A Critical Overview of the Web Services Choreography Description Language (WS-CDL)

Alistair Barros
Marlon Dumas
Phillipa Oaks

1 Introduction

There is an increasingly widespread acceptance of Service-Oriented Architectures (SOA) as a paradigm for integrating software applications within and across organizational boundaries. In this paradigm, independently developed and operated applications are exposed as (Web) services which are then interconnected using a stack of standards including SOAP, WSDL, UDDI, WS-Security, etc. While the technology for developing basic services and interconnecting them on a point-to-point basis has attained a certain level of maturity and adoption, there remain open challenges when it comes to managing service interactions that go beyond simple sequences of requests and responses or involve large numbers of participants.

Standardization is a key aspect of the Web services paradigm. Web services standardization initiatives such as SOAP and WSDL, as well as the family of WS-* specifications (e.g., WS-Policy, WS-Security, WS-Coordination) aim at ensuring interoperability between services developed using competing platforms. Figure 1 provides a quick (though partial) overview of the existing stack of Web services standards.

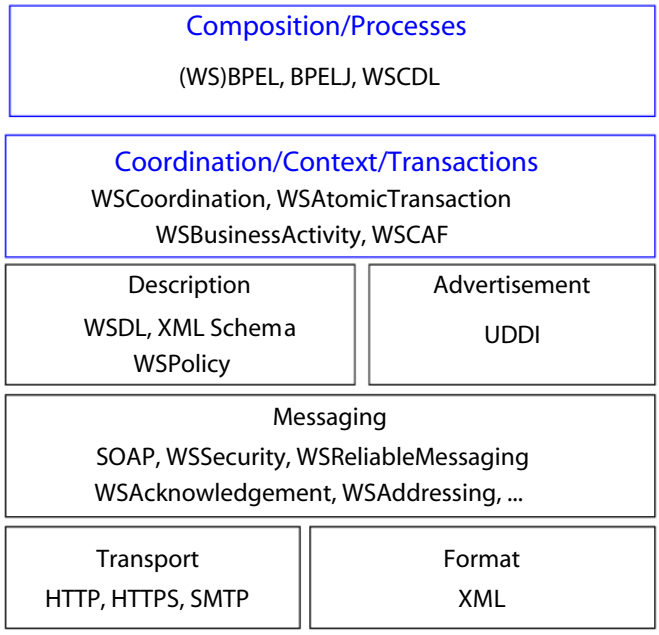

Figure 1. A view on the *Web Services Stack*

The standards in the category *composition/processes* deal with the interplay between services and business processes. A number of discontinued standardization proposals in this category have been put forward over the past years (e.g., WSFL, XLang, BPML, WSCL, and WSCI), leading to two ongoing standardization initiatives: the Business Process Execution Language for Web Services (BPEL4WS, also known as BPEL or WS-BPEL) [1] and the Web Services Choreography Description Language (WS-CDL) [5]. The significant attention raised by this category of standards reflects the fundamental links that exist between Business Process Management (BPM) and SOA. On the one hand, emerging BPM techniques rely on SOA as a paradigm for managing resources (especially software ones), describing process steps, or capturing the interactions between a process and its environment. On the other hand, a

service may serve as an entry point to an underlying business process, thereby inducing an inherent relation between the service model and the process model. Also, services may engage in interactions with other services in the context of collaborative business processes.

Standards for service composition cover three different, although overlapping, viewpoints:

- *Choreography* (also called *global model* in WSCI and *multiparty collaboration* in ebXML[1]): This viewpoint captures collaborative processes involving multiple services where the interactions between these services are seen from a global perspective.
- *Behavioral interface* (also called *abstract process* in BPEL and *collaboration protocol profile* in ebXML): This viewpoint captures the behavioral dependencies between the interactions in which a given service can engage.
- *Orchestration* (also called *executable process* in BPEL): This viewpoint deals with the description of the interactions in which a given service can engage with other services, as well as the internal steps between these interactions (e.g., data transformations).

This paper focuses on the choreography viewpoint. The following section provides more precise definitions of the above three viewpoints together with some examples. Section 3 provides an overview of an ongoing standardization proposal in the area of service choreography, namely WS-CDL. Section 4 provides a critique of WS-CDL. Finally, we outline some directions for further research and development in Section 5. High-level and detailed meta-models of WS-CDL are given in appendix.

2 Viewpoints in Service Composition

2.1 Choreography

A *choreography model* describes a collaboration between a collection of services in order to achieve a common goal. It captures the interactions in which the participating services engage to achieve this goal and the dependencies between these interactions, including control-flow dependencies (e.g., a given interaction must occur before another one), data-flow dependencies, message correlations, time constraints, transactional dependencies, etc. A choreography does not describe any internal action that occurs within a participating service that does not directly result in an externally visible effect, such as an internal computation or data transformation. A choreography captures interactions from a global perspective, meaning that all participating services are treated equally. In other words, a choreography encompasses all the interactions between the participating services that are relevant with respect to the choreography's goal.

An example of a choreography is shown in the form of an UML activity diagram in Figure 2. Three services are involved in this choreography: one representing a *customer,* another one a *supplier,* and a third one a *warehouse* . The elementary actions in the diagram represent business activities that result in messages being sent or received. For example, the action *order goods* undertaken by the customer results in a message being sent to the supplier (this is described as a textual note below the name of the action). Of course, every message sending action has a corresponding message receipt action but to avoid cluttering the diagram, only the sending or the receipt action (not both) is shown for each message exchange.

[1] http://www.ebxml.org

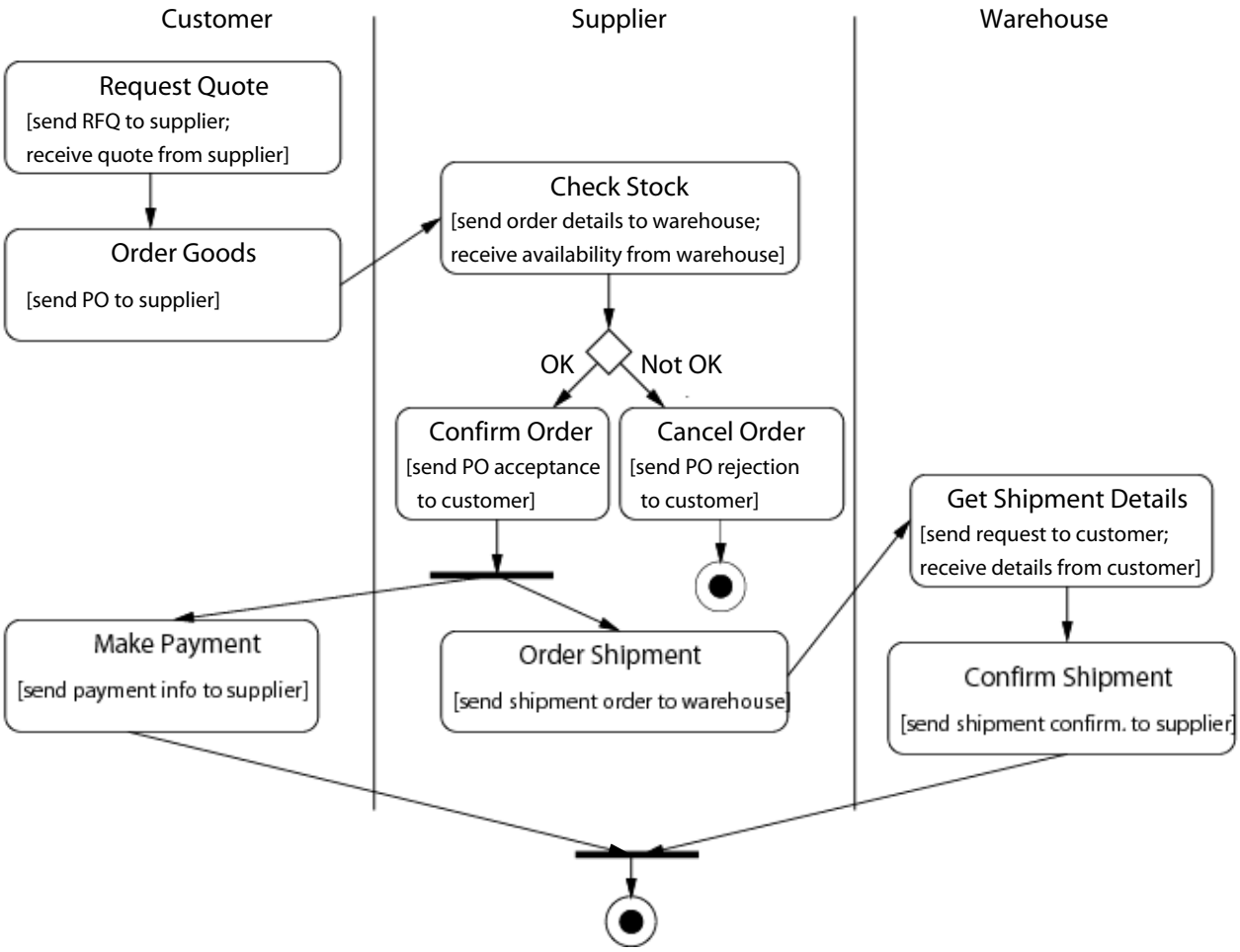

Figure 2. Choreography example

A choreography constitutes an agreement between a set of parties as to how a given collaboration should occur. This agreement may be established by common consultation or designed within a standardization committee. For example, RosettaNet's Partner Interface Processes (PIPs) can be seen as choreographies designed by standardization[2] .

2.2 Behavioral Interface

A *Behavioral interface model* captures the behavioral aspects of the interactions in which a particular service can engage to achieve a goal. It complements structural interface descriptions such as those supported by WSDL that capture the elementary interactions in which a service can engage, and the types of messages and the policies under which these messages are exchanged (e.g., with respect to security and reliability). A behavioral interface captures dependencies between interactions such as control-flow dependencies (e.g., that a given interaction must precede another one), data-flow dependencies, time constraints, message correlations, and transactional dependencies, etc.

