Maintaining Data Dependencies Across BPEL Process Fragments

Rania Khalaf[1], Oliver Kopp[2], and Frank Leymann[2]

[1] IBM TJ Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
rkhalaf@us.ibm.com
[2] University of Stuttgart, Universitätsstr.38,70569 Stuttgart, Germany
{kopp, leymann}@iaas.uni-stuttgart.de

Abstract. Continuous process improvement (CPI) may require a BPEL process to be split amongst different participants. In this paper, we enable splitting standard BPEL - without any extensions or new middleware. We present a solution that uses a BPEL process, partition information, and results of data analysis to produce a BPEL process for each participant. The collective behavior of these participant processes recreates the control and data flow of the non-split process. Previous work presented process splitting using a variant of BPEL where data flow is modeled explicitly using 'data links'. We reuse the control flow aspect from that work, focusing in this paper on maintaining the data dependencies in standard BPEL.

Keywords: Web services, fragments, business process, BPEL.

1 Introduction

When outsourcing non-competitive parts of a process or restructuring an organization, it is often necessary to move fragments of a business process to different partners, companies, or simply physical locations within the same corporation. We provided a mechanism in [8] that takes a business process and a user-defined partition of it between participants, and creates a BPEL [16] processes for each participant such that the collective behavior of these processes is the same as the behavior of the unsplit one. The process model given as input was based on a variant of BPEL, referred to as BPEL-D, in which data dependencies were explicitly modeled using 'data links'.

Our work in this paper aims to study splitting a process specified in standard compliant BPEL, in which data dependencies are – by definition – implicit. We want to do so while maintaining transparency and without requiring additional middleware. Transparency here means that (1) the same process modeling concepts/language are used in both the main process and the processes created from splitting it; (2) process modifications made to transmit data and control dependencies are easily identifiable in these processes, as are the original activities. This enables the designer to more easily understand and debug the resulting processes, and enables tools to provide a view on each process without the generated communication activities.

B. Krämer, K.-J. Lin, and P. Narasimhan (Eds.): ICSOC 2007, LNCS 4749, pp. 207–219, 2007.

Data analysis of BPEL processes returns data dependencies between activities. On a cursory glance, it seems that it would provide enough information to create the necessary BPEL-D data links. In fact, that was the assumption made in [8] when discussing how the approach could be used for standard BPEL. While for some cases that would be true, section 2 will show that the intricacies of the data sharing behavior exhibited by BPEL's use of shared variables, parallelism, and dead path elimination (DPE) in fact require a more sophisticated approach. DPE [16] is the technique of propagating the disablement of an activity so that downstream activities do not hang waiting for it. This is especially needed for an activity with multiple incoming links, which is always a synchronizing join.

Our work explains the necessary steps required to fully support splitting a BPEL process based on business need without extending BPEL or using specialized middleware. A main enabler is reproducing BPEL's behavior in BPEL itself.

2 Scenario and Overview

Consider the purchasing scenario in Figure 1: It provides a 10% discount to members with 'Gold' status, a 5% discount to those with 'Silver' status, and no discount to all others. After receiving the order (A) and calculating the appropriate discount (C, D, or neither), the order status is updated (E), the order is processed (F), the customer account is billed (G), and a response is sent back stating the discount received (H). We will show how data is appropriately propagated between participant processes, created by splitting this example, using only BPEL constructs.

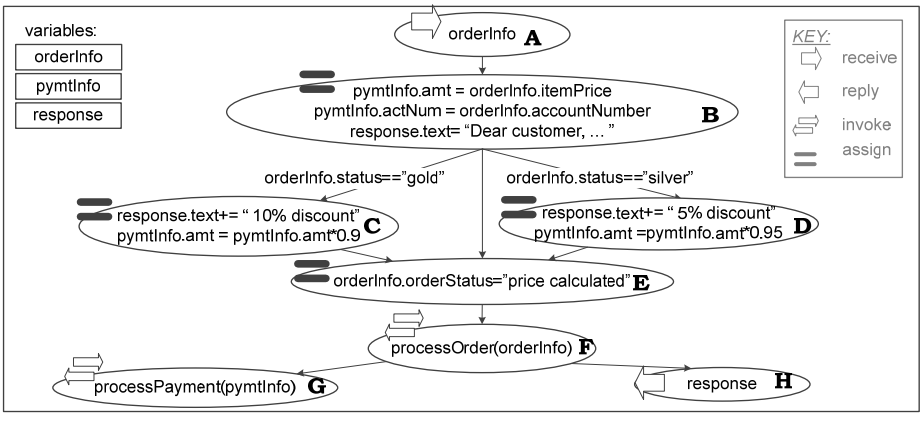

Fig. 1. Sample: an ordering process that provides discounts to Gold and Silver customers

