Role-based Decomposition of Business Processes using BPEL

Rania Khalaf
IBM T.J. Watson Research Center
1 Rogers Street, Cambridge, MA 02142, USA
rkhalaf@watson.ibm.com

Frank Leymann
University of Stuttgart, Stuttgart;
IBM Software Group, Boeblingen, Germany
Leymann@iaas.uni-stuttgart.de

Abstract

This paper addresses role-based decomposition of a business process model (based on a subset of WS-BPEL, using explicit data link)s. A mechanism is presented for partitioning a business process so that each partition can be enacted by a different participant. An important goal is to disconnect the partitioning itself from the design of the business process, simplifying the reassignment of activities to different entities. The result is several (compliant) BPEL processes, one for each participant, as well as the information needed to wire them together at deployment time and ensuring correct instance-level connections at runtime. We present details of partitioning and successfully running a sample process with three participants.

1. Introduction

Business process reengineering (BPR) and continuous process improvement (CPI) is a well-established practice in many enterprises today: Execution histories of collections of process instances are periodically analyzed to detect potential deviations from business measures and to derive corresponding improvements by modifying the underlying process model. In case appropriate improvements cannot be achieved by model modifications more dramatic changes are done: Non-competitive parts of the process model are “cut out” and delegated to a third-party that commits to perform these parts within the given business measures (“business process outsourcing” BPO).

BPO sometimes is as drastic as delegating a complete process to a third party, or less drastic by outsourcing only fragments of the process. When outsourcing a fragment the third-party may have the liberty to substitute the cut-out fragment by some other process model as long as the overall business functionality is achieved. Effectively, the cut-out fragment becomes a traditional subprocess [14], i.e. it is used as an encapsulated function. But often, the cut-out fragment interacts with the “rest” of the process model: For example, there may be ordering dependencies or participation in the same unit of work between cut-out activities and activities left behind. In such cases, the fragment cannot be considered a simple subprocess but the fragment and the remaining process model are interwoven by message exchanges and coordination requirement.

Specifying these dependencies is a non-trivial endeavor. To keep the corresponding modeling task simple, the swim-lane approach is a useful abstraction: A swim-lane represents a “partner”, i.e. either the original (i.e. outsourcing) enterprise or a third-party to which a fragment is outsourced. Then, each activity of a process model is simply placed into a corresponding swim-lane without any other impact on the process model (figure 1 (a)). Based on this assignment of activities to swim-lanes both, the corresponding fragments can be derived automatically as well as the interdependencies and corresponding model changes (figure 1 (b)).

As a consequence, process fragmentation can be specified by providing partner partitioning on a given process from the outset, or the activities of an existing process may be partitioned based on choosing responsibility partners later. After deciding which partner must execute which subset of activities, the algorithms in this paper are used to create a separate processes model for each partner such that local control and data dependencies between activities in the original process are translated into (application or control) message flows between the derived process models.

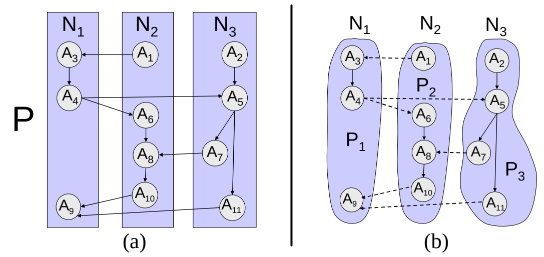


Figure 1. a) Process P with swim-lanes (b) Cutting off local process from P, and reconnecting them using message flows (dashed lines).

For example, in process P in figure 1a, A_1 may be the submission of an order by an actor in organization N_2 (customer) while A_3 may be the processing of the order by an actor in organization N_1 (travel agent), etc. In the spirit of process agility, parts of a process may be outsourced to an external partner due to corporate outsourcing, re-orgs, mergers, or acquisitions. Consequently, swim-lanes may correspond to elements of an organizational structure of another enterprise, making it natural to use the swim-lane notation to model the various partners.

Figure 1b shows local processes P_1, P_2, and P_3 running at N_1, N_2, and N_3 resulting from P. This paper is concerned with defining how these local processes can be derived automatically, leveraging the Web services platform in particular the Web Services Definition Language (WSDL) [22] and the Business Process Execution Language for Web Services (WS-BPEL, BPEL, BPEL4WS) [2]. We use the term ‘participant’ for an agent assigned to a subset of the process’s activities, and ‘swim-lane’ for the graphical representation of a participant.

Another emerging area that motivates enabling different partitions of a business process is that of the mobile workforce, where workers go out in the field or work remotely (excavation, traveling service and support personnel, remote employees, etc.). Early work [5] focuses on device mobility of an already partitioned process model where simple, single-interaction workflows are allowed on the client. We have seen a desire with our industry and university partners to enable a worker to be able to ‘check out’ the part of the work that she needs to do while away (possibly with intermittent connectivity), or a server may send parts of a workflow for enactment to mobile clients. This is a complex, evolving problem of many dimensions that are not the focus of this paper, but to which understanding arbitrary, work-item based, process partitioning is important.

