# Making BPEL Flexible – Adapting in the Context of Coordination Constraints Using WS-BPEL

Yunzhou Wu *and* Prashant Doshi
LSDIS Lab, Dept. of Computer Science
University of Georgia
Athens, GA 30602
wuyzh@uga.edu, pdoshi@cs.uga.edu

## Abstract

*While WS-BPEL is emerging as the prominent language for modeling executable business processes, it provides limited support for designing flexible processes. An important need of adaptive processes is for concurrent activities in the process to respect* coordination *constraints. These require that concurrent activities coordinate their behaviors in response to events otherwise the process may become inconsistent. We show how the constraints that necessitate coordination may be represented in WS-BPEL, and use generalized adaptation and constraint enforcement models to provide a way to transform the traditional BPEL process to an adaptive one. The final outcome is an executable WS-BPEL process without extensions capable of executing on standard BPEL implementations and able to adapt to events while respecting coordination constraints.*

## 1 Introduction

Web service business process execution language (WS-BPEL; [4]) is emerging as the language of choice for modeling executable processes. This is because WS-BPEL provides an appropriate set of constructs to design a process in an intuitive way. One of these is the ability to synchronize concurrent flows within a process using synchronization constraints. While these constraints are indeed a critical feature of concurrent process flows, additional types of constraints may also exist between them. For example, consider a trip planning process that executes the activities of booking an airline ticket and a hotel concurrently to improve its efficiency. Of course, the date of arrival in the airline ticket must coincide with the date of checking in at the hotel. In the event that the reserved airline ticket becomes unavailable, say due to overbooking, a new reservation with a different date must be *coordinated* with a new hotel booking for that date.

In the above example, the constraint that the airline and hotel booking dates must coincide require that actions in response to certain events be coordinated. We label such types of constraints between concurrent activities as *coordination constraints*. These constraints assume significance as processes become flexible and seek to adapt to events to preserve their optimality. Although WS-BPEL is now in its second version and business processes face a growing need to be agile, WS-BPEL continues to natively support only the synchronization constraint between concurrent flows.

In this paper, we show how coordination constraints may be represented in WS-BPEL by utilizing its extensibility constructs. In particular, we view constraints as rules, and use the syntax of the semantic Web rule language (SWRL) to specify the constraints and embed references to the SWRL based constraints within WS-BPEL. Activities that must be coordinated are identified to be a part of the appropriate coordination constraint(s). Previous research [9, 10] has shown how concurrent activities participating in coordination constraints may adapt to events optimally while respecting the constraints. From these approaches we deduce a generalized model of adaptation in conjunction with a constraint enforcement mechanism. WS-BPEL process designers may utilize the generic models for automatically adapting the process while respecting constraints.

Existing implementations of WS-BPEL are not required to support the extensions made to a WS-BPEL document. Therefore, we show how we may transform a process extended with coordination constraints and adaptation models into a core WS-BPEL process that does not utilize the extensibility constructs and is capable of executing on standard WS-BPEL implemen-

tations. Specifically, we solve the combined adaptation and constraint enforcement models to obtain a *policy* that recommends adaptive actions while respecting the constraints. We empirically evaluate the performance of the adaptive WS-BPEL process in comparison to a traditional process that does not adapt and provide favorable results in support.

## 2 Motivating Scenarios

**Trip Planning** Consider a Web services based process for organizing a trip that consists of concurrently booking an airline ticket, and hotel and rental car at the destination. We consider the event where a reservation of the airline would subsequently need modification because the ticket becomes unavailable, perhaps due to overbooking. In response, the trip planner may choose to change the date of departure, change the destination airport to another one in the city, change the company, or simply wait in the hope that a vacancy may arise. Note that a change in the departure date will require *coordinated re-bookings* of all three – airline, hotel and the rental car. A change in the destination airport will need modification in the booking of the airline and the rental car concurrently. We illustrate the trip planning process and the coordination constraints in Fig. 1.

