Implementing WS-CDL

Lars-Åke Fredlund

LSIIS, Facultad de Informática, Universidad Politécnica de Madrid

fred@babel.ls.fi.upm.es

Abstract

The WS-CDL definition, currently a W3C candidate recommendation, describes a language which is meant for characterising interactions between distinct web services. However, so far there are few tools that allow to check and execute specifications written in the new WS-CDL language. Apart from missing tool support, many basic language features are currently poorly understood. In contrast to traditional languages for describing Web Service interaction, the WS-CDL does not describe traditional two-party communications from the point-of-view of one of the participants. Rather, the WS-CDL language describe interaction scenarios involving multiple parties and from a viewpoint independent of its process participants. To allow us to experiment with the new language, and in general gain more insight in the design principles of the WS-CDL standard proposal, we have implemented a tool that allow us to syntax check, execute and debug WS-CDL descriptions, and which in the near future will allow model checking of WS-CDL descriptions.

1. Introduction

WS-CDL, the Web Services Choreography Description Language [3], is a new XML [2] based language for describing interactions between web services. In November 2005, it reached the status of W3C Candidate Recommendation.

In contrast to, say, WS-BPEL [4], in WS-CDL there is no notion of a unique Web Services being defined (through use of other Web Services in its definition). In WS-CDL Web Service interactions are described in a peer-to-peer manner, there is no notion of a coordinating Web Service nor of a coordinated Web Service. In popular terms the developers of WS-CDL likes to distinguish between traditional *orchestration languages*, like WS-BPEL that uses existing services to orchestrate a new service, and between *choreography languages*, like WS-CDL, that describe the interaction points between existing Web Services without defining new such services.

The developers of WS-CDL have also made public claims that the language is based on a particular formal method, the π-calculus [7] process algebra, and that therefore the proposed language is particularly well-suited language for describing concurrent processes and dynamic interconnection scenarios.

It is not clear, however, that the reality of the current WS-CDL standard proposal really fulfils the lofty vision of its designers. Already the language proposal have met with widespread criticism. WS-CDL is considered not useful for developing concrete web services (in contrast to, say, WS-BPEL), nor is it directly useful for deriving implementations of Web Services but is rather intended to serve some abstract purpose (defining abstract interactions among services) that so far has seen little use in industry. As a further drawback it is difficult to write WS-CDL specifications that can be considered elegant or intuitive. As the language is based on XML, and lacking a graphical notion, specifications even of smaller interactions become rather lengthy. Moreover the language introduces a wealth of new design concepts such as e.g. channel passing, and constructs for expressing directly concurrent execution and nondeterministic choices, choreographies, and so on that are currently not very well understood in the world of Web Services. With regards to the connection to formal methods, so far there exists no accepted formal semantics of WS-CDL that shows exactly in what way the new language is related to the π-calculus (but see [8] for an early attempt).

Moreover, so far precious few tools exists that allow to experiment with the new WS-CDL language. Perhaps the most widely known tool is `pi4soa` [1], which provides an Eclipse plugin that provides a graphical editor to compose WS-CDL choreographies and generate from them compliant BPEL.

To address some of these problems we decided to develop an implementation of the WS-CDL language. Concretely our implementation is a tool that allows to parse and syntactically check WS-CDL specifications, to simulate (or run) specifications, and to model check specifications against correctness properties specified as safety automatons.

The value of the tool is however greater than its core func-

[1] `http://sourceforge.net/projects/pi4soa`

tionality. The simulation functionality of the tool is provided by implementing an interpreter for WS-CDL programs in the Erlang functional programming language. As the implementation of the interpreter is written in a clean way, the source text of the interpreter can in fact serve as a formal (operational) semantics of the WS-CDL language, something which so far is missing.

In the following we will in Section 2 first introduce the WS-CDL language, and then in Section 3 describe our implementation of WS-CDL. To illustrate the language features and the type of specifications that the tool can manage we include in the paper a small case study example; the source code is available in figures 2 and 3. Finally we draw some conclusions from the implementation effort, and outline a number of items for further work in Section 5.