1.1. Approach Overview

Architecturally, the approach presented follows the steps in figure 2. The ‘Transform’ block represents the mechanisms presented in this paper. The transformation takes as input the initial process model, in a modified version of BPEL that we call BPEL-D (described in section 3), its corresponding WSDLs, and a specification *N* of which activities are to be carried out by which participant. Using the mechanisms presented in this paper, the result is the creation of one spec-compliant BPEL (not BPEL-D) process and one WSDL file *per participant*, as well as a simple global wiring definition. The processes, being in vanilla BPEL, can therefore be handled using widely available BPEL tools.

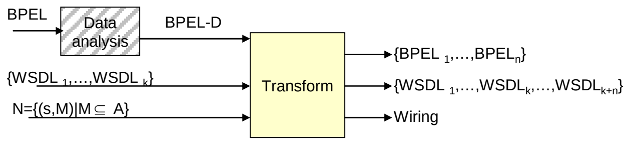


Figure 2. Approach. Shaded region is future work.

The observation that guides the partitioning mechanism is that the creation of the participant processes reduces to a strategy of defining precisely how to ‘break’ existing links that cross swim-lane boundaries, and then wiring together the now isolated components. The breaking of the links, detailed in section 4, is done by changing control and data dependencies into Web services message exchanges. Section 5 provides the necessary information for the wiring the processes together at runtime.

We currently manually perform the transformation, describing results of executing a partitioned process in section 6. The shaded block presents future work described in section 3.

2. BACKGROUND: BPEL

BPEL is an XML workflow language for Web services. Instances of typed connectors (“partnerLinks") provide either one or both of a role that the process implements (myRole) and one that it expects from a partner (partnerRole). The roles refer to defined WSDL portTypes.

Primitive activities include invoking a Web service(invoke), receiving and replying to invocations to operations offered on the process’s WSDL(receive/reply). The <flow> activity is a structured activity used for parallelism. It can have control links that impose ordering on its enclosed activities. An activity that is the target of links can only start once the status of all incoming links is known, it has control from its enclosing structured activity, and its Boolean ‘joinCondition’ evaluates to true (default is the disjunction of the status of incoming links). Once it completes, it fires its outgoing links with the value of evaluating its ‘transitionCondition’.

Data is written to and read from scoped variables. A correlation mechanism is used to route messages to correct instances of a running process. A correlation set refers to a set of properties; each property is aliased to fields in one or more WSDL messages. Incoming messages are checked for the set, which is matched against existing values mapped to running process instances.

If a match is not found, the process definition is checked for the ability to create one based on the message and ‘receive’ activities in the process with the “createInstance” attribute set to “yes”. Once an instance is created and the message reaches its designated ‘receive’ activity, all other ‘receive’ activities in that instance that also have “createInstance” set to “yes” loose their creation ability.

2.1. Scopes, Faults and Dead-Path Elimination

The structured ‘scope’ activity groups related activities. Among other things, it provides them with fault handlers. A ‘scope’ can be the source/ target of links, and links can cross scope boundaries. If a fault is thrown, all activities in its scope stop executing, and a fault handler is looked up by going up the scope hierarchy. Once an appropriate handler is found, the activities of the handler are executed and all links whose sources but not targets belong inside the scope on which the handler is defined fire *negatively*. If the handler completes successfully and does not rethrow the fault, its scope completes and the links that have the scope itself as their source fire with the value of the evaluating their transition condition. If the handler itself faults, then the fault is again propagated up the hierarchy.

The “Dead Path Elimination” technique (DPE) is used in Graph oriented workflow languages (e.g. [12]) to automatically disable activities along a path that is determined to be no longer reachable. In BPEL, it is achieved using fault handling: An activity’s false ‘joinCondition’ causes a ‘joinFailure’ fault to be thrown. If an activity has the "suppressJoinFailure" attribute set to true, the behavior is equivalent to surrounding the activity with a scope having an empty fault handler for joinFailures. Process-wide DPE semantics are achieved by setting this attribute to true on the whole process. [6] details advanced issues on joinFailure.

3. Defining the Main Process

The process model used as input to our transformation is simply a subset of BPEL, but replaces the programming style (scoped variable) data handling in BPEL with data links defined in section 3.1. For easy reference, we call this BPEL-D. There is no fundamentally new concept there: The data dependencies already exist in vanilla BPEL. In BPEL-D we simply make them explicit.

Clearly, there is a need to accept spec-compliant (subset) BPEL as the main input, making BPEL-D an *intermediate* representation. We are working on a BPEL to BPEL-D transform using ideas from [9], which claims a mechanism to derive data dependencies from BPEL using techniques from compiler theory. As we would still go through BPEL-D before fragment creation, the mechanisms in this paper would not be affected once this is introduced.

However, BPEL-D is usable directly since explicit data links are old and common in workflow design (WSFL [13], FDL [14]), supported by commercial products in use for many years such as IBM MQ Workflow.

The subset of BPEL used to define a main process model is:

- Process, with suppressJoinFailure set to 'yes' (DPE on)
- Exactly one correlation set.
- PartnerLinks
- A single top level 'flow' activity, and its links.
- All simple activities, except 'terminate', 'throw', 'compensate', and the form of 'copy' in an 'assign' activity that copies into a process's endpoint references(EPR).
- A 'receive' and its corresponding 'reply' are disallowed from being placed in different participants.