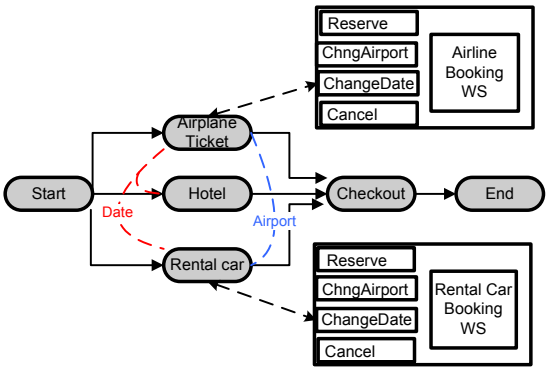


**Figure 1.** The trip planning process with coordination constraints. A change in the date of arrival requires rebooking for all three activities, while a change in destination airport requires modifications to two activities.

**Supply Chain for Computer Assembly** Another scenario to consider is the supply chain of a computer manufacturer (e.g. Dell) which operates on minimal inventory, and therefore incurs significant costs if its order is delayed. The manufacturer typically orders in bulk different computer parts from different suppliers concurrently. Since the parts must be assembled into a single computer, they must be *compatible* with each other. For example, RAM must interoperate with the motherboard. If the delivery of RAM is delayed and the manufacturer chooses to change the RAM supplier, the supplier of motherboards may also need to be changed concurrently to preserve the compatibility constraint.

## 3 Coordination Constraints in BPEL

We utilize the trip planning example to illustrate our approach. We first note that WS-BPEL supports extensibility by allowing namespace-qualified attributes to appear in any WS-BPEL element and by allowing elements from other namespaces to appear within WS-BPEL defined elements. The extensions may be mandatory or optional. In case of mandatory extensions, WS-BPEL implementations that are unable to understand them must reject the entire process. For optional extensions, WS-BPEL implementations may choose to ignore them.

If constraints that induce coordination exist between concurrent activities, we specify them within the <flow> activity in a WS-BPEL document. Analogous to the specification of synchronization constraints, we utilize the <links> construct to list the different coordination constraints. Within this construct, we propose a new namespace-qualified element <ccLink> to declare each unique coordination constraint. We illustrate this specification using a snippet in Fig. 2.

```
<process name="tripPlannerProcess"
  targetns="http://example.com/ws-bp/trip"
  xmlns:tpns="http://reservation.org/wsdl/air+hotel+car">
...
<variables>
     <variable name="ticketDate"
               messageType="tpns:reservMsg">
     <variable name="hotelDate"
               messageType="tpns:reservMsg">
     <variable name="rentalCarDate"
               messageType="tpns:reservMsg">
</variables>
...
<flow>
   <links>
     <cc_ns:ccLink name="date" ref="uri:date.swrl">
     <cc_ns:ccLink name="airport" ref="uri:airport.swrl">
   </links>
...
</flow>
```

**Figure 2.** Specification of coordination constraints *date* and *airport* using new ccLink element within ⟨flow⟩.

Notice that <ccLink> in Fig. 2 is qualified with

the namespace cc_ns that is defined in a XML schema which would be imported in the header of the WS-BPEL document. The *date* constraint is declared as a rule using the syntax of the semantic Web rule language (SWRL) in the file located at uri:date.swrl where uri gives the location of the file. The rule simply states that the variables ticketDate, hotelDate and rentalCarDate should all have equal values:
$true \Rightarrow equals(ticketDate, hotelDate, rentalCarDate)$
Because the antecedent is true, the consequent must be true under any interpretation.

# 4 Coordinated Adaptation

In Section 3, we showed how a type of constraint – which requires certain concurrent operations to be coordinated – could be generally represented in a WS-BPEL process using its extensibility constructs. As processes become agile, they seek to adapt to exogenous events in optimal ways. For example, if a reserved ticket becomes unavailable, a naive response is to simply wait until it becomes available again because of cancellations, although the wait could be expensive. Instead, an optimal response may be to change the date of departure to a low traffic day, if schedule permits.