2. A Short Introduction to WS-CDL

The WS-CDL language is, as are most other languages for describing Web services, based on XML. Given that no graphical syntax exists, descriptions of web service interactions become large for even small examples. The WS-CDL language can be understood to have two parts: a part describing *static* relationships which are invariant, e.g., for characterising types of XML data, for defining types of communication channels, types of communication partners, and so on. The second part describes *dynamic* behaviour: interactions between communication partners (web services) and their temporal interdependencies.

2.1. Static WS-CDL

The static part defines the collaborating entities in a WS-CDL specification.

Role Types All interactions takes place between different *role types*, for example between a Customer role type and a Retailer role type. A typical definition of role types can be found on lines 30–39 in figure 2.

It would probably have been preferable to call these entities roles instead of role types, as there is no possibility to create separate instances from a role type definition, but in this paper we will continue using the vocabulary of the WS-CDL definition. The same confusion with regards to types versus instances holds also for Participant and Channel Types (covered next).

Participant Types A participant type groups together role types, that are conceptually going to be realised by the same physical entity. For instance, if both the Retailer and the Warehouse role types of a business interaction are implemented by the same organisation, the organisation could be modelled using one Participant Type. In the example the participant types are defined on lines 9–15.

Relationship Types A relationship type declares that two Role Types (invariably two) have a need to interact to realise some set of web services, as seen on lines 41–44 in the static part of the example.

Channel Types Channel types describe the medium used to communicate between role types. A communication always, surprisingly given the supposedly high-level nature of the specification language, involves two parties. In the WS-CDL standard multi-party communication is not possible. WS-CDL interactions have two-phases, a *request* followed by an optional *response*. Two channel types for communicating with a retailer, and a channel for directly communicating with a customer, are provided on lines 46–57 of the example.

A channel type identifies the responding role type but not the requesting role type. Furthermore a channel type may include a section that describes how channel type instances[2] may be transmitted over another channel instance in interactions. Interestingly, the channel type may also restrict how many times and in what circumstances channel instances are used; e.g. stipulating for instance that a given instance can only be used *once*, in a single communication or *shared* that it can be used in multiple communications. However, the usage of these restriction modes for channel instances is poorly explained in the standards document, nor are such features present in the core π-calculus formalism either. Finally the static part also contains definitions of types, and of functions, defined in the XPath formalism [1], for retrieving subfields of XML based document data.

2.2. Dynamic WS-CDL

The core of WS-CDL can be found in the dynamic part. Here the notion of a *choreography* is defined. In WS-CDL a choreography represents, essentially, a use case.

A choreography concerns a set of relationships (interactions between role types) and includes a definition of the set of variables used to realise the data dependent behaviour of the choreography. A particular variable is located at a particular participant type, and all role types at a participant type share access to such a variable. A role type that is located at a different participant type from where a variable is located cannot access the variable. Thus, in the normal case, there is no implicit communication between web services located at different participants using shared variables.

[2] yes, there are instances of channel types

The choreography *activity* construct describes the behaviour of the choreography (as seen next), while an exception block defines the handling of exceptional events during the execution of the choreography activity, and a finalizer block describes the actions to be taken upon termination of the choreography. In addition a choreography can define sub-choreographies, and can make available to these sub-choreographies its own variables.

2.2.1. Choreography Activities

An activity is the basic programmatic building block of WS-CDL. Essentially there are three classes of activities: structural ones, workunit activities, and basic activities.

Basic Activities The principal basic activity is the `interaction`; an example interaction can be seen on lines 25-53 in figure 3. An interaction describes the binary communication between two role types, as a request–response set of events exchanging information about the value of variables located at the respective role type (or rather: participant types). Other basic activities include the starting of a sub-choreography (using the `perform` statement) and the assigning of variables (`assign`).

Workunits A `workunit` is essentially a conditional and a looping construct combined in one, enclosing an activity. A workunit has a guard, which if true permits the execution of the enclosed activity. If false, and if a blocking execution strategy has been specified, and the guard has a chance of becoming true in the future, the execution of the guards is halted until it becomes true. Upon successful execution of the workunit activity the repeat condition of the workunit is analysed, and if the condition is true, the workunit is enabled for future execution.

An example workunit that checks whether a variable has been initialised is shown on lines 15–23 in figure 3.

