On the Suitability of WS-CDL for Choreography Modeling (Extended Version)

Gero Decker,[*] Hagen Overdick
Hasso-Plattner-Institute, University of Potsdam, Germany
(gero.decker,hagen.overdick)@hpi.uni-potsdam.de

Johannes Maria Zaha
Queensland University of Technology, Australia
j.zaha@qut.edu.au

Abstract: The Web Service Choreography Description Language (WS-CDL) has been put forward as language for capturing sets of web service interactions and their control and data dependencies, seen from a global perspective. However the suitability of WS-CDL for this purpose has not been assessed in a systematic manner. This paper presents such an assessment by adopting a two-pronged approach. First, the paper studies the relation between WS-CDL and π-calculus, a well-known formalism for specifying communicating systems with dynamic topologies. Second, WS-CDL is assessed in terms of its support for two collections of patterns: the Workflow Patterns which capture recurrent control-flow dependencies in business processes, and the Service Interaction Patterns which capture recurrent compositions of interactions between services.

1 Introduction

With increasing maturity of implementation languages for Service-Oriented Architectures, languages for describing service interactions on a higher level of abstraction and in the early phases of the development lifecycle are emerging. WS-CDL allows one to define interactions, and to compose these interactions through control-flow dependencies in order to capture collaborative business processes. The Workflow Patterns [vdAtHKB03] are an established framework for evaluating the control-flow perspective of business process definition languages, while the Service Interaction Patterns [BDtH05] have been put forward as a benchmark for evaluating languages that support the definition of interactions and compositions thereof. Therefore, the combination of these two frameworks provides a basis for evaluating the suitability of WS-CDL for choreography definition. As a starting point, the relationship between WS-CDL and π-calculus is investigated. This allows for identifying the capabilities of WS-CDL to elevate the level of abstraction by providing higher-level constructs than π-calculus.

The paper is structured as follows. In Section 2 an introduction to π-calculus is given

[*]Research conducted while with SAP Research Centre Brisbane

and the similarity between WS-CDL and π-calculus is shown. Sections 3 and 4 provide solutions for the Workflow Patterns and the Service Interaction Patterns. Finally, Section 5 concludes.

2 WS-CDL and π-calculus

The π-calculus is a process algebra for mobile systems [MPW92]. In π-calculus, communication takes place between different π-processes. Names are a central concept in π-calculus. Links between processes as well as messages are names. This allows for link passing from one process to another. The scope of a name can be restricted to a set of processes but may be extruded as soon as the name is passed to other processes. The syntax for π-calculus looks as follows:

Listing 1: π-calculus syntax

$$\begin{aligned} P &::= M \mid P|P' \mid (\nu\ z)P \mid !P \\ M &::= 0 \mid \pi.P \mid M + M' \\ \pi &::= \overline{x}\langle y\rangle \mid x(y) \mid \tau \end{aligned}$$

Concurrent execution is denoted as $P|P'$, the restriction of the scope of z to P as $(\nu\ z)P$ and an infinite number of concurrent copies of P as $!P$ (“bang-operator”). Inaction of a process is denoted as 0. A non-deterministic choice between M and M' as $M + M'$ and sending y over x as $\overline{x}\langle y\rangle$. The prefix $x(y)$ receives a name over x and continues as P with y replaced by the received name. τ is the unobservable action. Communication between two processes can take place in the case of matching send- and receive-prefixes. It has been shown that all Service Interaction Patterns can be expressed using π-calculus ([DPW06]). As an example the Request with referral pattern, where a party A sends a request to party B indicating that a follow-up response should be sent to another party C, can be encoded as follows:

Listing 2: Request with referral

$$\begin{aligned} &A = (\nu\ a)\ \overline{b}\langle req, c, a\rangle\ .\ a(resp)\ .\ 0 \\ &B = b(req, x, y)\ .\ \overline{x}\langle resp, y\rangle\ .\ 0 \\ &C = c(msg, z)\ .\ \overline{z}\langle resp\rangle\ .\ 0 \end{aligned}$$

Party A creates a fresh name a which will be used as response channel later on. A passes the request together with a as well as a channel c to B. B receives the request, processes it and sends a response to the specified party. B does not need to know this third party in advance. In this example x is replaced with c so B uses this channel to pass on his response. C receives this message and sends his response back over the return channel a. Each π-process describes the publicly visible behavior of a participant (“interface process”). The combination of these behaviors determines the global interaction protocol.