Unlike a choreography, a behavioral interface focuses on the perspective of one single party. As a result, a behavioral interface does not capture *complete interactions* since interactions necessarily involve two parties. Instead, a behavioral interface captures interactions from the perspective of one of the participants and can therefore be seen as consisting of communication actions performed by that participant. Like choreographies, behavioral interfaces do not describe internal tasks such as internal data transformations.

[2]Note, however, that RosettaNet PIPs are very high-level descriptions of collaborative processes; they do not provide detailed descriptions of the involved interactions and are not targeted specifically towards SOA.

Figures 3 and 4 show two examples of behavioral interfaces in the form of UML activity diagrams where activities correspond to message sending and receipt. These behavioral interfaces jointly cover the behavior expected from the supplier's role in the choreography of Figure 2. The first behavioral interface (Figure 3) deals with the *quote request* portion of the choreography, while the second one (Figure 4) deals with the Purchase Order (PO) entry – both of them from the perspective of the supplier.

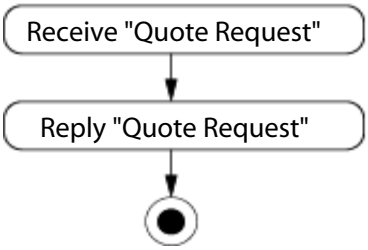

Figure 3. Quote request behavioral interface

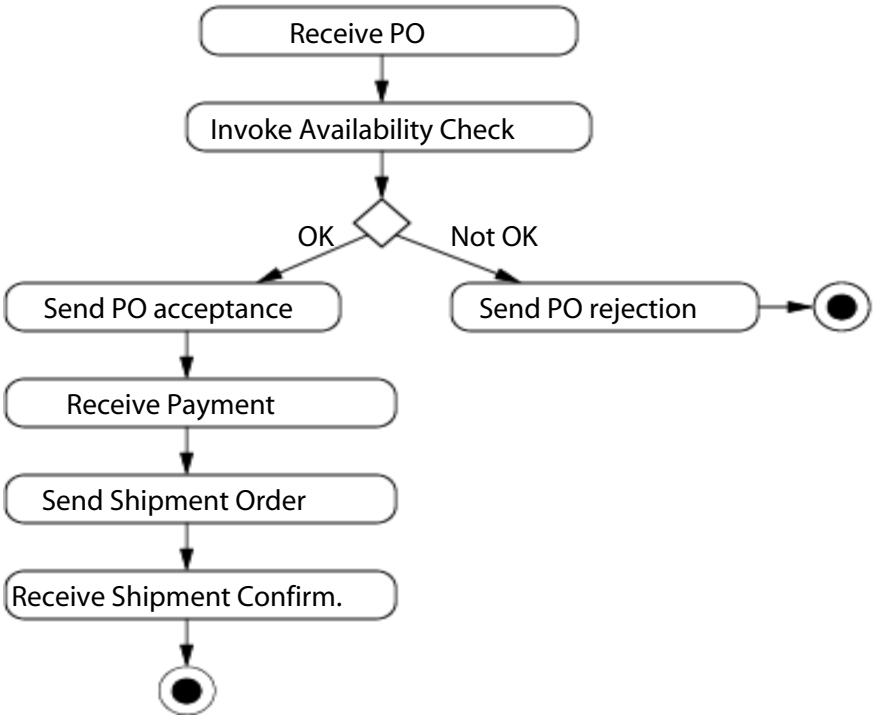

Figure 4. PO entry behavioral interface

The example illustrates the fact that in a given Business-to-Business (B2B) collaboration, a role in a choreography may be associated with multiple behavioral interfaces (and thus multiple WSDL interfaces). Moreover, given a choreography and a role within this choreography, an arbitrary number of behavioral interfaces may be defined that would *fit* (or *conform to*) the behavioral constraints imposed by the choreography on that particular role. Concretely, the behavioral interface described in Figure 5 could perfectly well be used instead of the one in Figure 4 in the context of the working choreography example. Indeed, this alternative behavioral interface will exhibit the same behavior as the one in Figure 4, provided that the message *PO cancellation* is never sent to a participant playing the role of supplier. In the running example, this is the case since the choreography of Figure 2 does not foresee this message being sent.

In Architecture Description Languages (ADLs), a distinction is often made between *provided* and *required* interfaces (also called *outgoing* and *incoming* interfaces in some nomenclatures). This distinction is based on the types of communication actions that an interface is allowed to use and the way in which these communication actions can be related.

A provided behavioral interface can only use communication actions in such a way that a receive action always occurs first, after which an optional choice of send actions may occur. The idea is that the

receive action corresponds to the *request* for a certain business function to be performed by the service, and the send actions correspond to the possible *responses* that may be the result of performing this business function. For example, in terms of the communication actions supported by BPEL, a provided behavioral interface must be composed of receive and reply tasks, where the replies are always coupled (i.e., correlated) to a receive.

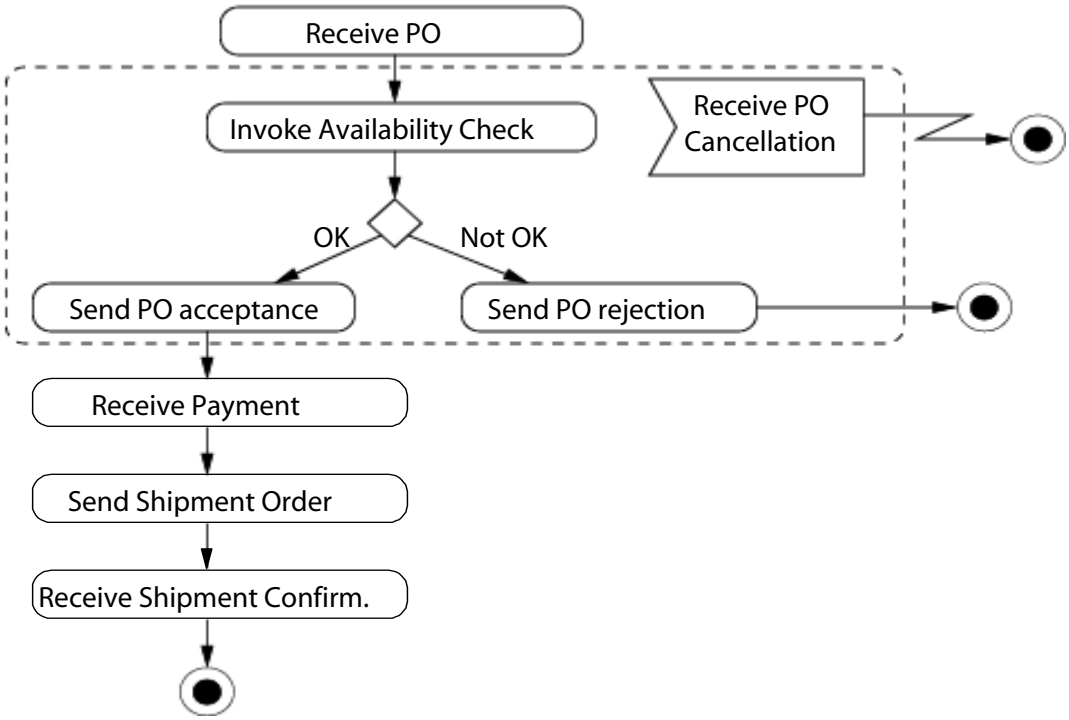

Figure 5. Alternative PO entry behavioral interface

On the other hand, a required behavioral interface can only use communication actions in such a way that a send action always occurs first, after which an optional choice of receive actions may occur. Essentially, the send action corresponds to the *request* for a certain business function to be performed by a counter-part, and the subsequent receive actions correspond to the possible *responses* that may result from performing this business function. In terms of the communication actions supported by BPEL, a required behavioral interface must be composed of asynchronous or synchronous invoke tasks.

At present, the distinction between required and provided interfaces does not seem to have been explicitly adopted by existing SOA standardization efforts. However, it is likely to play a role in one way or another in the context of service composition methodologies.

2.3 Orchestration

An *orchestration model* describes both the communication actions and the internal actions in which a service engages. Internal actions include data transformations and invocations to internal software modules (e.g., *legacy applications* that are not exposed as services). An orchestration may also contain communication actions or dependencies between communication actions that do not appear in any of the service's behavioral interface(s). This is because behavioral interfaces may be made available to external parties, and, thus, they should only show the information that actually needs to be visible to these parties. Orchestrations are also called *executable processes* since they are intended to be executed by an *orchestration engine*.

Figure 6 shows an example of an orchestration in the form of an UML activity diagram[3] . This orchestration adds an *internal action* (shown in dotted lines in the diagram) to the behavioral interface of Figure 4.

[3] http://www.uml.org

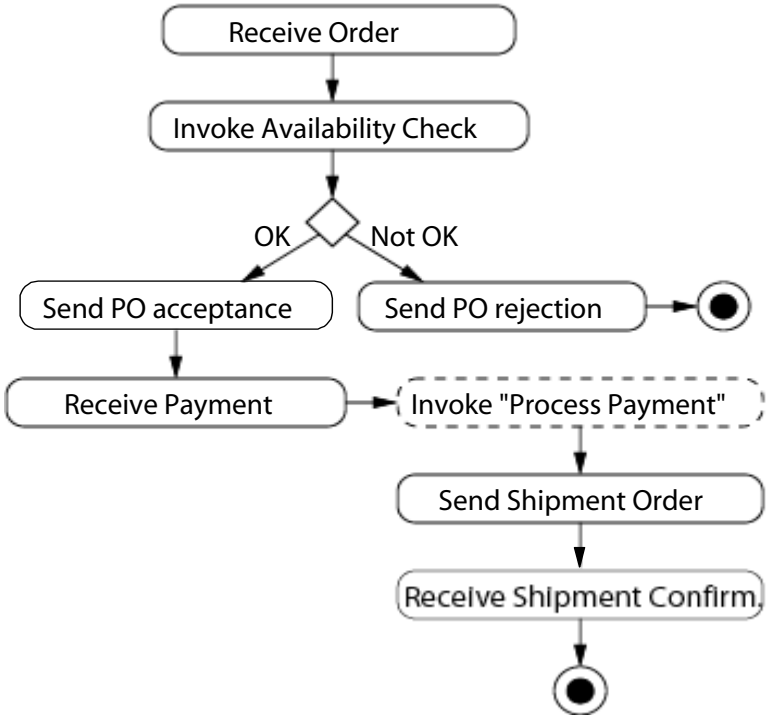

Figure 6:. An orchestration corresponding the behavioral interface of Figure 4.