However, designing adaptive processes is challenging due to multiple reasons: $(i)$ Processes specified using WS-BPEL tend to be inflexible, utilizing a *fixed* flow of activities that must be carried out in the specified order. This is in part because WS-BPEL is not intuitively amenable to describing a dynamic process. $(ii)$ Adaptation becomes complicated in the presence of constraints between activities that must be respected. An example of this is the coordination constraint between concurrent activities described previously.

## 4.1 Background: Adaptation Models

As mentioned, the decision of how to react to external events becomes difficult in the presence of inviolable constraints between process activities. A central *process manager* that has global knowledge of the entire process including the states of the individual activities is capable of adapting the process optimally to external events while satisfying the constraints. The M-MDP approach in [9], describes a *centralized* way of doing this that guarantees global optimality of the adaptation, but does not scale efficiently to large processes.

One way to scale reasonably well to processes with several activities is to associate a *service manager* with each activity who individually decides how to adapt to the external events. We show example *decision models* of the managers responsible for booking the airline ticket and hotel for the trip planner in Fig. 3($a$).

The MDP-CoM approach [9] formalizes each manager's behavior as an individual decision model, and ensures coordination between the concurrent activities involved in a constraint using a simple coordination enforcement mechanism (CoM). The mechanism is a finite state machine (FSM), whose state is perfectly observable to all the managers. The FSM is defined to have two general states: an *uncoordinated* (U) state and a *coordinated* (C) state. The state of the FSM signifies whether the managers must coordinate. Initially, the actions of the managers are uncoordinated – each one is free to follow the optimal action conditioned on its local state. If a manager decides to perform an action that must be coordinated, it signals its *intent* first. When any service manager signals its intent to perform a coordinating action, the FSM transitions to the coordinated state. In this state, all the managers are required to perform their respective coordinating actions. Their actions will also reset the FSM back to the uncoordinated state. Notice that when any of the manager signals its intent to perform a coordinating action, each manager must follow suit, no matter whether it's an optimal decision for the other manager. This is precisely the source of the loss in optimality for the decentralized approach.

## 4.2 Generalized Adaptation Model

Given the example decision models used by the service managers in the previous approach outlined in Section 4.1, we deduce a general-purpose decision making model for adapting to multiple events. Our objective is to provide the process designer with a general template model that could be used by the individual activities in the process to adapt optimally to external events.

Although the decision models in Figs. 3($a$) are for two different activities, their structural similarity imply that the models could be generalized. Specifically, nodes $S2$ and $S3$ represent states of an activity given that the two events (ticket or room is unavailable and ticket or room is confirmed) occurred. Nodes $S4$ and $S5$ represent states of the activity after the adaptive actions (change date or change hotel) were performed. Although the decision models distinguished between states $S4$ and $S5$, notice that only a single identical action is performed from both states in the two models, transitioning to $S1$. Thus, we may combine the states $S4$, $S5$ and $S1$ into a single state by augmenting the adaptive actions to include rebooking as well if needed, without loss of optimality.

We show the generalized decision model for two external events in Fig. 3($b$). Within the model, the oc-

