Pattern Based Analysis of BPEL4WS

Petia Wohed[1⋆] Wil M.P. van der Aalst[2] Marlon Dumas[3]
Arthur H.M. ter Hofstede[3]

[1] Department of Computer and Systems Sciences
Stockholm University/The Royal Institute of Technology
petia@dsv.su.se
[2] Department of Technology Management
Eindhoven University of Technology
w.m.p.v.d.aalst@tm.tue.nl
[3] Centre for Information Technology Innovation
Queensland University of Technology
{m.dumas, a.terhofstede}@qut.edu.au

Abstract. Web services composition is an emerging paradigm for enabling application integration within and across organisational boundaries. A landscape of languages and techniques for web services composition has emerged and is continuously being enriched with new proposals from different vendors and coalitions. However, little or no effort has been dedicated to systematically evaluating the capabilities and limitations of these languages and techniques. The work reported in this paper is a first step in this direction. It presents an in-depth analysis of the Business Process Execution Language for Web Services (BPEL4WS). The framework used for this analysis is based on a collection of workflow and communication patterns.

1 Introduction

Web Services is a rapidly emerging paradigm for architecting and implementing business collaborations within and across organisational boundaries. In this paradigm, the functionalities provided by business applications are encapsulated within web services: software components described at a semantical level, which can be invoked by application programs or by other services through a stack of Internet standards including HTTP, XML, SOAP, WSDL, and UDDI [6]. Once deployed, web services provided by various organisations can be inter-connected in order to implement business collaborations, leading to *composite web services*.

Business collaborations require long-running interactions driven by an explicit process model [1]. Accordingly, a current trend is to express the logic of a composite web service using a business process modelling language tailored for web services. Recently, many languages have emerged,

⋆ Research conducted while at the Queensland University of Technology.

including WSCI [17], BPML [4], BPEL4WS [7], and BPSS [16], with little effort spent on their evaluation with respect to a common benchmark. Such a comparative evaluation will contribute to establishing their overlap and complementarities, to delimit their capabilities and limitations, and to detect inconsistencies and ambiguities.

As a first step in this direction, this paper reports an in-depth analysis of one of these emerging languages, namely BPEL4WS (Business Process Execution Language for Web Services). It is expected that a similar analysis will be conducted for other alternative languages in the future.

The reported analysis is based on a framework composed of a set of *patterns*: abstracted forms of recurring situations found at various stages of software development [10]. Specifically, the framework brings together a set of *workflow patterns* documented in [3], and a set of *communication patterns* documented in [14].

The workflow patterns (WPs) have been compiled from an analysis of existing workflow languages and they capture typical control flow dependencies encountered in workflow modelling. More than 12 commercial Workflow Management Systems (WFMS) as well as the UML Activity Diagrams, have been evaluated in terms of their support for these patterns [3, 8]. The WPs are arguably suitable for analysing languages for web services composition, since the situations they capture are also relevant in this domain.

The Communication Patterns (CPs) on the other hand, are related to the way in which system modules interact in the context of Enterprise Application Integration (EAI). They are structured according to two dichotomies: synchronous vs. asynchronous, and point-to-point vs. multicast. They are arguably suitable for the analysis of the communication modelling abilities of web services composition languages, given the strong overlap between EAI and web services technologies.

Two other frameworks for analysing and comparing business process modelling languages have been proposed by Rosemann & Green [13] and Söderström et al. [15]. While these two frameworks are motivated by the same problem that motivates this paper, i.e. the continuously increasing number of process modelling languages and the need to understand and compare them, they differ from the pattern-based framework in that they target a different audience namely, IS/IT-managers, business strategists and other business stakeholders involved in business process management. Accordingly, they adopt a higher level of granularity.

The rest of the paper is structured as follows. Section 2 provides an overview of the BPEL4WS language. In sections 3 and 4 the BPEL4WS

language is analyzed using the set of workflow and communication patterns respectively. Finally, section 5 concludes the work.

2 BPEL4WS

BPEL4WS builds on IBM's WSFL (Web Services Flow Language) and Microsoft's XLANG (Web Services for Business Process Design) and combines accordingly the features of a block structured language inherited from XLANG with those for directed graphs originating from WSFL. The language is intended to support the modelling of two types of processes: executable and abstract processes. An *abstract*, (not executable) *process* is a business protocol, specifying the message exchange behaviour between different parties without revealing the internal behaviour for anyone of them. An *executable process*, which is also the focus of this paper, specifies the execution order between a number of *activities* constituting the process, the *partners* involved in the process, the *messages* exchanged between these partners, and the *fault* and *exception handling* specifying the behaviour in cases of errors and exceptions.

