Pattern Based Analysis of BPML (and WSCI)

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

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

Abstract. Web services composition is an emerging paradigm for enabling 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 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 step in this direction. It presents an in-depth analysis of the Business Process modeling Language (BPML). The framework used for this analysis is based on a collection of workflow and communication patterns. In addition to BPML, the framework is also applied to the Web Services Choreography Interface (WSCI). WSCI and BPML have several routing constructs in common but aim at different levels of the web services stack.
Keywords: business process modeling, web services composition, BPML, WSCI

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 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 [9]. Once deployed, web services provided by various organizations can be inter-connected in order to implement business collaborations, leading to *composite web services*.

⋆ Research conducted while at the Queensland University of Technology.

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 modeling language tailored for web services. Recently, many languages have emerged, including WSCI [23], BPML [5], BPEL4WS [10], XLANG [8], WSFL [14], and BPSS [22], 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 step in this direction, this paper reports an in-depth analysis of two of these emerging languages, namely BPML (Business Process Modeling Language), and its "sibling" language WSCI. 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 [13]. Specifically, the framework brings together a set of *workflow patterns* documented in [3], and a set of *communication patterns* documented in [19].

The workflow patterns (WPs) have been compiled from an analysis of existing workflow languages and 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, have been evaluated in terms of their support for these patterns [3, 11]. The WPs are arguably suitable for analyzing 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 modeling abilities of web services composition languages, given the strong overlap between EAI and web services technologies.

A similar analysis using the same framework as in this paper has been previously conducted on BPEL4WS and its parent languages XLANG and WSFL [24]. This paper and [24] together therefore provide a basis for comparing these emerging business process languages. Some conclusions of this comparative study are given in the conclusion of this paper.

Two other frameworks for analyzing and comparing business process modeling languages have been proposed by Rosemann & Green [18] and Söderström et al. [21]. While these two frameworks are motivated by the same problem that motivates this paper, i.e. the continuously increasing number of process modeling 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.

There have been several comparisons of some of the languages mentioned in this paper. These comparisons typically do not use a framework and provide an opinion rather than a structured analysis. A positive example is [20] where XPDL, BPML and BPEL4WS are compared by relating the concepts used in the three languages. Unfortunately, the paper raises more questions than it answers.

The rest of the paper is structured as follows. Section 2 provides an overview of the BPML language. In sections 3 and 4 the BPML language is analyzed using the set of workflow and communication patterns respectively. Section 5 introduces WSCI and discusses the differences and commonalities between WSCI and BPML. Finally, Section 6 concludes.

2 BPML

BPML (Business Process Modeling language) is a standard developed and promoted by BPMI.org (the Business Process Management Initiative). At the time of writing this paper, BPMI.org is supported by several organizations, including Intalio, SAP, Sun, and Versata. BPML can be seen as a language competing with other standards such as IBM's WSFL (Web Services Flow Language) and Microsoft's XLANG (Web Services for Business Process Design) which recently merged into BPEL4WS (Business Process Execution Language for Web Services).

The main ingredients of BPML are: activities, processes, contexts, properties, and signals. *Activities* are components performing specific functions. There are two types of activities: *simple* activities and *complex* activities. Simple activities are atomic while complex activities are decomposed into smaller activities. A *process* is a complex activity which can be invoked by other processes. A process that is defined independently of other processes is called a *top-level process*. A process that is defined to execute within another process is called a *nested process*. Furthermore there are two types of special processes: exception processes and compensation processes.

Contexts are very important in BPML. A context defines an environment for the execution of related activities. The context can be used to exchange information and coordinate execution. A context contains local definitions that only apply within the scope of that context. Local defini-

tions include properties, processes, and signals, among others. Contexts can be nested and a child context (recursively) inherits the definitions of its parent contexts and may override them. A process can have a context which is shared by all activities that execute as part of that process. *Properties* are used to exchange information and can only exist in a context. A property definition has a name and a type while each property instance has a value in the range of the type of the corresponding definition. One can think of properties as attributes (or instance variables) of a process. *Signals* are used to coordinate the execution of activities executing within a common context other than through basic routing constructs such as sequence. One can think of signals as messages, and there are constructs to send such a message (raise signal) and wait for a message (synchronize signal).