Activity F reads data from A and E. In BPEL-D [8], data links from different activities were allowed to write to the same location of the target activity's input container with a fixed conflict resolution policy of 'random'. Data was considered valid if the activity that wrote it had completed successfully. For cases where data is needed from only one activity (e.g.: A to B, C, D above), data links suffice. However, consider G. It reads *pymtInfo*, whose value of *amt* comes from B, and possibly from C

or D. If one had drawn a data link from all three and the *status* is *gold*, then B and C would have run successfully but not D. There would be a race between B and C's writes of *amt*, when only C should have won. A different resolution policy, such as 'last writer wins', is needed here. However, this cannot be realized using the order of the incoming messages carrying the required data: they may get reordered on the network. Even if synchronized clocks [5] are used, BPEL does not have constructs to handle setting variable values based on time stamps.

A high level overview of the approach we propose is: Given a BPEL process, a partition, and the results of data analysis on that process, we produce the appropriate BPEL constructs in the process of each participant to exchange the necessary data. For every reader of a variable, writer(s) in different participants need to send both the data and whether or not the writer(s) ran successfully. The participant's process that contains the reader receives this information and assembles the value of the variable. The recipient uses a graph of receive and assign activities reproducing the dependencies of the original writers. Thus, any writer conflicts and races in the non-split process are replicated.

In more detail, the steps of our approach are: (1) Create a writer-dependency-graph (*WDG*) that encodes the control dependencies between the writers. (2) To reduce the number of messages, use information about a particular partition: Create a participant-writer-dependency-graph (*PWDG*) that encodes the control dependencies between regions of writers whose conflicts can be resolved locally (in one participant). (3) Create *Local Resolvers* (*LR*) in the processes of the writers to send the data. (4) Create a '*receiving flow*' (RF) in the process of the reading activity that receives the data and builds the value of the needed variable.

Criteria: The criteria we aim to maintain is that conflicting writes between multiple activities are resolved in a manner that respects the *explicit control order,* as opposed to runtime completion times, in the original process model.

Restriction: We assume that data flow follows control flow. We disallow splitting processes in which a write and a read that are in parallel write to the same location. BPEL does allow this behavior, but it is a violation of the Bernstein Criterion [1,12]. The Bernstein Criterion states that if two activities are executed sequentially and they do not have any data dependency on each other, they can be reordered to execute in parallel.

3 Background

Our work builds on [8], for which we now provide an overview. We reuse the parts of the algorithm that create the structure of the processes, the endpoint wiring, and splitting of control links. In order to enable splitting *standard* BPEL (i.e. without explicit data links) we need to specify (1) how data dependencies are encoded without appropriate BPEL extensions (see partition dependent graphs introduced below) and (2) how data dependencies are reflected in the generated BPEL processes by using just standard BPEL constructs.

A designer splits a process by defining a partition of the set A of all its simple activities. Consider P, a set of participants. Every participant, $p \in P$, consists of a

participant name and a set of one or more activities such that: (i) a participant must have at least one activity, (ii) no two participants share an activity or a name, and (iii) every simple activity of the process is assigned to a participant. The result is one BPEL process and one WSDL file *per participant*, as well as a global wiring definition. Figure 2 shows a partition of the process in Figure 1.

$$P1 = \{p_w = (w, \{G\}), p_x = (x, \{A, B, H\}), p_y = (y, \{E, C\}), p_z = (z, \{D, F\})\}$$

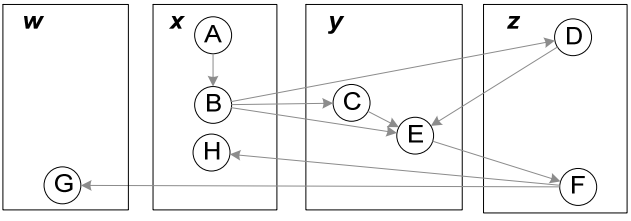

Fig. 2. A partition, P1, of the process in Figure 1

The subset of BPEL constructs that our algorithm, in both [8] and this paper, can actually consume is: (i) processes with 'suppressJoinFailure' set to 'yes' (DPE on), (ii) exactly one correlation set, (iii) any number of partnerLinks, (iv) a single top level 'flow' activity, (v) links, (vi) simple BPEL activities (except 'terminate', 'throw', 'compensate', and endpoint reference copying in an 'assign'). Additionally, (vii) a 'receive' and its corresponding 'reply' are disallowed from being placed in different participants. The single correlation set restriction is to enable properly routing inter-participant messages that transmit control and data dependencies.