2.4 Relations between viewpoints

The viewpoints presented above have some overlap. This overlap can be exploited within service composition methodologies to perform consistency checks between viewpoints or to generate code. For example, a choreography model can be used for the following purposes:

- To generate the behavioral interface that each participating service must provide in order to participate in a collaboration. As explained below, this *behavioral interface* can then be used during the development of the service in question. For example, given the choreography of Figure 2, it would be possible to derive the behavioral interface that a supplier service would be required to provide in order to participate in this choreography (and same thing for the customer or the warehouse).
- To check (at design time) whether the behavioral interface of an existing service conforms to a choreography and, thus, whether the service in question would be able to play a given role in that choreography.

Similarly, a behavioral interface can be used as a starting point to generate an *orchestration* skeleton that can then be filled up with details, regarding internal tasks, and refined into a full orchestration. On the other hand, an existing orchestration could be checked for consistency against an existing behavioral interface. In this way, it would be possible, for example, to detect situations where a given orchestration does not send messages in the order in which these are expected by other services.

A formal definition of the service modeling viewpoints presented above and the relations between them can be found in [3].

3 Overview of WS-CDL

This section provides an overview of the elements and structure of WS-CDL, as described in the editor's draft of the WS-CDL v1.0 specification, dated 22 September 2004. A high level conceptual view of the structure of WS-CDL descriptions is shown in Figure 7.

WS-CDL choreography descriptions provide a global or unbiased view of the interactions between two or more parties, as defined in the previous section. Web services will be developed by many different providers, but the ways of using them individually and in concert with other services cannot be described

by the Web Services Description Language (WSDL). The W3C Choreography working group[4] was chartered to address the need to provide a common and shareable means of describing how web services can be used individually and with one another to achieve goals.

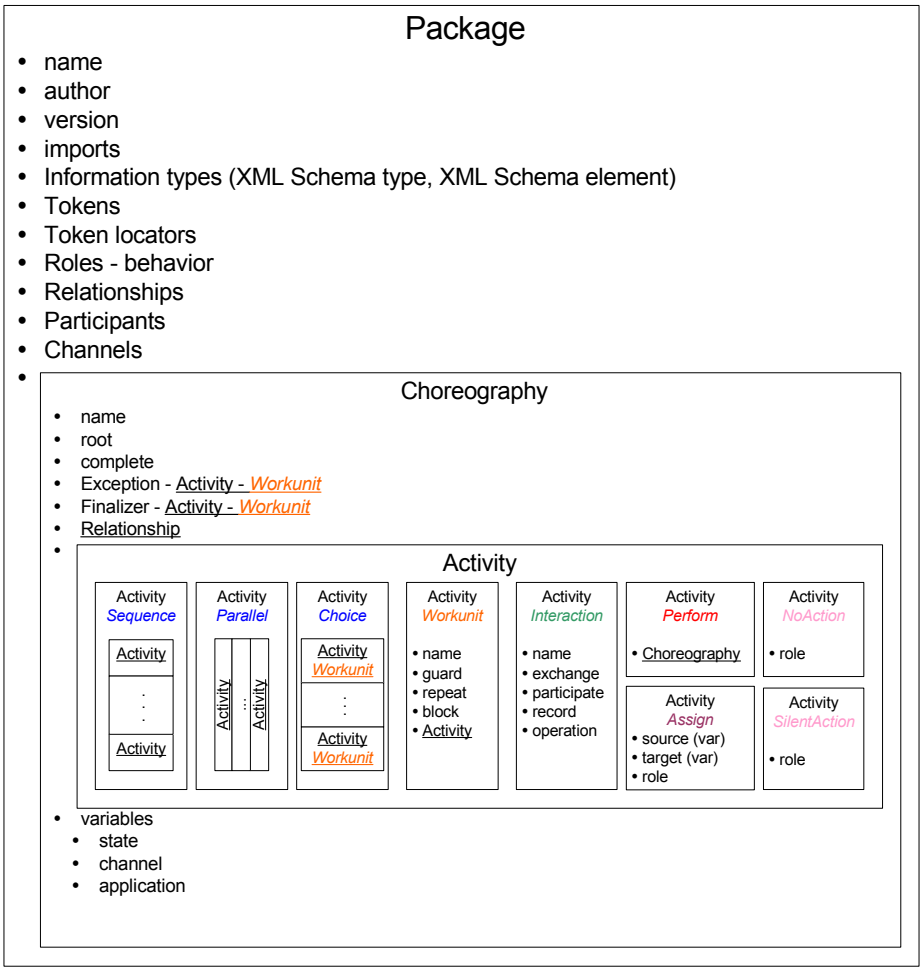

Figure 7. WS-CDL Package

The purpose of the Web Services Choreography Description Language (WS-CDL), as defined in the working group charter[5] and the requirements document[6], is to define multi-party contracts, which describe the externally observable behavior of web services and their clients (usually other web services), by describing the message exchanges between them. It is intended that WS-CDL choreography descriptions be used to generate code skeletons for Web services or BPEL abstract processes; this would be akin to generating the models in Figures 3 and 4 from the model shown in Figure 2.

A WS-CDL choreography description is contained in a package and is essentially a container for a collection of activities that may be performed by one or more of the participants. There are three types of activity in WS-CDL (see Figure 8), namely *control-flow* activities, *WorkUnit* activities and *basic activities* .

There are three (types of) activities in the first category, namely *Sequence*, *Parallel*, and *Choice*. These activities are block-structured in the sense that they enclose a number of sub-activities. A Sequence activity describes one or more activities that are executed in sequential order. A Parallel activity describes one or more activities that can be executed in any order or at the same time.

A Choice activity describes the execution of one activity chosen among a set of alternative or competing activities. Although the WS-CDL specifications do not explicitly state this, the Choice activity captures two types of choices:

[4] http://www.w3.org/2002/ws/chor/
[5] http://www.w3.org/2005/01/wscwg-charter.html
[6] http://www.w3.org/TR/2004/WD-ws-chor-reqs-20040311/

- Data-driven choice (or *immediate* choice): The choice between the activities is based upon a Boolean condition involving data variables that are evaluated immediately when the Choice activity is reached.
- Event-driven choice (or *deferred* choice): The choice depends on the occurrence of one among a set of competing events. Such events may be the occurrence of an interaction or the occurrence of an action elsewhere in the choreography that results in certain variables being populated, thus allowing certain conditions to be evaluated. The choice is called deferred because when the Choice activity is reached, the execution holds until an event occurs.

Whether a given Choice activity corresponds to the first type of choice or the second one depends on the nature of the activities enclosed by the choice. If all the enclosed activities have guard conditions and their evaluation is *non-blocking* (as explained below), then the choice is data-driven; otherwise it may correspond to a purely event-driven choice or a combination of data-driven and event-driven choices. In this respect, WS-CDL differs from BPEL, which treats these two types of choices as separate constructs (namely *switch* and *pick*).

The second type of activity in WS-CDL is called *WorkUnit*. A WorkUnit activity describes the conditional and, possibly, repeated execution of an activity. Syntactically, a WorkUnit activity has several parts, including a reference to the enclosed activity, a guard, a *block* condition, and a repetition condition (modeled as Boolean expressions). The guard and repetition conditions are used to determine whether or not the enclosed activity is executed one or more times. Specifically, the execution of the WorkUnit activity will start with the evaluation of the guard condition. If the condition evaluates to true, the enclosed activity is executed; otherwise it is *skipped*. If the activity is executed, and is once completed, the repetition condition is evaluated, and, depending on the outcome, the activity may be executed again or not, thereby providing a means to capture structured loops. The *block* condition is used to determine whether the guard and repeat conditions should wait for variables to become available before being evaluated. In typical usage scenarios, WorkUnits are used in conjunction with the Choice activity, and, in these cases, the choice may end up being immediate or deferred depending on whether the *block* conditions of the WorkUnits enclosed in a Choice evaluate to true or false.

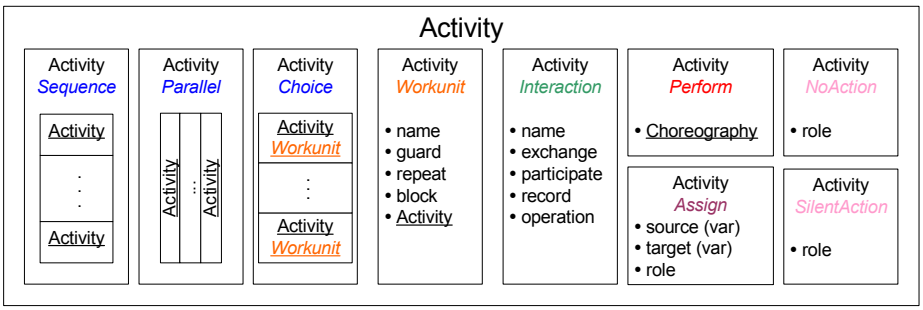

Figure 8. WS-CDL Activities

It can be seen that the Sequence, Parallel, Choice, and WorkUnit activities of WS-CDL allow one to capture the basic control-flow constructs found in typical imperative programming languages. It can also be seen that these activities correspond to the Sequence, Flow, While, Switch, and Pick activities in BPEL, with the exception noted above that the mapping from Choice and WorkUnit activities, on the one hand, to Switch and Pick activities, on the other, may be non-trivial due to the need to take into account WS-CDL's *block* conditions.