Each π-interaction in this setting represents an interaction between participants. In contrast this example, π-interactions can also be used for control flow coordination within one participant. This strategy was followed in the formalization of all the Workflow Patterns ([PW05]) and some Service Interaction Patterns. Each activity within a process is represented by a π-process and coordination between these processes takes place via interactions. If a process participates in a collaboration with other processes, the interactions used for control flow coordination are not visible to the outside world.

Comparison. In the π-calculus example in Listing 2 we described "stitched" interface processes. Control flow is specified between communication actions which leads to an endpoint-centric view. The Web Services Choreography Interface standard (WSCI [A$^+$02]) follows this approach, too. In contrast to this, WS-CDL treats interactions between services as first-class citizens. Therefore, control flow is defined between interactions rather than communication actions.

Besides the direct support for channel passing in WS-CDL, especially the basic control flow constructs correspond to concepts in π-calculus. Some of them have different semantics though:

- `<sequence>` directly corresponds to a sequence in π-calculus (".").
- `<parallel>` corresponds to parallelism in π-calculus ("|"). However, the `<parallel>` construct of WS-CDL also implies merging after the parallel branches have completed. In π-calculus this is not the case. Here, we would have to introduce an additional interaction for synchronization purposes.
- *Non-blocking performs* (`<perform block="false">`) of choreographies also correspond to parallelism in π-calculus. No synchronization is needed after the performed choreography has completed. Using a non-blocking perform within an infinitely repeated work unit, the bang-operator "!" can be realized in WS-CDL.
- `<choice>` corresponds to a choice in π-calculus ("+"). Once again, merging the alternative branches is implied in WS-CDL which would have to be encoded differently in π-calculus.

Some WS-CDL constructs cannot be directly found in π-calculus:

- *(Non-blocking) guarded work units* (`<workunit guard=".." block="false">`) contain interactions that can only occur if the given condition evaluates to true. In π-calculus we could express this using a choice with two guarded alternatives where one alternative would be a τ_0-action. Furthermore, merging before follow-up activities would have to be encoded.
- *Repeated work units* (`<workunit repeat="..">`) are a convenient means to introduce repetitions into choreographies. In π-calculus either recursion or the bang-operator have to be used.

Like already mentioned in the previous section, π-interactions can be used to encode complex control flow. A direct support for this facility is missing in WS-CDL. However, *blocking work units* can be used to realize a similar semantics in WS-CDL. Assume the following π-process:

$$(\nu\, h, i)\ (\tau_{AB}\ .\ (\bar{i}\ .\ 0 \mid \tau_{BC}\ .\ h\ .\ \tau_{BA}\ .\ 0) \mid \tau_{DB}\ .\ i\ .\ \tau_{BD}\ .\ \overline{h})$$

The τ-actions represent service interactions. The resulting global interaction protocol is: After interactions AB and DB, BD can happen. However, BC can already happen directly after AB without waiting for DB. Finally, BA can happen after BC and BD. Figure 1 illustrates this.

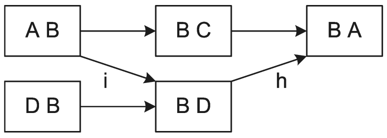

Figure 1: Sample choreography

We have introduced the fresh names h and i for realizing the desired control flow. In WS-CDL we can also introduce two variables h and i that cause the blocking of interaction BA and BD, respectively. For the rest of the control flow logic we use two sequences that run in parallel and assign a value to variables h and i at the appropriate places so that the blocked work unit are activated:

Listing 3: Sample choreography in WS-CDL

```
<parallel>
  <sequence>
    interactionAB
    <assign roleType="B"><copy>
      <source expression="true" />
      <target variable="cdl:getVariable('i')" />
    </copy></assign>
    interactionBC
    <workunit guard="cdl:getVariable('h')" block="true">
      interactionBA
    </workunit>
  </sequence>
  <sequence>
    interactionDB
    <workunit guard="cdl:getVariable('i')" block="true">
      interactionBD
    </workunit>
    <assign roleType="B"><copy>
      <source expression="true" />
      <target variable="cdl:getVariable('h')" />
    </copy></assign>
  </sequence>
</parallel>
```

