Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

Note on Syntactic Details of Split BPEL-D Business Processes

Rania Khalaf

Report 2007/2,

July 18, 2007

Institut für Architektur von Anwendungssystemen

Universitätsstr. 38
70569 Stuttgart
Germany

CR: C.2.4, D.2.7, D.4.4, H.4.1

Summary

The document presents the syntactic details of the generation of process fragments created by the algorithm in [1], which defined the mechanisms for splitting a business process defined using a variant of WS-BPEL. The variant, BPEL-D, replaced BPEL's variable style of data handling with explicit data links. We add new support in this note for syntactically splitting loops and scopes.

Contents

1. Introduction

This note complements the definition of the approach in [1] for generating arbitrary partitions of a business process. It complements it by providing additional details of the syntax generation for the process fragments, and by including new support for syntactically splitting loops and scopes.

2. Defining the partition

A designer creates a decomposition of a process by defining a partition of the set *A* of all *simple* activities in the process. A scope or loop is split by assigning the simple activities inside it to different participants. The user does *not* place the scope or loop itself in a participant.

Recall from [1]: Consider P_n a set of participant names. Every participant, *p*, belonging to the set of participants, *P*, consists of a participant name and a set of one or more activities such that:

$$P \subseteq (P_n \times A) \tag{1}$$

The restrictions on *P* are the following:

$$\forall p \in P, \pi_2(p) \neq \phi \tag{2}$$

$$\forall p_i, p_j : i \neq j \Rightarrow \pi_2(p_i) \cap \pi_2(p_j) = \phi \tag{3}$$

$$\bigcup_{p \in P} \pi_2(p) = A \tag{4}$$

In other words, a participant must have at least one activity, no two participants share an activity or a name, and every simple activity of the process is assigned to a participant.

The only structured activities allowed are <while>, <scope> and <flow>. The <flow> construct is only allowed where BPEL requires exactly one activity, such as inside a <while> or inside the <process> itself. No links may enter or leave it.

Restriction: A ‘receive’ and its corresponding ‘reply’ are disallowed from being placed in different participants.

1.1 Splitting Loops

Any loop whose nested activities are placed in at least one participant becomes a *split loop*. One *fragment* of the loop is in each participant that contains at least one activity nested in the split loop. To split a loop the designer places activities inside the loop in different participants and designates the participant responsible for the loop condition.

In order to encode the fragments responsible for the loop condition, we define L_n to be the set of names of split loops. Then, we define a map L_c :

$$L_c : L_n \rightarrow P_n \tag{5}$$

It associates every split loop name, $l \in L_n$ with its responsible participant name, $L_c(l)$.

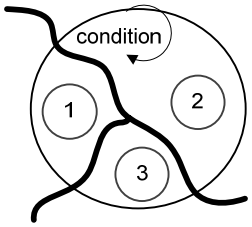

Figure 1: Splitting a loop with three activities amongst three participants. The participant that gets activity '2' gets the fragment responsible for the loop condition.

For example, in the Figure 1, the designer would create three participants, $P = \{p_1, p_2, p3\}$, set their activities, and set the fragment responsible for the loop condition. That would result in the following:

- $\{1\} \subseteq \pi_2(p_1)$,
- $\{2\} \subseteq \pi_2(p_2)$,
- $\{3\} \subseteq \pi_2(p_3)$, and
- $L_c(\text{"}loop1\text{"}) = \text{"}p_2\text{"}$, assuming that the loop is named "loop1" and that $\pi_1(p_2) = \text{"}p2\text{"}$.

Restriction: The loop condition is assigned to one and only one loop fragment.

Restriction: The join condition will be the conjunction of the local join conditions at each fragment.

1.2 Splitting Scopes

Any scope whose nested activities, including activities in fault and compensation handlers, are placed in more than one participant becomes a *split scope*. If the fault handler or compensation handler also has activities in more than one handler, we call it a *split handler*. Similarly, a *fragment* of the scope is in each participant that contains at least one activity nested in the split scope.

Consider the example in Figure 2, in which one splits a scope that has a fault handler. The oval represents the scope, which has two activities *a* and *b* with a link between them. The rectangle represents a fault handler. It has two activities *x* and *y*. The figure shows several of the ways in which this scope can be split between two partners.

