Analysis of Web Services Composition Languages: The Case 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, Sweden
petia@dsv.su.se
[2] Department of Technology Management
Eindhoven University of Technology, The Netherlands
w.m.p.v.d.aalst@tm.tue.nl
[3] Centre for Information Technology Innovation
Queensland University of Technology, Australia
{m.dumas, a.terhofstede}@qut.edu.au

Abstract. Web services composition is an emerging paradigm for application integration within and across organizational 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 effort has been dedicated to systematically evaluate the capabilities and limitations of these languages and techniques. The work reported in this paper is a step in this direction. It presents an in-depth analysis of the Business Process Execution Language for Web Services (BPEL4WS) with respect to a framework composed of workflow and communication patterns.
Keywords: web services composition, business process modeling, interorganizational workflow, BPEL4WS

1 Introduction

Web Services is a rapidly emerging paradigm for architecting and implementing business collaborations within and across organizational boundaries. In this paradigm, the functionalities provided by business applications are encapsulated within web services: software components described at a semantic 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 [7]. Once deployed, web services provided by various organizations can be inter-connected in order to implement business collaborations, leading to *composite web services*.

Business collaborations require long-running interactions driven by explicit process models [1]. Accordingly, it is a natural choice to capture the logic of a composite web service using business process modeling

---

[*] Research conducted while at the Queensland University of Technology.

languages tailored for web services. Many such languages have recently emerged, including WSCI [21], BPML [5], BPEL4WS [8], BPSS [20], and XPDL [22], with little effort being dedicated to their evaluation with respect to common benchmarks. The comparative evaluation of these languages would contribute to ongoing standardization and development efforts, by identifying their relative strengths and weaknesses, delimiting their capabilities and limitations, and detecting ambiguities.

As a step in this direction, this paper analyses one of these emerging languages, namely BPEL4WS. An evaluation of BPML can be found in [2] and is briefly summarized in Section 4. Similar evaluations of other languages for web services composition will be conducted in the future.

Approach The reported analysis is based on a framework composed of *patterns*: abstracted forms of recurring situations encountered at various stages of software development [13]. Specifically, the framework brings together a set of *workflow patterns* documented in [4], and a set of *communication patterns* documented in [17].

The workflow patterns (WPs) have been compiled from an analysis of workflow languages. They capture typical *control flow* dependencies encountered in workflow modeling. More than 12 commercial Workflow Management Systems (WFMS) as well as the UML Activity Diagrams notation, have been evaluated in terms of their support for these patterns [4, 10]. Since the functionalities abstracted by the WPs are also required for capturing interactions between web services, these patterns are arguably suitable for analysing languages for web services composition.

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). Given the strong overlap between EAI and web services composition, both requiring the representation of *communication flows* between distributed processes, the communication patterns defined for EAI provide an arguably suitable framework for the analysis of web services composition languages.

The evaluation framework therefore focuses on the control-flow and the communication perspectives. In particular, it excludes the data manipulation and the resource allocation perspectives (e.g. partner selection). The argument is that data manipulation and resource allocation can be treated separately from control-flow and communication, and that a separate framework could be designed for evaluating languages with respect to these other perspectives. Although data manipulation (e.g. counters and boolean variables used as flags) can be used for captur-

ing control-flow aspects, this is undesirable, not only because it breaks the principle of separation of concerns, but more importantly, because it hinders the applicability of verification and analysis techniques to the resulting process models [14].

Overview of BPEL4WS BPEL4WS builds on IBM's WSFL (Web Services Flow Language) and Microsoft's XLANG (Web Services for Business Process Design). Accordingly, it combines the features of a block structured process language (XLANG) with those of a graph-based process language (WSFL). BPEL4WS is intended for modeling two types of processes: executable and abstract processes. An *abstract process* is a business protocol specifying the message exchange behavior between different parties without revealing the internal behavior of any of them. An *executable process* specifies the execution order between a number of constituent *activities*, the *partners* involved, the *messages* exchanged between these partners, and the *fault* and *exception handling* mechanisms.

A BPEL4WS process specification is a kind of flow-chart. Each element in the process is called an *activity*. An activity is either *primitive* or *structured*. The primitive activity types are: invoke (to invoke an operation of a web service described in WSDL); receive (to wait for a message from an external source); reply (to reply to an external source); wait (to remain idle for some time); assign (to copy data from one data *container* to another); throw (to indicate errors in the execution); terminate (to terminate the entire service instance); and empty (to do nothing).