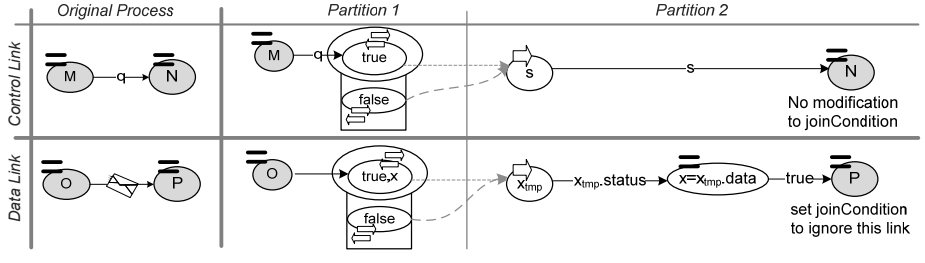

Fig. 3. Summary of link splitting in [8]: the rectangle is a fault handler that catches the BPEL 'joinFailure' fault. Dashed line is a message.

The main idea of [8] is to split of control and data links by adding activities in the participant processes as shown in Figure 3. The top row shows splitting a control link with a transition condition *q* between M and N. To transmit the value of the link to N in participant 2, a scope with a fault handler for 'joinFailure' is used in participant 1. The body of the scope contains an invoke with 'suppressJoinFailure' set to 'no'. The invoke sends 'true()' if the link from M evaluates to true. If not, then that invoke throws a joinFailure, because DPE is off at the invoke (suppressJoinFailure=no). The joinFailure is caught by the fault handler, which contains an invoke that sends 'false()'. Participant 2 receives the value of the link, using a 'receive' activity that is

in turn linked to N with a transition condition set to the received value. This is the status, determined at runtime, of the link from M to N in the original process.

The bottom row shows splitting a data link between P and O. We use, in participant 1, a similar construct to that of a split control link. 'true()' is used as the transition condition and the data is sent if O completes successfully. If O fails or is skipped, the invoke in the fault handler sends 'false()' and an empty data item is sent.

In participant 2, a receiving block is created. Such a receiving block consists of (1) a receive activity receiving the data into a uniquely named variable r, (2) an assign activity copying from *r.data* to the desired variable, and (3) a link between them conditional on *r.status*. The message from participant 1 is written in x_{tmp}. If the status is true, the assign writes the data to *x*. Otherwise the assign is skipped. P must wait for the data but does not depend on whether *x* was written, so the join condition of P is modified to ignore this new incoming link.

4 Related Work

There is a sizable body of work on splitting business processes, covered in more details in [8]. The most relevant using BPEL is [6] where a process is broken down into several BPEL processes using program analysis and possibly node reordering, with the aim of maximizing the throughput when multiple instances are running concurrently. They claim data analysis on BPEL can lead to enough information to easily propagate data. However, they support a limited set of dependencies because they do not handle faults – in particular those needed for Dead-Path-Elimination.

Alternative approaches for maintaining data dependencies across processes are those that do not require standard BPEL, use new middleware, or tolerate fragmentation obfuscation. In the non-BPEL arena, the most relevant in splitting processes are the use of BPEL-D [8] (explicit data links) which is a simpler case of this paper's algorithms, van der Aalst and Weske's P2P approach [19] for multi-party business processes using Petri Nets, and Muth et. al's work on Mentor [15] using State Charts. In the P2P work, a public workflow is defined as a Petri Net based Workflow Net, with interactions between the parties defined using a place between two transitions (one from each). Then, the flow is divided into one public part per party. Transformation rules are provided to allow one the creation of a private flow from a single party's public one. In Mentor, a state and activity chart process model is split so that different partners can enact its different subsets. Data flow in activity charts, however, is explicitly modeled using labeled arcs between activities - much simpler to split than BPEL's shared variables.

For new middleware instead of our approach, one could explore a wide variety of other ways of propagating data. Examples include: shared data using space-based computing [11]; distributed BPEL engines like the OSIRIS system [18]; modifying a BPEL engine to support using data from partially ordered logical clocks [5] along with write conflict resolution rules.

Dumas et. al [4] translate a process into an event based application on a space-based computing runtime, to enable flexible process modeling. While not created for decomposition, it could be used for it: the process runs in a coordination space and is

thus distribution-friendly. The SELF-SERV Project [3] provides a distributed process execution runtime using state-charts as the process model. In both these works, the result is not inline with our goals: the use of a non-BPEL model (UML Activity diagrams, state charts), the requirement of new middleware (coordination space, SELF-SERV runtime), and lack of transparency because runtime artifacts are in a different model (controllers, coordinators) than the process itself.

