricts the series of enclosed activities to be
2095 enabled sequentially.

2096 The syntax of this construct is:

```
2097                <sequence>
2098                   ActivityNotation+
2099                </sequence>
```

2100

2101 #### 4.11.1.2 Parallel

2102

2103 A Control structure contains two or more activities that are enabled when the parallel
2104 activity is enabled. The parallel element allows all enclosed activities to be enabled
2105 concurrently.

2106 The syntax of this construct is:

```
2107                <parallel>
2108                   ActivityNotation+
2109                </parallel>
```

2110

2111 #### 4.11.1.3 Choice

2112 A Control structure contains one or more activities from which one is enabled when the
2113 condition specified by the predicate attribute of the choice-branch element evaluates to
2114 true.

2115 The syntax of this construct is:

```
2116                <choice>
2117                  <choice-branch  name="ncname"?
2118                          predicate="xsd:boolean XPath-expression"? >
2119                     ActivityNotation
```

```
            </choice-branch>+
          </choice>
```

4.11.2 Basic activities

4.11.2.1 Perform activity

The perform activivity allows composing recursive Choreographies to form new Choreographies. Within the perform element the choreographyName references a base, non-root Choreography defined in the same or in a different collaboration Package.

The syntax of this construct is:

```
<perform  choreographyName=”qname”>
     <alias>
         <this  variable=“ncname”/>
         <free  variable="ncname"/>
     </alias>+
</perform>
```

4.11.2.2 Interaction activity

A WS-CDL **interaction** always involves the exchange of information between two Roles in a Relationship.

The syntax of this construct is:

```
<interaction
           channel="ncname"
           operation="ncname"
           time-to-complete="xsd:duration"
           initiateChoreography="true"|"false">

      <participate  relationship=”ncname”
                    fromRole="ncname"
                    toRole="ncname" />

      <exchange
          messageContentType="qname"
          viewpoint=“common”|”participant”?
          align=“true”|”false”?
          action=“request”|”receive”|”respond” >

          <use       name=“ncname”/>?
          <populate  name="ncname"/>?
      </exchange>
```

```
            <record
                viewpoint="common"|"participant"?
                align="true"|"false"?
                action="request"|"receive"|"respond" >

                <fromRole  name="ncname"
                           informationType="qname">
                   <source    name="ncname"/> |
                     <source  expression="XPath-expression"/>
                   <target    name="ncname"
                          predicate="xsd:boolean XPath-expression"? />
                </fromRole>?

                <toRole  name="ncname"
                         informationType="qname">
                   <source    name="ncname"/> |
                     <source  expression="XPath-expression"/>
                   <target  name="ncname"
                          predicate="xsd:boolean XPath-expression"? />
                </toRole>?
            </record>*
      </interaction>
```

The **interaction** construct allows a Role to interact with another Role by requesting an observable operation offered by that Role. The interaction materializes when a requester Role sends variable information and an accepter Role simultaneously receives the variable information through a shared Channel as described by the channel attribute.

This means an Interaction can be one of two types:

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

An Interaction also contains "references" to:

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

2207

2208 The attribute operation specifies a one-way or a request-response WSDL 1.2 operation.

2209 An interaction activity can be marked as a Choreography initiator when the
2210 initiateChoreography attribute is set to "true".

2211 Within the participate element, the relationship attribute specifies the Relationship this
2212 Choreography participates in and the fromRole and toRole attributes specify the
2213 requesting and the accepting Roles respectively.

2214 The time-to-complete attribute identifies the time an interaction MUST take to complete.

2215 Within an Interaction, a Role needs to have a common understanding of the state
2216 changes of one or more *State Variables* that are complimentary to one or more State
2217 Variables of its partner Role. Additionally within an Interaction, a Role needs to have a
2218 common understanding of the values of the Information Exchange Variables at the
2219 partner Role. The optional align element with the variable and with-variable attributes,
2220 specifies the shared variable information that MUST be atomically aligned and made
2221 available to the two interacting participants, as specified by the fromRole and toRole
2222 attributes and the relationship attribute within the participate element, when the interaction
2223 activity completes normally. In the case where the operation is of request-response type,
2224 then the attribute direction within the align element defines if the Channel will be aligned
2225 during the request or during the response.

2226

2227 **4.11.2.2.1 Interaction Roles**

2228 Interactions always have a "direction" in that there is a *From Role* that sends the original
2229 message and a *To Role* that receives the message. In the case of a request/response
2230 MEP, the "To Role" will also send a response message back to the "From Role".

2231

2232 **4.11.2.2.2 Interaction Message Content**

2233 *Message Content* identifies the type of information that is exchanged between the roles
2234 and the *Information Exchange Variables* used as follows:

2235 ?? *One Way From Message* is the variable that is the source for a One-Way Message
2236 at the *From Role*

2237 ?? *One Way To Message* is the variable that is the destination for a One-Way Message
2238 at the *To Role*

2239 ?? *Request From Message* is the variable that is the source for Request Message at the
2240 *From Role*

2241 ?? *Request To Message* is the variable that is the destination for Request Message at
2242 the *To Role*

2243 ?? *Response To Message* is the variable that is the source for Response Message at
2244 the *To Role*

2245 ?? *Response From Message* is the variable that is the destination for Response
2246 Message at the *From Role*

2247

2248 The type of information that is referenced will vary depending on the type of the
2249 Choreography.

2250

Choreography Type	Message Content
Abstract	In an Abstract Choreography, the message content that is exchanged is described by: ✍ A unique identifier, e.g. a URI, that identifies the message content and ✍ A semantic definition that explains the purpose of the message and outlines its content. No detail is provided of the actual message content, e.g. XSD definitions
Portable	In a Portable Choreography, the Abstract definition of Message Content is extended to include a WSDL Message Type or an XSD element type
Concrete	In a Concrete Choreography, Message Content is defined in the same way as for a Portable Choreography