The third type of WS-CDL activities, *basic activities*, includes *Interaction*, *NoAction*, *SilentAction*, *Assign*, and *Perform*. The activities NoAction and SilentAction describe points in a choreography where one (named) *role* performs no action or performs an action *behind the scenes* that does not affect the rest of the choreography. The Assign activity is used to transfer the value of one variable to another variable within one role.

The Perform activity is used to *call* another choreography to be performed within the context of the executing choreography. The *called* choreography may be defined within the same package as the *caller*, or it may be from a completely separate package that has been imported. The perform activity has a binding mechanism to allow the callers' variables to be bound to free variables in the called choreography.

The most important element of WS-CDL is arguably the Interaction activity (Figure 9 and Figure 15 in Appendix A). An Interaction describes an exchange of information between parties with a focus on the receiver of the information. Interactions have one of three purposes; to make a *request*, or to make a response (*respond*), or to make a request that requires a response (*request-respond*).

An interaction activity description has three main parts corresponding to (i) the *participants* involved; (ii) the *information* being exchanged; and (iii) the *channel* for exchanging the information. In Figure 9, an example of two interaction activities (corresponding to the *Request Quote* activity in the UML diagram of Figure 2) is given in WS-CDL's XML syntax[7]. It is worth noting the asymmetric nature of interaction activity descriptions in WS-CDL, which are biased towards the receiver rather than the sender. In particular, WS-CDL interaction activity descriptions refer to the operation performed when information is received, but not the action(s) (or operations) leading to the sending of information.

The *from* (sender) and *to* (receiver) roles, and the relationship between them for this interaction, are made explicit in the *participate* part of the Interaction description. Although the relationship is associated, via its roles, to one or more behaviors, the exact operation the receiver is expected to perform with the information it receives, is defined in the overall interaction definition rather than in the subordinate *participate* definition.

```
<interaction name=              "QuoteRequest"
             initiate=          "true"
             align=             "false"
             operation=         "ReceiveRFQ"
             channelVariable=   "requestQuoteChannel">
    <participate toRole=        "supplier"
                 fromRole=      "customer"
                 relationship=  "GetQuote"/>
    <exchange name=             "e1"
              action=           "request"
              informationType=  "quoteRequest" >
         <send variable=        "rfq"/>
         <receive recordReference="record1"
                  variable=     "rfq"/>
   </exchange>
   <record name=                "record1"
           when=                "after" >
      <source variable=         "rfq"/>
      <target variable=         "quoteForm"/>
   </record>
</interaction>
<interaction name=              "QuoteResponse"
             initiate=          "false"
             align=             "false"
             operation=         "ReceiveQuote"
             channelVariable=   "receiveQuoteChannel">
   <participate toRole=         "customer"
                fromRole=       "supplier"
                relationship=   "GetQuote"/>
   <exchange name=              "e2"
             action=            "respond"
             informationType=   "quoteDocument">
        <send variable=         "quote"/>
        <receive variable=      "quote"/>
   </exchange>
</interaction>
```

Figure 9. WS-CDL interaction example

[7] Note that the definition of a graphical notation to describe choreographies is not in the scope of the WS-CDL charter.

The information sent or received during an interaction is described by a named v*ariable* and an optional r*ecordReference* element in the e*xchange* description. The *record reference* identifies the variables sent and received and optionally describes how the variable is populated on the sender's side *before* the interaction, and/or how the receiver extracts data items from it *after* the interaction.

Variables in WS-CDL are used to represent three different types of information: application-dependent information (e.g. product code), state information (e.g. order received), and channel information. Variables contain values and have an i*nformationType*. Variables are accessed using WS-CDL XPath 1.0 extension functions such as *getVariable(name, path, role)* .

Channel variables are of a channel type. A channel type description indicates the role the receiver (of the request or response message) is playing and which behavior the receiver performs on this channel. The behaviors of roles are (optionally) described by WSDL interfaces.

```
<channelType name=    "requestQuoteChannel"
             action=  "request"
             usage=   "unlimited">
   <role type=        "supplier"
         behavior=    "ReceiveRFQ"/>
   <reference>
      <token name=    "supplierRef"/>
   </reference>
   <identity>
      <token name=    "quoteId"/>
   </identity>
   <passing action=   "respond"
            new=      "true"
          channel=    "receiveQuoteChannel"/>
</channelType>

<channelType name=    "receiveQuoteChannel"
             action=  "respond"
             usage=   "once">
   <role type=        "customer"
         behavior=    "ReceiveQuote"/>
   <reference>
      <token name=    "customerRef"/>
   </reference>
   <identity>
      <token name=    "quoteId"/>
   </identity>
</channelType>
```

Figure 10. WS-CDL channel type example

In this respect, the notion of channel, as used in WS-CDL, is very similar to that found in system description languages such as SDL.[8] Indeed, in SDL-2000 channels are associated with interfaces, with one-directional channels being associated to one interface and two-directional channels being associated to two interfaces (one per direction). However, unlike SDL-2000, WS-CDL does not directly distinguish between one-directional and two-directional channels; instead, a WS-CDL channel can be associated to one of three actions: request, respond, or request-respond. In principle, a relationship would exist between these three types of actions and WSDL Message Exchange Patterns (MEPs), but this relationship is not explicitly addressed in the WS-CDL specification.

[8] http://www.sdl-forum.org/SDL

In any case, channels are the link between WS-CDL choreographies and operations described in WSDL interfaces. Every behavior to be exhibited in a choreography has a corresponding channel type description, and this relation is one-to-one since each channel type variable can enact exactly one behavior

Channels can also be used to pass channel type variables in addition to passing application and state variables. Accordingly, a channel type description may show the names of the other channel types it can exchange. The code example in Figure 10 shows the requestQuoteChannel *passing* the receiveQuoteChannel.

The *reference* token in the channel describes how to contact the receiver, and the *identity* tokens are used to correlate the messages exchanged through the channel with the choreography. *Tokens* are associated with an information type, and *token locators* provide the means to describe how tokens are located using XPath expressions.

The *informationTypes* of tokens and variables are described at the package level, making them available to all enclosed choreographies and activities. Information types are either XML Schema elements or WSDL 2.0 schema elements, or XML Schema types or WSDL 1.1 message types. Application, state, and channel variables are all specified with an information type. An example is shown in Figure 11.

The variables that are used within the choreography's activities are described within the choreography definition, thus scoping variables at the choreography level. These top-level variables can be made available or shared with *called* choreographies using the variable binding mechanism in the perform activity.

```
<informationType name=   "address"
                 type=   "xsd:anyURI"/>
<informationType name=   "correlationId"
                 type=   "xsd:string"/>
<informationType name=   "quoteRequest"
                 type=   "xsd:anyURI"/>
<informationType name=   "quoteDocument"
                 type=   "xsd:anyURI"/>
<token name=             "customerRef"
      informationType=   "address"/>
<token name=             "supplierRef"
      informationType=   "address"/>
<token name=             "quoteId"
      informationType=   "correlationId"/>
<tokenLocator tokenName= "quoteId"
              query=     "quoteRequest/docId"
      informationType=   "quoteRequest">
</tokenLocator>
```

Figure 11. WS-CDL information and token types example

Variables can belong to one named role or to all roles in a choreography. The variable *quoteForm*, shown in Figure 12, belongs only to the suppler, whereas the other two variables *rfq* and *quote* are available at all the roles participating in the choreography. The transfer of variable values between roles must be done using messages in an interaction activity[9] .

[9] A recent proposal (http://lists.w3.org/Archives/Public/public-ws-chor/2004Aug/0006.html) suggests that the *ParticipantType* be used as grouping mechanism to enable roles being performed by the same service provider to share variable values with one another without the overhead of engaging in an interaction activity to exchange variable values via messages.


```
<choreography name=              "QuoteAndOrder"
               isolation=        "dirty-write"
               root=             "true">
  <relationship type=            "GetQuote"/>
  <relationship type=            "PlaceOrder"/>
  <variableDefinitions>
     <variable name=             "rfq"
               mutable=          "false"
               free=             "false"
               informationType=  "quoteRequest"
               silentAction=     "false"/>
     <variable name=             "quoteForm"
               mutable=          "true"
               free=             "false"
               informationType=  "quoteDocument"
               silentAction=     "true"
               role=             "supplier"/>
     <variable name=             "quote"
               mutable=          "true"
               free=             "false"
               informationType=  "quoteDocument"
               silentAction=     "false"/>
  </variableDefinitions>
  <sequence> ... activities ... </sequence>
  <exception name=               "NoAction">
     <workunit block=            "false"
               repeat=           "false"
               name=             "NoAction"
               guard=            "true">
        <noAction role=          "customer"/>
     </workunit>
  </exception>
  <finalizer name=               "None">
     <workunit block=            "false"
               repeat=           "false"
               name=             "None"
               guard=            "true">
        <noAction role=          "customer"/>
     </workunit>
  </finalizer>
</choreography>
```

Figure 12. WS-CDL choreography example

A *Choreography* description is a container for a top-level activity and optional exception and finalizer work units (Figure 12 and Figure 14, Appendix A). The exception work unit may be activated if exceptions occur. The finalizer work unit may be activated when the choreography has completed successfully, but a failure in another choreography means that this completed choreography must be *rolled back* .

One or more WS-CDL choreography descriptions are contained in a Package (Figure 13 and Figure 16 in Appendix B). The package is used to collect the information that is common to all the choreographies contained in the package.