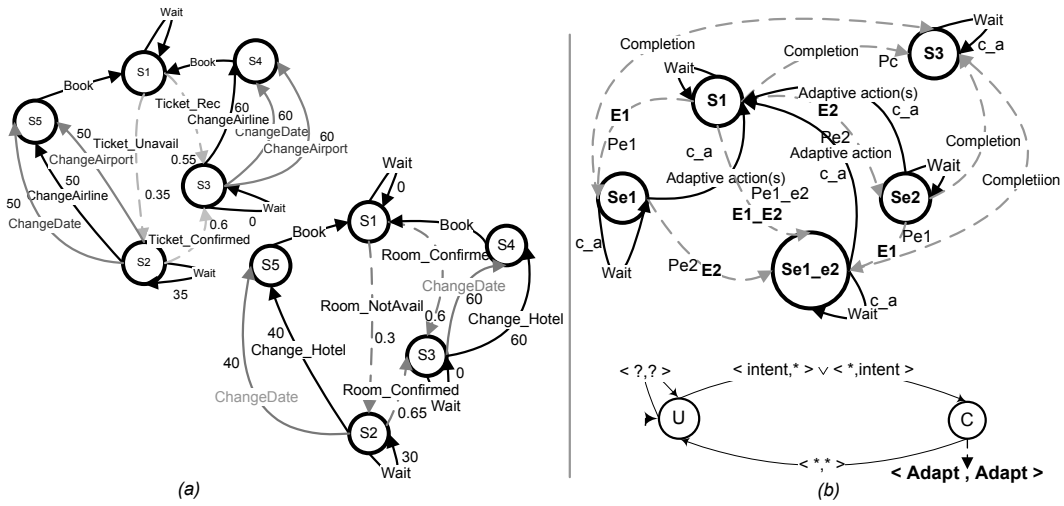


**Figure 3.** ($a$) Simplified decision models for booking an airline ticket (left), and hotel (right), illustrating actions, events, probabilities and costs. Actions are invocations of service operations. Transitions due to actions are depicted using solid lines and are deterministic. Exogenous events are shown dashed. Decimals accompanying event arcs denote example probabilities of occurrence of events conditioned on states. Numbers beside action arcs show example costs of performing actions conditioned on the state. ($b$) Generalized decision model for adapting to external events (top). CoM for coordinating concurrent activities (bottom). Transitions (solid arrows) are due to joint actions of the activities. '*' indicates any action, '?' indicates remaining actions. Dashed arrow gives the action rule for each activity when FSM is in that state.

currence of each external event (say $E1$) with a probability ($Pe1$) leads to a state which signifies that the event occurred ($Se1$). Some states signify that multiple events have occurred. If the favorable event, labeled as 'completion' occurs the state of the activity transitions to $S3$. In addition to performing a single or composite 'adaptive action' in response to event(s), we also have the option of a no-op (labeled as 'wait') which signifies that we wait, ignoring the event until 'completion' may occur. In general, for $n$ external events excluding 'completion', the state space size will be $C_1^n + C_2^n + ... + C_n^n + 2$, where $C_m^n = \frac{n!}{(n-m)!m!}$. We point out that each adaptive action has an associated cost, $c\_a$, that could depend on the state at which the action is performed.

To ensure that adaptive actions are coordinated among concurrent activities, we utilize the simple CoM that appeared in [9] and briefly described previously in Section 4.1. We show this CoM in Fig. 3($b$). Although more sophisticated CoMs are possible (see for example [10]), CoM in Fig. 3($b$) may be implemented easily using a single variable that denotes its state.

## 5 Embedding Adaptation Models

We observe that the event handling ability available in WS-BPEL provides a simple way to adapt by performing predefined actions if certain events occur. For example, a trip planning process may handle the event that the airline ticket becomes unavailable by always changing the date of the travel. However, we may not always choose to modify the ticket date in response to the event; instead we may decide to change the arrival airport if another airport exists at the destination. Furthermore, the adaptive action needs to be coordinated with analogous actions for other concurrent activities.

We utilize WS-BPEL's extensibility and introduce a new namespace-qualified element, <cc_ns:alt_activity>, to provide a general way to specify a choice of actions that could be performed on receiving an event. The namespace cc_ns would be defined in the header of the WS-BPEL process. We illustrate this in Fig. 4 by specifying the alternative operation ChangeAirport that could also be performed if the ticket is unavailable. What remains now is to determine which action should be selected in response to the event. As we mentioned previously, this question may not be trivially answered, more so due to the presence of coordination constraints.