Structural Activities The most basic structural activity is the sequence, specifying that its arguments should be evaluated in sequence. The parallel activity, in contrast, permits parallel execution of its activity arguments. Finally the choice activity selects, based on the executability of its arguments, one of its arguments for continued execution. If, for instance, one branch continuously blocks (a workunit waiting for its guard condition to become true) then that choice branch will never be chosen. These constructs bear a strong resemblance to the corresponding process operators in the π-calculus in contrast to the more flow-oriented concurrency constructs available in WS-BPEL.

3. An Implementation of WS-CDL

As mentioned in the introduction we have implemented a tool to execute WS-CDL descriptions, and with the secondary goal of developing a semi-formal semantics for the language. The ambition is to better understand the design decisions of the language through a proper semantics, and moreover to provide the much needed facility of debugging the dynamic part of WS-CDL descriptions.

Although the standard says, “WS-CDL is not an ”executable business process description language” or an implementation language”, the purpose of our implementation is exactly to provide a tool for executing WS-CDL specifications. Essentially we have implemented an executable formal semantics for WS-CDL as a structured operational semantics transition relation over WS-CDL states (see later discussion on what constitutes a WS-CDL state). The implementation of the semantics allows to compute the possible next states from a given WS-CDL specification (in case data is not completely determined we cannot fully deduce which states are reachable).

The semantics is implemented in the Erlang programming language [5]. Erlang is, basically, an untyped eager functional programming language which offers excellent support for concurrency, communication and distribution. In contrast to most functional programming languages Erlang has actually been widely used in industry, at Ericsson (the chief language provider) as well as at a significant number of other companies.

Thanks partly to the industry usage, there exists in Erlang already good library support for working with XML based documents, e.g., the `xmerl` library provides a parser for XML documents, and the `xmerl_xpath` library permits executing `XPath` 1.0 queries against XML documents.

An alternative to formulating the operational semantics in Erlang would have been to use a tool such as, for instance, Maude system [6]. However clearly the level of XML and XPath support in Erlang is far superior to what is available in Maude. In fact thanks to these salient features of Erlang we were able to provide a relatively clean implementation of the semantics of WS-CDL; the functional part of Erlang provides a nice syntax for expressing the transition relation, whereas the XML and XPath support already available in Erlang is key to the evaluation of WS-CDL expressions. For instance, retrieving the channel variable from a purchase order (lines 48–52 in figure 3) is done by accessing the subfield `'/PO/CustomerRef'` (using an XPath expression) of the choreography variable `tns:purchaseOrder` (located at the Retailer) using the `cdl:getVariable` function. As can be seen, use of XML and XPath is typically used everywhere in a WS-CDL specification (XPath is also used in the specification of token locators), and having them avail-

able without effort constitutes a significant advantage when it comes to providing an executable implementation of the language semantics.

3.1. WS-CDL States and Transitions

In our semantics the notion of a WS-CDL state is an Erlang record type named `chor` with the fields *name*, *state*, *vars*, *blocking*, and *running*. This record type essentially represents the running instance of the top choreography of the WS-CDL specification: *name* is the name of that choreography, *state* is the state of the choreography (whether *enabled*, *in_exception*, *finalising*, *completed*) and *vars* is a mapping from the variables defined in the choreography to their current values.

A variable resides a particular participant, and is either uninitialised, or has a value, or is aligned with another value at another participant (one of the more advanced concepts of WS-CDL permits tight coordination of role type interactions leading to aligned – with the same value – variables), or its value is defined by an outer choreography.

The *running* field refers to the currently executing activity in the context of the current choreography. Initially `running` is equal to the first statement in the choreography, but the contents of the field changes during the execution of the choreography. In the general case, for instance, an activity may refer to a sub-choreography, leading the `running` field to contain a a sub-choreography WS-CDL record state. This sub-choreography WS-CDL of course has its own variables, and a separate activity.

The implementation of the set of transition rules for our WS-CDL semantics then is a function `doStep` that given three arguments: (i) the WS-CDL record state of the choreography to execute, (ii) a stack of surrounding choreographies in which the currently executing one is embedded, and (iii) and the static data present in the WS-CDL specification, computes and returns (as a list) the set of possible next states.