BPML offers the following types of simple activities:

action To perform or invoke a single operation that involves the exchange of input and output messages that are mapped onto properties. Actions allow two top-level processes to communicate.
assign To assign a new value to a property.
call To instantiate a process (whether a top-level process or a subprocess) and wait for it to complete.
compensate To invoke compensation for a given process.
delay To wait for a specified period or a specified time.
empty An action that does nothing.
fault To throw a fault in the current context.
raise To raise a signal.
spawn To instantiate a process without waiting for it to complete.
synch To wait for a signal to be raised.

Meanwhile, the types of complex activities offered by BPML are:

all To execute activities in parallel.
choice To choose among multiple alternatives based on the occurrence of an event.
foreach To execute activities once for each element in a list.
sequence To execute activities in sequence.
switch To conditionally execute one among a set of activities.
until To execute activities once or more times based on an exit condition.
while To execute activities zero or more times based on an exit condition.

For more information on BPML, we refer to [5].

3 The Workflow Patterns in BPML

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, 12, 17] is also relevant for web services composition languages like BPML, 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 BPML. In particular, we indicate whether the pattern is directly supported by a BPML construct. If this is not the case, we sketch a workaround solution. Most of the solutions are presented in a simplified BPML notation which is intended to capture the key ideas of the solutions while avoiding coding details. In other words, the fragments of BPML definitions provided here are not "ready to be run".

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 BPML construct sequence as illustrated in in Listing 1. This listing uses a simplified notation: activityA and activityB need to be replaced by concrete activities when the pattern is applied to a given situation.

In BPML, the activities directly contained within a process are by default assumed to be executed in sequential order. Therefore, the operator sequence can be omitted as is shown in Listing 2.

Listing 1 (Sequence)

```
<process name="SequentialProcess1">
  <sequence>
    activityA
    activityB
  </sequence>
</process>
```

Listing 2 (Sequence)

```
<process name="SequentialProcess2">
  activityA
  activityB
</process>
```

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 [7]. **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 [7]. 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.

Listing 3 (Split/Synchronization)

```
1 <process name="ParallelProcess1">
2   <sequence>
3     <all>
4       activityA1
5       activityA2
6     </all>
7     activityB
8   </sequence>
9 </process>
```

Listing 4 (Split/Synchronization)

```
1 <process name="ParallelProcess2">
2   <context>
3     <signal name="tns:completedA1">
4      tns is the namespace of this BPML
5      process definition
6     <signal name="tns:completedA2">
7     <process name="activityA1">
8       ...
9       <raise signal="tns:completedA1">
10    </process>
11    <process name="activityA2">
12      ...
13      <raise signal="tns:completedA2">
14    </process>
15  </context>
16  <all>
17    <spawn process="activityA1"/>
18    <spawn process="activityA2"/>
19  </all>
20  ...
21  <sequence>
22    <all>
23     <synch signal="tns:completedA1">
24     <synch signal="tns:completedA2">
25    </all>
26    activityB
27  </sequence>
28 </process>
```

Solutions, WP2 & WP3 The parallel split is realized by defining the activities to be run in parallel as components of a complex activity of type all (see Listing 3, lines 3 to 6). Adding an activity after the flow, as

for example activity B in line 7 of this listing, yields the solution to the Synchronization pattern.

Instead of using the all complex activity to launch and synchronize parallel activities it is also possible to use the spawn and synch activities as shown in Listing 4. The context declares two signals (completedA1 and completedA2) and two processes (activityA1 and activityA2). First the two processes are initiated using two spawn activities. Some time later the two processes are synchronized through two synch activities. Note that each of the synch activities in the main process waits for the execution of the raise activity in the corresponding subprocess. This combination of raise and synch activities effectively captures the Synchronization pattern. Note that in Listing 4 the spawn activities appear within an all activity. However, this all activity can be replaced by a sequence activity without changing the logic of the overall process since the "spawn activities take an insignificant amount of time to complete" [5, page 48]. A similar remark applies to the synch activities appearing within an all activity.

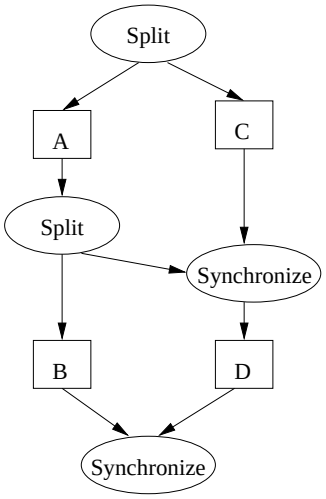
The solution of patterns WP2 and WP3 using spawn and synch (Listing 4) is more general than the solution using just the all construct (Listing 3). For example, the synchronization point appearing between activities C and D in Figure 1 cannot be captured using a combination of activities all and sequence, without using a synch activity as in Listing 5. For a proof that it is not possible to express the process described in Figure 1 by simply combining structured workflow constructs (i.e. all, switch, sequence, and while), we refer to [16, 15].