To enable the representation of complex structures the following *structured activities* are provided: 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. Given a set of activities contained within the same flow, the execution order can further be controlled through (control) links, which allow the definition of dependencies between two activities: the target activity may only start when the source activity has ended. Activities can be connected through links to form directed acyclic graphs.

Related work BPEL4WS was released as part of a trio of standard proposals, together with WS-Coordination (WS-C) and WS-Transaction (WS-T). However, WS-C and WS-T deal with issues orthogonal to control-flow and communication and hence fall outside the scope of this paper. WS-C and WS-T are concerned with the coordination of distributed pro-

cesses, in particular for the purpose of performing ACID transactions and business activities (i.e. long-running transactions). A comparison of WS-C and WS-T with a competing proposal, namely the Business Transaction Protocol (BTP), is reported in [9].

Existing frameworks for comparing process modeling languages [12, 19, 18] are coarse-grained and syntactical in nature, answering questions such as: "does a language offer more operators than another and which ones?", or "does it integrate a given ontological construct?". This contrasts with the functional nature of the workflow patterns approach which addresses questions such as: "does a language provide a given functionality and how?". As a result, the above frameworks are more appropriate for IS/IT-managers requiring a high-level view of business processes [19]. They are therefore complementary to the analysis presented here.

2 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 [3, 11, 16] is also relevant for web services composition languages like BPEL4WS, XLANG, and WSFL. In this section, we consider the 20 workflow patterns presented in [4], 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 avoiding irrelevant coding details.

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 This pattern is directly supported by the construct sequence.

WP2 Parallel Split A point in the process where a single thread of control splits into multiple threads which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order [6]. **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 [6]. 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 *send tickets* and *receive payment* 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 (Listing 1, lines 2–5). If no link is defined within a flow, the activities within it are executed in parallel. Adding an activity after the flow, e.g. activity B in line 6, yields the solution to the Synchronization pattern.

Alternative solutions using control links are given in Listing 2. Links L1 and L2 are defined in a flow F which consists of 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). In order to ensure that B is only executed after both A1 and A2, an AND joinCondition is included (line 11).

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

Listing 2
```
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 1 and 2 illustrate the two styles of process modeling supported by BPEL4WS. Listing 1 shows the "XLANG-style" of modeling (i.e., routing through structured activities). Listing 2 shows the "WSFL-style" of modeling (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 3, where the sequences Sa and Sb are defined to run in parallel. The definition of link L (lines 3, 7 and 13) implies that activity B2 (which follows B1) can be executed only after activity A1 has completed. In other words, link L captures an intermediate synchronization point between the parallel threads Sa and Sb. This inter-thread synchronization cannot be expressed using structured activities only (for a proof see [15]), so that the solution of the pattern that uses links, is more general than the one without. Figure 1 illustrates the example in graphical form.

Listing 3

```
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>
```
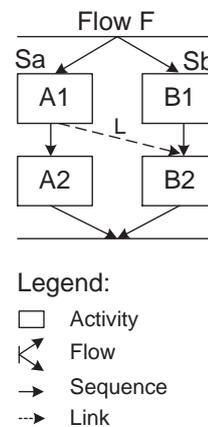
Figure 1

WP4 Exclusive Choice A point in the process where, based on a decision or 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.

Solutions, WP4 & WP5 As in the previous patterns, two solutions are proposed. The first one relies on the activity switch inherited from XLANG (Listing 4). Each case specifies the activity to be performed when a condition is fulfilled. The second solution uses control links (see Listing 5

---

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

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 4

```
1 <switch>
2     <case condition="C1">
3         activityA1
4     </case>
5     <case condition="C2"/>
6         activityA2
7     </case>
8 </switch>
9 activityC
```

Figure 2

Listing 5