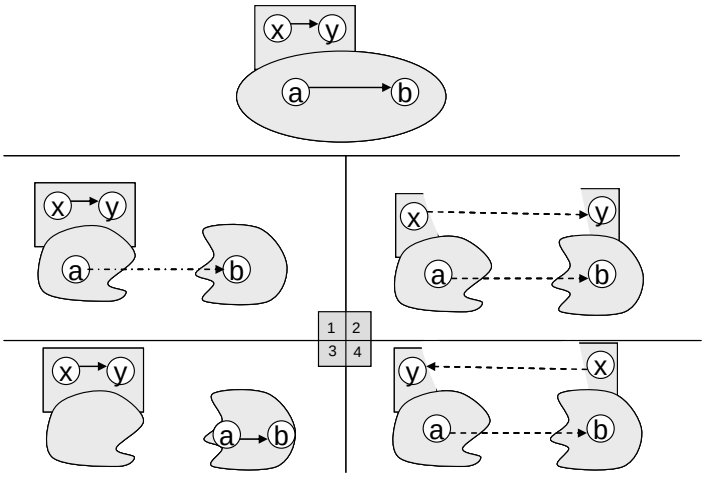

Figure 2: Splitting a scope (oval) with a fault handler (rectangle) into two fragments.

Therefore, to split a scope the designer places the activities inside the scope in different participants. This includes activities inside the fault handler. For example, to split the scope above as shown in the bottom right corner (4) of Figure 2, one would have:

- a partition with two participants, $P = \{p_1, p_2\}$ and
- $\{a, y\} \subseteq \pi_2(p_1)$ and $\{b, x\} \subseteq \pi_2(p_2)$.

Restriction: The join condition will be the conjunction of the local join conditions at each fragment, as is also the case for loops.

1.3 The Rubber Band Effect

In BPEL, scopes and loops must be strictly nested. This restriction is maintained when they are split. It results in a property we define as the 'rubber band' effect [2]. Note that a process itself is treated as a scope in BPEL.

Corollary: If *x* is a split scope or loop, all the ancestor scopes and loops of *x* (including the process itself) must also be split.

Particularly, if *x* has fragments in participant *p1* and participant *p2*, then all ancestor scopes/loops of *x* also have fragments in *p1* and *p2*. Consider that the scope in Figure 2 is in a process *p*. Then, the process is also split between two partners for all the cases shown.

Note that a scope *s'* in a fault handler of scope *s* and having no parent scope in that fault handler is treated as a child of *s* for the purposes of the rubber band effect. Therefore, *s'* cannot be split if *s* is not split.

2 Grouping the fragments

The following XML element that identifies the set of processes making up a split process. The correlation set element will be described in the next section. At deployment time, additional information is needed (minimal wiring), but that is not needed for creating the BPEL fragments and is therefore omitted here.

```
<xs:splitProcess name="xsd:QName" targetNamespace="xsd:anyURL">
  <xs:participant name="xsd:string"
        processDefinition="xsd:anyURI"/>+
  <xs:globalCorrelationSet>?
    <xs:correlationSetName>…</xs:correlationSetName>
  </xs:globalCorrelationSet>
</xs:splitProcess>
```

Table 1: <splitProcess> element tying the process fragments together.

2.1 Identifying Instances

Recall from [1] that there must be one correlation set for the entire process, named the 'global correlation set'. It must be copied it into the messages that transmit dependencies. It is named in the splitProcess definition. It will be used throughout as the instance identifier.

3 BPEL Language Extensions

Several language extensions are needed to indicate to the engine the fact that a loop or scope is split[2]. The syntax for them is shown in [3], using the iaas namespace (*http://www.iaas.uni-stuttgart.de*) as follows:

- On the <process> element:
  - The optional attribute *belongs-to="NCNAME"* that specifies the name of the overall process. The name of the process (fragment) itself will be that of the participant. If a process has this attribute, it also indicates that the process is fragmented.
- On the <scope> element:
  - The attribute *fragmented="yes|no"* that denotes whether the scope is a fragment or not. The default value is "no".
- On the <while> element:
  - The attribute *fragmented="yes|no"* that denotes whether the loop is a fragment or not. The default value is "no".
  - The attribute *is-responsible="yes|no"* that specifies whether this fragment of the loop is responsible for the loop condition. The default value is "no".