WP4 Exclusive Choice A point in the 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 with another one (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 Listing 6 shows the realization of the Exclusive Choice and Simple Merge patterns using a switch activity. There are three possibilities. If condition C1 (not specified) evaluates to true, activityA1 is executed. If condition C1 evaluates to false and condition C2 evaluates to true, activityA2 is executed. If both condition C1 and

Figure 1: Synchronization point in the middle of two threads



Listing 5 (Split/Synchronization)

```
1 <process name="ParallelProcess3">
2   <context>
3     <signal name="tns:completedT1">
4     <signal name="tns:completedT2">
5     <signal name="tns:completedA">
6     <process name="activityT1">
7       activityA
8       <raise signal="tns:completedA">
9       activityB
10      <raise signal="tns:completedT1">
11    </process>
12    <process name="activityT2">
13      activityC
14      <synch signal="tns:completedA"
15      activityD
16      <raise signal="tns:completedT2">
17    </process>
18  </context>
19  <all>
20    <spawn process="activityT1"/>
21    <spawn process="activityT2"/>
22  </all>
23  ...
24  <sequence>
25    <all>
26     <synch signal="tns:completedT1">
27     <synch signal="tns:completedT1">
28    </all>
29    ...
30  </sequence>
31 </process>
```

condition C2 evaluate to false, the default activity activityA3 is executed. Clearly, the switch activity (Listing 6, lines 3 to 15) can be used to model situations where there is a one-to-one correspondence between the Exclusive Choice and the Simple Merge. After executing one of the three activities, activityB is executed.

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.

Listing 6 (Exclusive Choice)

```
<process name="ChoiceProcess1">
  <sequence>
    <switch>
      <case>
        <condition> C1 </condition>
        activityA1
      </case>
      <case>
        <condition> C2 </condition>
        activityA2
      </case>
      <default>
        activityA3
      </default>
    </switch>
    activityB
  </sequence>
</process>
```

Listing 7 (Multi-Choice)

```
<process name="ChoiceProcess2">
  <all>
    <switch>
      <case>
        <condition> C1 </condition>
        activityA1
      </case>
    </switch>
    <switch>
      <case>
        <condition> C2 </condition>
        activityA2
      </case>
    </switch>
  </all>
</process>
```

Solution, WP6 BPML does not provide direct support for the Multi-Choice pattern. Fortunately, as indicated in [3], a workaround solution of this pattern can be obtained by combining patterns WP2 and WP4. This solution is sketched in Listing 7. Depending on conditions C1 and C2, activityA1 and/or activityA2 are 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, WP7 As for the Multi-Choice pattern, there is no BPML construct that directly corresponds to this pattern, although there are workaround solutions to capture it by combining the solutions of patterns WP2, WP3, WP4, and WP5. The idea is to either express the pattern by nesting switch and all statements (a solution that only works for structured processes), or to use combinations of raise and synch state-

ments. The latter approach is more general than the former as discussed in Pattern WP3.

Depending on the place in the process where the synchronizing merge appears, designing a workaround solution for this pattern may be relatively difficult. For example, if in Figure 1 the *Split* node at the top of the diagram was replaced by a *Multi-Choice* node, and the two *Synchronize* nodes were replaced by *Synchronizing Merge* nodes, then the process would need to somehow keep track of the activation of the left thread (containing activities A and B) in order to determine whether activity D should be activated immediately after activity C completes, or whether it should also wait for activity A to complete. Hence, in the general case, some bookkeeping of the paths that are taken during the execution of a process has to be encoded into the process definition in order to implement this pattern.

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 BPML provides a partial solution for this pattern as shown in Listing 8. Using the all and two switch activities, activityA and/or activityB are started (or none). After executing either of these activities, a new instance of the process activityC is created. Therefore, if both C1 and C2 evaluate to true, two instances of activityC are created thus realizing the Multi-Merge pattern. We consider this to be only partial support because the remainder of the process has to be put in a separate process thus limiting its application (especially if the Multi-Merge is embedded in a loop). Moreover, the construction involves 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).