Mainstream data flow analysis techniques are presented in [14], but BPEL presents special challenges due to parallelism and especially Dead-Path-Elimination. The application of the Concurrent Single Static Assignment Form (CSSA, [10]) to BPEL is shown in [13]. The result of the CSSA analysis is a possible encoding of the use-definition chains, where the definitions (write) of a variable for every use (read) are stated. Thus, the CSSA form can be transformed to provide a set of writers for each reading activity which can be in turn used as one of the inputs to our approach.

We are not aware of any work that propagates data dependencies among fragments of a BPEL process in the presence of dead-path elimination and using BPEL itself.

5 Encoding Dependencies

In this section, we describe how the necessary data dependencies are captured and encoded. The Figure 1 scenario is used throughout to illustrate the various steps. The presented algorithms require the results of a data analysis on the process. In parallel, we are working on such an algorithm (in preparation for publication), optimized for our approach, but whose details are out of scope for this paper. Any data analysis algorithm on BPEL is usable provided it can handle dead path elimination, parallelism, and provide the result (directly or after manipulation) explained below.

One challenging area is in handling writes to different parts of a variable. Our approach handles not only writes to an entire variable, but can handle multiple queries of the form that select a named path (i.e.: (/e)*, called *lvalue* in the BPEL specification) and do not refer to other variables. For example, consider w_1 writes *x.a*, then w_2 writes *x.b*, then r reads *x*; r should get data from both writers and in such a way that *x.b* from w_1 does not overwrite *x.b* from w_2 and vice versa for *x.a*. However, if they had both written to all of *x*, r would need *x* from just w_2. On the other hand, whether an activity reads all or part of a variable is treated the same for the purposes of determining which data to send.

The data algorithm result should provide for each activity *a*, and variable *x* read by *a* (or any of the transition conditions on *a*'s outgoing links), a set $Q_s(a,x)$. $Q_s(a,x)$ groups sets of queries on *x* with writers which may have written to the same parts of *x* expressed in those queries by the time *a* is reached in the control flow. Thus, $\boldsymbol{Q_s(a,x)}$ is a set of tuples, each containing a query set and a writer set. Consider w_1, w_2, and w_3 that write to *x* such that their writes are visible to *a* when *a* is reached. Assume they respectively write to $\{x.b, x.c\}$, $\{x.b, x.c, x.d\}$, and $\{x.d, x.e\}$. Then $Q_s(a,x) = \{(\{x.b, x.c\}, \{w_1, w_2\}), (\{x.d\}, \{w_2, w_3\}), (\{x.e\}, \{w_3\})$. Consider $A_d(a,x)$ to provide the set of all writers that *a* depends on for a variable *x* that it reads. In other words, using $\pi_i(t)$ to denote the projection to the ith component of a tuple t, $A_d(a,x) = \bigcup_{q_s \in Q_s(a,x)} \pi_2(q_s)$.

5.1 Writer Dependency Graph (WDG)

We define a *writer dependency graph (WDG)* for activity *a* and variable *x* to be the flow representing the control dependencies between the activities in $A_d(a, x)$. As we are dealing with the subset of BPEL that is a flow with links, the structure is a Directed Acyclic Graph. We have: $WDG_{a,x} = (V, E)$ where the nodes are the writers:

$$V = A_d(a, x) \subset A$$

As for the edges, if there is a path in the process between any two activities in A_d that contains no other activity in A_d, then there is an edge in the WDG connecting these two activities. Consider a function *Paths(a,b)* that returns all paths in the process between *a* and *b*. A path is expressed as an ordered set of activities. Formally, and where $\{v_1, v_2\} \in V$:

$$(v_1, v_2) \in E \Leftrightarrow |Paths(v_1, v_2)| > 0 \wedge \forall p \in Paths(v_1, v_2), p \cap V = \{v_1, v_2\}$$

A WDG is not dependent on a particular partition. Consider F in Figure 1. $A_d(F, orderInfo) = \{A, E\}$. E is control-dependent on A; therefore, $WDG_{F,orderInfo} = (\{A, E\}, \{(A, E)\})$. Another example is $WDG_{G,pymtInfo} = (\{B, C, D\}, \{(B, C), (B, D)\})$.

To reduce the number of messages exchanged between partitions to handle the split data, one can: (i) use assigns for writers in the partition of the reader; (ii) join results of multiple writers in the same partition when possible. The next section shows how to do so while maintaining the partial order amongst partitions.