The navigation semantics, from a control point of view, of these processes is well understood and clearly defined both in the BPEL specification and in mathematical mappings of BPEL to lower level formalisms such as [6], [18]. Since we are using only BPEL *flow*, with *suppressJoinFailure*, then navigation semantics are very similar to those of FDL.

3.1. Data Links

Multiple mechanisms were considered for dealing with the issue of sharing data, with trade-offs presented in section 7. As in [13], [14], each activity here gets optional input and output data containers. Consider the output container, *o(A1)*, of activity A1 and the input container *i(A2)* of activity A2. The contents of a container are one tree of data items, whose definition is either an XML Schema Element, an XML Schema Simple Type, or a WSDL Message. XPath's data model [23] explicitly defines the representation of XML data as a tree. In practice, we use XPath to select a data item from a container. A *transitionCondition* on a link has access to the containers of the link's source activity.

An activity's input container is populated by data link(s) from parts of the output containers of other activities; its output container is populated by the activity itself. A data link d(A_1, A_2) specifies a map that assigns parts of the output container of A_1 to parts of the input container of A_2. A control flow dependency *must* be present between any two activities, A_1 and A_2 joined by a data link d.

Consider $o'(A_1)$, the ordered depth-first set of the tree in $o(A_1)$. Consider $i'(A_1)$, the ordered depth-first set of the tree in $i(A_1)$. Note that the depth-first set of a tree is the set of nodes, in order, visited in a depth-first traversal of the tree. We define a data connector map, adapted from [14], where A is the set of activities in the process, and $\wp$(X) denotes the powerset of X:

$$\Delta : A \times A \to \bigcup_{A_1 \in A, A_2 \in A} \wp(o'(A_1) \times i'(A_2))$$

with the following two conditions summarizing the restrictions above:

(i) $\Delta(A_1, A_2) \in \wp(o'(A_1) \times i'(A_2))$

(ii) $\Delta(A_1, A_2) \neq \phi \Rightarrow A_2$ is reachable from A_1

Basic fault handling at the global level can ensure that instances don't hang if a local process fails. Similar to [11], a fault handler can be defined on each local process to catch any fault from that process and notify the other created partners. The latter would have event handlers that terminate the instance upon receipt of such a fault message. We are investigating breaking up fault handling scopes.

4. Partitioning the Process

This section describes the mechanics of the partitioning algorithm. Each of the following sections defines part of the approach. In order to illustrate the mechanics involved in a concrete manner, we provide concrete BPEL snippets from the loan approval process in figure 3, derived from [2], [3]. While small, this sample is a good candidate for illustration because it presents some of the edge cases of the approach.

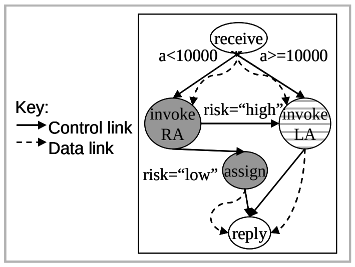


Figure 3. Loan Approval. Shading denotes partition.

This process receives a loan application. If the amount is below a threshold, it sends the application to a risk assessor (RA), otherwise it sends it to a loan approver (LA) whose output is the result. If the assessment was performed and the risk was low, then it sets the loan result (assign); otherwise, it invokes the approver (LA). Then, the result is sent back to the applicant (reply).

4.1. Defining the Participants

The decomposition is created by defining a partition of the set *A* of all activities in the process. Every participant, *n*, belonging to the set of participants, *N*, consists of a name, *s*, and a set of one or more activities, *M*, such that $N = \{n\} = \{(s, M) \mid M \subseteq A\}$ Using $\pi_i(f)$ to represent the i-th projection map, the restrictions on *N* are the following:

$$\forall n_i \in N, \pi_2(n_i) \neq \phi$$

$$\forall n_i, n_j \in N, i \neq j \rightarrow n_i \cap n_j = \phi$$

$$\bigcup_{n \in N} \pi_2(n) = A$$

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

For the loan approval example, three participants were chosen, shown by the different shadings in figure 3:

n_1=("loanApprovalParticipant", (receive, reply))
n_2=("approverParticipant", (invoke-RA, assign))
n_3=("assessorParticipant", (invoke-LA)).

4.2. Preparation: WSDL and Process Skeletons

The WSDL of each participant gets new portTypes for the operations created, and new partnerLinkTypes referring to these portTypes. 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 only reflect communications between the *newly* created processes; they do not affect operations exposed to or exposed by outside parties.

Each participant gets one BPEL process named the same as the participant. PartnerLinks in the main process model used in a fragment (local process) must be copied into the latter's definition. Then, the following are added to each local process:

- New partnerLink(s) linking it to each of the other participants so they may communicate with each other.
- New variables to handle the data, as in section 4.4.
- A correlation set as in section 5.
- For example, the following partnerLink and correlation set were created for the loan approver participant (n2).

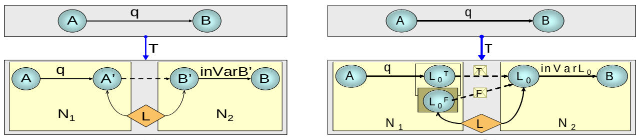
```
<partnerLink name="n1n2"   partnerLinkType="ns1:n1n2LinkType"
                myRole="approver" partnerRole="requestor"/>
     <correlationSet name="name" properties="loandef:name"/>
```

4.3. Passing Control via Messages

The control from a control link broken across participants is flowed via explicitly exchanged messages: sending and receiving activities linked by corresponding partnerLinkTypes.