Listing 8 (Multi-Merge)

```
1  <process name="MultiMerge1">
2    <context>
3      <process name="activityC">
4        ...
5      </process>
6    </context>
7    <all>
8      <switch>
9        <case>
10         <condition> C1 </condition>
11         activityA
12         <call process="activityC"/>
13       </case>
14     </switch>
15     <switch>
16       <case>
17         <condition> C2 </condition>
18         activityB
19         <call process="activityC"/>
20       </case>
21     </switch>
22   </all>
23 </process>
```

Listing 9 (Discriminator)

```
1  <process name="Discriminator1">
2    <context>
3      <signal name="tns:completedA">
4      <process name="activityA1">
5        ...
6        <raise signal="tns:completedA">
7      </process>
8      <process name="activityA2">
9        ...
10       <raise signal="tns:completedA">
11     </process>
12   </context>
13   <sequence>
14     <all>
15       <spawn process="activityA1"/>
16       <spawn process="activityA2"/>
17     </all>
18     ...
19     <synch signal="tns:completedA">
20     activityB
21   </sequence>
22 </process>
```

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 BPML. However, there is a workaround solution using spawn, raise, and synch activities as shown in Listing 9. In lines 15 and 16 two subprocesses are instantiated. Each of these subprocess instances will raise the signal completedA (line 6 and line 10) just before completion. In Line 19 the process waits for the first one to complete using a synch activity. Although this solution works, it is indirect and requires some bookkeeping. In addition, there are also some issues to be resolved when this construction is embedded in a loop, since the signal is then raised twice for each iteration of the loop, but it is only used once in each iteration. It is also not straightforward to apply this workaround solution in situations where the discriminator is used for inter-thread communication. For example, assume that in Figure 1 the "Synchronize" node which appears between tasks C and D was replaced by a "Discriminator" node, meaning that activity D can start as soon as either activity A or activity C completes. To capture this situation using spawn, raise, and synch, 3 subprocesses would need to be defined:

one containing activity A and B, one containing activity C alone, and one containing activity D alone. One can imagine from this example, that in more complex scenarios, capturing the Discriminator pattern in BPML would not be straightforward.

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. Note that block-oriented languages and languages providing constructs such as "while do", "repeat until" typically impose such restrictions, e.g., it is not possible to jump from one loop into another loop.

Solution, WP10 This pattern is not fully supported in BPML. Although the while, foreach, and until activities allow for various types of 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.[4]

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. The goal of this pattern is to avoid having to join divergent branches into a single point of termination.

Listing 10 (Imp. Termination)

```
<process name="ImpTermination">
  <context>
    <process name="activityA1">
      ...
     </process>
    <process name="activityA2">
      ...
    </process>
  </context>
  <all>
   <spawn process="activityA1"/>
   <spawn process="activityA2"/>
  </all>
  ...
</process>
```

Listing 11 (MI without sync)

```
<process name="MIwithoutSync1">
  <context>
    <property name="tns:set_of_objects"
              type="..."/>
    <process name="activityA">
      ...
    </process>
  </context>
  <foreach select="$tns:set_of_objects">
    <spawn process="activityA"/>
  </foreach>
</process>
```

[4] For a discussion on non-structured cycles that can not be unfolded into structured cycles see [16, 15].

Solution, WP11 Using the spawn activity type it is possible to have divergent branches which do not need to come together into a single point of termination, i.e., there is no need to join concurrent or conditional threads. Therefore, Implicit Termination is supported through the existence of the spawn activity type. Listing 10 shows two spawn activities creating process instances that are never synchronized. Note that the spawn activity type is the only activity type enabling such behavior: all other activity types (including call, all, and while) have a single point of termination.

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 spawn activity embedded in a while, until, or foreach loop. Listing 11 shows an example where for each element in some set of objects an instance of activityA is created.

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 (or with an external process).

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 an all activity. In fact, in BPML there is no need to replicate the activity as a process can be instantiated multiple

times as was shown in Listing 12. The all activity will only complete when all four instances of activityA have completed. The solution becomes more complex if the number of instances to be created and synchronized is only known at run time (WP14). Listing 13 shows a solution using a property called counter to keep track of the number of instances created/completed, and a property called nofi which records the total number of instances to be created. The property nofi needs to be set through an assign statement before the instantiation of subprocesses starts. The first while activity creates instances using a spawn activity. The second while activity records the number of completed instances using a synch activity.

This solution can be extended to deal with pattern WP15. However, it is clear that there is no direct support for WP14 and WP15 because any solution will involve explicit bookkeeping of the number of active instances.