```
1 <flow>
2   <links>
3     <link name="L1"/>
4     <link name="L2"/>
5     <link name="L1s"/>
6     <link name="L2s"/>
7   </links>
8   <empty>
9     <source linkName="L1"
10       transitionCondition="C1"/>
11    <source linkName="L2"
12      transitionCondition="C2"/>
13   </empty>
14   activityA1
15     <target linkName="L1">
16     <source linkName="L1s">
17   activityA2
18     <target linkName="L2">
19     <source linkName="L2s">
20   activityC
21     joinCondition="L1s OR L2s"
22     <target linkName="L1s">
23     <target linkName="L2s"> ...
24 </flow>
```
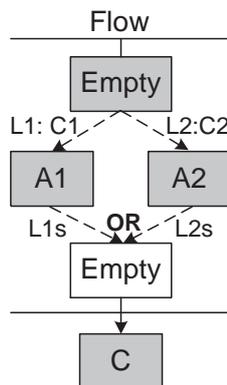
A difference between these two solutions is that in the solution of Listing 4 only one activity is triggered, the first one for which the specified condition evaluates to true. Meanwhile, in the solution of Listing 5 multiple branches may be triggered if more than one of the conditions evaluates to true. To ensure that only one of the branches is triggered, the conditions have to be disjoint.

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 ac-

tivity *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 The solution of WP6 and WP7 are identical to the WSFL-style solutions of WP4 and WP5 (Listing 5). This follows from the *dead-path elimination* principle, according to which the truth value of an incoming link is propagated to its outgoing link. In the example of Listing 5, if condition C1 (C2) evaluates to true, activity A1 (A2) receives a positive value and 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:** Two activities *audit application* and *process applications* running in parallel should both be followed by an activity *close case*. Activity *close case* should be executed twice if both activities *audit application* and *process applications* are executed.

Solution, WP8 BPEL4WS offers no direct support for WP8. Neither XLANG nor WSFL allow multiple (possibly concurrent) activations of an activity following a point where multiple paths converge. In the above

example, the *close case* activity cannot be activated once after the completion of *audit application* and another time after the completion of *process applications*.

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 links with an OR joinCondition is that a joinCondition is only evaluated 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. The while activity can only capture structured cycles, i.e. loops with one entry point and one exit point. The restrictions made in BPEL4WS that links should not cross the boundaries of a loop, and that links should not create cycles, entails that there is no support for arbitrary cycles. Note that there exist non-structured cycles that cannot be unfolded into structured ones [15].

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 The pattern is directly supported since in BPEL4WS there is no need to explicitly specify a 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 6). The invoked process, i.e., process B, has to have the attribute createInstance within its receive activity assigned to "yes" (see Listing 7).

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 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 8. 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 encoded by means of a loop and a counter: the counter is incremented when a new instance is created, and 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.

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

Listing 6

```
1 <processA>
2   <while cond="C1">
3     <invoke processB ... >
4     </invoke>
5   </while>
6 </process>
```

Listing 7

```
1 <processB>
2   <receive processA ...
3     createInstance="yes">
4   </receive>
5 </process>
```

Listing 8

```
1 moreInstances:=True
2 i:=0
3 <while moreInstances OR i>0>
4   <pick>
5     <onMessage StartNewActivityA>
6       invoke activityA
7       i:=i+1
8     </onMessage>
9     <onMessage ActivityAFinished>
10      i:=i-1
11    </onMessage>
12    <onMessage NoMoreInstances>
13      moreInstances:=False
14    </onMessage>
15  </pick>
16 </while>
```

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. Both the director and the operations manager would be notified and 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 It is possible to capture this pattern in BPEL4WS using the concept of serializable scopes (see Listing 9). A serializable scope is an activity of type scope whose containerAccessSerializable attribute is set to "yes", thereby guaranteeing concurrency control on shared containers.

The activities to be interleaved are placed in different containers which all write to a single shared container (container C in Listing 9). Since the activities are placed in different containers, they can potentially be executed in parallel. On the other hand, since the serializable scopes that contain the activities write to the same container, no two of them will be "active" simultaneously, but instead, they will be executed one after the other. Three things are worth pointing out with respect to this solution. Firstly, the semantics of serializable scopes in BPEL4WS is not clearly defined. The BPEL4WS specification only states that this semantics is "similar to the standard isolation level 'serializable' of database transactions", but it does not specify where the similarity stops (e.g. how does the underlying transaction model deal with or prevent serialization conflicts?). Secondly, it is not possible in this solution to externally influence (at runtime) the order in which the activities are executed: instead, this order is fixed by the transaction manager of the underlying BPEL4WS engine. Finally, since serializable scopes are not allowed to be nested, this solution is not applicable if one occurrence of the interleaved parallel routing pattern is embedded within another occurrence.

To overcome these limitations, a work-around solution using deferred choice (i.e. pick) as proposed in [4] can be applied (see Listing 10). The drawback of this solution is its complexity, which increases exponentially with the number of activities to be interleaved.

Listing 9