In this context, our approach is to allow the process designer to reference the generalized adaptation model and the CoM described previously in Section 4.2. As demonstrated in [9], these models are sufficiently sophisticated to decide which adaptive action to perform while respecting the coordination constraint. We also

```
<eventHandlers>
  <onEvent partnerLink="AirlineWSLnk"
    operation="ReservationUpdate"
    variable="ticketUnavail">
    <scope>
      ...
      <invoke name="invoke" partner="AirlineWS"
        portType="tpns:ReservationPT"
        operation="ChangeDate"
        inputVariable="ticketDate">
        <cc_ns:ccLinkInst name="date" />
        <cc_ns:adaptModel ref="uri:genericMdl1"/>
        <cc_ns:comModel ref="uri:CoM1"/>
      </invoke>
      <cc_ns: alt_activity>
        <invoke name="invoke" partner="AirlineWS"
          portType="tpns:ReservationPT"
          operation="ChangeAirport"
          inputVariable="ticketAirport">
          <cc_ns:ccLinkInst name="airport" />
          <cc_ns:adaptModel ref="uri:genericMdl1"/>
          <cc_ns:comModel ref="uri:CoM1"/>
        </invoke>
      </cc_ns:alt_activity>
    </scope>
  </onEvent>
...
</eventHandlers>
<flow>
  <links> ... </links>
  <sequence>
    <invoke name="invoke" partner="AirlineWS"
        portType="tpns:ReservationPT"
        operation="Reserve"
        inputVariable="ticketDate"/>
    <receive name="receive" partner="AirlineWS"
        portType="tpns:BookingPT"
        operation="Completion"
        variable="confirmation"/>
  <sequence>
...
<flow>
```

**Figure 4.** We utilize WS-BPEL's extensibility to specify alternative actions that could be performed in response to an event. To decide on the adaptive actions in the context of constraints, the designer may utilize the generalized adaptation model and CoM. Events for other activities such as hotel reservation are handled using additional <onEvent> sections.

require that the designer indicate which of the concurrent activities and its operations are part of some coordination constraint previously defined in Fig. 2. This is analogous to specifying if an activity is the source or target of a synchronization constraint.

Of course, as the generalized adaptation model is a template, we need to instantiate it for the process. In doing so, we do not assume any knowledge about the operation of the adaptation model and CoM from the process designer. Notice that the generalized model in Fig. 3($b$) in part requires that we specify the external events and the corresponding adaptive action choices in response to the events and their combinations. Fortunately, this information could easily be provided in the <eventHandlers> section of the WS-BPEL process as partially shown in Fig. 4. Here, each <onEvent> element specifies an event, say $E1$, and the activities within the nested <scope> are the appropriate adaptive action(s) that could be performed from the corresponding state, $Se1$. The fact that the *ChangeDate* operation within the <invoke> activity induces coordination is indicated by referring to the *date* constraint defined in Fig. 2. The generalized adaptation model and CoM are also referenced using the new namespace-qualified elements as shown in Fig. 4. By doing so, the designer indicates that the WS-BPEL process should be adaptive and it should utilize the adaptation model and CoM to decide the adaptive actions. The blocking <receive...> activity terminates when the completion event is received leading to state $S3$.

The remaining parameters of the decision model such as the cost of performing an adaptive operation, for example changing the date of an existing reservation, and the probability of events, for example the maximum percentage of reservations that could be canceled due to say overbooking, are often specified in the agreements (using say, WS-Agreement [2]) between the process and the partner Web services.

## 6 Transforming the Extended BPEL

Although we have extended the WS-BPEL process using the extensibility constructs, the process may still be executed in a standard BPEL implementation by instructing it to ignore the extensions. If our extensions are ignored, the original (non-adaptive) behavior of the process is retained. Consequently, a designer may extend any existing BPEL process in order to transform it into a process that is capable of adapting to external events while respecting the constraints, if needed.

We outline the steps needed to transform an extended WS-BPEL process into a fully standards-compatible adaptive BPEL process below:

• Given the BPEL process extended with elements and attributes describing the coordination constraint(s), events, references to the generalized decision and CoM models and the accompanying WS-agreements, we parse these documents using a BPEL parser such as the

open source ActiveBPEL library [1], a custom SWRL parser for the constraint(s), and a general XML parser for the WS-Agreement documents.

• We load the generalized decision models and CoMs referenced by the BPEL process and instantiate them. Specifically, we utilize the information about the events and the corresponding action choices to establish the structure of the decision model. We utilize the costs of performing the different operations and probabilities of events obtained from the service agreements to parameterize the models.

• We combine each decision model specific to an activity with the CoM and formalize the combined model as a Markov decision process (MDP), as shown in [9]. We may utilize standard solution techniques such as value iteration [7] to solve it.

