Maintaining Data Dependencies Across BPEL Process Fragments

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

[1]IBM TJ Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
rkhalaf@us.ibm.com

[2]Institute of Architecture of Application Systems, University of Stuttgart, Germany
{kopp,leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@article{KKL08,
  author    = {Rania Khalaf and Oliver Kopp and Frank Leymann},
  title     = {Maintaining Data Dependencies Across
               {BPEL} Process Fragments},
  journal   = {International Journal of
               Cooperative Information Systems (IJCIS)},
  year      = {2008},
  volume    = 17,
  number    = 3,
  pages     = {259--282},
  doi       = {10.1142/S0218843008001828},
  publisher = {World Scientific}
}
```


See also http://www.worldscinet.com/ijcis/17/1703/S0218843008001828.html

International Journal of Cooperative Information Systems

MAINTAINING DATA DEPENDENCIES ACROSS BPEL PROCESS FRAGMENTS

RANIA KHALAF
IBM TJ Watson Research Center, 19 Skyline Drive
Hawthorne, NY 10532, USA

OLIVER KOPP, FRANK LEYMANN
IAAS, Universität Stuttgart, Universitätsstr. 38
70569 Stuttgart, Germany

Received (Day Month Year)
Revised (Day Month Year)

Continuous process improvement (CPI) may require a BPEL process to be split amongst different participants. In this paper, we enable splitting standard BPEL – without requiring any new middleware for the case of flat flows. The solution also supports splitting loops and scopes that have compensation and/or fault handlers. When splitting loops and scopes, we extend existing Web services standards and frameworks in a standard compliant manner in order to support the resulting split control (not data) between the fragments. Data dependencies, however, are handled directly using BPEL constructs placed in the fragments even for split loops and scopes.

We present a solution that uses a BPEL process, partition information, and results of data-flow 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 as well as the control flow aspect from our work on splitting loops and scopes, 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. In Ref. 1 we provided a mechanism that takes a business process and a user-defined partition of it between participants, and creates a BPEL^2 process 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 going as far as possible without requiring additional middleware. New middleware is only used for control dependencies and only for the advanced cases of splitting loops and fault/compensation handling scopes. Even in that case, it is added as an extension in a modular, standards compliant manner. 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.

Data-flow 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 our previous work[1] when discussing how the approach could be used for standard BPEL. While for some cases that would be true, Sec. 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[2] 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: An activity must wait until all its incoming links have fired, upon which time it evaluates a 'joinCondition'. This condition is in terms of the status of the incoming links and its default value is an 'or'. If the join condition evaluates to true, the activity runs; otherwise, it gets disabled and fires all its outgoing links with the value 'false'. In BPEL, this behavior occurs through the use of the joinFailure built-in fault which is thrown if an activity's join condition evaluates to false. The activity may choose to suppress this fault by setting (either at the activity or process level) the attribute 'suppressJoinFailure' to true. The result is similar to surrounding the activity with a scope containing an empty fault handler for the joinFailure fault. By empty fault handler, we mean a fault handler whose body contains an 'empty' activity. More information on BPEL fault propagation and DPE in particular is provided in Ref. 3.

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

2. Scenario and Overview

Consider the purchasing scenario in Fig. 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,

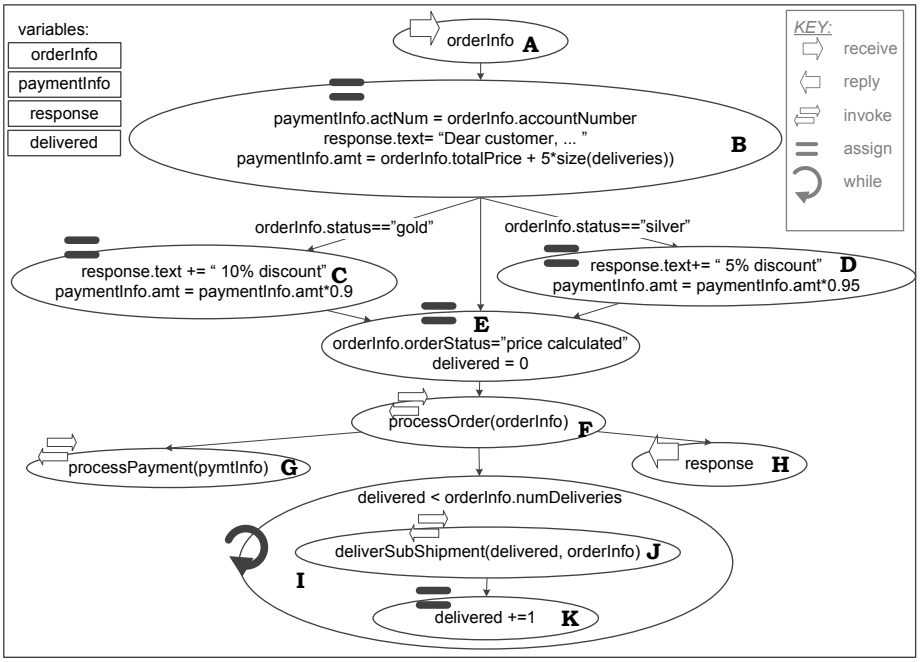

Fig. 1. Sample: an ordering process that provides discounts to Gold and Silver customers.