The BPEL4WS process itself is a kind of flow-chart, where each element in the process is called an *activity*. An activity is either a primitive or a structured activity. The set of *primitive activities* contains: invoke, invoking an operation on some web service; receive, waiting for a message from an external source; reply, replying to an external source; wait, waiting for some time; assign, copying data from one place to another; throw, indicating errors in the execution; terminate, terminating the entire service instance; and empty, doing nothing.

To enable the presentation of complex structures the following *structured activities* are defined: sequence, for defining an execution order; switch, for conditional routing; while, for looping; pick, for race conditions based on timing or external triggers; flow, for parallel routing; and scope, for grouping activities to be treated by the same fault-handler. Structured activities can be nested and combined in arbitrary ways. Within activities executed in parallel the execution order can further be controlled by the usage of links (sometimes also called control links, or guarded links), which allows the definition of directed graphs. The graphs too can be nested but must be acyclic.

3 The Workflow Patterns in BPEL4WS

Web services composition and workflow management are related in the sense that both are concerned with executable processes. Therefore, much

of the functionality in workflow management systems [2, 9, 12] is also relevant for web services composition languages like BPEL4WS, XLANG, and WSFL. In this section, we consider the 20 workflow patterns presented in [3], and we discuss how and to what extent these patterns can be captured in BPEL4WS. Most of the solutions are presented in a simplified BPEL4WS notation, which is rich enough for capturing the key ideas of the solutions, while at the same time avoiding a detailed coding-oriented representation.

WP1 Sequence An activity in a workflow process is enabled after the completion of another activity in the same process. **Example:** After the activity *order registration* the activity *customer notification* is executed.

Solution, WP1 There are two possible solutions for this pattern in BPEL4WS: one using the operator sequence inherited from XLANG (see Listing 1), and one using the concept of control link inherited from WSFL (see Listing 2). In this case a link needs to be defined first (lines 2 to 4) and then the activity to be executed first is specified as source activity for this link (line 6) while the subsequent activity is specified as target for the link (line 8). All these activities are embedded within a single flow activity.

Listing 1

```
1 <sequence>
2   activityA
3   activityB
4 </sequence>
```

Listing 2

```
1 <flow>
2   <links>
3     <link name="L"/>
4   </links>
5   activityA
6     <source linkName="L"/> ...
7   activityB
8     <target linkName="L"/> ...
9 </flow>
```

WP2 Parallel Split A point in the process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order [5]. **Example:** After activity *new cellphone subscription order* the activity *insert new subscription* in Home Location Registry application and *insert new subscription* in Mobile answer application are executed in parallel.

WP3 Synchronization A point in the process where multiple parallel branches converge into one single thread of control, thus synchronizing

multiple threads [5]. It is an assumption of this pattern that after an incoming branch has been completed, it cannot be completed again while the merge is still waiting for other branches to be completed. Also, it is assumed that the threads to be synchronized belong to the same global process instance (i.e., to the same "case" in workflow terminology). **Example:** Activity *archive* is executed after the completion of both activity *send tickets* and activity *receive payment*. Obviously, the synchronization occurs within a single global process instance: the *send tickets* and *receive payment* must relate to the same client request.

Solutions, WP2 & WP3 The parallel split is realized by defining the activities to be run in parallel as components of an activity of type flow (see Listing 3, lines 2 to 5). If no control link is defined within a flow, the activities within the flow are executed in parallel. Adding an activity after the flow, as for example activity B in line 6, yields the solution to the Synchronization pattern.

Similarly to the solution for WP1, a solution based on control links is also possible for WP2 and WP3 (see Listing 4). In this solution the links L1 and L2 are defined in a flow F. Furthermore, F consists of the activities A1, A2 and B. The sources of L1 and L2 are A1 and A2 respectively (lines 7 and 9) and the target for both links is activity B (lines 12 and 13). To execute B after both A1 and A2 have been completed successfully an AND joinCondition is defined for activity B (line 11).