Consider the control link, *l(A,B,q)* in figure 4 between A and B of transition condition 'q', within the main process model. Partition such that A is in N1 and B is in N2. For *l(A,B,q)*, the operation is added to the definition of the portType used by N2's role in the partnerLink L between N1 and N2. The portType of N1 is not affected (BPEL 'invokes' refer to portTypes of *invoked* services).

The transformation for sending control, conceptually, is shown in figure 4a: The link is transformed by adding a sending activity A' and a control link *l(A,A',q)* at partner N1 and a receiving activity B' and control link *l(B',B,inVarB)* at partner N2. In order to propagate the the status of the original link from A in N1 to B in N2, the value sent from A' to B' must be the status of the link *l(A,A',q)*: either the value of evaluating *'q'* or simply *'false'* due to a fault or DPE. Therefore, A' is used to signal the link status to N2. This signal is received by B', and used as the value of the transition condition of the link between B' and B.



(a) (b)

Figure 4. Splitting a control link across local processes (a)Conceptually, (b) BPEL; dark square is a fault handler

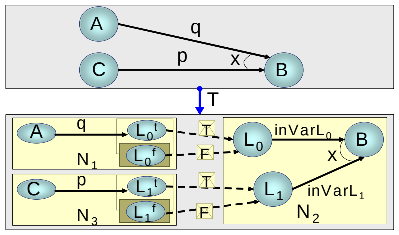


Figure 5. joinCondition x, link sources separated.

Next, we consider the specific issues of mapping directly to BPEL. The relevant BPEL activities are created (Figure 4b), to ensure that that the link status is properly propagated. Clearly, A' is an *invoke* and B' is a *receive*. The only place in BPEL where one can read the status of a link is in the 'joinCondition' of the link's target activity. To propagate the link's status, a scope *S* is used instead of A' in N1. *S* contains an invoke activity L_0^T which is now the target of the link from A, *l(A,* L_0^T*,q)*. L_0^T has *suppressJoinFailure='false'*, and its input variable has value 'true'. *S* has a fault handler *f('joinFailure',* L_0^F*)*, where 'joinFailure' is the name of the fault caught by *f*, and L_0^F is an invoke activity whose input variable has value ‘false’. Both invokes call the same operation on N2. In N2, the signal is received by the ‘receive’ activity L_0., with link *l(Lo,B,true)*. The join condition of B is untouched.

Consider what happens. If A fails the *status*($l(A, L_0^T,q)$) will be negative and the join condition of L_0^T will fail. *Therefore, the status of our original link is just whether or not there was a join failure at activity* L_0^T. Now this information must be passed to N2, which is done by setting ‘suppressJoinFailure’ attribute on L_0^T to false, and using the fault handler on S. At runtime, if the join fails L_0^F propagates a false signal to L_0 ; otherwise, L_0^T propagates a ‘true’ signal L_0. The corresponding behavior reaches B, due to the condition of the control link $l(L_0, B, inVar\ L_0)$.

If an activity B is the target of multiple links, each from a different partner (Figure 5) the join condition is not affected in this approach, as long as the links between the ‘receives’ (L_0 and L_1) and B have the same name as the link in the main model.

The following snippets show the result of breaking up the control link in the loan approval example between *receive* in n_1 and *invokeLA* in n_2:

Sending control (in loanApprovalParticipant.bpel):

```
<scope name="n1n2control-scope">
 <faultHandlers>
  <catch faultName="jfns:joinFailure">
    <invoke partnerLink="n1n2"     portType="apnsf:approverPT"
                operation="n1n2Link" inputVariable="falseAndCorrel" />
  </catch>
 </faultHandlers>
 <invoke name="n1n2control"   suppressJoinFailure="no" partnerLink="n1n2"
        portType="apnsf:loanApprovalPT" operation="n1n2Link"
        inputVariable="trueAndCorrel">
  <target linkName="receive-to-approval"/>
 </invoke>
</scope>
```

Receiving control (in approverParticipant.bpel)

```
<receive name="receive1" partnerLink="n1n2"  portType="apnsf:loanApprovalPT"
           operation="n1n2Link" variable="statusn1n2"   createInstance="yes">
 <correlations> <correlation set="name" initiate="yes"/> </correlations>
 <source linkName="receive-to-approval"
   transitionCondition=   "bpws:getVariableData('statusn1n2', 'status')"/>
</receive>
```

4.4. Sharing Data: Passing Data via Messages.

Using data links reduces the problem of data sharing between partitions to properly breaking up data links that cross swim-lane boundaries. Sharing context data, such as correlation value updates, is circumvented through: (1) the restriction of one correlation set on the main process (reused in all interactions between partners), and (2) deployment-time binding of EPRs.