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). The deliveries are made in more than one shipment as shown by the loop (I). Each shipment is processed (J) by a service that takes the order information and a counter of how many shipments have already been sent out. The number of shipments is updated (K) and the loop iterates until all shipments are made. We will show how data is appropriately propagated between participant processes, created by splitting this example, using BPEL constructs.

Activity F reads data from A and E. In BPEL-D[1], 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[4] 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-flow 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[5,6]. 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

This paper builds on our prior work presented in Ref. 1, 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 (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.

Scopes and loops are split by assigning the activities in them to different partners.

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

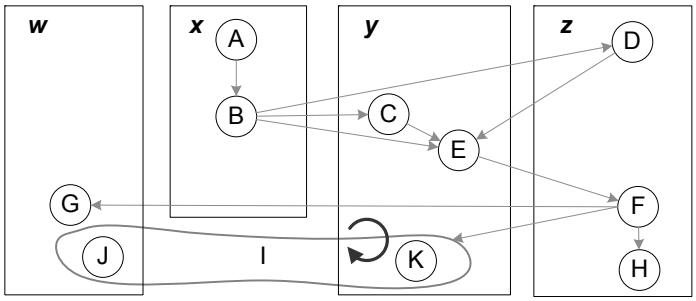

Fig. 2. A partition, P1, of the process in Fig. 1

The designer does not assign the scope or loop itself in a participant. The designer must also associate, for every split loop, one participant to be responsible for evaluating the loop's condition. This is done using the map L_c that associates every split loop name, with its responsible participant name.

The result is one BPEL process and one WSDL file *per participant*, as well as a global wiring definition. Fig. 2 shows a partition of the process in Fig. 1.

The subset of BPEL constructs that our algorithm can consume is:

- Processes with 'suppressJoinFailure' set to 'yes' (DPE on)
- Exactly one correlation set, to enable properly routing inter-participant messages that transmit control and data dependencies
- Any number of partnerLinks
- The supported structured activities are: 'flow', 'while' 'scope'.
  - Links are allowed, but not from/to the boundary of a 'flow' activity.
  - Scopes may have compensation and/or fault handlers but not event or explicit termination handlers.
  - Loops and scopes must be uniquely named within a process.
  - Data written in a split loop and needed after the loop is restricted to be only for whole variables.
  - The join condition of a split loop or scope is restricted to a conjunction of the local join conditions of each fragment of a split loop or scope.
  - Compensation is a recovery mechanism, and thus must not fail. Compensation handlers data behavior is that of BPEL 1.1: Compensation handlers read data from a snapshot of the process state taken when their associated scope instance completed and do not write data visible outside of the compensation handler itself.
  - A fault handler is restricted to only read from the faulting activity but may write to any variable.

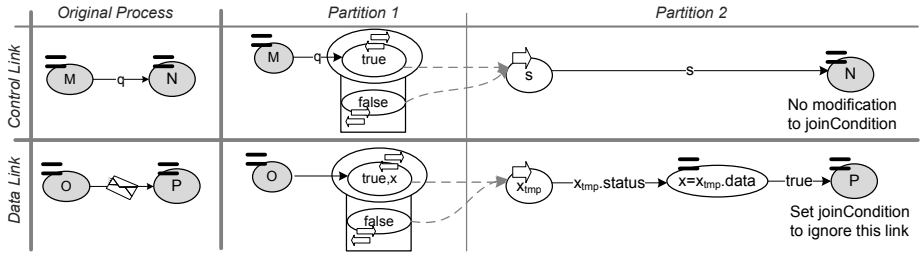

Fig. 3. Summary of link splitting presented in Ref. 1: the rectangle is a fault handler that catches the BPEL 'joinFailure' fault. Dashed line is a message.

- Simple BPEL activities (except 'terminate', 'throw', 'compensate', and endpoint reference copying in an 'assign').
- A 'receive' and its corresponding 'reply' are disallowed from being placed in different participants.

The main idea of the approach of our prior work[1] is to split of control and data links by adding activities in the participant processes as shown in Fig. 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 O and P. 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.

Having explained how explicit control and data are split, we move to the two other types of dependencies: implicit control dependencies found in loops and scopes

and implicit data dependencies found in BPEL shared variables. For the former, we provide a brief overview of our solution. On the other hand, the latter is the focus of the main contributions of this paper and will be handled throughout the remaining sections.

3.1. *An Overview of Splitting Implicit Control Flow: Loops and Scopes*

An initial idea for handling the control dependencies resulting from splitting scopes and loops was to use variations on the above in-fragment sending and receiving patterns. However, this drastically increases the amount of activities and the complexity of the fragments: one part of a split loop may lack enough information to determine when it may iterate again or what the value of the loop condition is. In the case of a split scope, one part needs to be notified if another part has thrown a fault and that needs to be handled before it can complete. In some cases this becomes intractable because no fragment alone has all the necessary information about the original non-split process, such as is the case for determining default compensation order of nested scopes which depends on the control paths between these scopes in the unsplit process. An example of where it would be possible yet complex to use sending and receiving patterns is in splitting loops: if one fragment of a loop completes then it has to wait for all others to complete and also notify them that it itself has completed. Therefore, it is worth bringing in additional capabilities than what the BPEL language can provide if one needs to split the control behavior of BPEL loops and scopes.