A Package describes the *RoleTypes* that exhibit *Behaviors* and *RelationshipTypes* that occur between two roleTypes. Relationship types specify a subset of the behaviors of the roles that are to be exhibited in that relationship. Finally, the package may contain a description of *ParticipantTypes* which represent logical groupings of roles.


```
<package name=  "Buying"
         xmlns= "http://www.w3.org/2004/10/ws-chor/cdl/"
  ...
  <roleType name=         "supplier">
    <behavior interface= "Supplier.wsdl"
              name=      "ReceiveRFQ"/>
    <behavior interface= "Supplier.wsdl"
              name=      "ReceivePO"/>
  </roleType>
  <roleType name=         "customer">
    <behavior interface= "Customer.wsdl"
              name=      "ReceiveQuote"/>
    <behavior interface= "Customer.wsdl"
              name=      "ReceivePaymentRequest"/>
  </roleType>
  <relationshipType name="GetQuote">
    <role type=           "supplier"
          behavior=       "ReceiveRFQ"/>
    <role type=           "customer"
          behavior=       "ReceiveQuote"/>
  </relationshipType>
  <relationshipType name="PlaceOrder">
    <role type=           "supplier"
          behavior=       "ReceivePO"/>
    <role type=           "customer"
          behavior=       "ReceivePaymentRequest"/>
  </relationshipType>
  <participantType name= "CustomerA">
    <role type=           "customer"/>
  </participantType>
  <participantType name= "SupplierB">
    <role type=           "supplier"/>
    <role type=           "customer"/>
  </participantType>
  ...
</package>
```

Figure 13. WS-CDL package example

To clarify, a role exhibits at least one behavior. A relationship defines exactly two roles and the subset of the behaviors of each role that are used in the relationship. A participant is an entity that plays a particular set of roles with a defined subset of the behaviors of those roles. ParticipantTypes are defined at the package level of WS-CDL but are not used elsewhere in the specification of choreographies or activities. At present, the WS-CDL specification does not address the issue of bridging the static design of roles, behaviors, relationships, and participants with the runtime binding to service providers and concrete services.

Although all elements of WS-CDL are uniquely identified by name, many elements (such as choreographies, activities, roles, relationships, and channels) will also require unique identities for their runtime instances. For example, a choreography with an *interaction* activity for requesting quotes that is *performed* inside a *parallel* activity with several different quote providers will need a mechanism to identify which instance of the choreography and interaction activity is related to each response received.

4 Issues and discussion

There are several places where the WS-CDL specification is not yet fully developed and a number of known issues remain open. Some of these issues are the subject of ongoing debate and should be *resolved* in one way or another in the near future. However, some issues of a more fundamental **or practical** nature will be more difficult to address and are likely to require a significant review of the

language's underlying meta-model and implied techniques. In this section, we discuss some of these major issues.

One of the requirements of WS-CDL[10] is to provide a means for tools to validate conformance to choreography descriptions to ensure interoperability between web services. To enable design time or static validation and verification of choreographies to ensure correctness properties such as livelock, deadlock, and leak freedom, or to ensure that the runtime behavior of participants conforms to the choreography plan, WS-CDL must be based on or related to a formal language that provides these validation capabilities. Although WS-CDL appears to borrow terminology from pi-Calculus [6], the link to this or any other formalism is not clearly established.

Positioned over the web services composition layer of the Web Services stack, WS-CDL is required to interoperate with a number of web service standards, notably WSDL and WSDL-MEPs, for static service binding – WS-BPEL for endpoint orchestrations and WS-Reliable-Messaging for lower level quality of messaging. Yet the mapping remains open, and conceptual sufficiency in aligning WS-CDL with these standards is arguably limited.

The mapping between WS-CDL and WSDL, mandated by the Web Service Choreography group's requirements, is largely open. As it stands, interacting senders and receivers in WS-CDL, captured through a Role, define at least one behavior, and a behavior may optionally refer to a WSDL document. WS-CDL Interactions take place in the context of a channel variable, representing an instance of a Channel Type. The channel can have one of three actions (request, respond, and request-respond). How these map to the eight WSDL 2.0 Message Exchange Patterns (MEPs) has yet to be precisely determined.

On the other hand, the existing association between WS-CDL and WSDL is arguably too restrictive. A choreography *wired* to specific WSDL interfaces (either indirectly through references to operations or more directly through an association between roles and their behaviors specified by reference to WSDL interfaces) cannot utilize functionally equivalent services with different WSDL interfaces. In other words, the choreography is statically bound to specific operation names and types, which may hinder the reusability of choreography descriptions. Cast more generally, choreography descriptions which abstractly describe behavior at a higher level, in terms of capability, would allow runtime selection of participants able to fulfil that capability, rather than restricting participation in the choreography to participants based on their implementation of a specific WSDL interface or WSDL operations.

A more subtle dependency is semantic consistency of a global choreography and local process orchestrations. As described in Section 2, a choreography definition introduces message ordering constraints over the interface views of local process orchestration definitions. These need to be supported at the orchestration level in which they are mapped. The expressive power of orchestration semantics, at the same time, should be not be limited by the choreography layer.

Close inspection of WS-CDL's expressive power suggests that it was developed with basic assumptions of process orchestration expressiveness and, therefore, with a basic level of messaging supported by this functionality. This point is best borne out by examining the extent and relationship of interactions supported in WS-CDL. Interactions occur between a pair of roles; in other words, only binary interactions are supported. Missing in WS-CDL is the explicit support for multi-party interactions and more complicated messaging constraints which these bring.

Some key requirements to consider are those which emerge in multi-party scenarios – for example, the multiple instances of interactions which can arise for the same interaction types at the same time; the processing of a purchase order, for instance, can involve several competing suppliers (known only at runtime due to the specific content of the purchase). Responses might be time-critical, and all suppliers might be required to receive the request and respond within a specified duration. The preparation of requests, the sending of requests, and the receipt of responses might need to be done in parallel. This may place a constraint over the number of suppliers that are required to successfully receive the request in order for the overall issue of the request to go-ahead. Moreover, when the number of suppliers is large, assumptions about the number of responses need to be relaxed. A minimum number might be required before further steps in the workflow are taken, while any remaining responses might be ignored or accepted.

Such a scenario, which in our view is not unusual for real-scale B2B applications involving tens to hundreds of parties, requires complicated orchestration support. These include multiple instances of

[10] http://www.w3.org/TR/2004/WD-ws-chor-reqs-20040311/, C-CR-5001

interactions, atomicity constraints on interactions, and partial synchronization of responses. In fact, one such scenario was discussed in the collection of use cases during the Web Services Choreography group's requirements gathering.[11] As it stands, it is unclear how WS-CDL can conveniently support these sorts of multi-party interactions without serializing the interaction and/or using low-level bookkeeping mechanisms based on arrays and counters. Workflow languages, capable of supporting multi-party instances, would be constricted by the single instance, binary interactions supported in WS-CDL.

In terms of WS-BPEL specifically, it remains open how some of WS-CDL's WorkUnit functionality can be mapped to WS-BPEL. Here we refer to the blocked wait for an action that occurs when the *block* condition associated to a WorkUnit evaluates to true as discussed in Section 3. In this case, an activity is allowed to proceed once an interaction or variable assignment action, which may occur in a completely different part of the choreography, supplies the required data. It is not clear how this would be mapped in terms of WS-BPEL's Pick and Switch constructs or what the complexity of the mapping would be. Also, as shown in Section 2, the relationships between choreography and behavioral interface (called *abstract process* in WS-BPEL) may be non-trivial, and there are currently no precise notions of conformance between WS-CDL choreographies and WS-BPEL abstract processes. Understanding these relations is crucial if these two specifications are to be used together in practice. It is worth noting that the definition of such relationships, as well as the mapping from WS-CDL to BPEL, would be much simpler if WS-CDL had a similar set of control-flow constructs as BPEL (i.e., Sequence, Flow/Parallel, While, Switch, Pick, and possibly also control-links). Ultimately, the fundamental difference between the concept of choreography on the one hand, and the concept of behavioral interface (i.e., BPEL abstract process) on the other, is that a choreography focuses on interactions seen from a global viewpoint, while behavioral interfaces focus on communication actions seen from the viewpoint of one of the participants. This difference is orthogonal to control flow, and, arguably, the same control flow constructs could be used for describing choreographies and interface behaviors. In other words, WS-CDL and BPEL (or at least the subset of BPEL for describing abstract processes) could share the same set of control flow constructs.

In terms of messaging quality of service, WS-CDL relies on WS-Reliable-Messaging principally. The extent of quality of service messaging on which WS-CDL depends is not fully established, and the mapping for reliable messaging at the very least remains open. In general, no *a priori* configurability of WS-CDL specifications for different quality of messaging service is in place. This in our view limits the layering and exploitation of choreography for lower level services from current and oncoming messaging standards.

Finally, it is worth noting that WS-CDL is an XML-based language standard. The development of a graphical language for capturing choreographies is not within the current choreography charter of W3C. Any exploitation of WS-CDL should be based on a graphical language, which supports user convenience in capturing specifications, model verification, and validation, as well as configuration for specific deployments utilizing different aspects messaging. The generation of WS-CDL fabricated with the configuration settings to utilize the relevant lower level messaging services is of paramount importance through such a conceptual provision. Ultimately, it is worth remembering that choreography models (unlike orchestration models) are inherently design-level artifacts and are not intended to be directly executed.

5 Conclusion and plan for future work

After placing WS-CDL within the broader context of SOA, this technical briefing has provided a relatively detailed overview of WS-CDL and has discussed some major issues with the current version of its specification. These issues primarily stem from three factors:

- *Lack of separation between meta-model and syntax*. The WS-CDL specification tries to simultaneously define a meta-model for service choreography and an XML syntax. As a result, there is no strict separation between semantic and syntactic aspects. Our view is that a service choreography meta-model should be developed independent of a particular interchange format. A cleanly defined meta-model would set the ground not only for the definition of an interchange format, but also for the definition of corresponding modeling notation(s). Ultimately, it is important to remember that service choreographies are more a design than an implementation artifact, and, therefore, a modeling notation for service choreographies may end up being more useful than an XML syntax (although the latter would still be useful for interchange purposes).