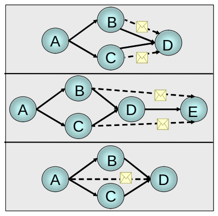
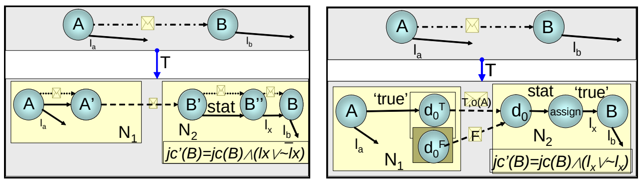


Figure 6. Necessary use cases for data links

The first idea considered is to simply send data using an ‘invoke’ and ‘receive’. However, consider the following case where the source is in a branch of the flow that is not taken (DPE): the data link from *invokeLA* to *reply* in the loan approval example if *a<10000*. *invokeLA* is killed, along with the ‘invoke’ we suggest adding to send *invokeLA’s* data. The suggested ‘receive’ at the partner would hang forever, and the *reply* would never run. Part of our solution is to also send the source’s completion status. If successful, ‘true’ is sent with the data. Otherwise, ‘false’ and ‘null’ are sent (left side figure 7a).

The data at the receiving partner should be passed to the target activity’s input container just before the target executes. In case of a conflict with writes to the same location, the winner is chosen at random. The motivation, also in [14], for allowing such conflicts is to support the very common situation where two paths merge at an activity but only one of them will ever run in any instance (figure 6, top). Consider the loan approval process, which has two such branches joining at the final *reply* and each writing the application result to the response variable. The above approach *without* provisions for either source being killed by DPE could result in the process overwriting the successful branch’s result with ‘null’.

The solution is in ensuring *that the data link from a source activity that doesn’t complete never writes to a value seen by the target of that data link.* In practice, this corresponds to adding a new activity after the “receive” and before the actual target activity (B’’ in Figure 7a, ‘assign’ in 7b). If the original source activity fails, the new activity would be skipped. However, whether or not the target activity will run is a function of its control, not data, links. The join condition of the target activity is therefore modified to be agnostic to the status of the new link from the added activity. This creates the proper behavior supporting all the cases in figure 6.



(a) (b)

Figure 7. Splitting up a data link across local processes (a) Conceptual (b)BPEL, jc’(B) refers to status of lx.

When using BPEL (figure 7b), we use in N1 a scope *S*, containing an invoke activity d_0^T connected to A with *l(A,* d_0^T*, ‘true’).* d_0^T has *suppressJoinFailure=’false’*, and its input variable of two parts with values ‘true’ and the data needed by B. *S* has a fault handler *f(‘joinFailure’*, d_0^F), where d_0^F is an invoke activity whose input variable has ‘false and ‘null’. Both invokes call the same operation on N2. In N2, the message is received by ‘receive’ activity d_0. An ‘assign’ copies the data needed by B into the appropriate parts of its input variable as specified by the data map on *d(A,B)*. Links *l(d_0., assign, inVard$_0$/status)* and lx=*l(assign,B,true)* are added. The join condition of B is changed such that *jc'(B)=jc(B)∧(status(l_x)∨~status(l_x))*.

Here, the fault handling is used to propagate whether 'A' was successful or not instead of control link status. The differences are that additional data is sent in the positive case, an activity buffers the 'receive' from the original target, and the join condition of the target is modified.

Data links that stay within one process are replaced with an 'assign' linked between the data source activity and the data target activity. The join condition of the target is amended as for 'B' above. Variables are added for each activity container and the activities modified to refer to them.

A possible optimization is shown in figure 8 if a control link and data link have *the same source and target*.

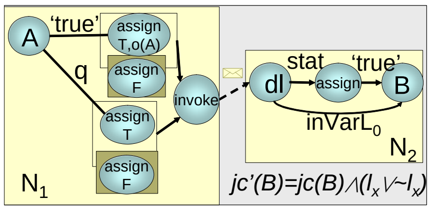


Figure 8. Possible optimization

The following snippets show breaking up the data link in loan approval between *assign* in n_3 and *reply* in n_1.

Sending data (from assessorParticipant.bpel)

```
<scope>
 <faultHandlers>
   <catch faultName="jfns:joinFailure">
     <invoke partnerLink="n1n3" portType="lns:loanApprovalPT"
                  operation="n3n1Data" inputVariable="falseAndCorrel" />
   </catch>
 </faultHandlers>
 <sequence suppressJoinFailure="no">
  <target linkName="data-assign-to-main"/>
    <assign>
      <copy> <from><status  xmlns="" xsi:type="xsd:boolean">true</status></from>
                 <to variable="statusAndApprovalInfo" part="status"/> </copy>
       <copy>  <from variable="trueAndCorrel" part="name"/>
                   <to variable="statusAndApprovalInfo" part="name"/> </copy>
       <copy>  <from variable="approvalInfo" part="accept"/>
                    <to variable="statusAndApprovalInfo" part="accept"/></copy>
    </assign>
    <invoke partnerLink="n1n3" portType="lns:loanApprovalPT"
operation="n3n1Data" inputVariable="statusAndApprovalInfo" />
   </sequence>
</scope>
```