This is the point that using only BPEL itself is not adequate. However, a goal of this work was to minimize the use of specialized middleware and not require replacing one's existing platforms and systems. We solve this problem in a manner that is compliant with the Web services standards, by (1) providing extensions to the BPEL language in the form of three new attributes to denote that a scope or loop is a fragment and to determine which loop fragment is responsible for the loop condition. These extensions are only needed in the resulting fragments and are not used in the non-split process; (2) providing two new protocols that plug into the WS-Coordination framework[7] to enable the proper behavior to take place between the fragments of split loops and scopes.

The protocols complement and control local engine behavior; they do not replace it. This approach requires providing the coordinator with a-priori information, such as scope nesting, number of fragments, location of handlers, and compensation order. These are encoded in structures created when creating the fragment processes from the non-split process. They are sent to the coordinator at deployment time.

Note that if the only split scope is the process itself and this scope does not contain fault handlers or nested scopes with compensation handlers or split loops, then coordination may be used but is not needed. Instead a pattern of fault and event handlers is offered that propagates a fault from any of the fragments to the

others and causes termination of them all. This pattern is illustrated in our BPEL solution for RosettaNet[8].

An implementation of the presented solution for propagating control between fragments of split loops and scopes requires: (1) implementations of the new protocols to plug into a WS-Coordination implementation, (2) Extending a BPEL engine to: (a) Understand new language extensions for split loops and scopes, (b) affect navigation based on protocol messages, (c) trigger protocol messages based on navigations. One implementation of this approach is presented in Ref. 9: It extends a BPEL engine in a pluggable modular manner, as detailed by a pluggable BPEL architecture[10] and using the loop and scope coordination protocol definitions[11].

One must note that this is one solution for addressing split control dependenices for loops and scopes. The problem of data dependencies that is the main focus of this paper is nearly orthogonal and in fact does not use the coordination protocols. However, we provide this overview of the split loops/scopes control solution to provide the reader with a complete picture of how splitting a BPEL process can take place. As this paper is focused on data dependencies, further details of the protocols and the coordination approach are not elaborated here. For more details on the end-to-end splitting approach for BPEL, we refer the reader to Ref. 12.

4. Related Work

There is a sizable body of work on splitting business processes, covered in more details in Ref. 1. The most relevant using BPEL is Nanda et. al's work[13] 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-flow 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 completely new middleware, or tolerate fragmentation obfuscation. In the non-BPEL arena, the most relevant in splitting processes are the use of BPEL-D[1] (explicit data links) which is a simpler case of this paper's algorithms, van der Aalst and Weske's P2P approach[14] 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[16]; distributed BPEL engines like the OSIRIS system[17]; modifying a BPEL engine to support using data from partially ordered logical clocks[4] along with write conflict resolution rules.

Dumas et. al translate a process into an event based application on a space-based computing runtime, to enable flexible process modeling[18]. 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[19] 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[20] do not address BPEL's special challenges due to its use of parallelism and especially Dead-Path-Elimination. The application of the Concurrent Single Static Assignment Form (CSSA)[21] to BPEL is shown in Ref. 22. 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 Fig. 1 scenario is used throughout to illustrate the various steps. The presented algorithms require the results of a data-flow analysis on the process. One such algorithm, detailed in Ref. 23 and Ref. 24, was created specifically for this approach. Its details are out of scope for this paper. Any data-flow 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-flow 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. $Q_s(a, x)$ is thus a set of tuples, each containing a query set and a writer set. The representation of a query does not include the variable itself, i.e. '.a' instead of 'x.a'. The symbol ϵ is used to indicate the whole variable and is treated as the empty string in the algorithms that follow.

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

$$Q_s(a, x) = \{(\{x.b, x.c\}, \{w_1, w_2\}), (\{x.d\}, \{w_2, w_3\}), (\{x.e\}, \{w_3\})$$

Loops and fault/compensation handlers are collapsed in these sets. Therefore, the writers in $Q_s(a, x)$ have the following properties:

- If a is in a loop, then all the writers must precede a in the loop and belong to the same loop
- Writers that are in a loop not containing a are represented in $Q_s(a, x)$ as one writer representing the largest loop containing these activities that does not also contain a.
- If a is in a compensation handler, then all the writers must also belong to this compensation handler.
- Each query set contains the largest queries which these writers write to. For example, if w_1, w_2 both write to x.b and x.b.e, then the resulting tuple would be $(\{.b\}, \{w_1, w_2\})$

In order to support split loops and scopes, the data-flow algorithm should also be able to provide (directly or after manipulation) the following sets that have the same structure as $Q_s(a, x)$, but the writer sets differ as follows:

- For each loop l represented as a writer in a $Q_s(a, x)$, where x is a variable read by a and a is any activity not nested in l, a set $Q_s^{postloop}(l, x)$ whose writers are all writers in the loop whose writes reach a.
- For each loop l and each variable x, a set $Q_s^{preloop}(l, x)$ whose writers are all writers not nested in the loop that are read by any activity nested in the loop.
- For each loop l and each variable x read by an activity in l, a set $Q_s^{intraloop}(l, x)$ whose writers of x are only those which are nested in l and not in any other loop l' in l and whose writes reach the next iteration of l.