4 Details on creating the WSDL and Skeleton BPEL for each participant

At this point, we consider the creation of the process structure and the placement of the original activities in these new processes.

4.1 WSDL definitions

Each participant will have a new WSDL definition. Communication between fragments will occur using uniquely named operations. These operations, which we will refer to as *connecting-operations*, are the ones that will be used to transmit data and control dependencies between the fragments.

One new *portType* gets added to the WSDL definition of a process fragment for each other process fragment that it needs to receive communications from. One new *partnerLinkType* for each of these portTypes is then added to the WSDL definition. This is a portType that the process will provide (i.e.: referred to in the 'myRole' attribute of one of the process's partnerLink elements). An operation, with a unique name within each local process, will be added for every link in the main process where the source and target activities are assigned to different partners. These WSDLs so far only reflect communications between the *newly* created processes; they do not affect operations exposed to or exposed by outside parties. We consider these next.

We will call the portTypes that the main process offered to partners (before the split) as the process's *original portTypes*. We currently restrict splitting these original portTypes: all <receive> activities that refer to the same partnerLink must appear in the same fragment. To reflect the interactions with external clients on the fragments of the process, the corresponding partnerLink from the main process is moved to that fragment's BPEL process.

4.2 Creating the Skeletons of the BPEL Fragments

Each participant gets one BPEL process named after the participant name, and containing a *iaas:fragment-of* attribute containing the name of the main process. Then, the following are added to each local process:

1) A copy of each partnerLink in the main process model that is used by the current fragment (local process).
2) New partnerLink(s) linking it to each of the other process fragments that it interacts with.
3) New variables to handle the data, as detailed below.
4) The global correlation set.
5) Next, handle activities:
   a) Starting at the process itself, and recursing through nested split scopes and loops:
      i) For a fragment of a scope:
         (1) If the scope is *not* the process itself:
            (a) Determine the scope's split parent.
            (b) Place a <scope> activity in the split parent. Give it the same name as that of the scope it is a fragment of. Set the newly defined attribute *fragmented="yes"*.
         (2) If there are activities in the fragment that belong to a fault or compensation handler of the scope, place a new fault or compensation handler on the newly created scope. Place the activity inside it. If there is more than one activity, place them in a top-level <flow> in the handler.
         (3) If there are activities in that scope fragment's body, place them in the new scope. Surround them with a <flow> if there is more than one.
         (4) If no activities are in that fragment's body (for example, in the left participant in case 3 in Figure 2) place an <empty> as the main activity of the new scope.

ii) For a fragment of a loop:
(1) Determine the loop's split parent scope or loop.
(2) Place a <while> activity in the process fragment such that it is nested in the proper parent determined above. If it is in the handler of a split scope then place it in the same handler in the fragment of that scope. Give it the same name as that of the split loop. Set the newly defined attribute *iaas:fragmented="yes"*.
(3) For the fragment that is responsible for the loop condition, place the condition itself. Set the newly defined attribute *iaas:is-responsible="yes"*.
(a) For other fragments, set the condition to 'false()'.
(4) Find the activities in this fragment of the loop that are not nested in a child scope or loop, and place them in the newly created <while>. The recursion will handle nested scopes and loops.
(5) If there is more than one activity in the body of the loop, surround them in a <flow>.

b) Finally, handle links
i) Remove any dangling links by removing the link <source> or <target> element from links whose source or target is in another fragment. Keep the link definition as they will be used in the receiving/sending blocks.
ii) Handle links whose source or target is a split loop or scope activity.

4.3 Placement of Links and Inter-fragment Communication Activities

In this approach, a control or data dependency between fragments is transmitted using a set of BPEL constructs in one fragment that send the needed information, and a corresponding set at another fragment that receives and handles this information. We call the former a 'sending block' and the latter a 'receiving block'. Basically, these blocks contain the activities concerned with inter-fragment communication.