Listing 3

```
1 <sequence>
2   <flow>
3     activityA1
4     activityA2
5   </flow>
6   activityB
7 </sequence>
```

Listing 4

```
1 <flow name="F">
2   <links>
3     <link name="L1"/>
4     <link name="L2"/>
5   </links>
6   activityA1
7     <source linkName="L1"/>...
8   activityA2
9     <source linkName="L2"/>...
10  activityB
11    joinCondition="L1 AND L2"
12    <target linkName="L1"/>
13    <target linkName="L2"/>...
14 </flow>
```

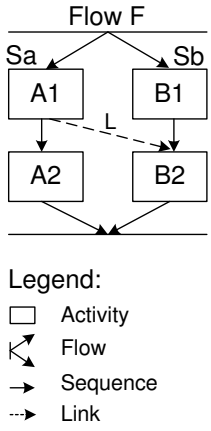
Listings 3 and 4 illustrate the two styles of process modelling supported by BPEL4WS. Listing 3 shows the "XLANG-style" of modelling (i.e., routing through structured activities). Listing 4 shows the "WSFL-

style" of modelling (i.e., using links instead of structured activities). It is also possible to mix both styles by having links crossing the boundaries of structured activities.[4] An example is given in Listing 5, where the sequences Sa and Sb are defined to run in parallel. The definition of a link L (lines 3, 7 and 14) implies that activity B2, following after activity B1 in sequence Sb, can be executed first after activity A1 from sequence Sa have completed its execution. In other words, link L captures an intermediate synchronization point between the two parallel threads Sa and Sb. This inter-thread synchronization cannot be expressed using structured activities only (for a proof see [11]). Figure 1 illustrates the example in graphical form.[5]

Listing 5

```
1 <flow name="F">
2   <links>
3     <link name="L"/>
4   </links>
5   <sequence name="Sa">
6     activityA1
7       <source linkName="L"/>
8     activityA2
9   </sequence>
10  <sequence name="Sb">
11    activityB1
12    activityB2
13      <target linkName="L"/>
14  </sequence>
15 </flow>
```

Figure 1



WP4 Exclusive Choice A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen. **Example:** The manager is informed if an order exceeds $ 600, otherwise not.

WP5 Simple Merge A point in the workflow process where two or more alternative branches come together without synchronization. It is an assumption of this pattern that none of the alternative branches is ever executed in parallel (if it is not the case, then see the patterns Multi Merge and Discriminator). **Example:** After the payment is received or the credit is granted the car is delivered to the customer.

[4] However, in order to prevent deadlocks, links are not allowed to cross the boundaries of while loops, serializable scopes, or compensation handlers.

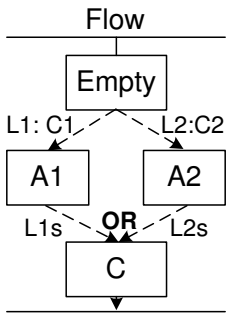
[5] Since BPEL4WS does not provide a graphical notation, the use of figures is limited to some patterns only.

Solutions, WP4 & WP5 As in the previous patterns, two solutions are proposed. The first one relies on the activity switch inherited from XLANG (Listing 6). Each case specifies the activity to be performed when a condition is fulfilled. The second solution uses control links (see Listing 7 and Figure 2). The different conditions (C1 and C2 in the example) are specified as transitionConditions, one for each corresponding link (L1 or L2). This implies that the activities specified as targets for these links (A1 and A2 in the example) will be executed only if the corresponding conditions are fulfilled. An empty activity is the source of links L1 and L2, implying that conditions C1 and C2 are evaluated as soon as the flow is initiated. Activity C is the target of links L1s and L2s whose sources are A1 and A2 respectively, thereby capturing the Simple Merge pattern.

Listing 6

```
<switch>
    <case condition="C1">
        activityA1
    </case>
    <case condition="C2">
        activityA2
    </case>
</switch> activityC
```

Figure 2



Listing 7

```
<flow>
  <links>
    <link name="L1"/>
    <link name="L2"/>
    <link name="L1s"/>
    <link name="L2s"/>
  </links>
  <empty>
    <source linkName="L1"
      transitionCondition="C1"/>
    <source linkName="L2"
      transitionCondition="C2"/>
  </empty>
  activityA1
    <target linkName="L1">
    <source linkName="L1s">
  activityA2
    <target linkName="L2">
    <source linkName="L2s">
  activityC
    joinCondition="L1s OR L2s"
    <target linkName="L1s">
    <target linkName="L2s"> ...
</flow>
```

A difference between these two solutions is that in the solution of Listing 6 only one activity is trigerred, the first one for which the specified condition evaluates to true. Meanwhile, in the solution of Listing 7 multiple branches may be trigerred if more than one of the conditions evaluates to true. To ensure that only one of the branches is trigerred,

the conditions have to be disjoint. If this is not the case, Listing 7 rather provides a solution to the Multi Choice pattern described below.