- For each compensation handler h and variable x read by an activity in h, a set $Q_s^{prehandler}(h, x)$ whose writers are not in h and whose writes reach activities in h.

Consider $A_d(a, x)$ to provide the set of all writers that a depends on for a variable x that it reads: using $\pi_i(t)$ to denote the projection to the i^{th} 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 graph representing the control dependencies between the activities in $A_d(a, x)$ following the steps shown below. Recall that loops are collapsed due to the construction of Q_s whereby writers in a loop that does not contain a are treated as a single node corresponding to the loop. Thus 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.

In the presence of fault handler on a scope, we consider that there exists a path from all activities in the body of the scope to all activities with no incoming links in that scope's fault handlers. Recall that we restrict data flow from outside the fault handler to inside it to be only from the faulting activity. The value of data from the faulting activity will be transmitted to the fault handler fragments using the coordinator. Therefore, if a is in a fault handler then all writers in $A_d(a, x)$ will belong in the same fault handler.

On the other hand, all the writers in the WDG for a reader in a compensation handler will belong in that compensation handler due to the restriction that activities in compensation handlers cannot write data visible outside the handler itself.

A WDG is not dependent on a particular partition. Consider F in Fig. 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:

(i) For each node that corresponds to a split loop, choose an 'owning participant': the one with the lowest number of necessary inter-fragment messages based on the location of the writers nested in the loop and of the reader. A tie is broken through random selection.

(ii) Place a temporary (root) node for each partition, and draw an edge from it to every WDG activity having no incoming link whose source activity is in that partition. This root node is needed to build the subgraphs in the next step.
(iii) Form the largest weakly connected subgraphs where no path between its activities contains any activities from another partition.
(iv) The regions are formed by the subgraphs after removing the temporary nodes added in step ii.

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 P1 in Fig. 2. The PWDG for F and variable *orderInfo*, and the PWDG of G and variable *pymtInfo* are therefore 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 (Sec. 5.3). The latter will be part of the Receiving Flow for the entire PWDG (Sec. 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 (Fig. 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 activities to the process of participant p as described in Algorithm 1.

We define $Q_{sp}(n, a, x)$ to be a function that takes a PWDG node n, a reader a, and a variable x, and returns a set of tuples where each tuple has a set of queries and a set of possible writers such that the writers all belong to the same PWDG node n. These query sets are the same as those in $Q_s(a, x)$, but the writer sets are subsets of those in in $Q_s(a, x)$ such that they only contain writers that are included in the PWDG node, n. A tuple resulting in an empty writer set is not included in $Q_{sp}(n, a, x)$.

Algorithm 1 Creation of a Local Resolver

```
1: procedure CREATE-LOCAL-RESOLVER-MULTIPLE-WRITERS(Node n, Activity a, Variable x)
2:     Q ← Q_sp(n, a, x)
3:     if p = p_r then
4:         Add b=new empty, v=new variable, v.name=idn(n)
5:         t ←v.name
6:     end if
7:     if |Q| = 1 then let Q = {q_s}
8:         if p ≠ p_r then
9:             b ← CREATE-SENDING-BLOCK(name(x))
10:        end if
11:        for all w ∈ π_2(q_s) do
12:            if type(w)=loop then
13:                // see Sec. 5.6
14:            else
15:                Add link l = (w, b, true())
16:            end if
17:        end for
18:    else // more than one query set
19:        if p ≠ pr then
20:            Add b=new invoke, v=new variable, v.name=idn(n)
21:            b.inputVariable=v, b.toPart=("data",x), b.joinCondition="true()"
22:            t ←v.name
23:        end if
24:        for all q_s ∈ Q do
25:            s ← CREATE-ASSIGN-SCOPE(t,qs)
26:            Add link l = (s, b, true())
27:        end for
28:    end if
29: end procedure
```

Algorithm 1 presents the creation of a Local Resolver: if the reader is in the same partition as the writers in this node, then we wait with an 'empty' (line 3-6).

If all writers write to the same set of queries, and the node is not in the reader's participant, use a sending block (line 9). Create a link from every writer to b, which is either the empty or the sending block's invoke (line 7-17). Fig. 4 shows such use of an invoke for C and D in partition y.

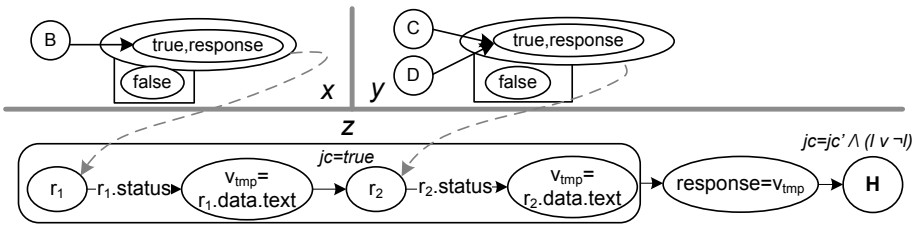

Fig. 4. Snippets from processes from the process in Fig. 1 w/ partition P2.

If there is more than one query set (line 18-28), 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 19-23). For each query, use a structure similar to a sending block (i.e.: scope, fault handler) to get the writers' status (line 25), 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 (line 11-17 in Create-Assign-Scope, Algorithm 2). Create a link from the scope to either the empty from line 4 or the invoke from line 20 (line 26).