```
1 <flow>
2   <scope name=S1
3     containerAccessSerializable:="yes">
4     <sequence>
5        write to container C
6        activityA1
7        write to container C
8     </sequence>
9   </scope>
10  <scope name=S2
11    containerAccessSerializable:="yes">
12    <sequence>
13       write to container C
14       activityA2
15       write to container C
16    </sequence>
17  </scope>
18 </flow>
```

Listing 10

```
1 <pick>
2   <onMessage m1>
3     <sequence>
4        activity A1
5        activity A2
6     </sequence>
7   </onMessage>
8   <onMessage m2>
9     <sequence>
10       activity A2
11       activity A1
12     </sequence>
13  </onMessage>
14 </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 a point in the process where a given activity A has finished and a subsequent activity B 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 direct support for capturing this pattern. Therefore, a work-around solution using deferred choice, as proposed in [4], has to be applied.

WP19 Cancel Activity & WP20 Cancel Case A cancel activity terminates a running instance of an activity, while canceling 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.

Solutions, WP19 & WP20 WP20 maps directly to the basic activity terminate, which is used to abandon all execution within a business process instance. All currently running activities must be terminated as soon as possible without any fault handling or compensation behavior. WP19 is supported through fault and compensation handlers.

3 The Communication Patterns in BPEL4WS

In this section we evaluate BPEL4WS with respect to the communication patterns presented in [17]. Since communication is realized by exchanging messages between different processes, it is explicitly modeled by sending and receiving messages. Two types of communications are distinguished, namely synchronous and asynchronous communication.

3.1 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 modeled in BPEL4WS is by the invoke activity included in the requesting process, process A (see Listing 11) and a couple of receive and reply activities in the responding process, process B (see Listing 12). 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 output-Container, where the incoming data is stored (or the output data from the 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.