π-calculus has a minimal number of language constructs in order to allow for elegant reasoning. In contrast to this, WS-CDL is more of an engineering medium to facilitate the design of service choreographies and therefore includes more high-level concepts. Unlike π-calculus, WS-CDL is typed and has a high expressiveness for specifying data structures ("information types") and conditions such as repetition conditions and guard conditions. Role types, relationship types, participant types and channel types can be easily described. Identity tokens definitions are a convenient way to capture correlation.

Since WS-CDL is tightly coupled with web service technology and does not provide any graphical representations, we argue that WS-CDL should only be used in the latest stages of the choreography definition process. Like it is proposed to use the Business Process Modeling Notation (BPMN [bpm06]) prior to encoding process descriptions in BPEL, there should be a similar choreography modeling notation for early choreography design stages. UML2.0 Activity Diagrams have an extensible meta-model that could used for defining a choreography notation. However, to the best knowledge of the authors there is no publication introducing such an extension.

3 Workflow Pattern support

The Workflow Patterns were introduced by van der Aalst et al. in [vdAtHKB03]. They serve as a reference framework for assessing process modeling languages and workflow systems in terms of control flow expressiveness. WSCI as well as a number of other standards (e.g. BPEL, UML 2.0 Activity Diagrams, BPMN) have already been assessed ([vdADtHW02], [WvdADtH03], [RvdAtHW06], [WvdAD^{+}06]). These assessments serve as reference for identifying if direct, partial or no support for a particular pattern is given.

3.1 Basic control flow patterns

Sequence is directly supported through the structure type `<sequence>` in WS-CDL. As shown above, control flow dependencies between interactions could also be realized through blocking work units and variable assignments.

Listing 4: Sequence

```
<sequence>
  activity1
  activity2
</sequence>
```

Corresponding ***Parallel Splits*** and ***Synchronizations*** can be expressed using `<parallel>` structures. Although only block structures can be captured, we follow the assessments for these two patterns in XLang ([WvdADtH03]) and BPML / WSCI ([vdADtHW02]) and therefore conclude that there is direct support for them. A workaround for encoding arbitrary *Synchronizations* is the usage of blocking work units and variables (cf. Listing 3). A *Parallel Split* without synchronization can alternatively be expressed using non-blocking `<perform>` structures.

Listing 5: Parallel Split & Synchronization

```
<parallel>
  activity1
  activity2
</parallel>
```

The ***Exclusive Choice*** pattern appears as the structure type `<choice>` in WS-CDL. It specifies that only one activity within the structure is selected and all other activities are disabled. Different control flow behaviors apply depending on whether or not work units with guard conditions are contained in the structure. In the case where such work units are present the behavior of an *Exclusive Choice* applies. The first work unit that matches the guard condition is selected. `<choice>` also implements ***Simple Merge*** at the same time. Thus direct support for both patterns. Arbitrary *Simple Merges* can again be implemented using blocking work units as a workaround.

Listing 6: Exclusive Choice & Simple Merge

```
<choice>
  <workunit guard=".."> ..
  </workunit>
  <workunit guard=".."> ..
  </workunit>
</choice>
```

3.2 Advanced Branching and Synchronization patterns

In the case of the ***Multiple Choice*** pattern a number of branches are chosen. This can be expressed using a `<parallel>` structure containing guarded work units. A matching ***Synchronizing Merge*** is also covered by this structure.

Listing 7: Multiple Choice & Synchronizing Merge

```
<parallel>
  <workunit guard=".." ..>...</workunit>
  <workunit guard=".." ..>...</workunit>
</parallel>
```

Multiple Merge: A point in a process where two or more branches reconverge without synchronization. If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every activation of every incoming branch. Solution: Non-blocking `<perform>` structures can be used to implement this pattern. A sub-choreography is defined and instances are performed from within parallel activities. This style of implementation is similar to spawning off new process instances in BPML, which causes problems if the *Multi Merge* is placed within a cycle. In analogy to BPML, we opt for partial support for this pattern.