5.2 Partitioned Writer Dependency Graph (PWDG)

The *partitioned writer dependency graph* for a given WDG is the graph representing the control dependencies between the sets of writers of *x* for *a* based on a given partition of the process. A PWDG node is a tuple, containing a partition name and a set of activities. Each node represents a '*region*'. A region consists of activities of the same partition, where no activity from another partition is contained on any path between two of the region's activities. The regions are constructed as follows:

1) Place a temporary (root) node for each partition, and draw an edge from it to every WDG activity having no incoming links in that partition. This root node is needed to build the proper subgraphs in step 2.
2) Form the largest strongly connected subgraphs where no path between its activities contains any activities from another partition.
3) The regions are formed by the subgraphs after removing the temporary nodes.

Each edge in the PWDG represents a control dependency between the regions. The edges of the PWDG are created by adding an edge between the nodes representing two regions, r_1 and r_2, if there exists at least one link whose source is in r_1 and whose target is in r_2.

Consider the partition P1in Figure 2. The PWDG for F and variable *orderInfo*, and the PWDG of G and variable *pymtInfo* would therefore be as follows:

$$PWDG_{F,orderInfo,P1} = (\{n_1 = (x, \{A\}), n_2 = (y, \{E\})\}, \{(n_1, n_2)\})$$
$$PWDG_{G,pymtInfo,P1} = (\{n_1 = (x, \{B\}), n_2 = (y, \{C\}), n_3 = (z, \{D\})\}, \{(n_1, n_2), (n_1, n_3)\})$$

Next, consider a different partition, P2, similar to P1 except that C is in p_z with D, instead of p_y, then the PWDG of H and response has only two nodes:

$PWDG_{H,response,P2} = (\{n_1 = (x, \{B\}), n_2 = (z, \{C, D\})\}, \{(n_1, n_2)\})$

If all writers and the reader are in the same partition, no *PWDG* is needed or created. Every PWDG node results in the creation of constructs to send the data in the writer's partition and some to receive it in the reader's partition. The former will be the Local Resolvers (section 5.3). The latter will be part of the Receiving Flow for the entire PWDG (section 5.4).

5.3 Sending the Necessary Values and the Use of *Local Resolvers*

A writer sending data to a reader in another participant needs to send both whether or not the writer was successful and if so, also the value of the data. We name the pattern of activities constructed to send the data a *Local Resolver (LR)*.

If there is only one writer in a node of a PWDG, then: if the node is in the same partition as the PWDG, do nothing. Otherwise, the Local Resolver is simply a *sending block* as with an explicit data link (Figure 3, partition 1).

If there is more than one writer, the algorithm below is used. Basically, conflicts between writers in the same PWDG node, *n=(p,B)*, are resolved in the process of *p*: An activity waits for all writers in *n* and collects the status for each set of queries.

Assume a PWDG for variable *x*, and the reader in partition p_r. Consider *id* to be a map associating a unique string for each set of queries in Q, and *idn* to do the same for each PWDG node. For each PWDG node, *n=(p,B)*, *with more than one writer,* add the following activities to the process of participant p:

```
CREATE-LOCAL-RESOLVER-MULTIPLE-WRITERS(Node n, String x)
1   Q = Q_s(n, x)
2   If p=p_r
3       Add b=new empty, v=new variable, v.name = idn(n)
4       t = idn(n)
5   If |Q| = 1, let Q = {q_s}
6       If p!=p_r
7           b=CREATE-SENDING-BLOCK(x)
8       ∀w ∈ π2(q_s)
9           Add link l = (w, b, true())
10  Else  // more than one query set
11      If p!=pr
12          Add b = new invoke, v = new variable
13          b.inputVariable=v,  b.toPart=("data",x),  b.joinCondition="true()"
14          t=name(v)
15      ∀q_s ∈ Q
16          s = CREATE-ASSIGN-SCOPE(t,qs)
17      Add link l_1 = (s, b, true())
```

CREATE-ASSIGN-SCOPE(String t, Set qs):
Add *s= new scope*
s.addFaultHandler('joinFailure', a_sf=new assign)
s.setActivity(a_s=new assign),
a_s.suppressJoinFailure='no'
a_sf.addCopy(QSTATUS-STR(t,qs),false())
a_s.addCopy(QSTATUS-STR(t,qs), true())
$\forall w \in \pi_2(q_s \in Q)$
Add link $l = (w, a_s, true())$
Return s

QSTATUS-STR (String t, Set qs)
Return t + ".status" + id(qs)

CREATE-SENDING-BLOCK(String x)
Add *s= new scope*
s.addFaultHandler("joinFailure", invf=new invoke)
Add *v = new variable*
invf.inputVariable=v
invf.toPart=("status",false())
s.setActivity(inv=new invoke)
inv.inputVariable=v
inv.toPart=("status",true())
inv.toPart=("data",x)
inv.suppressJoinFailure="no"
Return inv

If the reader is in the same partition as the writers in this node, then we wait with an 'empty' (line 3). If all writers write to the same set of queries, and the node is not in the reader's participant, use a sending block. Create a link from every writer to b, which is either the empty or the sending block's invoke (line 6-9). Figure 4 shows such use of an invoke for C and D in partition y.