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 choice construct. The semantics of choice, i.e. awaiting for the arrival of an event and depending on the event selecting a prespecified route, captures the key idea of this pattern, namely a choice is not made immediately when a certain point (i.e. the choice activity) is reached, but delayed until receipt of some kind of external trigger. Listing 14 shows the application of the choice activity to choose between two activities based on the first action to take place. There are three types of event-triggered activity types: action, synch, and delay. In Listing 14 both events are of type action. Listing 15 shows another realization of the deferred choice using two events of type delay and one of type synch. The first event catches the signal startA, the second event waits for a specified amount of time (duration), and the third event waits until a specified time (instant).

Listing 12 (MI with sync)

```
<process name="MIwithSync1">
  <context>
    <process name="activityA">
      ...
    </process>
  </context>
  <all>
    <call process="activityA"/>
    <call process="activityA"/>
    <call process="activityA"/>
    <call process="activityA"/>
  </all>
</process>
```

Listing 13 (MI with sync)

```
<process name="MIwithSync2">
  <context>
    <property name="tns:counter"
              type = "xsd:integer">
      <value>0</value>
    </property>
    <property name="tns:nofi"
              type = "xsd:integer" />
    <process name="activityA">
      ...
      <raise signal="tns:completedA">
    </process>
  </context>
  <while>
    <condition>
      $tns:counter <= $tns:nofi
    </condition>
    <spawn process="activityA"/>
    <assign property=""tns:counter">
      <value>$tns:counter + 1</value>
    </assign>
  </while>
  ...
  <assign property="tns:counter">
    <value>0</value>
  </assign>
  <while>
    <condition>
      $tns:counter <= $tns:nofi
    </condition>
    <synch signal="activityA"/>
    <assign property="tns:counter">
      <value>$tns:counter + 1</value>
    </assign>
  </while>
</process>
```

Listing 14 (Deferred Choice)

```
<process name="DeferredChoice1">
  <choice>
    <event>
      <action portType="tns:portX"
              operation="messageA"/>
      activityA
    </event>
    <event>
      <action portType="tns:portX"
              operation="messageB"/>
      activityB
    </event>
  </choice>
</process>
```

Listing 15 (Deferred Choice)

```
<process name="DeferredChoice2">
  <context>
    <signal name="tns:startA"/>
    <property name="tns:duration"
              type="xsd:duration"/>
    <property name="tns:instant"
              type="xsd:dateTime"/>
  </context>
  ...
  <choice>
    <event>
      <synch signal="tns:startA"/>
      activityA
    </event>
    <event>
      <delay duration="tns:duration/>
      activityB
    </event>
    <event>
      <delay instant="tns:instant/>
      activityC
    </event>
  </choice>
</process>
```

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 BPML does not offer direct support for this pattern. There are several ways to work around this problem. One work-around solution uses the deferred choice (i.e. the choice construct in BPML) as proposed in [3]. The drawback of such a solution is its complexity, which grows exponentially with the number of activities to be interleaved. Another possible solution is to place each activity to be interleaved in a separate subprocess, and to have a "routing" process which invokes these subprocesses in turn using the call construct. This "routing" process would need to do some bookkeeping in order to determine which subprocesses

have already been invoked at a given point in time, and which have not. It will also encode the policy to be used in order to determine which activity will be executed next.

WP18 Milestone A given activity 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 has finished and another activity 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 BPML does not provide a direct support for capturing this pattern. Therefore, a work-around solution has to be used. Once again it is possible to construct solutions inspired by the ideas in [3]. For example, it is possible to have one of the processes raising signals like EnableMilestone and DisableMilestone using activities of type raise. The other process which is running in parallel uses a choice activity with events for catching these signals, etc. Clearly, this leads to complex process definitions for simply checking the state in a parallel branch.

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 Both patterns are solved using exceptions. When an exception process is instantiated, all other activities in that context are terminated. Putting the exception process in a certain context can therefore be used to scope the cancellation. Listing 16 shows WP19: activityA is cancelled the moment signal cancelA is received. As a result everything in the context, i.e., just activityA, is terminated. WP20 (Cancel Case) is supported by putting the exception process in the context of the top level process. This way the whole process is cancelled, cf. Listing 17.