Listing 8: Multiple Merge

```
<choreography name="chor1">
  activity1
</choreography>
<choreography>
  <parallel>
    <sequence>
      activity2
      <perform choreographyName="chor1" block="false"/>
    </sequence>
    <sequence>
      activity3
      <perform choreographyName="chor1" block="false"/>
    </sequence>
  </parallel>
</choreography>
```

The ***Discriminator*** is a point in a process that waits for one of the incoming branches to complete before activating the subsequent activity. Solution: A blocking work unit can be used to emulate a *Discriminator*. Several activities running in parallel assign a value to a synchronization variable. As soon as this happens the blocking work unit can start. Since there is no direct language construct for the *Discriminator* and implementations always involve bookkeeping through variables, there is no support for this pattern.

Listing 9: Discriminator

```
<parallel>
  <workunit ..>
    ...
    <assign roleType="Role1">
      <copy name="activate subsequent activity">
        <source expression="true" />
        <target variable="cdl:getVariable('enable','','','Role1')" />
      </copy>
    </assign>
  </workunit>
  <workunit ..>
    ...
    <assign roleType="Role1">
      <copy name="activate subsequent activity">
        <source expression="true" />
        <target variable="cdl:getVariable('enable','','','Role1')" />
      </copy>
    </assign>
  </workunit>
  <workunit guard="cdl:getVariable('enable','','','Role1') == true"
```

```
      block="true" ..>
    ...
  </workunit>
</parallel>
```

3.3 Structural patterns

Arbitrary Cycles is a point in a process where one or more activities can be done repeatedly. *Arbitrary Cycles* are not supported in WS-CDL. Cycles with one entry and one exit point are implemented using repeated work units.

Implicit Termination: A given sub-process should be terminated when there is nothing else to be done. This pattern covers how to describe the point of termination of an instance of the model. In the case of WS-CDL there is direct support for this pattern: Sub-choreography instances can be activated using non-blocking performs and there is thus no defined point in the choreography where termination of the choreography instance happens.

3.4 Patterns involving Multiple Instances

These patterns describe scenarios where multiple instances of an activity can be created in the context of a single case. If the number of instances is known at design-time (***MI with a priori design time knowledge***) the activities could be replicated and placed in a `<parallel>` structure. These structures also indicate synchronization after completion of the activities. Alternatively, a sub-choreography can be defined and activated several times using `<perform>` actions. These actions can either be blocking or non-blocking while in the latter case no synchronization takes place (***MI without synchronization***).

Listing 10: Workaround for MI without synchronization

```
<choreography name="subchoreography">
  ..
</choreography>
..
<choreography ..>
  <workunit .. repeat="..">
    <perform choreographyName="subchoreography" block="false">
      ..
    </perform>
  </workunit>
</choreography>
```

Listing 11: Workaround for MI with a priori design-time knowledge

```
<choreography name="SubChoreography">
  ..
```

```
</choreography>
..
<choreography ..>
  <parallel>
    <perform choreographyName="SubChoreography">
      ..
    </perform>
    <perform choreographyName="SubChoreography">
      ..
    </perform>
    <perform choreographyName="SubChoreography">
      ..
    </perform>
  </parallel>
</choreography>
```

The case where the number of instances is not known at design-time (***MI with a priori runtime knowledge*** and ***MI with no a priori runtime knowledge***) cannot directly be expressed in WS-CDL. As a workaround, blocking work units and variables can be used: A sub-choreography is defined and performed several times within a repeated work unit. If the `<perform>` action is non-blocking, the sub-choreography instances are activated in parallel. In the case of no a priori knowledge the repeated work unit is also guarded so instances can be activated at a later point in time. A counter is used to record how many instances have already completed. A blocking work unit is activated as soon as the counter has reached a certain value.