If there is more than one query set, the status for each one needs to be written. If the reader is in another participant we create an invoke that runs regardless of the status of the writers (line 11-16). For each query, use a structure similar to a sending block (i.e.: scope, fault handler) to get the writers' status (line 16), but using assigns rather than invokes. The assigns write true or false to a part of the status variable corresponding to the query. Create links from each writer of the query set to the assign in the scope. Create a link (line 17) from the scope to either the empty from line 3 or the invoke from line 12.

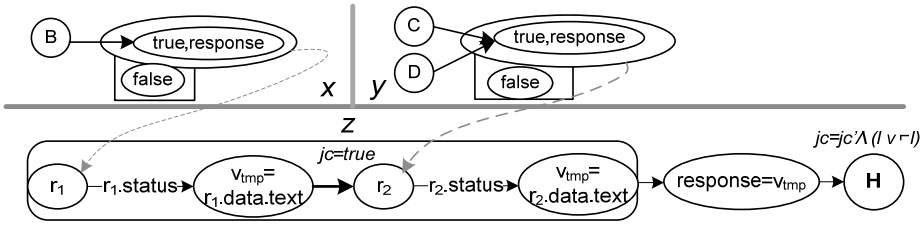

Fig. 4. Snippets from processes from the process in Fig. 1 w/ partition P2

5.4 Receiving Flow (RF)

A Receiving Flow, for a reader *a* and variable *x,* is the structure created from a PWDG that creates the value of *x* needed by the time *a* runs. It contains a set of receive/assign activities, in *a*'s process, to resolve the write conflicts for *x.*

Consider p_r to be the reader's partition, and *G* to be the PWDG from *WDG(a,x)*. An RF defines a variable, $v_{tmp,}$ whose name is unique to the RF. The need for v_{tmp} is explained in the next section. A receiving flow is created from *G* as follows:

```
CREATE-RF(PWDG G)
1   Create a <flow> F
2   For all  n = (p, B) ∈ π1(G)
3       PROCESS-NODE(n)
4   For all   e = (n1, n2) ∈ π2(G)
5       For all   d ∈ ea_n1
6           Add a link l=(d, ba_n2 true())
7   Add a_f=new assign
8   a_f.addCopy(v_tmp, x)
9   Add links l_f = (F, a_f, true()) and l_r = (a_f, a, true())
10  joinCondition(a) = joinCondition(a) ∧ (l_r ∨ ¬l_r)
```

Create a flow activity (line 1). For each node (line2-3), we will add a block of constructs to receive the value of the variable and copy it into appropriate locations in a temporary, uniquely named variable $v_{tmp.}$ Link the blocks together (line 4-6) by connecting them based on the connections between the partitions, using *ba* and *ea* as the first and last activities of a block, respectively. The subscript is used to identify which node's block they are for (i.e.: ea_{n1} is the *ea* set created in PROCESS-NODE(n1)). Link the flow to an assign (line 7-9) that copies from v_{tmp} to *x*. Link the assign to *a* and modify *a*'s join condition to ignore the new link's status (line 10).

```
PROCESS-NODE(Node n) //recall  n=(p,B)
1     Q = Q_s(n, x),  ea = ∅
2     //All activities added in this procedure are added to F
3     If p = p_r
4         If |Q| = 1,  let  Q = {q_s}
5             ba =new assign,
6             For all q ∈ q_s,  ba.addCopy(v_tmp.q, x + "." + q)
7             If |B| = 1, Add link  l_0 = (b ∈ B, ba, true())
8             ea ← ea ∪ {ba}
9         Else
10            ba =new empty
11            For all q_s ∈ Q
12                CREATE-Q-ASSIGN(q_s, "x", QSTATUS-STR(idn(n), q_s))
13        If |B| ≠ 1
14            Add link l_0 = (em, ba, true()), where em=empty from LR
15        joinCondition(ba)=status(l_0)
16    Else  //p is not pr
17        Add rrb= new receive,   joinCondition(rrb)=true(), rrb.variable=r_i
18        ba = rrb    //note that ea will be created in lines 20,23
19        If |Q| = 1,  let Q = {q_s}
20            CREATE-Q-ASSIGN(q_s, "r_i.data", "r_i.status")
21        Else
22            For all q_s ∈ Q
23                CREATE-Q-ASSIGN(q_s, "r_i.data", QSTATUS-STR(r_i, q_s))
```