For completeness we show below the `doStep` function clause concerning the transition steps (execution) of a `perform` activity.

```
1  doStep(Chor, Chors, Package)
2    when Chor#chor.running={perform,P} ->
3
4    C = getChor(P#perform.chor,
5                Chor, Package),
6    Vars =
7      lists:foldl
8      (fun (VD,Vars) ->
9          if
10           VD#variableDef.free ->
11             [bind_var(VD,[Chor|Chors])|
12               Vars];
13           true ->
14             init_vars(VD,Package)++Vars
15         end
16       end,
17       {[],[]}, C#chor.variableDefs),
18
19   NewChor =
20     #chor{name=P#chor.name,
21           state=enabled,
22           vars=Vars,
23           blocking=P#perform.blocking,
24           running=[P#chor.activity]},
25
26   [Chor#chor{running={chor,NewChor}}];
```

Although the example uses concrete Erlang syntax it is fairly easy to understand. In Erlang variables begin with a capital letters and atoms (literals) with a lowercase letter. Variables can be assigned once only (meaning there is never two consecutive assignments to the same variable). The Erlang syntax `Variable#RecordName.Field` is used to access a field `Field` of the variable `Variable` of record type `RecordName`, the syntax `#RecordName{Field1=Value1,...}` is used to create a record and `Variable#RecordName{Field=Value}` returns a new record where the value of the referenced field has changed but otherwise the record is identical to the record stored in `Variable`.

Thus the above function clause is chosen when called with a first argument record whose `running` field matches the functional pattern `{perform,P}`. Essentially the function retrieves the definition of the choreography (lines 4–5), and then initialises the variables of the choreography in lines 6–17. Free variables are bound by the outer choreographies (`[Chor|Chors]`) while the location (participant types) of fresh variables are computed using `init_vars` function. Then a new choreography `running` state record is initialised in lines 19–24. Its name is derived from the choreography description, it is `enabled` to execute, it may be blocking, and its initial activity (`running`) is also fetched from the choreography description.

Finally in the last line the function returns the new set of states, which in the `perform` clause is just a single state that is identical to the incoming state, except the `running` record component now contains a new sub-choreography. For completeness the `doStep` clause for such a sub-choreography is also shown below. Any step by the sub-choreography gives rise to a set of new states, which are embedded again in the outer choreography.

```
1  doStep(Chor, Chors, Package)
2    when Chor#chor.running={chor,Ch} ->
3      lists:map
4        (fun (NewCh) ->
5           Chor#chor
```

```
6           {running={chor,NewCh}}
7         end,
8         doStep
9         (Ch,[Chor|Chors],Package));
```

3.2. Tool Functionalities

As an extension to standard WS-CDL language the tool provides the facility to simulate instantiated specifications through the use of a XPath function `cdl:doc` for embedding an XML document into the body of the WS-CDL code. In the example this construct is used on line 19 in figure 3 to initialise the `purchaseOrder` variable at the `Customer` role type.

On top of the WS-CDL semantics we have built a simple debugging environment for single-stepping WS-CDL specifications, setting breakpoints (based on variable assignments, and role type interactions), and for executing (simulating) WS-CDL specification without stopping. As WS-CDL specifications are inherently non-deterministic in simulation a next state is chosen randomly, while in debugging mode the user may influence the choice.

Moreover there is a possibility to model check a specification against a correctness property formulated as a safety automaton. Such an automaton has the power to inspect the WS-CDL specification state. For example, to check that in all possible interactions a choreography variable always receives a well-defined value at the conclusion of a choreography. Note that such correctness properties investigate the internal soundness of the WS-CDL specification itself, and do not address the problem of whether a WS-CDL specification has been correctly implemented (orchestrated). Such verification is possible, but left for future work.

3.3. The WS-CDL Example Simulated

The small WS-CDL example in figures 2 and 3 was simulated in our tool. The result can be depicted symbolically as in figure 1. The example has 7 transitions in our simulator. The first transition corresponds to the Customer executing the guard of the workunit (line 16), and since the guard succeeds, then the body of the workunit (the assignment statement in lines 17 to 22 which initialises the `purchaceOrder` variable at the `Customer` side) provides the second transition.