Listing 12: Workaround for MI with a priori runtime knowledge

```
<choreography name="chor1">
  <sequence>
    activity
    <assign><copy><!-- increase counter -->
        <source expression="cdl:getVariable('counter') + 1" />
        <target variable="cdl:getVariable('counter')" />
    </copy></assign>
  <sequence>
</choreography>..
  <sequence>
    <assign><copy><!-- initialize counter -->
      <source expression="0" />
      <target variable="cdl:getVariable('counter')" />
    </copy></assign>
    <workunit .. repeat=".."><!-- spawn off multiple instances -->
      <perform choreographyName="chor1" block="false" />
    </workunit>
    <workunit guard="cdl:getVariable('counter')==.." block="true" ..>
      ...
    </workunit>
 </sequence>
```

3.5 State-based Patterns

A ***Deferred Choice*** is a point in the process where one of several branches is chosen. In contrast to the XOR-split, the choice is not made explicitly (e.g. based on data or a decision) but several alternatives are offered to the environment. This means that once the environment activates one of the branches the other alternative branches are withdrawn. It is important to note that the choice is delayed until the processing in one of the alternative branches is actually started. Solution: `<choice>` structures implement both *Exclusive Choice* and *Deferred Choice*. If no guarded work units are contained in a choice structure the decision criteria is hidden and *Deferred Choice* behavior applies.

Listing 13: Deferred Choice

```
<choice>
  interaction1
  interaction2
</choice>
```

In the case of ***Interleaved Parallel Routing*** a set of activities is executed in an arbitrary order: Each activity in the set is executed, the order is decided at run-time, and no two activities are executed at the same moment (i.e. no two activities are active for the same workflow instance at the same time). There is no support for Interleaved Parallel Routing in WS-CDL. A possible workaround could be to introduce a variable representing a mutex. Blocking work units are used the guard condition of which evaluates to true if the mutex is available (i.e. the variable a certain value). Then as a first activity within each work unit the mutex is made unavailable (i.e. another value is assigned). However, this is only possible if the first activity is performed before another work unit is activated. This might depend on the underlying implementation.

Milestone: The enabling of an activity depends on the case being in a specified state, i.e. the activity is only enabled if a certain milestone has been reached which did not expire yet. Solution: In WS-CDL state can be represented using variables ("state capturing variables"). Work units can be activated if a certain guard condition evaluates to true. Milestones are then expressed by non-blocking work units.

Listing 14: Milestone

```
<workunit guard="cdl:getVariable('PurchaseOrderAcknowledged','','',
  'Buyer') == true)" block="false" ..>
    ...
</workunit>
```

3.6 Cancellation patterns

In the case of ***Cancel Activity*** an enabled activity is disabled, i.e. a thread waiting for the execution of an activity is removed. Solution: In WS-CDL exceptions can be caused which lead to disabling activities within the scope of the nearest exception handler. Thus direct support.

Listing 15: Cancel Activity

```
<choreography ..>
  ..
  <parallel>
    <interaction ..>
      ..
      <exchange .. action="respond">
       <send .. causeException=".."/>
       <receive .. causeException=".."/>
     </exchange>
    </interaction>
    <interaction ..>
      ..
    </interaction>
  </parallel>
  <exceptionBlock ..>
    ..
  </exceptionBlock>
</choreography>
```

Cancel Case is directly supported, too. A case, i.e. workflow instance, is removed completely (i.e., even if parts of the process are instantiated multiple times, all descendants are removed). This can be achieved through exceptions that are handled in the root choreography.

4 Service Interaction Pattern support

The Service Interactions Patterns were introduced by Barros et al. in [BDtH05]. They present common interaction scenarios between two or more parties and can be used to assess choreography languages. Although [BDtH05] also contains hints about how different languages implement individual patterns no complete assessment of a standard has been carried out so far using this set of patterns.

4.1 Single-transmission bilateral interaction patterns

All three patterns ***Send***, ***Receive*** and ***Send/receive*** are directly supported in WS-CDL. The structure type `<interaction>` allows to define message exchanges between two services. The `action` attribute of a channel type specifies whether the message exchange is of type `request-only`, `response-only` or `request-response`. Binding a particular participant for an interaction is realized through assigning channel instances to variables. At any point in a choreography a new instance can be assigned. Therefore, design-time and runtime binding and even runtime re-binding of participants can be expressed.

4.2 Single-transmission multilateral interaction patterns

The ***Racing incoming messages*** pattern is similar to the Workflow Pattern *Deferred Choice*: A party expects to receive one among a set of messages. It can be expressed using a `<choice>` containing interactions with the same recipient.