4.3.1 Links from/to split loops and scopes

We place outgoing links from a split scope or loop as follows:

- If the target of the link belongs to a partner A, where A also contains a fragment of the split activity, then partner A gets the entire link.
- If the target of the link does not belong to any partner containing a fragment of the scope, then the choice of partner to own the source of the link is arbitrary and the link is split like any other.

For incoming links, one fragment is the target of all the incoming links, and it is the one on which the original join condition definition, if any, is placed.

4.3.2 'Receiving Block' and 'Sending Block' Placement

Property: a cross-scope link in the main process has to stay a cross-scope link across the partition.

Place the sending block as follows:

(1) If the participant containing the source of the dependency also contains a fragment of the target's immediate scope, place the sending block in the immediate scope of the target,

(2) Otherwise, place it in the first common ancestor scope between the source and the target.

The receiving block is placed in the scope of the target activity, in the fragment in which that activity was placed in the partition.

<receive> activities in receiving blocks

The 'createInstance' attribute is set to 'yes' on 'receive' activities added in receiving blocks targeted at activities that do not have incoming links in the same partition. In other words, links whose source is in the same fragment as they are.

5 Fragmenting Control and Data Dependencies

See [1] for how to split a control and data link and the exact resulting constructs.

Here, we do not repeat that, but instead focus on the new items due to loops and scopes.

5.1 Loops and Scopes in BPEL-D

Recall that in BPEL-D, data dependencies are defined using data links. In this section, we define how such dependencies are propagated between fragments. A new addition to BPEL-D is the ability to specify data links to a compensation handler, and between loop iterations. This is illustrated in Figure 3. The transmission of data is discussed in this section for each of these cases.

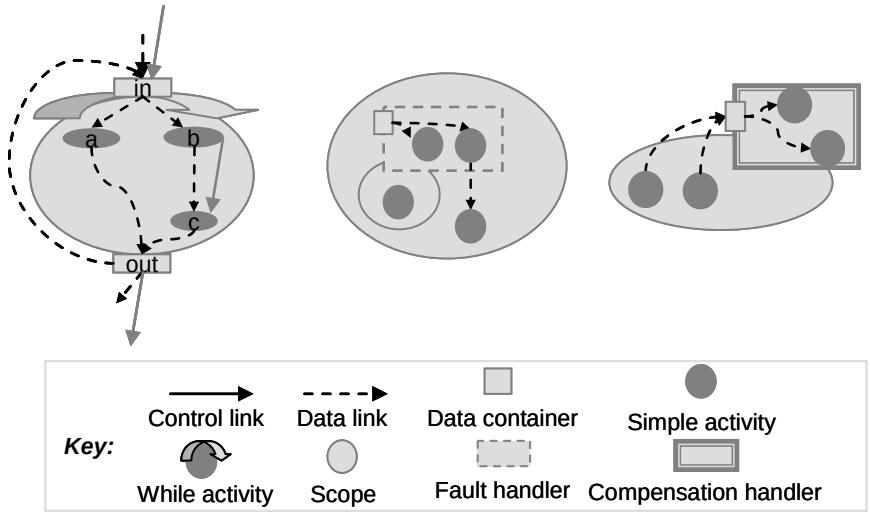

Figure 3: BPEL-D data links and containers for loops and scopes.

5.2 The end-of-split-activity data receiving block

When sending data between two different loop iterations or into a compensation handler, we will use a modified version of the receiving block that will receive and assign the data at the end of a scope or the end of an iteration of a loop. This will enable it to be read in the subsequent iteration or when a compensation handler runs.

The modification is as follows: No link will leave the assign activity. Additionally, a link is placed to the 'receive' of the receiving block from each of the activities, in the fragment of the scope/loop in which the receiving block is being placed, that have no outgoing links. The join condition of the receive is set to ignore the status of all these incoming links.

We call this modified version the *end-of-split-activity data receiving block*.