[11] http://lists.w3.org/Archives/Public/public-ws-chor/2003Aug/att-0016/mpi-use-case-ab.html

- *Lack of direct support for certain categories of use cases*. Although a set of use cases was identified by the Web Services Choreography Working Group prior to the release of the first WS-CDL specification, it appears that WS-CDL was not designed to provide direct support for (all of) these use cases. In particular, while many service choreography use cases involve one-to-many or multi-transmission interactions, the language does not provide direct support for these scenarios.
- *Lack of comprehensive formal grounding*. While WS-CDL borrows some terminology from pi-calculus, there is no comprehensive mapping from WS-CDL to pi-calculus or any other formalism. Even if a formalization of WS-CDL was undertaken in the future, it would be an *a posteriori* exercise rather than an *a priori* effort to ensure the coherence and consistency of the language.

Ultimately, it may be that the WS-CDL standardization effort came too early in the evolution of SOAs. Indeed, WS-CDL has attempted at the same time to be groundbreaking and to create a consensus. In this respect, it is insightful to compare the development of WS-CDL with that of BPEL. BPEL stemmed from two sources, WSFL and XLang, which derived themselves from languages supported by existing tools (namely MQSeries Workflow and BizTalk). Furthermore, together with the first draft of BPEL, a prototype implementation was released. In contrast, WS-CDL has been developed without any prior implementation and does not derive (directly) from any language supported by an implementation.

Whether or not WS-CDL becomes a de jure standard and is adopted by a wide user base, its development would have been instrumental in promoting and advancing the notion of service choreography as a basis for service-oriented development. Still, many issues remain to be resolved before the emergence and adoption of SOA infrastructures based on the notion of service choreography. To advance this vision, we propose a research agenda structured around three major tasks:

- *Identify and document a library of service choreography patterns.* Generally speaking, patterns document known solutions to recurrent problems that occur in a given software development context. A pattern captures the essence of a problem, provides examples, and proposes solutions. The value of patterns lies in their independence from specific languages or techniques and the fact that they capture common situations, abstracting away from specific scenarios or cases. In particular, a library of patterns for choreography modeling/definition would provide a foundation to analyze and improve existing languages and techniques for choreography modeling, and/or to design new ones.
- *Define a service choreography meta-model.* Based on the identified service choreography patterns, and an analysis of existing approaches to service choreography definition in terms of these patterns, it would be possible to identify a set of interrelated concepts directly relevant to service choreographies. This would provide a *kernel* service choreography meta-model that could then be enriched with concepts found in existing service choreography definition languages such as WS-CDL or ebXML BPSS [2]. Importantly, the meta-model should be formalized, for example by defining a type system or a mapping into a well-established formalism. This formalism could be a variant or an extension of Petri nets or process algebra (e.g., Coloured Petri nets or pi-calculus).
- *Define concrete syntaxes for service choreography modeling/interchange.* Once a service choreography meta-model has been defined, a visual notation and/or an interchange format can be specified. Effectively, the meta-model would serve as an *abstract syntax* for choreography definition while the visual notation and the interchange format would be seen as concrete syntaxes. The visual notation could be based upon existing languages rather than developed from scratch. Visual process modeling notations such as BPMN[12] or UML[13] activity and sequence diagrams could be used as the basis for defining a visual notation for choreography modeling. The interchange format, on the other hand, could be defined in terms of XML schema. Importantly, the elements in these concrete syntaxes would map directly to the concepts of the service choreography meta-model.

The choreography patterns, meta-model, visual notation, and interchange format would together provide the basis for a service choreography modeling infrastructure. From there, model transformations targeting the meta-models of *executable* languages such as BPEL, WSDL, and WS-Coordination could be developed, laying the ground for model-driven service-oriented development environments.

[12] http://www.bpmn.org
[13] http://www.uml.org

Disclaimer

The views and opinions expressed in these documents are those of the authors only and do not necessarily reflect those of SAP AG or Queensland University of Technology.

Acknowledgments

The work of the second author has been funded by a Smart State Fellowship co-sponsored by the Queensland Government and SAP Australia Pty Ltd.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services, version 1.1*, May 2003. Available at: http://www-106.ibm.com/developerworks/webservices/library/ws-bpel

[2] J. Clark, C. Casanave, K. Kanaskie, B. Harvey, J. Clark, N. Smith, J. Yunker, K. Riemer, (Eds) *ebXML Business Process Specification Schema Version 1.01*, UN/CEFACT and OASIS Specification, May 2001. Available at: http://www.ebxml.org/specs/ebBPSS.pdf.

[3] R. Dijkman and M. Dumas. Service-oriented Design: A Multi-viewpoint Approach. *International Journal of Cooperative Information Systems* 13(4), December 2004. Earlier version available as a technical report at: http://www.ub.utwente.nl/webdocs/ctit/1/000000ed.pdf

[4] T. Halpin. *Information Modeling and Relational Databases*. Morgan Kaufmann, 2001.

[5] N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon. *Web Services Choreography Description Language Version 1.0*, W3C Working Draft, October 2004. Available at: http://www.w3.org/TR/ws-cdl-10.

[6] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

Alistair Barros, SAP Research Centre, Brisbane, alistair.barros@sap.com
Marlon Dumas, Queensland University of Technology, m.dumas@qut.edu.au
Phillipa Oaks, Queensland University of Technology, p.oaks@qut.edu.au

A High-level meta-model in UML

The UML class diagrams in Figures 14 and 15 provide a high-level overview of WS-CDL's underlying meta-model. Figure 14 describes the concepts of *package* and *choreography* while Figure 15 drills down into the concepts of *activity* and *interaction* . These meta-models are based upon the editor's draft of the WS-CDL v1.0 specification dated 22 September 2004.

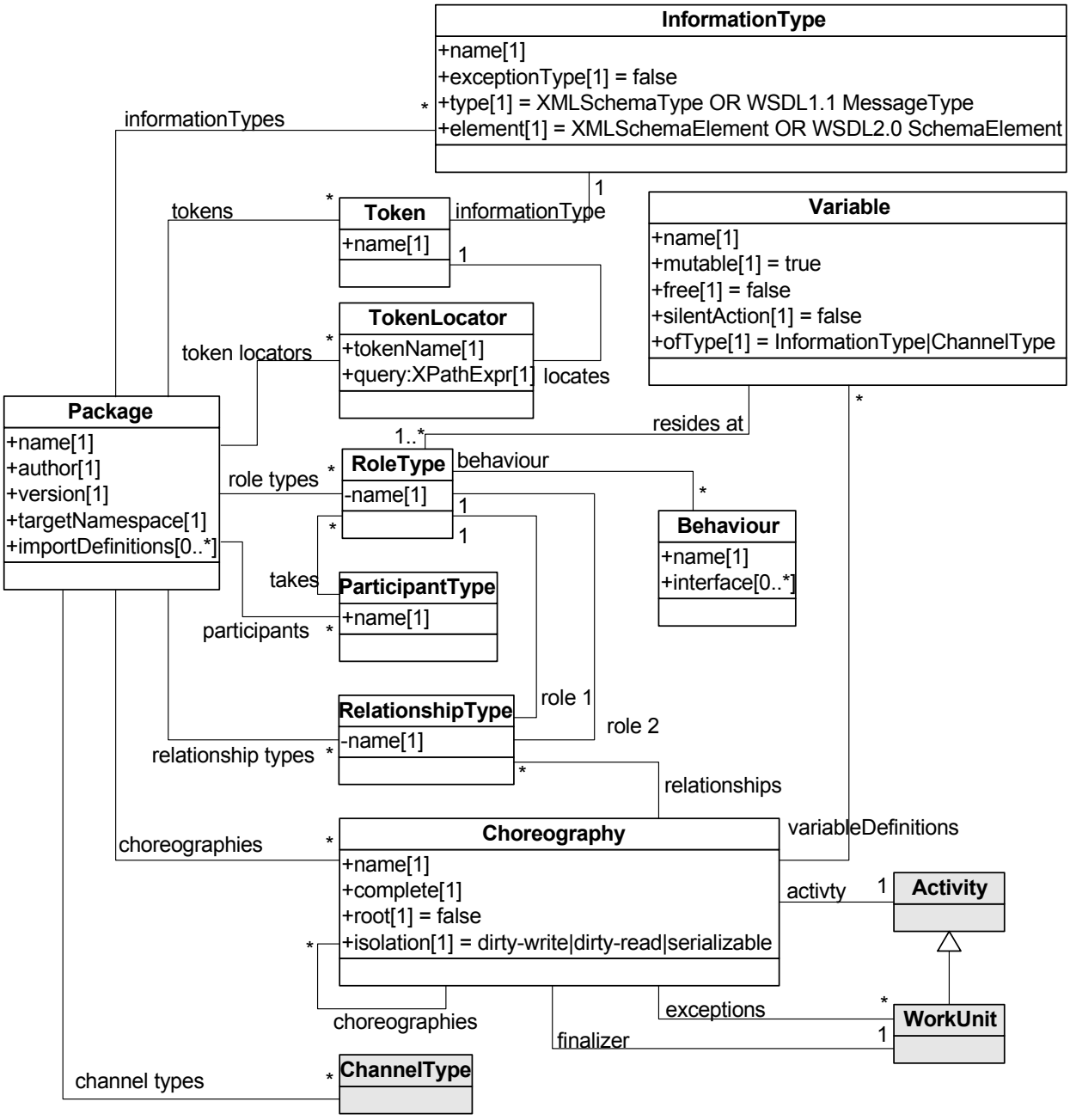

Figure 14. WS-CDL's meta-model in UML (part 1)