• Solution of the MDP is a policy which maps each combined state (state of the decision model and the CoM) to an action that is optimal at that state. This action is one of the adaptive action choices available to perform. We transform the policy into WS-BPEL code for constructing the final WS-BPEL process.

• We generate the adaptive WS-BPEL process using a custom serializing library based on the open source ActiveBPEL library. As shown in Figs. 6 and 7, we utilize the policies to guide the adaptive behavior of each concurrent activity in the process. Coordination, if needed, is ensured by signaling an internal exception using <throw> <catch> statements.

• Final WS-BPEL process may be passed to any BPEL implementation for execution. Note that the process does not contain any extensions and is therefore fully compliant with existing WS-BPEL 2.0 specification. On the occurrence of an event, the process utilizes the policy to guide which adaptive action or a sequence of actions to perform while respecting coordination constraints, if any.

To aid understanding, we describe the transformed BPEL process in three parts. We begin by showing the <flow> construct in Fig. 5. Observe that the invoke and receive activities remain unchanged (see Fig. 4 for comparison). Hence the process blocks until it receives a confirmation message, although it could receive other external events in the meantime. However, before the invocation of the Reserve operation of the Airline Web service, we initialize a variable, pStateA, which signifies the state of the activity. On completing the reservation, we transition the state again to signify completion.

Each activity within the transformed BPEL process uses a policy to guide its actions on the occurrence of an event (Fig. 6). For example, the airline booking activity uses policyA. On receipt of an event such as ticket unavailable, we change the pStateA variable to

```
...
<flow>
  <sequence>
    <assign> <copy>
      <from> <literal> 'S1' </literal> </from>
      <to variable="pStateA" />
    </copy> </assign>
    <invoke name="invoke" partner="AirlineWS"
        portType="tpns:ReservationPT"
        operation="Reserve" inputVariable="ticketDate" />
    <receive name="receive" partner="AirlineWS"
        portType="tpns:BookingPT"
        operation="Completion" variable="confirmation" />
    <assign> <copy>
      <from> <literal> 'S3' </literal> </from>
      <to variable="pStateA" />
    </copy> </assign>
  </sequence>
  ...
<flow>
```

**Figure 5.** 'S1' represents the initial state of the activity while 'S3' the completed state. Other concurrent activities such as reserving the hotel are defined analogously within the <flow> construct.

reflect the corresponding state in the decision model (see Fig. 3($b$)) and if the action prescribed by the policy is a coordination inducing operation, we signal an internal exception using <throw>.

We utilize a *fault handler* to catch the internal exception that is thrown when a policy prescribes a coordination inducing operation, as shown in Fig. 7. The CoM transitions to 'U' and the corresponding coordination inducing operation as obtained from the policy is performed. As all the activities must coordinate, we perform the appropriate operations for all the concurrent activities that are a part of the coordination constraint. Because the fault handler is in a different scope, any blocked <receive...> activity within the <flow> construct is not expected to terminate when a fault is thrown. For multiple constraints, distinct fault handlers are used.

## 7 Performance Evaluation

We implemented the task pipeline outlined in Section 6 using the open source ActiveBPEL library. As we mentioned previously, it takes as input an extended WS-BPEL process and produces a WS-BPEL process capable of adapting to external events.

We evaluate the adaptive performance of the transformed WS-BPEL in the context of trip planning

```
<scope name="eventScope">
  <eventHandlers>
    <onEvent  partnerLink="AirlineWSLnk"
      operation="ReservationUpdate"
      variable="ticketUnavail">
      <scope>
        <assign> <copy>
          <from> <literal> 'S2' </literal> </from>
          <to variable="pStateA"/>
        </copy> <assign>
        <if>
        <cond>$policyA:ele[$pStateA]:ele[$cState]=1</cond>
          <throw faultName="CoM" faultvariable="cState">
            <assign> <copy>
              <from> <literal> 'C' </literal> < /from>
              <to variable="cState"/>
            </copy> </assign>
          </throw> <elseif>
        <cond>$policyA:ele[$pStateA]:ele[$cState]=2</cond>
          <throw faultName="CoM" faultvariable="cState">
            <assign> <copy>
              <from> <literal> 'C' </literal> < /from>
              <to variable="cState"/>
            </copy> </assign>
          </throw> <elseif>
        <cond>$policyA:ele[$pStateA]:ele[$cState]=3</cond>
          <empty> </empty> </if>
      </scope>
    </onEvent>
  ...
  </eventHandlers>
</scope> ...
```