Receiving data (from loanApprovalParticipant.bpel)

```
<receive name="n3n1data" partnerLink="n1n3"
 portType="lns:loanApprovalPT" operation="n3n1Data"
 variable="statusAndApprovalInfoN3" createInstance="yes">
 <correlations> <correlation set="name" initiate="yes"/> </correlations>
 <source linkName="data-assign-to-reply" transitionCondition=
"bpws:getVariableData('statusAndApprovalInfoN3','status')"/>
</receive>
<assign>
  <copy><from variable="statusAndApprovalInfoN3" part="accept"/><to
variable="approvalInfo" part="accept"/></copy>
  <target linkName="data-assign-to-reply"/>  <source linkName="n3data-to-reply"/>
</assign>
<reply name="mainreply" partnerLink="customer"
      portType="apns:loanApprovalPT"   operation="approve"
variable="approvalInfo"
      joinCondition=
"(bpws:getLinkStatus('setMessage-to-reply') or bpws:getLinkStatus('approval-to-
reply'))  and  (bpws:getLinkStatus('n3data-to-reply') or
not(bpws:getLinkStatus('n3data-to-reply'))) and (bpws:getLinkStatus('n2data-to-
reply') or   not(bpws:getLinkStatus('n2data-to-reply')))">
   <target linkName="setMessage-to-reply"/><target linkName="approval-to-
reply"/>
   <target linkName="n3data-to-reply"/> <target linkName="n2data-to-reply"/>
</reply>
```

5. Wiring the Processes Together

For the local processes to successfully work together one must address: (1) the 'receive' activities that can create instances, (2) correlation sets for the newly created 'receive' activities, and (3) connecting the processes together at deployment time through a wiring model (a la WSFL Global Model [13]).

A greedy approach is used: 'createInstance' is set to 'yes' on all 'receive' activities that have no incoming links and that were newly introduced. This is a harmless overestimate: in BPEL only the first createInstance receive to get a message can actually create an instance; second, the direct mirroring of control from the main process means that introduced receive activities that were downstream in the main process but have ended up at the top of a single partner's process will never create an instance. For example, in figure 1b, 'A3' will never happen before 'A9' even though P1, in isolation, seems to allow it. Analysis to reduce the number of receives that can create an instance is possible but has no large effect on performance or execution behavior.

A single correlation set for the entire process is used for instance routing, and reused for inter-partner communication by copying it into the messages. Once split apart, any interactions with outside parties will be routed properly because BPEL correlation tokens are part of the application data that the partner already knows about. The value is set by any of the starting receive activities, and maintained for the lifecycle of each instance. For the loan approval example, the correlation was on the name of the applicant.

Next, the processes need to be wired together. The processes alone cannot be used to relate their partnerLinks to each other's (ie: N3 and N1 talk to the same N2), or deterministically pick a connection for a process that offers/requires the same portType over multiple partnerLinks. Global wiring models can get quite complex. The basic requirements are partnerLink aliasing and locators to set up initial connections. A locator directly provides or resolves to the address of a service (wsdl:port, WS-Addressing EPR, etc.). We provide the following minimal wiring: a set of pair-wise connectors where at least one party is a BPEL process. Each connector, c, consists of:

c =(([process-definition]?, [locator]?, [local-name]?),
([process-definition]?, [locator]?, [local-name]?))

At least one of the three values must be present for each party in *c*: The process-definition if a party is implemented by a BPEL process; the locator if it is invoked by the other party in the connector; the local-name if the party has a local name for the connector (ie: a partnerLink). The model must include the connectors to any partners invoked in the main process model as well as those created by the partitions. BPEL has no generic deployment descriptor, making deployment implementation dependent.

6. Executing the Loan Approval Sample

The loan approval process in Figure 3 is a good candidate to use because it exhibits the salient features of partitioning, (see Figures 5 and 6): forks, joins where the link sources are in another participant than the common target, ('reply' and 'invoke-LA'), data writing to the same location from two possibly exclusive paths. Additionally, it invokes other partner Web services (LA and RA) that are not part of the partition. In the chosen partitioning, the invokes to LA and RA are split among two different participants.

The transforms were manually performed yielding three BPEL processes, the corresponding WSDLs, and the global wiring. Actual snippets of relevant pieces were shown through-out the paper. Static locators were used in the wiring: (WSDL port). These processes were deployed into and executed by BPWS4J [3]. All three processes created instances and ran to completion. Two separate inputs were tried to follow different paths: (1)"name: rania khalaf, amount: 10": runs invokeRA, kills invokeLA, and approves the loan. (2)"name: john doe, amount:1000000": runs invokeRA and invokeLA and denies the loan. BPWS4J does not lock resources to prevent a race on initialization from incorrectly create several instances with the same correlation set, so an artificial delay was introduced between sending the first two starter messages to n_2 and n_3.

7. On Data Sharing

Several options were considered before using data links

- *Shared database, with engine level data replication to synchronize and retrieve data on-demand*. Requires a separate DB, shared by all partners, which is especially a trust concern and possible bottleneck. Variables get written out of band, resulting in invalid BPEL.
- *Data sent with the next control link*: Seems to be the natural approach. Static analysis could determine which variables to send. However, it is not clear what to send if the source of the control link is killed by DPE, and the writer of the data was upstream (A6 needs data written by A3 but A4's join is false).
- *Data links, and broken using a data service at each partner*. Participants invoke the data service to send/retrieve data. This hides the interactions, and causes data overwriting problems when two different activities that write to the same location are on exclusive paths to the same target. Some communities use 'third party copy', sending one's partners the data service's EPR. A variation is (invalid) BPEL with a 'replicate' attribute on activities as a data-retrieval hint to the engine.
- *Data links, broken using WS message exchanges*. The chosen option. The main benefits are a natural model for defining data in simple way that naturally translates into message exchanges, and avoiding the problems above. Drawbacks were: creating extra assign activities, and requiring explicit data link modeling. However, data links are common in workflow languages and can be derived by analyzing data process dependencies.