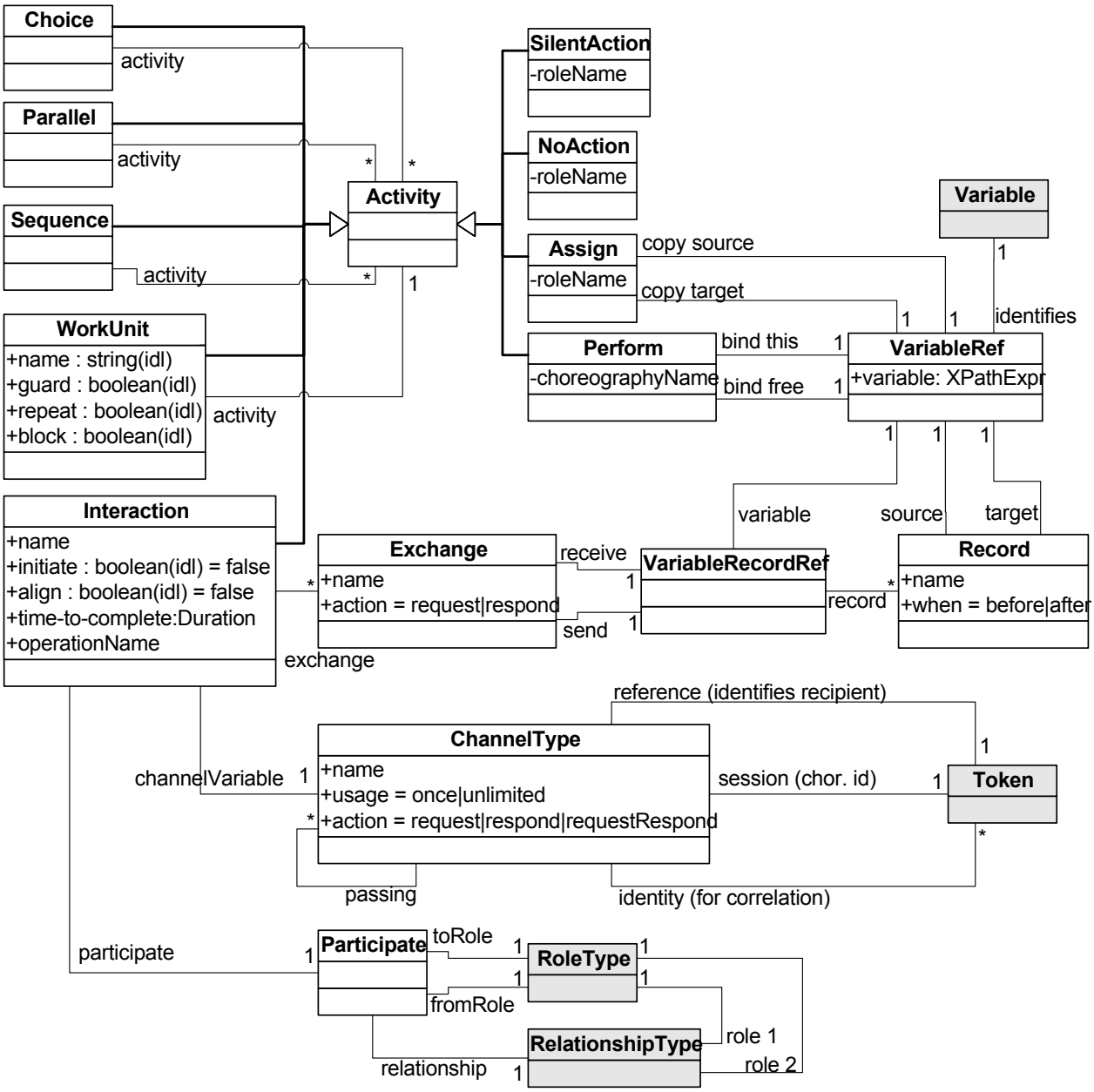

Figure 15. WS-CDL's meta-model in UML (part 2)

B Detailed meta-model in ORM

Figures 18provide a more detailed overview of WS-CDL's meta-model using the Object-Role modeling notation (ORM) [4].[14] The ORM notation permits capturing more constraints graphically than UML class diagrams. Annotations are used to provide even more details and to refer to specific sections of the WS-CDL specification. The ORM meta-models are based upon the editor's draft of the WS-CDL v1.0 specification dated 22 September 2004.

The models can also be downloaded from:
http://sky.fit.qut.edu.au/~dumas/WS-CDL/WS-CDL-MetaModels.zip