5.3 Merging sending blocks from the same activity.

Recall from 1] and figure 4 that sending blocks from the same activity can be merged as follows: all scopes are merged into one scope as follows: The body of the new scope will contain a <flow> activity and the scope will have a fault handler for the joinFailure fault. The new flow activity will be the target of the link from A, instead of the <invoke> activities. Every <invoke> in the body of the scopes of the separate receiving blocks is placed in this

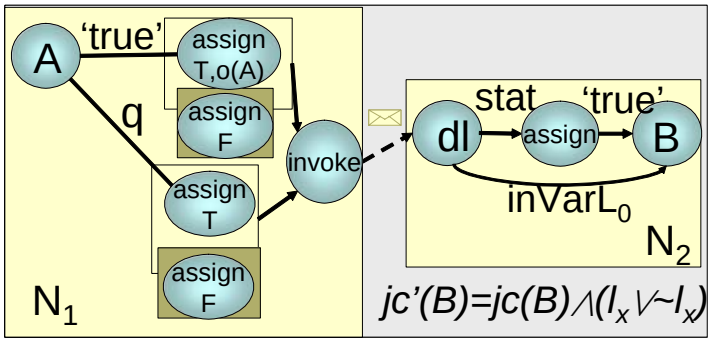

Figure 4: sending both data and control together when splitting a data and a control link that share the same source and target.

5.4 Data between loop iterations

Data needed by subsequent iterations will be sent by the producing activities at the end of an iteration. To ensure this, place one additional control link to the scope of the receiving block from each of the activities in the sender's fragment that are in the loop and have no incoming links. The join condition of the scope is then set to ignore the values of these new links.

An activity A in the body of the loop is determined to produce data needed by an activity B in the body of the loop during the next iteration if:

- There is a data link from A to the output container of the loop, copying data into a set of locations D of that container, and

- There is a data link from the output container of the loop to the input container of the loop, that copies from a subset of D into a set of locations E of the input container, and
- There is a data link from the input container of the loop to the input container of B that copies from a subset of E to a set of locations F in the input container of B.

If all of the above are true and A and B are in different fragments, we need to place data sending and receiving blocks between A and B. We place the receiving block in the fragment of the loop where the receiving activity is. The receiving block used is the modified *end-of-split-activity data receiving block.*

The data sent will be the value of the output container of A and the assign in the receiving block. The assign will perform first the copies that led to E and then the copies that led to F.

Note that one data link from A may result in several sending/receiving blocks. This will occur if it is determined that one of these cross-iteration data link creates writes needed by more than one activity in the next iteration. If this is determined to be the case, the sending blocks are merged as shown below.

5.4.1 Data to the loop condition

This data is needed only by the fragment that is responsible for the condition. It is treated in the same way as data needed between iterations, with the reader being the responsible fragment.

5.4.2 Data to a fault handler

Data to a fault handler in BPEL-D may only come from the faulting activity. This will be sent to the scope fragment with the fault handler through the coordination protocols for handling split scopes. No sending/receiving blocks are created for it.

5.4.3 Data to a compensation handler

Similarly to data across loop iterations, there is some indirection between the source of a data link to a compensation handler and the actual activit(ies) that need to receive the data.

Check the following for every activity, A, that is the source of a data link to the input container of a compensation handler of a split scope and copying into a set of D of locations in that input container.

- The compensation handler is in another fragment or is split between the fragment of A and another, and
- There exists a data link from the input container of the compensation handler to an activity, B, where B is in another fragment and the data link copies from a subset of D into a set of locations E of B's input container.

For every data link source activity A for which the above are true, we need to place a data sending block. Multiple sending blocks from the same activity are merged as shown

above. For every corresponding reading activity B, we place an *end-of-split-activity data receiving block* body of the scope fragment on which the compensation handler containing B is defined.

6 References

1. Khalaf, R. Leymann, F.: Role Based Decomposition of Business Processes Using BPEL. Proc. of the IEEE Int'l Conf. on Web Services (ICWS'06), Chicago, Il, Sep. 2006.
2. Khalaf, R., Leymann, F.: Coordination Protocols for Split BPEL Loops and Scopes. University of Stuttgart, Technical Report No. 2007/01, March 2007.
3. Paluszek, Michael: Coordinating Distributed Loops and Fault Handling, Transactional Scopes using WS- Coordination protocols layered on WS-BPEL services, Diplomarbeit Nr. 2586, 2007