And the creation of the assigns for each query is as follows:

CREATE-Q-ASSIGN(Set qs, String var, String statusP)
- c.1 Add *act =new assign*
- c.2 $ea \leftarrow ea \cup \{act\}$
- c.3 Add link $l = (ba, act, statusP)$
- c.4 **For all** $q \in q_s$, $act.addCopy(v_{tmp}.q, var + "." + q)$

For each node: If the node is in the same participant as *a* and has, one query set, add an assign copying from the locations in *x* to the same locations in v_{tmp} (line 3-6). If the node has only one writer, link from the writer to the assign (line 7). If it has more than one writer, an 'empty' was created in the Local Resolver (LR), so link from *that* empty to the assign (line 13-14). If the node has more than one query set, create an empty instead of an assign and (line 11-12) create one assign per query set. Create links from the empty to the new assigns whose status is whether the query set was successfully written (line c.3). Add a copy to each of these assigns, for every query in the query set, from the locations in *x* to the same locations in v_{tmp} (line c.4). Then, (line 15) set the joinCondition of the empty or assign to only run if the data was valid.

If the node is another partition, create a receiving block (line 17) instead of an assign. Set the joinCondition of the receive to true so it is never skipped. Again copy the queries into a set of assigns (line 19-23).

Figure 5 shows two examples for partition P1 of our scenario. The top creates *pymtInfo* for G: The value of *amt* may come from B,C, or D but *actNum* always from B. The bottom creates *orderInfo* for F. Notice how A's write is incorporated into the RF even though A and F are in the same participant.

Note that receiving flows reproduce the building of the actual variable using BPEL semantics. Thus, the behavior of the original process is mirrored, not changed.

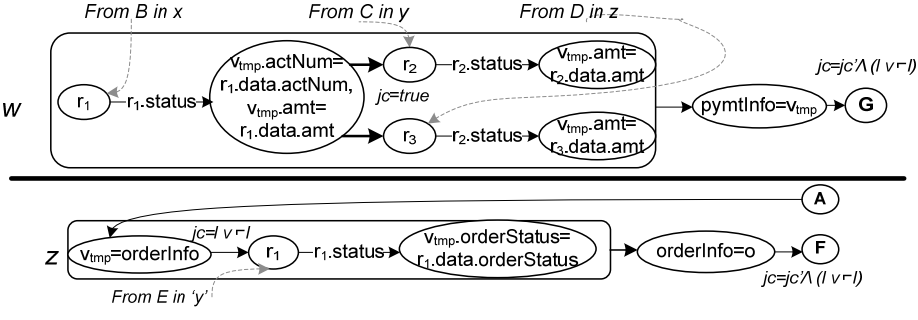

Fig. 5. Two RFs using partition P1. Top: *pymtInfo* to G in w. Bottom: *orderInfo* to F in z.

Multiple RFs and the trailing assign

Consider multiple readers of the same variable placed in the same participant Each RF writes to its local temporary variable, and only copies to the shared variable when the trailing assign at the end of the RF is reached. This temporary variable is used so that messages arriving out of order or to multiple RFs concerned with the same variable

do not incorrectly overwrite each other's values. The algorithms require that the original process adhere to the Bernstein Criterion; otherwise, one cannot guarantee that RFs with overlapping WDGs don't interfere with each other's writes.

6 Conclusion and Future Work

We provided an algorithm for splitting BPEL processes using BPEL itself for proper data communication between participants; furthermore, splits are transparent, i.e. it is clear where the changes are and they are done in the same modeling abstractions as the main process model. This has been achieved by use of LRs and RFs as long as the original process respects the Bernstein Criterion. If not, one would have to take into consideration actual completion times of activities, which goes beyond BPEL's capabilities. Having placed the activities that handle data and control dependencies at the boundaries of the process and used naming conventions on the operations, we enable graphical/text-based filters to toggle views between the activities of the non-split process and the 'glue' activities we have added. The difficulty in maintaining data dependencies in BPEL is due to unique situations (section 2), such as the ability to 'revive' a dead path with an 'or' join condition, resulting from dead-path elimination and parallelism.