**Figure 6.** On an event, we indicate the new state of the activity (for e.g., 'S2' indicating ticket unavailable) and consult the policy for corresponding action to perform. If the action is a coordination inducing operation, we do not perform it but instead signal an internal exception, which is equivalent to the intent in CoM.

```
<scope name=eventScope>
  <faultHandlers>
    <catch faultName="CoM" faultVariable="cState"
      faultMessageType="tpns:coordMsgType">
      <if>
        <cond>$policyA:ele[$pStateA]:ele[$cState]=1</cond>
          <sequence>
            <assign> <copy>
              <from> <literal> 'U' </literal> </from>
              <to variable="cState" part="symbol"/>
            </copy> </assign>
            <invoke  name="invoke" partner="AirlineWS"
                portType="tpns:ReservationPT"
                operation="ChangeDate"
                inputVariable="ticketDate"/>
          </sequence>
        <cond>$policyA:ele[$pStateA]:ele[$cState]=2</cond>
          <sequence>
            <assign> <copy>
              <from> <literal> 'U' </literal> </from>
              <to variable="cState" part="symbol"/>
            </copy> </assign>
            <invoke  name="invoke" partner="AirlineWS"
                portType="tpns:ReservationPT"
                operation="ChangeAirport"
                inputVariable="ticketAirport"/>
          </sequence> </if>
      <if>
        <cond>$policyH:ele[$pStateH]:ele[$cState]=1</cond>
          <invoke  name="invoke" partner="HotelWS"
              portType="tpns:ReservationPT"
              operation="ChangeDate"
              inputVariable="ticketDate"/>
      <elseif>
      ...
    </catch>
  </faultHandlers>
</scope>
```

**Figure 7.** We use the faultHandlers to recognize the transition of the CoM to the 'C' state. In this case, coordinated operations as prescribed by the corresponding policies (policyA, policyH) are performed. Fault handler is in the same scope as event handler so as to catch any thrown faults. If more coordination constraints exist, we utilize additional <catch> sections.

and supply chain scenarios outlined in Section 2. We compare the average reward obtained (completion reward - cost incurred) when executing a traditional non-adaptive WS-BPEL process (denoted as *originalBPEL*) that chooses to wait, with our transformed adaptive WS-BPEL process (denoted as *adaptiveBPEL*). We varied the probability that items of interest (airline ticket and CPU or RAM) are available given that they were not initially. Between plots, we varied the probability they are unavailable initially. Both processes are run under identical conditions.

We show the plots for the two scenarios in Fig. 8. Each data point in a plot is the average of 100 simulated runs of the process using the ActiveBPEL simulator. For the plots $(a, c)$, the event that the desired item is unavailable or is delayed occurs 1 in 2 times among the 100 runs while it occurs 1 in 10 times for the plots in $(b, d)$. The activities in *adaptiveBPEL* opted to wait for completion if the probability of availability (or received) is high, which explains why the two processes perform identically in the latter stages. This is because the activities believe that it is more likely that they will eventually receive the ticket (or order) once it is unavailable (or delayed). However, if the probabil-