Then, the `Customer` interacts with the `Retailer` issuing a request (lines 30–35), and the `Retailer` sends a response indicating either a successful or failed purchase operation. Note that, because the `align` option was not specified in the scenario, the sending and the receiving part of a `exchange` is logically separate transitions in the diagram,

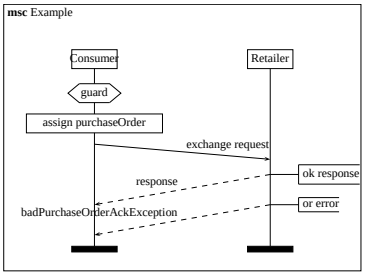

Figure 1: Message Sequence Chart for the WS-CDL Example

since there is a possibility that other concurrent activities could be interleaved with the exchange between `Customer` and `Retailer`.

4. Acknowledgments

The author was supported by a Ramón y Cajal grant from the Ministerio de Educación y Ciencia.

5. Conclusions and Further Work

It is fair to say that the tool still remains a prototype. The full WS-CDL language is not yet covered. We do, for instance, not handle the non-"isolated" choreographies (i.e., updates to free choreography variable take place immediately). Nor is the exception mechanism and the choreography coordination mechanism completely handled for now. We expect to address these limitations of the prototype in the near future.

However, even so the prototype represents a powerful tool to experiment with the new WS-CDL language as it is very instructive to trace message exchanges and variable updates as result of the dynamics of the WS-CDL specification. Moreover the implemented semantics clearly has a value in itself, it serves to clarify some obscure corners of the WS-CDL standard. We expect to publish a separate paper regarding the use of the tool semantics as a semi-formal semantics for WS-CDL in the near future. With regards to the WS-CDL standard itself the implementation exercise has highlighted a number of language constructs which for now are either poorly defined, or just poorly explained in the reference document. We enumerate a few such issues below:

- Coordination of exceptions: The exception (and finalising) mechanisms are in general poorly explained (and defined). For instance there is some confusion in the standards document regarding whether exceptions impact only certain role types in a choreography, or whether all role types are impacted. The likely intention is that choreography coordination is required to no-

tify all role types, but this is inadequately specified in the reference document.

- The underlying semantic model, for implementations of WS-CDL specifications, is in general vague. For instance, what assumptions regarding communication mechanism would be sufficiently powerful to implement the interaction activity with alignment and choreography coordination?

- The channel passing mechanism is poorly explained in the standard. Exactly which conditions does the reference text put on channel usage (the annotations of the `usage` optional attribute in a channel type declaration)? Moreover in general the channel passing mechanism seems overly clumsy. Why is it at all necessary to specify that channels can be passed in the channel type definition? The intention of the language designers with regards to the channel mechanisms are in general far from clear, even for the purpose of language implementation.

A further item for future work is to establish a connection to WSDL (Web Services Description Language), for the purpose of testing a set of interacting WSDL web services against the combined service prescribed by a WS-CDL specification.

References

[1] XML Path Language (XPath) Version 1.0. Technical report, W3C, November 1999.

[2] Extensible Markup Language (XML) 1.0. Technical report, W3C, February 2004.

[3] Web Services Choreography Description Language, Version 1.0 – W3C candidate recommendation 9 november 2005. Technical report, W3C, November 2005.

[4] Web Services Business Process Execution Language Version 2.0 (draft). Technical report, OASIS, 2006.

[5] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall International (UK) Ltd., 1996.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude system. In *Rewriting Techniques and Applications, 10th International Conference, RTA'99, Trento, Italy, July 2–4, 1999, Proceedings*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243. Springer-Verlag, 1999.

[7] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I and II. *Information and Computation*, 100(1):1–40 and 41–77, 1992.