Algorithm 2 Functions used by Create-Local-Resolver-Multiple-Writers

function Create-Assign-Scope(String t, Set q_s)
  s ←*new scope*
  a_{sf} ←*new assign*
  s.addFaultHandler('joinFailure',a_{sf})
  a_{sf}*.addCopy*
    *(*QStatus-Str(t,q_s)*, false())*
  s.setActivity(a_s*=new assign)*
  a_s*.suppressJoinFailure='no'*
  a_s*.addCopy*
    *(*QStatus-Str(t,q_s)*, true())*
  for all $w \in \pi_2(q_s)$ **do**
    if type(w)=loop **then**
      // see Sec. 5.6
    else
      Add link $l = (w, a_s, true())$
    end if
  end for
  return s
end function

function QStatus-Str(String t, Set q_s)
  return t + ".status" + id(q_s)
end function

function
Create-Sending-Block(String x)
  Add *s=new scope*
  invf=new invoke
  s.addFaultHandler('joinFailure',invf)
  Add *v=new variable*
  invf.inputVariable=v
  invf.toPart=("status",false())
  inv=new invoke
  s.setActivity(inv)
  inv.inputVariable=v
  inv.toPart=("status",true())
  inv.toPart=("data",x)
  inv.suppressJoinFailure="no"
  return *inv*
end function

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 and 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 presented in Algorithm 3:

Algorithm 3 Creating a Receiving Flow

```
1: procedure CREATE-RF(PWDG G)
2:     Create a flow F
3:     for all n = (p, B) ∈ π1(G) do
4:         PROCESS-NODE(n)
5:     end for
6:     for all e = (n1, n2) ∈ π2(G) do
7:         for all d ∈ ea_{n1} do
8:             Add a link l = (d, ba_{n2}, true())
9:         end for
10:    end for
11:    Add a_f=new assign
12:    a_f.addCopy(v_tmp, x)
13:    Add links l_f = (F, a_f, true()) and l_r = (a_f, a, true())
14:    a.joinCondition ← a.joinCondition ∧ (l_r ∨ ¬l_r)  // recall that a is the reader
15: end procedure
```

Create a flow activity (line 2). For each node, 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} (line 3-5). Link the blocks together by connecting them based on the connections between the partitions, using the first activity *ba* and the set of the last activities *ea* of a block (line 6-10). The subscript is used to identify which node's block they are for (i.e.: ea_{n_1} is the *ea* set created in PROCESS-NODE(n_1). Link the flow to an assign that copies from v_{tmp} to x (line 11-13). Link the assign to a and modify a's join condition to ignore the new link's status (line 14).

The processing of each PWDG node n, PROCESS-NODE, is shown in Algorithm 4. 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 6-9). If the node has only one writer, link from the writer to the assign (line 10-12). 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 20-22). If the node has more than one query set, create an empty instead of an assign (line 15-18) and 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 4 in Algorithm 5). Add a copy to each of these assigns, for every query in the query set, from the locations in x to the same

Algorithm 4 Processing of a PWDG node n

```
1: procedure PROCESS-NODE(Node n) // recall n = (p, B)
2:     Q ← Q_sp(n, a, x), ea ← ∅
3:     // All activities added in this procedure are added to F
4:     if p = p_r then
5:         if |Q| = 1 then let Q = {q_s}
6:             ba=new assign
7:             for all q ∈ q_s do
8:                 ba.addCopy(name(v_tmp) + q, x + q)
9:             end for
10:            if |B| = 1 then let B = {b}
11:                Add link l_0 = (b, ba, true())
12:            end if
13:            ea ← ea ∪ {ba}
14:        else
15:            ba=new empty
16:            for all q_s ∈ Q do
17:                CREATE-Q-ASSIGN(q_s,"x",QSTATUS-STR(idn(n), q_s))
18:            end for
19:        end if
20:        if |B| ≠ 1 then
21:            Add link l_0 = (em, ba, true()), where em=empty from LR
22:        end if
23:        ba.joinCondition ← status(l_0)
24:    else // p is not pr
25:        Add rrb=new receive, rrb.joinCondition = true(), rrb.variable = r_i
26:        ba = rrb
27:        if |Q| = 1 then, let Q = {q_s}
28:            CREATE-Q-ASSIGN(q_s, "r_i.data","r_i.status")
29:        else
30:            for all q_s ∈ Q do
31:                CREATE-Q-ASSIGN(q_s, "r_i.data", QSTATUS-STR(r_i, q_s))
32:            end for
33:        end if
34:    end if
35: end procedure
```

locations in v_{tmp} (line 5-7 in Algorithm 5). Then set the join condition of the empty or assign to only run if the data was valid (line 23).

If the node is another partition, create a receiving block instead of an assign (line 25). Set the join condition of the receive to true so it is never skipped. Again copy the queries into a set of assigns (line 27-34).