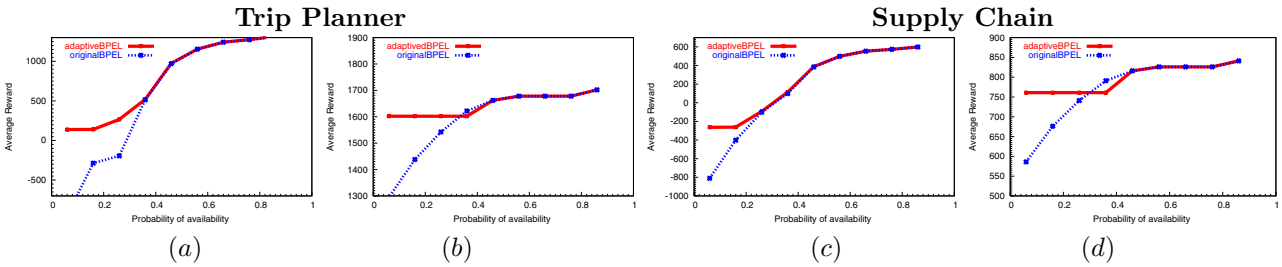


**Figure 8.** Performances of the traditional and adaptive WS-BPEL processes for the two problem domains. For plots $(a, c)$ the probability of the ticket,hotel and car being unavailable initially is 0.5 while it is 0.1 for $(b, d)$. The $x$-axis is the probability of availability (completion) of the items given that they were unavailable (or delayed) initially. Notice that the adaptiveBPEL performs significantly better (larger average rewards) than the traditional one when the probability is low.

ity of availability is low, the *adaptiveBPEL* chooses to change the date (or supplier) in a coordinated manner. In comparison, the *originalBPEL* continues to wait incurring a higher cost and lower reward. Notice that between plots for a scenario, the overlapping portion of the curves is larger when the probability of initial unavailability (or delay) is large. This is because we expect the items of interest to be again unavailable (or delayed) if we change the date (or supplier). Hence, waiting for completion is likely a better action.

## 8 Related Work and Discussion

Although the inability of WS-BPEL to intuitively support flexible processes has been pointed out previously, extensions of WS-BPEL to promote flexibility in processes have been sparse. Karastoyanova et al. [5] extend BPEL to allow for runtime selection of WS instances using a procedure called 'find and bind'. While the extensions provide a way for dynamic binding of Web services, a policy based approach is used to select the instances. However, the policy is assumed to be provided by the process designer. A04BPEL [3] extends WS-BPEL with aspect oriented concepts to support modularity and adaptability. Using dynamic weaving, the in-memory representation of a process is changed to insert new aspects. However, this requires that the process be suspended at different points. Outside the realm of WS-BPEL, Pesic et al. [6] propose constraint based models for processes and utilize the ConDec language for specifying the models. Ad-hoc and evolutionary changes to the process definition at run time are viewed as migration of instances between different constraint models. However, ConDec does not focus on adapting the behavior of activities to external events in the presence of constraints.

Our overriding concern in making WS-BPEL flexible and supporting coordination among concurrent activities was to introduce extensions in a *minimal* and *general* manner. As BPEL implementations may be instructed to ignore the extensions, we showed how the extensions may be utilized to transform a process into an adaptive one that is capable of coordinating operations across multiple concurrent activities in response to external events.

## References


[1] Active-Endpoints. Activebpel open source engine project. http://www.active-endpoints.com/active-bpel-engine-overview.htm, 2006.

[2] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. *WS-Agreement Spec.*, 2005.

[3] A. Charfi, M. Mezini. A04BPEL: An aspect-oriented extension to bpel. In *WWW*, 10:309–344, 2007.

[4] B. T. Committee. WS-BPEL 2.0. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel, 2007.

[5] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. Buchmann. Extending bpel for run-time adaptability. In *EDOC*, pages 15–26, 2005.

[6] M. Pesic, M. Schonenberg, N. Sidorova, and W. van der Aalst. Constraint-based workflow models: change made easy. In *OTM*, pages 77–94, 2007.

[7] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. Wiley-Interscience, 1994.

[8] K. Verma, R. Akkiraju, R. Goodwin, P. Doshi, and J. Lee. On accommodating inter service dependencies in web process flow composition. In *AAAI Spring Symp. on Semantic Web Services*, pages 37–43, 2004.

[9] K. Verma, P. Doshi, K. Gomadam, J. Miller, and A. Sheth. Optimal adaptation in web processes with coordination constraints. In *ICWS*, pages 257–264, 2006.

[10] Y. Wu and P. Doshi. Regret-based decentralized adaptation of web processes with coordination constraints. In *SCC*, pages 262–269, 2007.