In the case of the ***One-to-many send*** pattern a party sends messages to several parties. If the number of recipients is known at design-time, interactions can be placed in a `<parallel>` structure. This can represent both design-time and runtime binding of recipients of a specified role. However, if the number of recipients is not known at design-time we have the same problem like in the *MI with a priori runtime knowledge* pattern. As a workaround the interactions could be serialized in a repeated work unit. Or an interaction is placed in a sub-choreography which is activated several times using non-blocking `<perform>` actions. A variable is used for bookkeeping how many sends have already completed. Since WS-CDL only directly supports the case where the number of recipients is known at design-time, we opt for partial support for this pattern.

The ***One-from-many receive*** pattern describes that a party receives a number of logically related messages that arise from autonomous events occurring at different parties. The arrival of messages needs to be timely so that they can be correlated as a single logical request. Solution: This can be expressed using a repeated work unit containing a single interaction.

Listing 16: One-from-many receive

```
<workunit .. repeat="..">
  <interaction .. channelVariable="printer">
    ..
  </interaction>
</parallel>
```

The ***One-to-many send/receive*** is very similar to *One-to-many send*: A party sends a request to several other parties. Responses are expected within a given timeframe. The interaction may complete successfully or not depending on the set of responses gathered. Solution: The timeframe aspect is directly supported through the `<timeout>` structure in WS-CDL. Successful vs. unsuccessful completion is directly supported through exception mechanisms. However, we have the same problem with an unknown number of participants at design-time like it was the case for *One-to-many-send.* Hence, there is only partial support for this interaction pattern, too.

4.3 Multi-transmission interaction patterns

In the case of ***Multi-responses*** a party X sends a request to another party Y. Subsequently, X receives any number of responses from Y until no further responses are required. Solution: This pattern is directly supported through a repeated work unit containing the response interaction.

Contingent requests: A party X makes a request to another party Y. If X does not receive a response within a certain timeframe, X alternatively sends a request to another party Z, and so on. Responses from previous requests might be still considered or discarded. Solution: The limited timeframe can be specified through the `<timeout>` structure. A work unit is activated several times until a response arrives before the timeout occurs or the list of potential recipients has been reached. For every iteration a new channel instance is assigned to the channel variable. The selection of the next participant would be hidden. However, this only covers the cases where responses of previous requests are discarded. The case where other responses are still considered requires a parallel execution of the request-response-interactions. Blocking work units could be used to ensure that interactions are only activated a certain time after the previous request has been activated. Nevertheless, this only works for the case where the number of recipients is known at design-time. Since the discard-case is directly supported we still conclude that there is partial support for the pattern.

Listing 17: Contingent requests

```
<choreography name="FindPartyChoreography">
  <interaction ..>
    ..
    <timeout time-to-complete=".."/>
  </interaction>
  <exceptionBlock name="handleTimeoutException">
    <noAction/>
  </exceptionBlock>
</choreography>
..
<workunit repeat="cdl:hasExceptionOccurred('TimeoutException',
    '','','Role1')" block="true" ..>
  ...
</workunit>
<parallel>
  <perform choreographyName="FindPartyChoreography">
    <!-- bind first participant in the list -->
  </perform>
</parallel>
```

In the case of ***Atomic multicast notification*** a party sends notifications to several parties such that a certain number of parties are required to accept the notification within a certain timeframe. For example, all parties or just one party are required to accept the notification. In general, the constraint for successful notification applies over a range between a minimum and maximum number. There is no direct support for this pattern in WS-CDL. As a workaround we could resort to blocking work units and variables: A parallel structure contains several perform actions as well as two blocking work units. In the sequences the actual interactions take place. Depending on the outcome of the interaction a counter is increased. As soon as the counter reaches a certain value or a global timeout occurs, one of the blocking work units is activated.