Listing 16 (Cancel Activity)

```
<process name="CancelActivity1">
  <context>
    <signal name="tns:cancelA">
  </context>
  ...
  <sequence>
    <context>
     <exception name="cancelA">
      <synch signal="tns:cancelA"/>
     </exception>
    </context>
    activityA
  </sequence>
</process>
```

Listing 17 (Cancel Case)

```
<process name="CancelCase1">
  <context>
    <signal name="tns:cancelCase">
    <exception name="cancelCase">
     <synch signal="tns:cancelCase"/>
    </exception>
  </context>
  ...
</process>
```

4 The Communication Patterns in BPML

In this section we evaluate BPML according to the communication patterns presented in [19]. 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.

4.1 Synchronous Communication

Synchronous communication denotes the situation in which the sender and the receiver coordinate their processing according to their communication. This kind of communication is preferable when the sender needs input from the processing of the receiver, or requires notification of receipt, before it can continue the processing.

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's 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's side.

Solutions, CP1 & CP2 These patterns are captured in BPML by using the action construct, which essentially performs or invokes an operation

specified in WSDL. Both the sender's and the receiver's processes definitions should include an action element for the communication to occur. In the case of pattern CP1, the operation performed by the receiver's action should be of type "request-response" and the operation invoked by the sender's action should be of type "solicit-response". Accordingly, the action elements in both the sender's and the receiver's process should contain at least one input and one output element as sketched in listings 18 and 19.[5] We have omitted most details to avoid getting into WSDL. It is important to note the difference between both listings: In Listing 18 the output comes before the input (solicit-response: the sender first sends an output message, followed by the receipt of an input message) while in Listing 19 the order is reversed (request-response: the receiver first receives an input message and then sends a reply). In the case of CP2, the same solution applies, except that the output of the receiver's action statement, and the input of the sender's action statement are empty. Empty (or irrelevant) replies in two-way WSDL operations are thereby used to capture the notion of acknowledgment.

Listing 18 (Request/Reply: sender)

```
<process name="processA">
    ...
    <action portType=...
          operation=...
          locate = ... >
    <output ... />
    <input ... />
    </action>
    ...
</process>
```

Listing 19 (Request/Reply: receiver)

```
<process name="processB">
    ...
    <action portType=...
            operation=...
            locate=... />
    <input ... />
    <output ... />
     Activities to be executed
     when message is received
    </action>
    ...
</process>
```

If the purpose of the inter-process communication is to create a new instance of a process on the receiver's side, then pattern CP1 can also be captured through the call activity. Indeed, the call activity allows a process to create an instance of another process (specified in BPML) and wait for the completion of the created process instance. The call activity also supports the specification of input and output messages which are then mapped into output and input parameters of the invoked process.

[5] Note that the output element of an action statement corresponds to the input parameters of the WSDL operation invoked by this statement.

Similarly, pattern CP2 can be captured using the spawn activity, which allows a process to create an instance of another BPML process, but without waiting for the created process instance to complete. The spawn activity supports the specification of input parameters for the created process. Note that one can argue whether this is in fact a solution for CP2 since there is no explicit acknowledgement.

Finally, if the communication involves two BPML process instances running within a common context, then pattern CP2 can also be implemented using a raise activity on the sender's side, and a synch activity on the receiver's side. The contents of the message can then be passed as parameters of the generated signal.

CP3 Synchronous Polling Synchronous polling is a form of synchronous communication where a sender communicates a request to a receiver but instead of blocking (waiting for a reply), it continues processing. From times to time, the sender checks to see if the reply has arrived. 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 Since BPML supports parallelism, and since an action activity only blocks the thread in which it executes, this pattern is directly captured by a complex activity of type all with two threads: one for the receipt of the expected response, and another for the sequence of activities that can be performed without waiting for the response (see lines 6–11 in Listing 20). Communication is initiated beforehand through an action statement (lines 3–5 in the listing). To be able to proceed, this action is specified to send data (i.e. it contains at least one output element) but not wait for a reply (i.e. it contains no input element). The receiver process (not shown in the listing) should have an action to receive the request, and another one for sending the reply.

4.2 Asynchronous Communication

In contrast to synchronous communication, asynchronous communication does not require the sender to synchronize processing with communication. The sender sends a message and continues processing immediately. It does not concern itself with how its message is processed by the receiver, nor does it need feedback from the receiver in order to continue. This kind of communication arises when the purpose is information or control transfer.