2251
2252 #### 4.11.2.2.3 Interaction Channel Variables

2253 A Channel Variable contains information on where and how to send information to a
2254 specific instance of the To Role. This is because Concrete Channel information plus
2255 Correlation information about a Choreography contains sufficient information to identify
2256 how to send messages to a specific instance of a process.
2257 Additionally, Channel Variable information can be passed within Message Content. This
2258 allows the destination for messages in a choreography to be determined dynamically.
2259 For example, a Buyer could specify Channel information to be used for sending delivery
2260 information. The Buyer could then send the Channel information to the Seller who then
2261 forwards it to the Shipper. The Shipper could then send delivery information directly to
2262 the Buyer using the Channel Information originally supplied by the Buyer.
2263
2264 The content varies depending on the type of the choreography.
2265

Choreography Type	Channel
Abstract	In an Abstract Choreography, the channel is described by: ✍ A unique identifier, e.g. a URI that identifies the Channel within the Role ✍ A semantic definition, that describes the type of channel information that the Channel can accept
Portable	In a Portable Choreography, the abstract channel is extended by identifying its Channel Type, which defines what type of information the variable contains.

Choreography Type	Channel
Concrete	In a concrete choreography, the channel extends a portable channel by adding end point information for each interface such as complex Service References or simple URIs, digital certificates etc.

At run time, information about a channel variable is expanded further. This requires that the messages in the Choreography also contain Correlation information, for example by including:

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

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

4.11.2.2.4 Interaction Operations

An Operation specifies the particular part of an interface that is the target for a message.

The content varies depending on the type of choreography.

Choreography Type	Interaction
Abstract	In an abstract choreography, an operation is described by a unique name within the Interface within the Channel
Portable	In a portable choreography, an operation is described referencing a WSDL one-way or request-response Operation
Concrete	Same as portable.

4.11.2.3 Assign activity

The **assign** construct makes the variable value available to a participant using the use element, when the condition specified by the predicate attribute evaluates to true and the assign activity completes normally.

The syntax of this construct is:

```
<assign>
     <copy>
         <source   variable="ncname"/> |
            <source  expression="XPath-expression"/>
         <target   variable="ncname"
                    predicate="xsd:boolean XPath-expression"? />
```

```
2294            </copy>+
2295        </assign>
2296
2297
```

4.11.2.4 Noaction activity

The **noaction** construct allows the Choreography to perform an action that has no business effect on any participant.

The syntax of this construct is:

```
<noaction/>
```

4.11.2.5 Finalize activity

The **finalize** construct allows issuing a finalization command from a parent Choreography to the Finalization WorkUnit of an enclosed Choreography as identified by the choreography-name attribute.

The syntax of this construct is:

```
<finalize  choreographyName="qname" />
```

5. 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].

6. 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.

Using WS-CDL, two Web Service participants make progress by interacting. After they interaction, 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 WS- Reliability protocol, where agreement among participants can be reached even in the case of failures and loss of messages.

7. Relationship with the Transaction/Coordination framework

2343 # 8. Acknowledgments

2344

2344 # 9. References

2345 [1] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, Harvard
2346 University, March 1997

2347 [2] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax",
2348 RFC 2396, MIT/LCS, U.C. Irvine, Xerox Corporation, August 1998.

2349 [3] http://www.w3.org/TR/html401/interaction/forms.html#submit-format

2350 [4] http://www.w3.org/TR/html401/appendix/notes.html#ampersands-in-uris

2351 [5] http://www.w3.org/TR/html401/interaction/forms.html#h-17.13.4

2352 [6] Simple Object Access Protocol (SOAP) 1.1 "http://www.w3.org/TR/2000/NOTE-SOAP-
2353 20000508/"

2354 [7] Web Services Definition Language (WSDL) 1.2

2355 [8] Industry Initiative "Universal Description, Discovery and Integration"

2356 [9] W3C Recommendation "The XML Specification"

2357 [10] XML-Namespaces " Namespaces in XML, Tim Bray et al., eds., W3C, January 1999"

2358 http://www.w3.org/TR/REC-xml-names

2359 [11] W3C Working Draft "XML Schema Part 1: Structures". This is work in progress.

2360 [12] W3C Working Draft "XML Schema Part 2: Datatypes". This is work in progress.

2361 [13] W3C Recommendation "XML Path Language (XPath) Version 1.0"
2362

2363 [14] "Uniform Resource Identifiers (URI): Generic Syntax," RFC 2396, T. Berners-Lee, R.
2364 Fielding, L. Masinter, MIT/LCS, U.C. Irvine, Xerox Corporation, August 1998.
2365

2366 [15] WSCI: Web Services Choreography Interface 1.0, A.Arkin et.al
2367

2368 [16] XLANG: Web Services for Business Process Design
2369

2370 [17] WSFL: Web Service Flow Language 1.0
2371

2372 [18] BPEL: Business Process Execution Language 1.1
2373

2374 [19] BPML: Business Process Modeling Language 1.0
2375

2376 [20] XPDL: XML Processing Description Language 1.0
2377

2378 [21] WS-CAF: Web Services Context, Coordination and Transaction Framework 1.0
2379

2380 [22] Web Services Reliability 1.0
2381

2382 [23] The Java Language Specification
2383

2384 [24] Web Services Security
2385

2386 [25] J2EE: Java 2 Platform, Enterprise Edition, Sun Microsystems
2387

2387 # A. WS-CDL XSD Schemas

2388

2389

2389 # B. WS-CDL Supplied Functions
2390