8. Related Work

Two main trends stand out on interacting processes from a global point of view: Work using conversation languages explicitly modeling interactions as black/grey-boxed message exchanges; and, work on breaking up a single process model into smaller independent processes. In both cases, one will ultimately want to create or derive local processes related to the larger, initial process model.

The most relevant are Muth et al.'s [15], and van der Aalst's and Weske's work in [21]. [15] proposes an approach in which a process model, defined using state and activity charts, is split up so that different partners can enact different subsets of it. They offer several synchronization schemes, starting with one where a TP monitor communicates with all the workflows after every step. Then they optimize this to only synchronize when there is a control dependency between two participants. However, they always use a communication (TP) manager (centralized) which synchronizes the distributed processes, whereas we do not. In our work, all interactions (including state propagation) are directly between the participants. Additionally, our work takes advantage of advances in capabilities for distribution and heterogeneous system support by being natively service-oriented whereas the model in Mentor uses state and activity charts and was created pre-SOA. [21] creates a 'public workflow' that encodes both the logic at each party as well as the message exchanges between the parties. The public workflow is defined as a Workflow Net (based on Petri Nets), in which interactions between the parties are created using a place between two transitions (one from each). From there, the flow is divided into one public part per party. Each public part may be expanded into a private flow for that party, with conformance to the global flow guaranteed if given transformation rules are followed. The work in this paper is different in that the public flow in [21] explicitly models the boundaries between the organizations. In earlier work [20], they provided Message Sequence Charts [17] for defining the interaction protocol in the public flow. Both papers have a heavy focus on checking the correctness of the flows. In our case, we deal with arbitrary partitioning of already defined behavior, and not with specializing private flows. Therefore, the behavior in the resulting local processes will mirror the behavior in the main process model (and inherit any inconsistencies). Another difference here is our main model encodes the complete behavior and different parties are not allowed to change it. In [1], Casati and Discenza define and implement a framework for coordination and interaction between workflows, based on a pub/sub mechanism to wire together processes that can send/receive events. This provides a mechanism for connecting workflows together. That work does not focus on the decomposition of global processes. BPEL already has event handlers and interacts with other processes by sending/receiving messages (but without pub/sub).

In [10],[11], a BPEL process is broken down into several BPEL processes using program analysis and possibly reordering nodes, with each process deployed and executed on a separate machine, in order to maximize the throughput in cases where multiple instances of a process are running concurrently. It is not clear how they deal with propagating DPE across the process fragments. It is unclear what subset of BPEL they support. In [19], a single process model undergoes distributed execution using smart fragmentation and replication techniques. In these works, the partitions are computed by the system, and cannot be chosen at arbitrary points by the designer. In the latter case, the decision of where to execute each activity is dynamic and calculated at runtime based on available resources and actual load. All instance data is sent to each executing unit.

Approaches using conversation languages such as [9] and WS-CDL [12] define a "neutral" view of interactions, not going into the behavior of the implementations at each party or local process derivation rules. Work has been done to create local processes from such languages. [8] provides a top-down approach for observer verification using the state-machine-based conversation model in [9]. [16] provides a semi-automated approach, requiring designer input, for creating BPEL processes from WS-CDL. These are well–suited when one can assume little about the inner workings of each party (blackbox). This is not better or worse than the approach in this paper. The two approaches address different needs: Their first class citizen is the definition and order of messages exchanged between partners; our first class citizen is the order and role-based assignment of action items in a business process. For example, a reassignment of tasks in those approaches would require a redefinition of the main business process.

A semantically rich representation of a 'neutral' view protocol based on roles and commitments is presented in [6]. Message exchanges are represented in the commitment rules. The focus is on protocol refinement and protocol aggregation, enabled by reasoning about the semantics. They provide a way to derive compliant local processes, as nearly complete BPEL processes. Again, we see the same conflict in the purpose of the two works. In their work, interactions are first class. The commitments of a role are the baseline and must be preserved, and it is the work items at the local participants that may change. On the other hand, our baseline is the set of work items, and it is the interactions (and even business commitments) that change based on how a process is partitioned.

In our mechanism, distribution transparency, which has been around for a long time, is used only as far as it is provided through the underlying Web services stack (ie: implementation, (re)location, etc).

What is particularly interesting about this paper is: (1) the model's fragmentation is independent of its design: Instead of designing separate interacting parties and their message exchanges, we define the whole process and fragment it as needed: partner distinction is an add-on. (2) the creation of participant processes is simply the exercise of (automatically/algorithmically) breaking existing links. The behavior of each participant is a reflection of the designer's decisions. With such an intuitive modeling framework, one can understand what is going on without resorting to complex underlying mappings.