Fig. 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

Algorithm 5 Creation of the assigns for each query

```
1: procedure CREATE-Q-ASSIGN(Set q_s, String var, String statusP)
2:     Add act=new assign
3:     ea ← ea ∪ {act}
4:     Add link l = (ba, act, statusP)
5:     for all q ∈ q_s do
6:         act.addCopy(name(v_tmp) + q, var + q)
7:     end for
8: end procedure
```

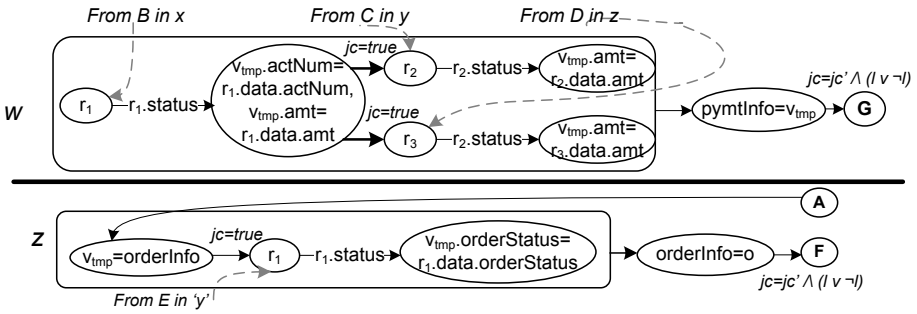

Fig. 5. Two RFs using partition P1. Top: pymtInfo to G in w. Bottom: orderInfo to F in z.

BPEL semantics. Thus, the behavior of the original process is mirrored, not changed.

5.4.1. *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 do not interfere with each other's writes.

5.5. *The End-of-Split-Activity Data Receiving Block*

A modification to the receiving block pattern presented in the Sec. 3 and illustrated in Fig. 3 is needed in the case of split activities (loops and scopes) when data is needed after the split activity completes. For example, data needed after a loop. We call this modified pattern the *end-of-split-activity data receiving block* and use it in Sec. 5.6 and Sec. 5.7. It is illustrated during the elaboration of data needed from

one loop iteration to the next in Sec. 5.6, Fig. 6.

The modification itself is as follows: the assign activity is not the source of a link. To ensure that the end-of-split-activity data receiving block is executed after all activities in the body of the split activity, a link is placed to the receive activity 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 activity is set to ignore the status of all these incoming links.

5.6. *Data in Split Loops*

As control in loops is not point-to-point as is the case for basic flows, providing the data needs to be handled somewhat differently. However, the intuition remains the same: patterns of BPEL activities are used to send and receive and reconstruct the values of the variables between fragments. The approach we take for dealing with loops is to treat the activities in a loop as a process themselves. Then, one may reuse the above approach for handling data flow for activities in the same loop iteration: Data from other fragments in the same iteration is treated like data from other parts of the process in the non-loop case. However, if the reader also needs the value of the variable from a previous iteration, then additional constructs are needed as explained in the subsequent paragraph handling data between loop iterations. To complete the picture, we must therefore address: data needed before the loop starts, data created in one iteration that is needed in a subsequent iteration, and data needed after the loop completes.

In order to provide the data needed before a loop iteration begins to all necessary loop fragments, the algorithms treat the entire loop itself as a reader. In our sample process, the split loop needs 'delivered' and 'orderInfo' before its iteration begins. For every variable v read inside a split loop l that is written outside of l, the algorithms in the previous sections are used in the same manner, except starting with $Q_s^{preloop}(l, v)$ instead of $Q_s(a, v)$. However, the loop is fragmented and more than one fragment may read the v. Each such fragment will need a receiving block. Thus, additional invokes will be added to the sending blocks to send to all the fragments. Fig. 6 illustrates this by showing the value of 'delivered' being sent from fragment 'x' and received at fragments 'y' and 'w' of the split loop I. Notice that the sending blocks are collapsed. This is an optimization allowed in the case of multiple data sending blocks from the same activity to different fragments by placing all the invokes in the same scope. Note that it is possible to also reduce complexity in the case that the Receiving Flow (of data needed before the loop) is complex: place it in one fragment and forwarding the final value of the variable from that fragment to the others.

Data needed from one loop iteration to the next, and/or from the values collected before the loop, is now considered. An example is the variable 'delivered' in our sample process. Data needed in the next iteration needs to be gathered at the end of the current iteration in the fragment(s) that need it. It is handled by using the algorithms provided for the non-loop case but starting with $Q_s^{intraloop}(l, v)$ instead of

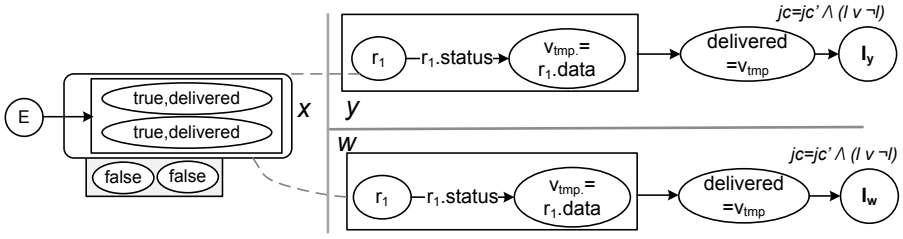

Fig. 6. Getting 'delivered' to the split loop fragments before iteration begins

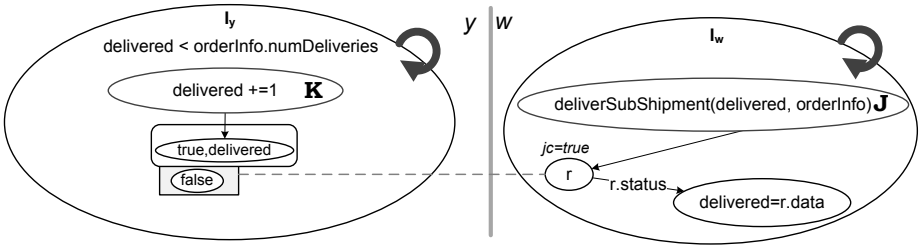

Fig. 7. Getting 'delivered' to the split loop fragments before iteration begins

$Q_s(a, v)$. The difference is that the receiving block is an end of split activity receiving block. The receiving block may need to be placed in more than one fragment and if so extra invokes are added in the sending blocks in the same manner. An example of using the value of 'delivered' written by J in one iteration of the split loop I and read by J in the next is shown in Fig. 7.