Listing 20 (Synchronous Polling)

```
1 <process name="SynchronousPolling"
2   <sequence>
3     <action portType=... operation=... locate=...>
4       <output>...</output>
5     </action>
6     <all>
7       <sequence>  actions not requiring the reply ... </sequence>
8       <action portType=... operation=... locate=...>
9       <input> ... </input>
10      </action>
11    </all>
12     action(s) requiring the reply ...
13  </sequence>
14 </process>
```

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 about sending the request (unless this knowledge is stored in properties) 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 shown as part of the solution for CP3. Specifically, the action statement with no input element (lines 3–5 in Listing 20) implements asynchronous message sending, while the action statement with no output element (lines 8–10) implements asynchronous message receiving [5, page 63].

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 in receiving that type of message. **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 (which will also include the newly added customers).

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 BPML. A workaround solution for CP5 is to have the sender process maintain an "expression of interest" list and then send the message to be published individually to each member who has expressed interest in messages of this type. This requires that the process designer implements the bookkeeping of the expressions of interest, the filtering according to message types, and the asynchronous sending of multiple messages. Similar remarks apply to pattern CP6, except that the filtering phase is not needed (the message is sent to all the partners known to the process), and the bookkeeping may be simpler than for CP5 if the partners are known at design time rather than determined at runtime.

5 WSCI

The Web Service Choreography Interface (WSCI) [23] was submitted in June 2002 to the W3C by BEA Systems, BPMI.org, Commerce One, Fujitsu Limited, Intalio, IONA, Oracle Corporation, SAP AG, SeeBeyond Technology Corporation, and Sun Microsystems. The reason we discuss WSCI in this paper is because there is a substantial overlap between BPML and WSCI.

The following simple activities types are supported by WSCI: call, delay, empty, fault, and spawn. In addition, the following complex activities types are supported: all, choice, foreach, sequence, switch, until, and while. Although there are subtle differences between these constructs in BPML and WSCI, their functionalities are comparable. Activity types that are in BPML but not in WSCI are assign, raise, and synch. Instead of the raise and synch activities there is another activity type named join which waits for spawned activities to complete. WSCI does not support the concept of a signal which limits its expressiveness. Nevertheless, all the solutions given in this paper can be realized using WSCI. Some things can be simplified. Consider for example Listing 4. When modeling this in WSCI, we can remove lines 3, 6, 9, and 13 and replace the activities in lines 23 and 24 by two join activities. Other things become more involved in WSCI. For example, to map the solution given in Listing 9 onto WSCI, we can remove lines 6 and 10 but have to replace Line 19 by a choice with two events: one for activityA1 and one for activityA2. Also noteworthy is the fact that it is not possible in WSCI to specify the inputs and outputs of action, call, and spawn activities as in BPML. Finally, the action element in WSCI has a single attribute for specifying both the port type and the operation, whilst in BPML these are specified as two separate attributes.

These examples illustrate that the differences between WSCI and BPML are minimal when it comes to supporting the patterns. Therefore, we do not give a detailed analysis of WSCI but simply refer to BPML. Another reason for not going into details is that the status of WSCI and its relation to BPML are not completely clear. WSCI is positioned as something in-between languages such as BPML and BPEL4WS on the one hand, and WSDL on the other. It is intended to model so-called "choreographed message exchanges". However, there is too much overlap between BPML and WSCI to justify their separation.

6 Conclusion

In this paper a framework based on existing workflow and communication patterns was used for an in-depth analysis of BPML. A summary of the results from the analysis are presented in Table 6. The table also shows a comparison of BPML with BPEL4WS, XLANG, WSFL and WSCI. The ratings for BPEL4WS, XLANG, WSFL in the table are taken from [24]. The ratings for BPML and WSCI are based on the evaluation reported 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 "–" in the table refers to no direct support. This does not mean though that it is not possible to realize the pattern through some workaround solution. Indeed, any of the constructs can be realized using a standard programming language, but this does not imply that such a programming language offers direct support for all of them. 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:

- 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 not all directly supported in the different languages.
- BPEL4WS as a language integrating the features of the block structured language XLANG and the directed graphs of WSFL, indeed supports all patterns supported by XLANG and WSFL. However, this makes BPEL4WS relatively complex, in the sense it provides many strongly overlapping constructs.
- BPML and WSCI offer basically the same functionality.

pattern	*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.

- BPML does not offer direct support for the Multi Choice and Synchronizing Merge while BPEL4WS does. This comes from the fact that BPEL4WS borrows the concept of "dead-path elimination" characteristic of WSFL/IBM MQSeries.
- Each of the languages supports Multiple Instances without Synchronization, Multiple Instances with a Priori Design Time Knowledge, Cancel Activity, and Cancel Case.
- Most of the languages support the Implicit Termination and the Deferred Choice.
- None of the compared languages support arbitrary cycles, although all of them directly support structured cycles.