9. Conclusion and Future Work

Global process views have often focused on the messages exchanged. Instead, we look at a process as a definition of work items. Distributing it agilely among partners consists of assigning different steps to different partners for execution. The result is several BPEL processes that will run locally at each partner, and use Web services messages to propagate control and data presented in the original model while still maintaining any communications with initial outside entities defined in it.

The paper highlights the challenges in splitting such a process model, while trying to be as close to the original representation as possible, and provides solutions for them. These challenges are: propagating both control and data, propagating DPE and its repercussions, reconciling data conflicts, and wiring the resulting processes together in a consistent manner.

There are several items in our future work agenda. A number of restrictions are placed on the input process model, ('BPEL-D'). Of these, our main focus is on lifting the restrictions on loops, fault handlers, and transactional scopes. Specifically, we are investigating using *coordination protocols* for handling split scopes and loops. We are also interested in enabling one to start with a vanilla BPEL process, and are investigating data analysis techniques on BPEL to create the BPEL-D representation. Then, the latter can be used either directly or as an intermediary format.

10. References

[1] F. Casati, A. Discenza. Supporting Workflow Cooperation Within and Across Organizations, SAC 2000, March 2000, ACM, Como, Italy.

[2] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, S. Weerawarana. Business Process Execution Language for Web Services. May 2003. Online at http://www.ibm.com/developerworks/library/ws-bpel

[3] F. Curbera, M. Duftler, R. Khalaf, N. Mukhi, W. Nagy, ands. Weerawarana. BPWS4J Online at http://www.alphaworks.ibm.com/tech/bpws4j.

[4] F. Curbera, R. Khalaf, F. Leymann, S. Weerawarana. Exception handling in the BPEL4WS language. BPM 2003, LNCS 2678, Eindhoven, the Netherlands, June 2003. Springer.

[5] J. Davis, D. Sow, D. Bourges-Waldegg, C. Jie Gou, C. Hoertnagl, M. Stolze, B. White Eagle, Y. Yin. Supporting Mobile Business Workflow with Commune, Workshop on Mobile Computing Systems and Applications, Washington, USA, April 2007.

[6] N. Desai, A. Mallya, A. K. Chopra, M. P. Singh, Interaction Protocols as Design Abstractions for Business Processes. IEEE Transactions on Software Engineering, Dec. 2005.

[7] R. Farahbod, U. Glässer, M. Vajihollahi, A Formal Semantics for the Business Process Execution Language for Web Services. Proc. of the Workshop on Web Services: Modeling, Architecture and Infrastructure ICEIS05, Miami, FL, May 2005, INSTICC Press

[8] X. Fu, T. Bultan, J. Su. A top-down approach to modeling global behaviors of web services. Requirements Engineering for Open Systems Workshop (REOS 2003), Monterey, California, Sep 2003.

[9] X. Fu, T. Bultan, J. Su. Conversation specification: A new approach to design and analysis of e-service composition. WWW2003, Budapest, Hungary, May 2003.

[10] M. Gowri, N. Karnik. Synchronization Analysis For Decentralizing Composite Web Services. Int. J. Cooperative Inf. Syst. 13(1): 91-119

[11] M. Gowri, S. Chandra, V. Sarkar, Decentralizing execution of composite Web services. OOPSLA 2004: 170-187

[12] N. Kavantzas, D. Burdett, G. Ritzinger (ed), Web Services Choreography Language (WS-CDL1.0), online at http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427

[13] F. Leymann et. al. Web Services Flow Language (WSFL) 1.0. May 2001. Online at http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[14] F. Leymann, D. Roller, Production Workflow, Prentice Hall, 2000.

[15] P. Muth, D. Wodkte, J. Wiessenfels, D.A. Kotz, G. Weikum, From Centralized Workflow Specification to Distributed Workflow Execution, Journal of Intelligent Information Systems, 10(2), 1998

[16] J. Mendling, M. Hafner, From Inter-Organizational Workflows to Process Execution: Generating BPEL from WS-CDL, Proc. of OTM 2005 Workshops. Srpringer LNCS 3762, Agia Napa, Cyprus, November 2005.

[17] E. Rudolph, J. Grabowski, P. Graubmann, Tutorial on Message Sequence Charts (MSC'96), Tutorials of the FORTE/PSTV 1996

[18] K. Schmidt, C. Stahl, A petri net semantic for BPEL4WS - validation and application. Proc. of 11th Workshop on Algorithms and Tools for Petri Nets. (2004)

[19] C. Schuler, R. Weber, H. Schuldt, H.J. Scheck, Peer-to-Peer Process Execution with OSIRIS, ICSOC 2003, Springer LNCS, Trento, Italy, Dec 2003.

[20] W.M.P van der Aalst, Interorganizational workflows: An approach based on message sequence charts and petri nets. Systems Analysis-Modelling-Simulation, 34(3), 1999.

[21] W.M.P. van der Aalst, M. Weske, The P2P Approach to Interorganizational Workflow, Proc. of CAiSE 2001, LNCS volume 2068, Springer, Berlin 2001

[22] W3C, Web Services Description Language (WSDL). Online at http://www.w3.org/2002/ws/desc/

[23] W3C, XPath Language (XPath), Version 1.0, Nov. 1999, Online at http://www.w3.org/TR/xpath