If there is at least one reader activity in the fragment of the loop that also needs the value of the variable from both its current iteration *and* its previous iterations (i.e.: it also needs a receiving block), then an assign activity is added in that fragment of the loop. For each such reader: (1) the assign activity copies from the variable to the target variable of the reader's receiving block and (2) a link is created from the assign to the first activity (receive or flow) in the receiving block.

The last case to consider is that of data written in a loop that is needed after the loop ends. This will be the case when a loop itself appears as a node in a PWDG. If the loop is split, then the splitting algorithms assign the corresponding PWDG node to one participant called the owning participant. This participant's fragment will be responsible for collecting and sending the data. The data is valid if it was written by a writer in at least one iteration of the loop. Therefore, a status variable is created in the owning participant's fragment and initialized to false before the loop starts.

If all writers are in the owning participant's fragment, then an assign activity is created for each writer. A link is created from each writer to the respective assign. The assign contains one copy statement that copies 'true' to the status of the data.

In the case that at least one writer is in another fragment of the loop, the data is collected at the owning participant's fragment by creating an end-of-split-activity data receiving block inside the loop. This receiving block and its corresponding sending blocks are constructed starting from $Q_s^{postloop}(l, v)$ and treated as being in the owning participant's fragment. The receiving block is modified such that the value of the status of the owning participant's fragment is set to true if at least one writer was successful: An additional copy statement is added to each assign in the receiving block that copies from 'true' to the status of the data. Thus, the value and status are made available at that fragment once the loop completes.

Having explained how the owning participant's fragment collects the value of the variable, we move on to explaining how this fragment sends the data to the fragments that need it: A sending block is created in the owning participant's fragment outside the loop. The change for loops is the writer is now the loop itself, and the sending block link from the writer has a transition condition: the status of the data. There is no special treatment needed for the corresponding receiving block(s).

5.7. *Data in Split Scopes*

The subset of BPEL we support enables splitting scopes with compensation handlers and/or fault handlers. Therefore, we focus in this section on how data flows into and out of these constructs when a scope (and possibly its handlers themselves) are split.

For the case of fault handlers, we restrict data flow into the handler to data coming from the faulting activity. This will be sent to the handlers using the coordination framework that supports re-enacting the control of scopes upon their fragmentation. Thus, this leaves data between activities in a split fault handler as well as data written in a fault handler that is needed outside the fault handler. The former is handled by simply treating the handler itself as a process as was done for activities in a flow or activities in the same iteration of a loop. The latter is already handled by the algorithms presented above for creating sending and receiving blocks; however, care must be taken in the placement of the sending and receiving blocks: *A cross-scope dependency must remain a cross-scope dependency.* In other words, the sending block for data written in a scope (whether in the scope body or any of its fault/compensation handlers) and whose reader is not in that same scope must be placed outside the scope. To be more precise, it must be placed in the smallest ancestor scope that contains both reader and writer. This ensures that the sending block is not killed by scope disablement while the receiving block is still active at another fragment. A side note is that this is also true for control sending/receiving blocks.

The last item to address is that of compensation handlers. A compensation handler is restricted to behave as in BPEL 1.1: to only read data visible at the time its scope instance completed and to write data only visible to other activities also in the same compensation handler. Therefore, data needed between activities of a

split compensation handler is treated in the same way as data between activities of a process. The novelty is in determining how data written outside the handler and read inside it is provided to the handler fragments.

Data needed for the compensation handler, h, is assembled by repeating the steps for a reader activity in the non-split scope case, but treating the handler itself as the reader. Therefore, one runs the algorithms above starting with $Q_s^{handler}(h, v)$ for every variable v read in the handler and written in the body of the scope. The difference is the placement of the receiving block: An end-of-split-activity data receiving block is used where the split activity is the corresponding scope. In line with the treatment for loops, if more than one fragment of the handler reads the variable, then a receiving block is placed in each such fragment and the sending blocks have extra invokes added to send to the fragments.

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 Local Resolvers and Receiving Flows 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 (Sec. 2), such as the ability to 'revive' a dead path with an 'or' join condition, resulting from dead-path elimination and parallelism.

Our future work includes optimizations such as merging overlapping RFs and targeted data-flow analysis. A first step for optimization is the application of the work presented by Balansundaram and Kennedy[25], Kennedy and Nedeljkovi[26], and Sarkar[27] to BPEL. 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.