Listing 18: Workaround for Atomic multicast notification

```
<choreography name="NotifyChoreography">
  <sequence>
    <interaction ..>
      ..
      <timeout time-to-complete=".."/>
    </interaction>
    <assign roleType="Role1">
      <copy name="increase counter">
        <source expression="cdl:getVariable('counter','','','Role1')+1" />
        <target variable="cdl:getVariable('counter','','','Role1')" />
      </copy>
    </assign>
  </sequence>
  <exceptionBlock name="handleTimeoutException">
    <noAction/>
  </exceptionBlock>
</choreography>
..
<sequence>
  <assign roleType="Role1">
    <copy name="get current time">
      <source expression="cdl:getCurrentTime('Role1')" />
      <target variable="cdl:getVariable('startTime','','','Role1')" />
    </copy>
    <copy name="init counter">
      <source expression="0" />
      <target variable="cdl:getVariable('counter','','','Role1')" />
    </copy>
  </assign>
  <parallel>
    <perform choreographyName="NotifyChoreography">
    </perform>
    <perform choreographyName="NotifyChoreography">
    </perform>
    <perform choreographyName="NotifyChoreography">
    </perform>
    ..
    <!-- work unit for success -->
    <workunit .. guard="cdl:getVariable('counter','','','Role1') > 4">
      ..
    </workunit>
    <!-- work unit for time out -->
    <workunit .. guard="cdl:getCurrentTime('Role1') >
        cdl:getVariable('startTime','','','Role1') + 1000">
      ..
    </workunit>
  </parallel>
</sequence>
```

4.4 Routing patterns

Request with referral: Party A sends a request to party B indicating that any follow-up response should be sent to a number of other parties (P1, P2, ..., Pn) depending on the evaluation of certain conditions. Solution: WS-CDL directly supports channel passing. An `<exchange>` structure within an `<interaction>` structure can contain a specification of the corresponding `channelType`. Although the follow-up responses might need to be sent to a number of participants that is only known at runtime (which would not be supported by WS-CDL), we argue that the notion of channel passing is at the very core of the pattern. So we conclude that there is direct support.

Listing 19: Request with referral

```
<channelType name="Channel1">..</channelType>
<interaction>
  <exchange channelType="Channel1">..</exchange>..
</interaction>
```

Relayed request: Party A makes a request to party B which delegates the request to other parties (P1, ..., Pn). Parties P1, ..., Pn then continue interactions with party A while party B observes a “view” of the interactions including faults. Solution: All responses are encoded in one-way interactions. Using a `<parallel>` structure responses are sent to A and to B.

Listing 20: Relayed request

```
<sequence>
  <interaction ..>
    ..
      <participate relationshipType="A-B" fromRole="A" toRole="B"/>
  </interaction>
  <parallel>
    <interaction ..>
      ..
      <participate relationshipType="A-P" fromRole="P" toRole="A"/>
    </interaction>
    <interaction ..>
      ..
      <participate relationshipType="B-P" fromRole="P" toRole="B"/>
    </interaction>
  </parallel>
</sequence>
```

Dynamic routing: A request is required to be routed to several parties based on a routing condition. The routing order is flexible and more than one party can be activated to receive a request. When the parties that were issued the request have completed, the next set of parties are passed the request. Routing can be subject to dynamic conditions based on data contained in the original request or obtained in one of the “intermediate steps”. Since the pattern description is very unprecise it is hard to judge whether or not this pattern is supported. Dynamic conditions in the sense that a participant can overwrite e.g. repetition or guard conditions at runtime is not supported. Static routing orders can easily be expressed using `<sequence>` and `<parallel>` structures. Dynamic routing orders in the sense that a participant can delete interactions or insert new interactions into the choreography at runtime is not supported. We skip this pattern in the assessment.

5 Conclusion

This paper has discussed the relationship between WS-CDL and π-calculus. It turns out that π-calculus and WS-CDL share some elements, but that a number of high-level constructs are provided in WS-CDL that result in more direct pattern support.