[8] Z. X. Yang Hongli and Q. Zongyan. A formal model of web service choreography description language (ws-cdl). Technical report, Department of Informatics, School of Math., Peking University, China, January 2006.


```
1  <informationType name="purchaseOrderType"
2                   type="tns:PurchaseOrderMsg"/>
3  <informationType name="purchaseOrderAckType"
4                   type="tns:PurchaseOrderAckMsg"/>
5  <informationType name="badPOAckType"
   type="xsd:QName" />
6  <informationType name="uriType"
   type="xsd:string" />
7  <informationType name="intType"
   type="xsd:integer" />
8
9  <participantType name="Consumer">
10   <roleType typeRef="tns:Consumer" />
11 </participantType>
12
13 <participantType name="Retailer">
14   <roleType typeRef="tns:Retailer" />
15 </participantType>
16
17 <token name="purchaseOrderID" informationType="tns:intType"/>
18 <token name="retailerRef" informationType="tns:uriType"/>
19 <token name="consumerRef" informationType="tns:uriType"/>
20
21 <tokenLocator tokenName="tns:purchaseOrderID"
22               informationType="tns:purchaseOrderType"
23               query="/PO/orderId"/>
24 <tokenLocator tokenName="tns:purchaseOrderID"
25               informationType="tns:purchaseOrderAckType"
26               query="/PO/orderId"/>
27 <tokenLocator tokenName="tns:retailerRef"
28               query="/PO/retailerId"
29               informationType="tns:purchaseOrderType"/>
30
31 <roleType name="Consumer">
32   <behavior name="consumerForRetailer"
33             interface="rns:ConsumerRetailerPT"/>
34   <behavior name="consumerForWarehouse"
35             interface="rns:ConsumerWarehousePT"/>
36 </roleType>
37 <roleType name="Retailer">
38   <behavior name="retailerForConsumer"
39             interface="rns:RetailerConsumerPT"/>
40 </roleType>
41
42 <relationshipType name="ConsumerRetailerRelationship">
43   <roleType typeRef="tns:Consumer"
44             behavior="consumerForRetailer"/>
45   <roleType typeRef="tns:Retailer"
46             behavior="retailerForConsumer"/>
47 </relationshipType>
48
49 <channelType name="ConsumerChannel">
50   <roleType typeRef="tns:Consumer"/>
51   <reference><token name="tns:consumerRef"/></reference>
52   <identity ><token name="tns:purchaseOrderID"/></identity>
53 </channelType>
54
55 <channelType name="RetailerChannel">
56   <passing channel="ConsumerChannel" action="request" />
57   <roleType typeRef="tns:Retailer"
58             behavior="retailerForConsumer"/>
59   <reference> <token name="tns:retailerRef"/>
   </reference>
60   <identity> <token name="tns:purchaseOrderID"/>
   </identity>
61 </channelType>
```

Figure 2: WS-CDL Example: Static Part

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

  <sequence>
  <workunit name="unit" block="false"
     guard="not(cdl:isVariableAvailable('tns:purchaseOrder','tns:Consumer'))">
  <assign roleType="tns:Consumer">
    <copy name="copy1">
      <source expression="cdl:doc('&lt;PO&gt; &lt;orderId name=&quot;10&quot;/&gt; &lt;CustomerRef name=&quot;1000&quot;/&g
      <target variable="cdl:getVariable('tns:purchaseOrder','','')"/>
    </copy>
  </assign>
  </workunit>

  <interaction name="createPO"
               channelVariable="tns:RetailerChannel"
               operation="handlePurchaseOrder" >
    <participate relationshipType="tns:ConsumerRetailerRelationship"
                 fromRoleTypeRef="tns:Consumer" toRoleTypeRef="tns:Retailer"/>
    <exchange name="request"
              informationType="tns:purchaseOrderType" action="request">
      <send variable="cdl:getVariable('tns:purchaseOrder','','')" />
      <receive variable="cdl:getVariable('tns:purchaseOrder','','')"
               recordReference="record-the-channel-info" />
    </exchange>
    <exchange name="response"
              informationType="purchaseOrderAckType" action="respond">
      <send variable="cdl:getVariable('tns:purchaseOrderAck','','')" />
      <receive variable="cdl:getVariable('tns:purchaseOrderAck','','')" />
    </exchange>
    <exchange name="badPurchaseOrderAckException" faultName="badPurchaseOrderAckException"
              informationType="badPOAckType" action="respond">
      <send variable="cdl:getVariable('tns:badPurchaseOrderAck','','')"
              causeException="tns:badPOAck" />
      <receive variable="cdl:getVariable('tns:badPurchaseOrderAck','','')"
              causeException="tns:badPOAck" />
    </exchange>
    <record name="record-the-channel-info" when="after">
      <source variable="cdl:getVariable('tns:purchaseOrder','',
                                        '/PO/CustomerRef')"/>
      <target variable="cdl:getVariable('tns:consumer-channel','','')"/>
    </record>
  </interaction>
  </sequence>
 </choreography>
</package>
```

Figure 3: WS-CDL Example: Dynamic Part