WP6 Multi-Choice A point in the process, where, based on a decision or control data, a number of branches are chosen and executed as parallel threads. **Example:** After executing the activity *evaluate damage* the activity *contact fire department* or the activity *contact insurance company* is executed. At least one of these activities is executed. However, it is also possible that both need to be executed.

WP7 Synchronizing Merge A point in the process where multiple paths converge into one single thread. Some of these paths are "active" (i.e. they are being executed) and some are not. If only one path is active, the activity after the merge is triggered as soon as this path completes. If more than one path is active, synchronization of all active paths needs to take place before the next activity is triggered. It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete. **Example:** After either or both of the activities *contact fire department* and *contact insurance company* have been completed (depending on whether they were executed at all), the activity submit report needs to be performed (exactly once).

Solutions, WP6 & WP7 As indicated before the solution of WP6 and WP7 are identical to the WSFL-style solutions of WP4 and WP5 (Listing 7). This follows from the *dead-path elimination* principle, which states that the truth value of an incoming link is propagated to its outgoing link. In the example of Listing 7, if condition C1 (C2) evaluates to true, activity A1 (A2) receives a positive value and it is therefore executed. On the other hand, if condition C1 (C2) evaluates to false, activity A1 (A2) receives a negative value, and it is not executed but still propagates the negative value through its outgoing link L1s (L2s). In particular, both A1 and A2 are executed if the two conditions C1 and C2 evaluate to true. In any case, the OR joinCondition attached to C, ensures that C is always executed, provided that one of the activities A1 or A2 is executed.

WP8 Multi-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 action of every incoming branch. **Example:** Sometimes two or more branches share the same ending. Two activities *audit application*

and *process applications* are running in parallel which should both be followed by an activity *close case*, which should be executed twice if the activities *audit application* and *process applications* are both executed.

Solution, WP8 BPEL4WS offers no direct support for WP8. Neither XLANG nor WSFL allow for two active threads following the same path without creating new instances of another process.

WP9 Discriminator A point in the workflow process that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and 'ignores' them. Once all incoming branches have been triggered, it resets itself so that it can be triggered again (which is important otherwise it could not really be used in the context of a loop). **Example:** To improve query response time a complex search is sent to two different databases over the Internet. The first one that comes up with the result should proceed the flow. The second result is ignored.

Solution, WP9 This pattern is not directly supported in BPEL4WS. Neither is there a structured activity construct which can be used for implementing it, nor can links be used for capturing it. The reason for not being able to use the link construct with an OR joinCondition, is the fact that a joinCondition is evaluated first when the status of all incoming links are determined and not, as required in this case, when the first positive link is determined.

WP10 Arbitrary Cycles A point where a portion of the process (including one or more activities and connectors) needs to be "visited" repeatedly without imposing restrictions on the number, location, and nesting of these points.

Solution, WP10 This pattern is not supported in BPEL4WS. Although the while activity allows for structured cycles, it is not possible to jump back to arbitrary parts of the process, i.e. only loops with one entry point and one exit point are allowed.[6] The restriction made that links can not cross the boundaries of a loop and that links may not create a cycle disables support for WP10.

[6] For a discussion on non-structured cycles that can not be unfolded into structured cycles see [11].

WP11 Implicit Termination A given subprocess is terminated when there is nothing left to do, i.e., termination does not require an explicit termination activity.

Solution, WP11 Implicit termination is supported by the flow construct. A structured activity (without flows and links) completes when its outermost activity completes and therefore corresponds to explicit termination. Using the flow construct and links, a subprocess can have multiple sink activities (i.e., activities not being a source of any link) without requiring one unique termination activity.

WP12 MI without Synchronization Within the context of a single case multiple instances of an activity may be created, i.e. there is a facility for spawning off new threads of control, all of them independent of each other. The instances might be created consecutively, but they will be able to run in parallel, which distinguishes this pattern from the pattern for Arbitrary Cycles. **Example:** When booking a trip, the activity *book flight* is executed multiple times if the trip involves multiple flights.

Solution, WP12 Multiple instances of an activity can be created by using the invoke activity embedded in a while loop (see Listing 8). The invoked process, i.e., process B, has to have the attribute createInstance within its receive activity assigned to "yes" (see Listing 9).