When comparing BPEL4WS, XLANG, WSFL, BPML and WSCI to contemporary workflow systems [3] on the basis of the patterns discussed

in this paper, they are remarkably strong. Note that only few workflow management systems support Cancel Activity, Cancel Case, Implicit Termination, and Deferred Choice. In addition, workflow management systems typically do not directly support message sending.

The trade-off between block-structured languages and graph-based languages is only partly reflected by Table 6. XLANG, BPML, and WSCI are block-structured languages. WSFL is graph-based. BPEL4WS is a hybrid language in the sense that it combines features from both the block-structured language XLANG and the graph-based language WSFL. Nearly all workflow languages are graph-based and emphasize the need of end-users to understand and communicate process models. Therefore, it is remarkable that of the five languages evaluated in Table 6, only WSFL is graph based. Moreover, in [14] no attention is paid to the graphical representation of WSFL. All the five languages are textual (XML-based) without any graphical representation. This seems to indicate that communication of the models is not considered as a requirement. In this context, we refer to the BPMI initiative toward a Business Process Modeling Notation (BPMN). BPMN is intended as a graphical language that can be mapped onto languages such as BPML and BPEL4WS [6]. Although not reflected by Table 6, the expressiveness of block-structured languages is limited to "well-structured" processes where there is a one-to-one correspondence between splits and joins. This forces the designer using a language like BPML to introduce entities of type signal, raise, and synch which appear to be workarounds to emulate a graph-based language.

Acknowledgments. We sincerely thank Ashish Agrawal and Assaf Arkin from Intalio for the highly valuable comments, feedback, and technical material that they provided to us during the writing of this report [4].

Disclaimer. We, the authors and the associated institutions, assume no legal liability or responsibility for the accuracy and completeness of any information about BPML, WSCI, XLANG, WSFL, or BPEL4WS, contained in this paper. However, we made all possible efforts to ensure that the results presented are, to the best of our knowledge, up-to-date and correct.

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. A. Agrawal and A. Arkin. Business process patterns (private communication). Intalio, San Mateo, CA, USA, 2002.
5. BPML.org. Business Process Modeling Language (BPML). Accessed November 2002 from `www.bpmi.org`, 2002.
6. BPML.org. Business Process Modeling Notation (BPMN), Working Draft (0.9). Accessed December 2002 from `www.bpmi.org`, 2002.
7. Workflow Management Coalition. Terminology and glossary. Document Number WFMC-TC-1011, Document Status - Issue 3.0, February 1999 `http://www.wfmc.org`.
8. Microsoft Corporation. Xlang web services for business process design. Accessed November 2002 from `www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`, 2001.
9. 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.
10. 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`.
11. 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.
12. L. Fischer, editor. *Workflow Handbook 2001, Workflow Management Coalition*. Future Strategies, Lighthouse Point, Florida, 2001.
13. Hillside.net. Patterns Home Page. `http://hillside.net/patterns`, 2000–2002.
14. IBM. Web services flow language. Accessed November 2002 from `www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`, 2001.
15. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows (Submitted)*. PhD thesis, Queensland University of Technology, 2002. `http://www.tm.tue.nl/it/research/patterns`.
16. 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.
17. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, 1999.
18. M. Rosemann and P. Green. Developing a meta model for the Bunge–Wand–Weber ontological constructs. *Information Systems*, 27:75–91, 2002.
19. W.A. Ruh, F.X. Maginnis, and W.J. Brown. *Enterprise Application Integration: A Wiley Tech Brief*. John Wiley and Sons, Inc, 2001.
20. R. Shapiro. A comparison of XPDL, BPML and BPEL4WS (Version 1.4). `http://xml.coverpages.org/Shapiro-XPDL.pdf`, 2002.
21. 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.

22. UN/CEFACT and OASIS. ebXML Business Process Specification Schema (Version 1.01). Accessed November 2002 from `www.ebxml.org/specs/ebBPSS.pdf`, 2001.
23. W3C. Web Service Choreography Interface (WSCI) 1.0. Accessed November 2002 from `www.w3.org/TR/wsci/`, 2002.
24. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Pattern-Based Analysis of BPEL4WS. QUT Technical report, FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002.