Workflow Patterns	WS-CDL	WSCI	BPEL
1. Sequence	+	+	+
2. Parallel Split	+	+	+
3. Synchronization	+	+	+
4. Exclusive Choice	+	+	+
5. Simple Merge	+	+	+
6. Multiple Choice	+	–	+
7. Synchronizing Merge	+	–	+
8. Multiple Merge	+/–	+/–	–
9. Discriminator	–	–	–
10. Arbitrary Cycles	–	–	–
11. Implicit Termination	+	+	+
12. MI without synchronization	+	+	+
13. MI with a priori design time knowledge	+	+	+
14. MI with a priori runtime knowledge	–	–	–
15. MI with no a priori runtime knowledge	–	–	–
16. Deferred Choice	+	+	+
17. Interleaved Parallel Routing	–	–	+/–
18. Milestone	+	–	–
19. Cancel Activity	+	+	+
20. Cancel Case	+	+	+
Service Interaction Patterns	WS-CDL		
1. Send	+		
2. Receive	+		
3. Send/receive	+		
4. Racing incoming messages	+		
5. One-to-many send	+/–		
6. One-from-many receive	+		
7. One-to-many send/receive	+/–		
8. Multi-responses	+		
9. Contingent requests	+/–		
10. Atomic multicast notification	–		
11. Request with referral	+		
12. Relayed request	+		

Table 1: Pattern support in WS-CDL

Table 1 summarizes which Workflow Patterns and Service Interaction Patterns are supported in WS-CDL. In analogy to the mentioned assessments of other process modeling languages we assign a "+" for direct support of a pattern, "+/–" for partial support and

"–" for lack of support. The table shows that WS-CDL supports more patterns than its predecessor WSCI. However, we propose introducing a construct similar to `<forEach>` in BPEL 2.0. This would lead to direct support of *Multiple Instances with a priori runtime knowledge* as well as the three Service Interaction Patterns that are currently only partially supported. At the moment variable assignments and blocking work units have to be resorted to as workarounds. This is problematic since mappings of blocking work units to implementation languages such as BPEL still remain unclear.

We stated that WS-CDL is not suited for early choreography design stages and we called for a standardized graphical notation, e.g. an extension to UML2.0 Activity Diagrams. WS-CDL could then be used as intermediary language introducing web-service-specific definitions.

Acknowledgments. The second author is funded in part by SAP. Many thanks to Marlon Dumas for his valuable feedback.

References

[A$^+$02] Assaf Arkin et al. Web Service Choreography Interface (WSCI) 1.0. Technical report, Aug 2002. `http://www.w3.org/TR/2002/NOTE-wsci-20020808`.

[BDtH05] Alistair Barros, Marlon Dumas, and Arthur ter Hofstede. Service Interaction Patterns. In *Proceedings 3rd International Conference on Business Process Management (BPM 2005)*, pages 302–318, Nancy, France, 2005. Springer Verlag.

[bpm06] Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification. Technical report, Object Management Group (OMG), February 2006. `http://www.bpmn.org/`.

[DPW06] Gero Decker, Frank Puhlmann, and Mathias Weske. Formalizing Service Interactions. In *Proceedings 4th International Conference on Business Process Management (BPM 2006)*, Vienna, Austria, Sept 2006. Springer LNCS.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–40, 1992.

[PW05] Frank Puhlmann and Mathias Weske. Using the pi-Calculus for Formalizing Workflow Patterns. In *Proceedings 3rd International Conference on Business Process Management (BPM 2005)*, pages 153–168, Nancy, France, 2005. Springer Verlag.

[RvdAtHW06] N. Russell, Wil M.P. van der Aalst, Arthur ter Hofstede, and Petia Wohed. On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. In *Proceedings 3rd Asia-Pacific Conference on Conceptual Modelling (APCCM 2006)*, volume 53 of *CRPIT*, pages 95–104, Hobart, Australia, 2006.

[vdADtHW02] Wil M.P. van der Aalst, Marlon Dumas, Arthur H.M. ter Hofstede, and Petia Wohed. Pattern-Based Analysis of BPML (and WSCI). QUT Technical report FIT-TR-2002-05, Queensland University of Technology, Brisbane, Australia, 2002.

[vdAtHKB03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[WvdAD$^+$06] Petia Wohed, Wil M.P. van der Aalst, Marlon Dumas, Arthur ter Hofstede, and N. Russell. On the Suitability of BPMN for Business Process Modelling. In *Proceedings 4th International Conference on Business Process Management (BPM 2006)*, LNCS, Vienna, Austria, 2006. Springer Verlag.

[WvdADtH03] Petia Wohed, Wil M.P. van der Aalst, Marlon Dumas, and Arthur ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In *Proceedings 22nd International Conference on Conceptual Modeling (ER 2003)*, volume 2813 of *LNCS*, pages 200–215. Springer Verlag, 2003.