[14] http://www.orm.net

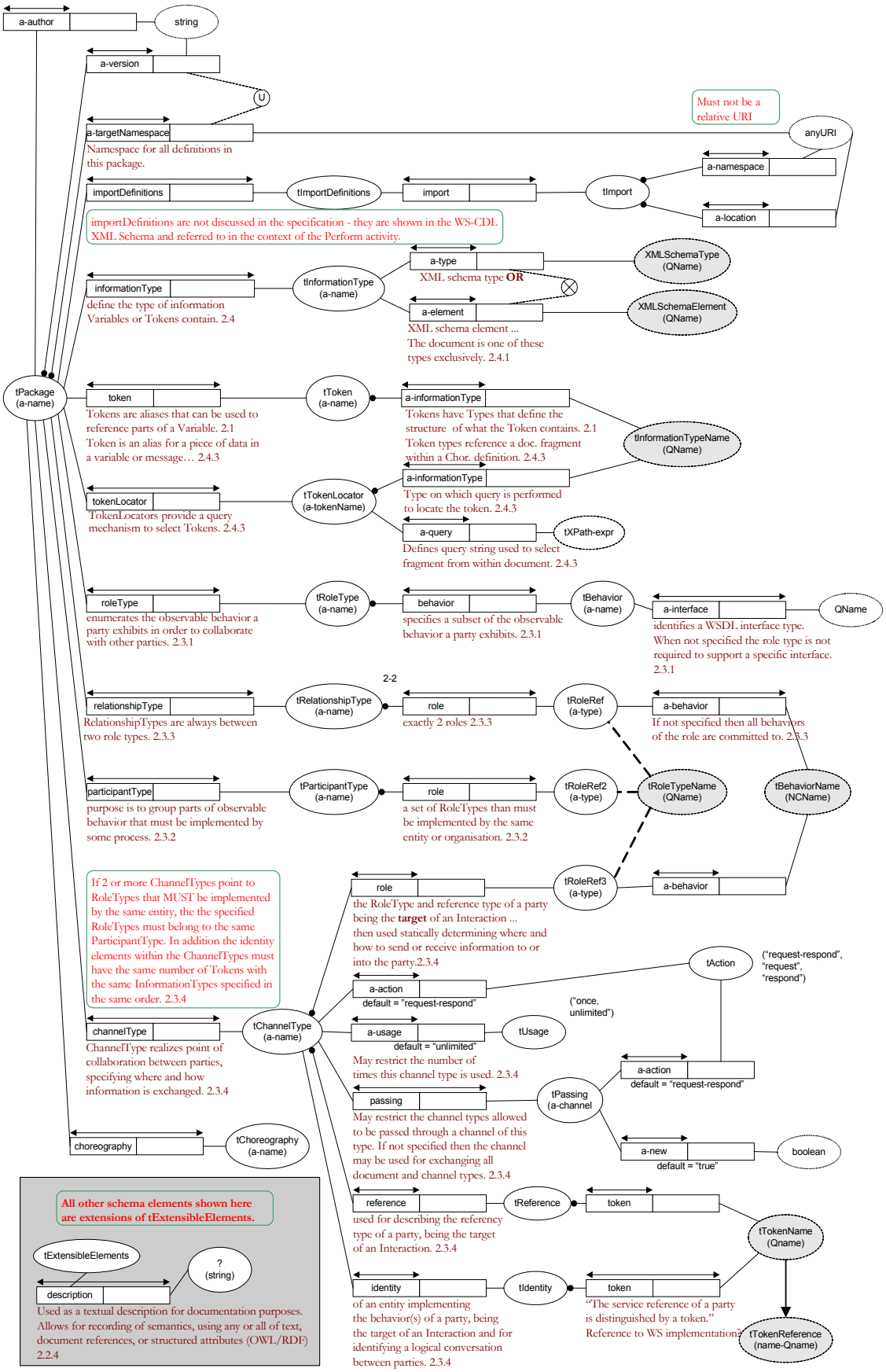

Figure 16. Fragment of WS-CDL's meta-model related to the concept of *Package*

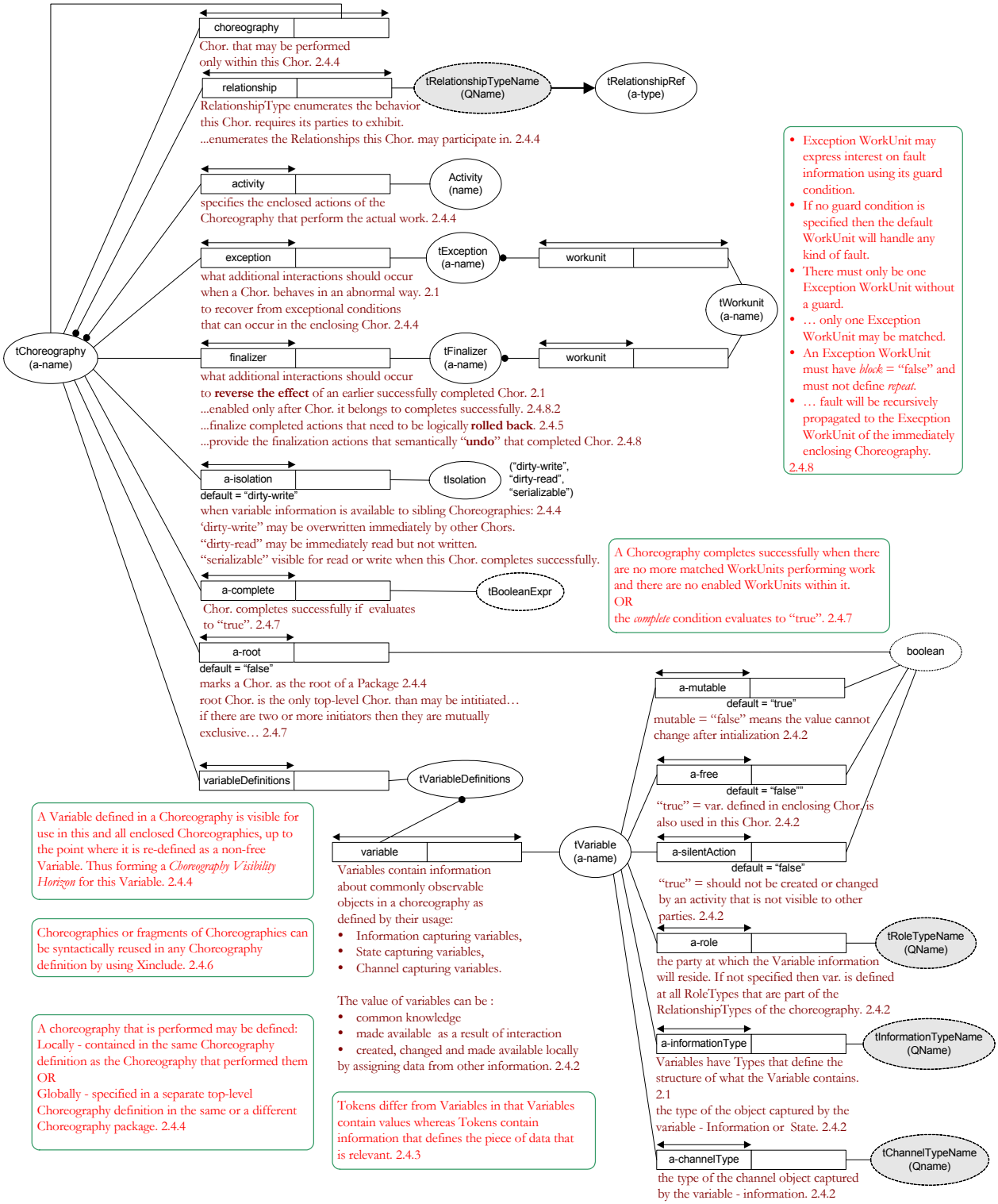

Figure 17. Fragment of WS-CDL's meta-model related to the concept of *Choreography*

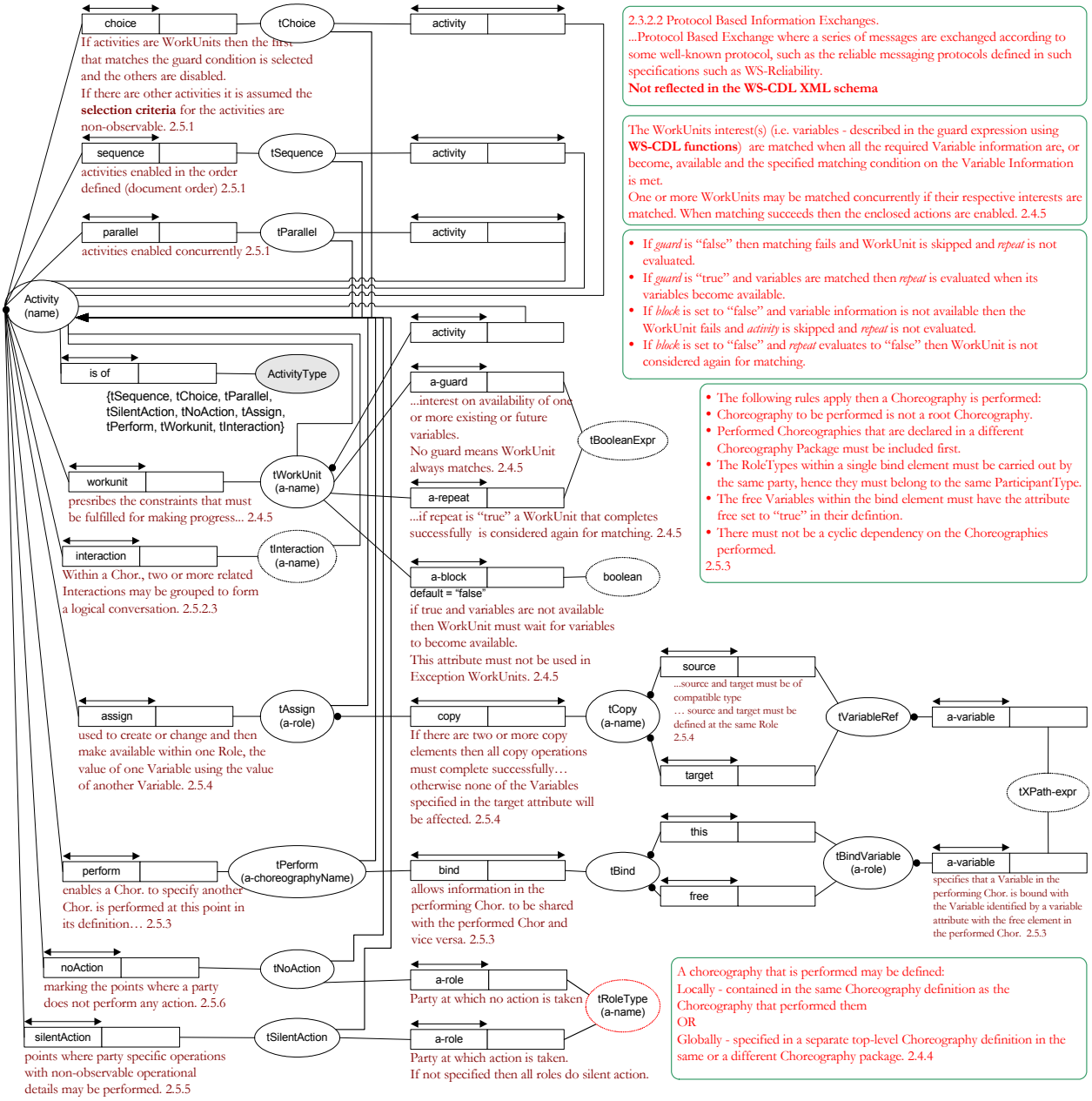

Figure 18. Fragment of WS-CDL's meta-model related to the concept of *Activity*

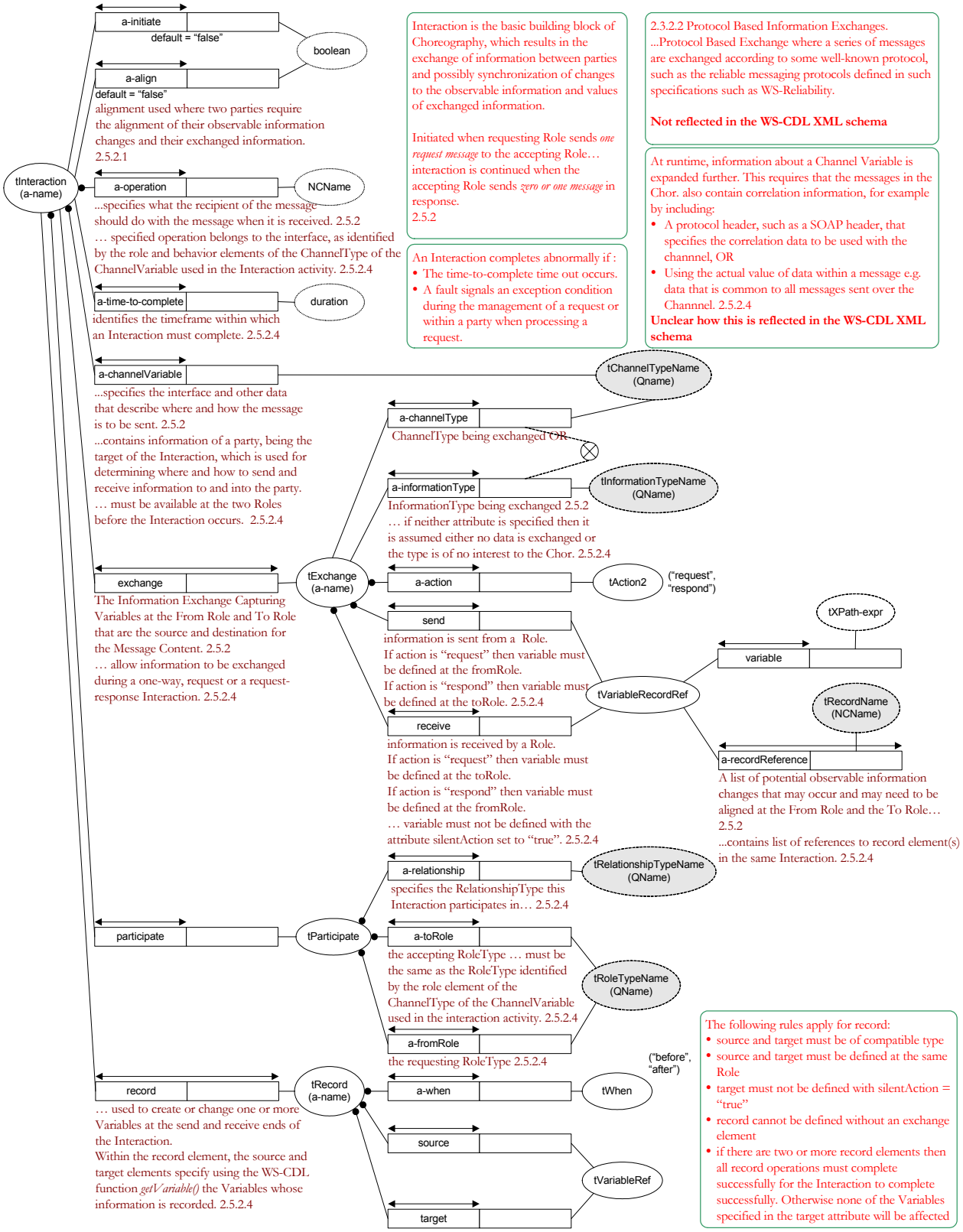

Figure 19. Fragment of WS-CDL's meta-model related to the concept of *Interaction*