Listing 11

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

Listing 12

```
1 <process name="processB"> ...
2   <sequence>
3     <receive partner="processA" ...
4       container="Request">
5     </receive>
6     ...
7     <reply partner="processA" ...
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, the sender checks to see if a reply has been sent. When it detects a reply it processes it and stops any further polling. **Example:** During a game session, the system continuously checks if the customer has terminated the game.

Solution, CP3 This pattern is captured through 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 13, 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 omitting the spec-

ification of an outputContainer. The communication for the responding process is the same as for the previous pattern (Listing 12).

Listing 13

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

3.2   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 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 part of the solution for CP3, namely an invoke activity with an inputContainer only (line 3 in Listing 13).

CP5 Publish/Subscribe A form of asynchronous communication where a request is sent by a process and the receivers are determined by a previous declaration of interest. **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 keeps track of 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.

4 Conclusion

A comparison of BPEL4WS with XLANG, WSFL, BPML and WSCI is given in Table 4. The ratings for BPML and WSCI in the table are taken from [2]. The ratings for BPEL4WS, XLANG and WSFL are based on the discussions in this paper. A '+' in a cell of the table refers to direct support (i.e. there is a construct in the language which directly supports the pattern). A '–' indicates that there is no direct support. This does not mean though that it is not possible to realize the pattern through some work-around solution. In fact, any of the patterns can be realized using a standard programming language but this is meaningless.[5] Sometimes there is a feature that only partially supports a pattern, e.g. a construct that directly supports the pattern but imposes some restrictions on the structure of the process. In such cases, the support is rated as '+/–'.

The following observations can be made from the table:

- As the first five patterns correspond to the basic routing constructs, they are directly supported by all languages.
- BPEL4WS as a language integrating the features of the block structured language XLANG and the directed graphs of WSFL, indeed supports the union of patterns supported by XLANG and WSFL.
- BPEL4WS, in contrast to BPML, does offer direct support for the Multi Choice and Synchronizing Merge. This is a consequence of the "dead-path elimination" characteristic inherited from WSFL.
- BPEL4WS does not support the Multi-Merge pattern, while BPML directly supports it with some restrictions. This is due to the fact that BPML, unlike BPEL4WS, supports invocation of sub-processes.
- Most of these languages support the Deferred Choice. This distinguishes them from many mainstream workflow languages.
- BPEL4WS, through the concept of serializable scopes, is the only one of the above languages to support the Interleaved Parallel Routing pattern, although with some restrictions.
- None of the compared languages supports *arbitrary* cycles.

---

[5] In 1936 Alan Turing defined the "Turing machine": A basic model of a computer consisting of a tape, state register, and action table. Languages such as BPEL4WS, XLANG, WSFL, BPML and WSCI are Turing complete if we assume perfect technology, e.g., unlimited storage. This implies that they can be used to emulate a Turing machine, and therefore, do any calculation thinkable. Unfortunately, this observation is not relevant from a practical point of view. Any programming language is Turing complete. However, this does not imply suitability for web services composition. Therefore, we consider "direct support" rather than providing formal expressiveness results.

pattern	product/standard				
	BPEL	XLANG	WSFL	BPML	WSCI
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, XLANG, WSFL, BPML and WSCI using both workflow and communication patterns.

BPEL4WS is an expressive language when compared to other languages for business process modeling, in particular those supported by workflow management systems which in general provide direct support for less than half of the workflow patterns [4]. On the negative side, BPEL4WS is a complex language in the sense that it offers many overlapping constructs (i.e. it lacks orthogonality). This is illustrated through the multiplicity of possible solutions for some of the patterns, i.e. "XLANG–style" solutions, "WSFL–style" solutions, and solutions combining both these styles. In addition, the semantics of BPEL4WS is not always clear, especially for advanced constructs such as serializable scopes. A simplification of the language and a formalization in terms of formal process modeling languages (e.g. π–calculus or Petri nets) are therefore desirable.

References

1. W.M.P. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–76, January/February 2003.
2. W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and P. Wohed. Pattern-Based Analysis of BPML (and WSCI). Brisbane, Australia, 2002. `http://www.citi.qut.edu.au/pubs/technical/pattern_based_analysis_BPML.p%df`.
3. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, Massachusetts, 2002.
4. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. Technical Report FIT-TR-2002-02, Faculty of IT, Queensland University of Technology, Brisbane, Australia, 2002. (Also see `http://www.citi.qut.edu.au/pubs/technical/workflow_patterns.pdf`.) To appear in *Distributed and Parallel Databases* Vol. 14:5–51, 2003.
5. BPML.org. Business process modeling language. Accessed November 2002 from `www.bpmi.org/`, 2002.
6. Workflow Management Coalition. Terminology and glossary. Document WFMC-TC-1011 Issue 3.0, February 1999 `http://www.wfmc.org`.
7. 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.
8. 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`.
9. S. Dalal, S. Temel, M. Little, M. Potts, and J. Webber. Coordinating business transactions on the web. *IEEE Internet Computing*, 7(1):30–39, January/February 2003.
10. 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.
11. L. Fischer, editor. *Workflow Handbook 2001, Workflow Management Coalition*. Future Strategies, Lighthouse Point, Florida, 2001.
12. P. Green and M. Rosemann. An ontological analysis of integrated process modelling. In *Proc. of the 11th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 225–240, Heidelberg, Germany, June 1999. Springer Verlag.
13. Hillside.net. Patterns Home Page. `http://hillside.net/patterns`, 2000–2002.
14. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows (submitted)*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2002. Available via http://www.tm.tue.nl/it/research/patterns.
15. 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.
16. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, 1999.
17. W.A. Ruh, F.X. Maginnis, and W.J. Brown. *Enterprise Application Integration: A Wiley Tech Brief*. John Wiley and Sons, Inc, 2001.

18. R. Shapiro. A Comparison of XPDL, BPML and BPEL4WS. Accessed February 2003, `http://xml.coverpages.org/Shapiro-XPDL.pdf`.

19. E. Söderström, B. Andersson, P. Johannesson, E. Perjons, and B. Wangler. Towards a framework for comparing process modelling languages. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 2348 of *LNCS*, Toronto, Canada, May 2002. Springer.

20. UN/CEFACT and OASIS. ebXML Business Process Specification Schema (Version 1.01). Accessed November 2002 from `www.ebxml.org/specs/ebBPSS.pdf`, 2001.

21. W3C. Web Service Choreography Interface (WSCI) 1.0. Accessed November 2002 from `www.w3.org/TR/wsci/`, 2002.

22. WfMC. Workflow process definition interface - XML Process Definition Language. Accessed November 2002 from `www.wfmc.org/standards/docs/TC-1025_10_beta_xpdl_073002.pdf`, 2002.