The complexity, in number of messages exchanged and activities added, depends on two factors: The amount of inter-participant data dependencies and the quality of the data algorithm. Poor data analysis leads to larger writer sets. At most one 'invoke' is added for each PWDG node, so the number of message exchanges added is at most the *total* number of PWDG nodes: $O(n_{PWDG}) = O(n)$. For the number of added activities, the upper bound is quadratic, $O(n_{PWDG} \times max(|Q_s|)) = O(n^2)$.

Our future work includes optimizations such as merging overlapping RFs and targeted data analysis. A first step for optimization is the application of the work presented in [2,7,17] to BPEL. Also of interest is enabling transmitting (some) data dependencies for split loops and scopes, whose control is split in [9], by grafting activities in the participant processes. Other directions include effects of toggling DPE, and using the information of whether a split is exclusive or parallel by analyzing link transition conditions. Another aspect is to provide an implementation of the algorithm and perform quantitive evaluation on the process fragments it outputs.

Acknowledgement. Jussi Vanhatalo, for suggesting local resolution with one invoke, inspiring the current *Local Resolver*. David Marston, for his valuable review.

References

1. Baer, J.L.: A Survey of Some Theoretical Aspects of Multiprocessing. ACM Computing Surveys 5(1), 31–80 (1973)
2. Balasundaram, V., Kennedy, K.: A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations. In: SIGPLAN Notices Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, vol. 24(7), pp. 41–53 (1989)

3. Benatallah, B., Dumas, M., Sheng, Q.Z.: Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services. Journal of Distributed and Parallel Databases 17(1), 5–37 (2005)
4. Dumas, M., Fjellheim, T., Milliner, S., Vayssiere, J.: Event-based Coordination of Process-oriented Composite Applications. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 236–251. Springer, Heidelberg (2005)
5. Fidge, C.: Logical Time in Distributed Computing Systems. IEEE Computer 24(8), 28–33 (1991)
6. Gowri, M., Karnik, N.: Synchronization Analysis for Decentralizing Composite Web Services. International Journal of Cooperative Information Systems 13(1), 91–119 (2004)
7. Kennedy, K., Nedeljkovi'c, N.: Combining dependence and data-flow analyses to optimize communication. In: Proceedings of the 9th International Parallel Processing Symposium, pp. 340–346 (1995)
8. Khalaf, R., Leymann, F.: Role Based Decomposition of Business Processes Using BPEL. In: ICWS 2006. Proceeding of the IEEE 2006 International Conference on Web Services, Chicago, Il, pp. 770–780. IEEE Computer Society Press, Los Alamitos (2006)
9. Khalaf, R., Leymann, F.: Coordination Protocols for Split BPEL Loops and Scopes. University of Stuttgart, Technical Report No. 2007/01 (March 2007)
10. Lee, J., Midkiff, S.P., Padua, D.A.: Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In: Huang, C.-H., Sadayappan, P., Sehr, D. (eds.) LCPC 1997. LNCS, vol. 1366, pp. 114–130. Springer, Heidelberg (1998)
11. Lehmann, T.J., McLaughry, S.W., Wyckoff, P.: T Spaces: The Next Wave. In: HICSS '99. Proceedings of the 32nd Hawaii International Conference on System Sciences, Maui, Hawaii (January 1999)
12. Leymann, F., Altenhuber, W.: Managing Business Processes as Information Resources. IBM Systems Journal 33(2), 326–348 (1994)
13. Moser, S., Martens, A., Görlach, K., Amme, W., Godlinski, A.: Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis. In: IEEE International Conference on Services Computing (SCC 2007), pp. 98–105. IEEE Computer Society Press, Los Alamitos (2007)
14. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
15. Muth, P., Wodkte, D., Wiessenfels, J., Kotz, D.A., Weikum, G.: From Centralized Workflow Specification to Distributed Workflow Execution. Journal of Intelligent Information Systems 10(2), 159–184 (1998)
16. OASIS: Web Services Business Process Execution Language Version 2.0, (April 11, 2007), Online at http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
17. Sarkar, V.: Analysis and Optimization of Explicitly Parallel Programs Using the Parallel Program Graph Representation. In: Huang, C.-H., Sadayappan, P., Sehr, D. (eds.) LCPC 1997. LNCS, vol. 1366, Springer, Heidelberg (1998)
18. Schuler, C., Weber, R., Schuldt, H., Scheck, H.J.: Peer-to-Peer Process Execution with OSIRIS. In: Orlowska, M.E., Weerawarana, S., Papazoglou, M.M.P., Yang, J. (eds.) ICSOC 2003. LNCS, vol. 2910, pp. 483–498. Springer, Heidelberg (2003)
19. van der Aalst, W.M.P., Weske, M.: The P2P Approach to Interorganizational Workflow. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 140–156. Springer, Heidelberg (2001)