WP13-WP15 MI with Synchronization A point in a workflow where a number of instances of a given activity are initiated, and these instances are later synchronized, before proceeding with the rest of the process. In WP13 the number of instances to be started/synchronized is known at design time. In WP14 the number is known at some stage during run time, but before the initiation of the instances has started. In WP15 the number of instances to be created is not known in advance: new instances are created on demand, until no more instances are required. **Example of WP15:** When booking a trip, the activity *book flight* is executed multiple times if the trip involves multiple flights. Once all bookings are made, an invoice is sent to the client. How many bookings are made is only known at runtime through interaction with the user.

Solutions, WP13-WP15 If the number of instances to be synchronized is known at design time (WP13), a simple solution is to replicate the activity as many times as it needs to be instantiated, and run the replicas in parallel by placing them in a flow activity. The solution becomes more

complex if the number of instances to be created and synchronized is only known at run time (WP14), or not known (WP15) – see Listing 10. In this solution a pick activity within a while loop is used, enabling repetitive processing triggered by three different messages: one indicating that a new instance is required, one indicating the completion of a previously initiated instance, and one indicating that no more instances need to be created. Depending on the message received an activity is performed/invoked in each iteration of the loop. However, this is only a work-around solution since the logic of these patterns is not directly captured by a BPEL4WS construct. Instead the logic is encoded by means of a loop and a counter: the counter is incremented each time that a new instance is created, and is decremented each time that an instance is completed. The loop is exited when the value of the counter is zero and no more instances need to be created.

Listing 8

```
<processA>
  <while cond="C1">
    <invoke processB ... >
    </invoke>
  </while>
</process>
```

Listing 9

```
<processB>
  <receive processA ...
    createInstance="yes">
  </receive>
</process>
```

Listing 10

```
moreInstances:=True
i:=0
<while moreInstances OR i>0>
  <pick>
    <onMessage StartNewActivityA>
      invoke activityA
      i:=i+1
    </onMessage>
    <onMessage ActivityAFinished>
      i:=i-1
    </onMessage>
    <onMessage NoMoreInstances>
      moreInstances:=False
    </onMessage>
  </pick>
</while>
```

WP16 Deferred Choice A point in a process where one among several alternative branches is chosen based on information which is not necessarily available when this point is reached. This differs from the normal exclusive choice, in that the choice is not made immediately when the point is reached, but instead several alternatives are offered, and the choice between them is delayed until the occurrence of some event. **Example:** When a contract is finalized, it has to be reviewed and signed either by the director or by the operations manager, whoever is available first. Both the director and the operations manager would be notified that

the contract is to be reviewed: the first one who is available will proceed with the review.

Solution, WP16 This pattern is realized through the pick construct. The semantics of pick, i.e. awaiting the receipt of one of a number of messages and continuing the execution according to the received message, captures the key idea of this pattern, namely a choice is not made immediately when a certain point (i.e. the pick activity) is reached, but delayed until receipt of a message.

WP17 Interleaved Parallel Routing A set of activities is executed in an arbitrary order. Each activity in the set is executed exactly once. The order between the activities is decided at run-time: it is not until one activity is completed that the decision on what to do next is taken. In any case, no two activities in the set can be active at the same time. **Example:** At the end of each year, a bank executes two activities for each account: *add interest* and *charge credit card costs*. These activities can be executed in any order. However, since they both update the account, they cannot be executed at the same time.

Solution, WP17 The existence of serializable scopes in BPEL4WS makes this pattern possible to express (see Listing 11). A serializable scope is a scope activity, whose containterAccessSerializable attribute is set to "yes", thereby guaranteeing concurrency control on shared containers. Defining the activities to be interleaved as activities belonging to different concurrent serializable scopes, implies their potential parallelism and avoids predefining an order between them. Defining, furthermore, the access of these scopes to one and the same container, implies that the activities will be executed consecutively, which is ensured by the blocking of the shared container during their corresponding executions. Two things are worth pointing out with respect to this solution. First, from the BPEL4WS specification it is not clear in what order the different activities are going to be executed. Furthermore, this order cannot be influenced externally. Secondly, since serializable scopes are not allowed to be nested, a more general solution allowing nesting of interleaved parallel routing can not be provided by the use of serializable scopes.

To overcome this limitation a work-around solution using deferred choice (i.e. the pick construct in BPEL4WS) as proposed in [3] can be applied, see Listing 12. The drawback of this solution is its complexity, which increases exponentially with the number of activities that have to be executed in arbitrary order.

Listing 11

```
<flow>
  <scope name=S1
    containerAccessSerializable:="yes">
    <sequence>
        write to container C
        activityA1
        write to container C
    </sequence>
  </scope>
  <scope name=S2
    containerAccessSerializable:="yes">
    <sequence>
        write to container C
        activityA2
        write to container C
    </sequence>
  </scope>
</flow>
```

Listing 12

```
<pick>
  <onMessage m1>
    <sequence>
      activity A1
      activity A2
    </sequence>
  </onMessage>
  <onMessage m2>
    <sequence>
      activity A2
      activity A1
    </sequence>
  </onMessage>
</pick>
```

WP18 Milestone A given activity E can only be enabled if a certain milestone has been reached which has not yet expired. A milestone is defined as a point in the process where a given activity A has finished and an activity B following it has not yet started. **Example:** After having placed a purchase order, a customer can withdraw it at any time before the shipping takes place. To withdraw an order, the customer must complete a withdrawal request form, and this request must be approved by a customer service representative. The execution of the activity *approve order withdrawal* must therefore follow the activity *request withdrawal*, and can only be done if: (i) the activity *place order* is completed, and (ii) the activity *ship order* has not yet started.

Solution, WP18 BPEL4WS does not provide a direct support for capturing this pattern. Therefore, a work-around solution has to be used (see Listing 13). Once again the solution is inspired by the ideas in [3]. A deferred choice between executing the activity B, or executing activity E, is made. A while loop is used to guarantee that as long as B is not chosen, E can be executed an arbitrary number of times. The limitation of this solution is that activity E can not be restricted by any parallel treads.

WP19 Cancel Activity & WP20 Cancel Case A cancel activity terminates a running instance of an activity, while cancelling a case leads to the removal of an entire workflow instance. **Example of WP19:** A customer cancels a request for information. **Example of WP20:** A customer withdraws his/her order.

Listing 13

```
activityA
B_completed:="false"
<while B_completed="false">
  <pick>
    <onMessage mE> activityE
    </onMessage>
    <onMessage mB>
      <sequence> B_completed:="true" </sequence>
    </onMessage>
  </pick>
</while>
activityB
```

Solutions, WP19 & WP20 WP20 is solved with the terminate activity, which is used to abandon all execution within a business process instance of which the terminate activity is a part. All currently running activities must be terminated as soon as possible without any fault handling or compensation behaviour. WP19 is dealt with using fault and compensation handlers, specifying the course of action in cases of faults and cancellations.

4 The Communication Patterns in BPEL4WS

In this section we evaluate BPEL4WS according to the communication patterns presented in [14]. Since communication is realized by exchanging messages between different processes, it is explicitly modelled by sending and receiving messages. Two types of communications are distinguished, namely synchronous and asynchronous communication.

4.1 Synchronous Communication

CP1 Request/Reply Request/Reply communication is a form of synchronous communication where a sender makes a request to a receiver and waits for a reply before continuing to process. The reply may influence further processing on the sender side.

CP2 One-Way A form of synchronous communication where a sender makes a request to a receiver and waits for a reply that acknowledges the receipt of the request. Since the receiver only acknowledges the receipt, the reply is "empty" and only delays further processing on the sender side.

Solutions, CP1 & CP2 The way in which synchronous communication is modelled in BPEL4WS is by the invoke activity included in the requesting process, process A (see Listing 14) and a couple of receive and reply activities in the responding process, process B (see Listing 15). Furthermore, two different containers need to be specified in the invoke activity within process A: one inputContainer, where the outgoing data from the process is stored (or input data for the communication); and one outputContainer, where the incoming data is stored (or the output data from this communication). The One-Way pattern differs from Request/Reply only by B sending its reply (i.e., confirmation) immediately after the message from A has been received, i.e., no processing is performed between receipt and reply activities.

Listing 14

```
1  <process name="processA">
2    <sequence>
3      ...
4      <invoke partner="B" ...
5        inputContainer="Request"
6        outputContainer="Response">
7      </invoke>
8      ...
9    </sequence>
10 </process>
```

Listing 15

```
1  <process name="processB"> ...
2    <sequence>
3      <receive partner="A" ...
4        container="Request">
5      </receive>
6      ...
7      <reply partner="A" ...
8        container="Response">
9      </reply>
10   </sequence>
11 </process>
```

CP3 Synchronous Polling Synchronous Polling communication is a form of synchronous communication where a sender communicates a request to a receiver but instead of blocking continues processing. At intervals defined by the developer, the sender checks to see if a reply has been sent. When it detects a reply it processes it and stops any further polling for a reply. **Example:** During a game session, the system continuously checks if the customer has terminated the game.

Solution, CP3 This pattern is captured trough utilization of two parallel flows: one for the receipt of the expected response, and one for the sequence of the activities not depending on this response (see Listing 16, lines 4 to 7). The initiation of the communication is done beforehand through an invoke action (line 3). To be able to proceed, the invoke action is specified to send data and not wait for a reply. This is indicated by the use of an inputContainer and by omitting the specification of an outputContainer. The communication for the responding process is the same as for the previous pattern (Listing 15).

Listing 16

```
1 <process name="A"
2   <sequence>
3     <invoke partner="B" ... inputContainer="Request"...> </invoke>
4     <flow>
5       <sequence> ... </sequence>
6       <receive partner="B" ... container="Result" ...> </receive>
7     </flow>
8     access container "Result" ...
9   </sequence>
10 </process>
```

4.2 Asynchronous Communication

CP4 Message Passing Message passing is a form of asynchronous communication where a request is sent from a sender to a receiver. When the sender has made the request it essentially forgets it has been sent and continues processing. The request is delivered to the receiver and is processed. **Example:** When an order is received, a log is notified, before the system executes the order.

Solution, CP4 The solution for this pattern has already been demonstrated as a part of the solution for CP3, namely an invoke activity with an inputContainer only (line 3 in Listing 16).

CP5 Publish/Subscribe A form of asynchronous communication where a request is sent by the sender and the receiver is determined by a declaration of interest by the receiver in the request. **Example:** An organization offers information about products to its customers. If the customers are interested in receiving such information, they have to notify a system, which lists interested customers. When product information is going to be distributed to the customers, the organization requests the current list, including the customers' addresses.

CP6 Broadcast A form of asynchronous communication in which a request is sent to all participants, the receivers, of a network. Each participant determines whether the request is of interest by examining the content. **Example:** Before a system is shut down for maintenance, every client connected to it is informed about the situation.

Solutions, CP5 & CP6 Publish/Subscribe and Broadcast are not directly supported in BPEL4WS.

5 Conclusion

In this paper a framework based on existing workflow and communication patterns was used for an in-depth analysis of BPEL4WS. A summary of the results from the analysis are presented in Table 1. The table also shows a comparison of BPEL4WS with XLANG, WSFL and two major Workflow Modelling Languages: Staffware PLC's Staffware and IBM's MQSeries Workflow. The ratings for Staffware and MQSeries Workflow in the table are taken from [3] where an analysis of more than 12 major commercial WFMS is provided. Since XLANG and WSFL correspond to subsets of the BPEL4WS, their ratings are straightforward given the discussions provided in this paper. Note that we indicate that Staffware and MQSeries Workflow are assumed to offer no support for the communication patterns. Although this may not be entirely true (e.g., Staffware has the concept of an event step), they are not intended for communication and therefore rated '–'.

A '+' in a cell of the table refers to direct support (i.e. there is a construct in the language which directly support the pattern). A '–' in the table refers to no direct support. Sometimes there is a feature that only partially supports a pattern, e.g., a construct that implies certain restrictions on the structure of the process. In such cases, the support is rated as '+/–'.

The following observations can now be made from the table: i) As the first five patterns correspond to the basic routing constructs, they are naturally supported by all languages. In contrast, the patterns referring to more advanced constructs are often poorly supported in the different languages. ii) BPEL4WS as a language integrating the futures of the block structured language XLANG and the directed graphs of WSFL, indeed supports all patterns supported by XLANG and WSFL. iii) BPEL4WS as a Web Service Composition language provides constructs for communication modelling which clearly distinguishes it from traditional workflow modelling languages.

Besides these positive remarks, we would also like to pose two negative comments. First of all, BPEL4WS is a complex language because if offers (too) many constructs. The simple fact that many of the patterns can be realized using "XLANG style" and "WSFL style" illustrates its complexity. Secondly, the semantics of BPEL4WS is not always clear. The precise semantics of advanced concepts like serializable scopes leave room for multiple interpretations thus complicating the adoption of the language.

pattern	*product/standard*				
	BPEL	XLANG	WSFL	Staffw.	MQS.
Sequence	+	+	+	+	+
Parallel Split	+	+	+	+	+
Synchronization	+	+	+	+	+
Exclusive Choice	+	+	+	+	+
Simple Merge	+	+	+	+	+
Multi Choice	+	−	+	−	+
Synchronizing Merge	+	−	+	−	+
Multi Merge	−	−	−	−	−
Discriminator	−	−	−	−	−
Arbitrary Cycles	−	−	−	+	−
Implicit Termination	+	−	+	+	+
MI without Synchronization	+	+	+	−	−
MI with a Priori Design Time Knowledge	+	+	+	+	+
MI with a Priori Runtime Knowledge	−	−	−	−	−
MI without a Priori Runtime Knowledge	−	−	−	−	−
Deferred Choice	+	+	−	−	−
Interleaved Parallel Routing	+/−	−	−	−	−
Milestone	−	−	−	−	−
Cancel Activity	+	+	+	+	−
Cancel Case	+	+	+	−	−
Request/Reply	+	+	+	−	−
One-Way	+	+	+	−	−
Synchronous Polling	+	+	+	−	−
Message Passing	+	+	+	−	−
Publish/Subscribe	−	−	−	−	−
Broadcast	−	−	−	−	−

Table 1. Comparison of BPEL4WS against XLANG, WSFL, Staffware and MQSeries Workflow using both workflow and communication patterns.

References

1. W.M.P. van der Aalst. Don't go with the flow: Web services composition standards exposed. to appear. *IEEE Intelligent Systems*, Jan/Feb 2003. Electronically accessible from `http://www.tm.tue.nl/it/research/patterns/ieeewebflow.pdf`.
2. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, Massachusetts, 2002.
3. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. Technical report FIT-TR-2002-2, Faculty of IT, Queensland University of Technology, July 2002. Accessed from `http://www.tm.tue.nl/it/research/patterns`. To appear in Distributed and Parallel Databases, Kluwer.
4. BPML.org. Business process modeling language. Accessed November 2002 from `www.bpmi.org/`, 2002.
5. Workflow Management Coalition. Terminology and glossary. Document Number WFMC-TC-1011, Document Status - Issue 3.0, February 1999 `http://www.wfmc.org`.

6. F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, March 2002.
7. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services. `http://dev2dev.bea.com/techtrack/BPEL4WS.jsp`.
8. M. Dumas and A.H.M. ter Hofstede. UML activity diagrams as a workflow specification language. In M. Gogolla and C. Kobryn, editors, *Proc. of the 4th Int. Conference on the Unified Modeling Language (UML01)*, volume 2185 of *LNCS*, pages 76–90, Toronto, Canada, October 2001. Springer Verlag.
9. L. Fischer, editor. *Workflow Handbook 2001, Workflow Management Coalition*. Future Strategies, Lighthouse Point, Florida, 2001.
10. Hillside.net. Patterns Home Page. `http://hillside.net/patterns`, 2000–2002.
11. B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. In B. Wangler and L. Bergman, editors, *Proc. of the 12th Int. Conference on Advanced Information Systems Engineering (CAiSE00)*, volume 1789 of *LNCS*, pages 431–445, Stockholm, Sweden, June 2000. Springer Verlag.
12. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, 1999.
13. M. Rosemann and P. Green. Developing a meta model for the Bunge–Wand–Weber ontological constructs. *Information Systems*, 27:75–91, 2002.
14. W.A. Ruh, F.X. Maginnis, and W.J. Brown. *Enterprise Application Integration: A Wiley Tech Brief*. John Wiley and Sons, Inc, 2001.
15. E. Söderström, B. Andersson, P. Johannesson, E. Perjons, and B. Wangler. Towards a framework for comparing process modelling languages. In A.B. Pidduck, J. Mylopoulos, C.C. Woo, and M.Tamer Özsu, editors, *14th International Conference on Advanced Information Systems Engineering, CAiSE 2002*, volume 2348 of *LNCS*, pages 600–611. Springer, 2002.
16. UN/CEFACT and OASIS. ebXML Business Process Specification Schema (Version 1.01). Accessed November 2002 from `www.ebxml.org/specs/ebBPSS.pdf`, 2001.
17. W3C. Web Service Choreography Interface (WSCI) 1.0. Accessed November 2002 from `www.w3.org/TR/wsci/`, 2002.