Acknowledgments

Jussi Vanhatalo, for suggesting local resolution with one invoke, inspiring the current *Local Resolver*. David Marston, for his valuable review. Oliver Kopp is funded by Tools4BPEL project, which in turn is funded by the German Federal Ministry of Education and Research (project no. 01ISE08).

This paper is the extended version of the paper of the same title published in the proceedings of International Conference on Service-Oriented Computing

(ICSOC 2007)[28]. The extensions included additional details as well as the support for split loops and scopes.

References

1. R. Khalaf and F. Leymann, *Role-based Decomposition of Business Processes using BPEL*, in *ICWS 2006* (IEEE Computer Society, 2006), pp. 770–780, doi:10.1109/ICWS.2006.56.
2. OASIS, *Web Services Business Process Execution Language Version 2.0 – OASIS Standard*, Tech. rep., Organization for the Advancement of Structured Information Standards (OASIS) (2007).
3. F. Curbera, R. Khalaf, F. Leymann and S. Weerawarana, *Exception Handling in the BPEL4WS Language*, in *Proc. of the Conference on Business Process Management (BPM 2003)*, edited by W. M. P. van der Aalst, A. H. M. ter Hofstede and M. Weske, *LNCS*, vol. 2678 (Springer, Eindhoven, the Netherlands, 2003), pp. 276–290, doi:10.1007/3-540-44895-0_19.
4. C. Fidge, *Logical Time in Distributed Computing Systems*, *IEEE Computer* **24**(8) (1991) pp. 28–33, doi:10.1109/2.84874.
5. J.-L. Baer, *A Survey of Some Theoretical Aspects of Multiprocessing*, *ACM Computing Surveys* **5** (1973) pp. 31–80, doi:10.1145/356612.356615.
6. F. Leymann and W. Altenhuber, *Managing business processes as an information resource*, *IBM Systems Journal* **33**(2) (1994) pp. 326–348.
7. OASIS, *Web Services Coordination (WS-Coordination) Version 1.1* (2007), oASIS Standard.
8. R. Khalaf, *From RosettaNet PIPs to BPEL processes: A three level approach for business protocols*, *Data Knowl. Eng.* **61**(1) (2007) pp. 23–38, doi:10.1016/j.datak.2006.04.006.
9. M. Paluszek, *Coordinating Distributed Loops and Fault Handling, Transactional Scopes using WS- Coordination protocols layered on WS-BPEL services*, Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany (2007), URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2586&engl=1`.
10. R. Khalaf, D. Karastoyanova and F. Leymann, *Pluggable Framework for Enabling the Execution of Extended BPEL Behavior*, in *Proc. of the 3rd ICSOC Int'l Workshop on Engineering Service-Oriented Application: Analysis, Design and Composition (WESOA 2007)*, LNCS (Springer, 2007).
11. R. Khalaf and F. Leymann, *Coordination Protocols for Split BPEL Loops and Scopes*, Tech. Rep. 2007/01, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Architecture of Application Systems (2007), URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2007-01&engl=1`.
12. R. Y. Khalaf, *Supporting Business Process Fragmentation While Maintaining Operational Semantics – A BPEL Perspective*, Doctoral Thesis, Universität Stuttgart (2008).
13. M. G. Nanda and N. M. Karnik, *Synchronization Analysis For Decentralizing Composite Web Services*, *Int. Journal of Cooperative Information Systems* **13**(1) (2004) pp. 91–119, doi:10.1142/S0218843004000900.
14. W. Aalst and M. Weske, *The P2P approach to Interorganizational Workflows*, in *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, edited by K. Dittrich, A. Geppert and M. Norrie, *LNCS*, vol. 2068 (Springer, 2001), pp. 140–156, doi:10.1007/3-540-45341-5_10.

15. P. Muth, D. Wodtke, J. Weissenfels, A. Dittrich and G. Weikum, *From Centralized Workflow Specification to Distributed Workflow Execution*, *Journal of Intelligent Information Systems* **10**(2) (1998) pp. 159–184, doi:10.1023/A:1008608810770.
16. T. J. Lehman, S. W. McLaughry and P. Wyckoff, *T Spaces: The Next Wave*, in *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS'99)* (Island of Maui, Hawaii, USA, 1999).
17. C. Schuler, R. Weber, H. Schuldt and H.-J. Schek, *Peer-to-Peer Process Execution with Osiris*, in *ICSOC 2003*, edited by M. E. Orlowska, S. Weerawarana, M. P. Papazoglou and J. Yang, *LNCS*, vol. 2910 (Springer, 2003), pp. 483–498, doi:10.1007/b94513.
18. M. Dumas, T. Fjellheim, S. Milliner and J. Vayssière, *Event-Based Coordination of Process-Oriented Composite Applications*, in *Business Process Management, 3rd International Conference, BPM 2005*, edited by W. M. P. van der Aalst, B. Benatallah, F. Casati and F. Curbera, vol. 3649 (2005), pp. 236–251, doi:10.1007/11538394_16.
19. B. Benatallah, M. Dumas and Q. Z. Sheng, *Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services*, *Journal of Distributed and Parallel Databases* **17**(1) (2005) pp. 5–37, doi:10.1023/B:DAPD.0000045366.15607.67.
20. A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison Wesley, 2006).
21. J. Lee, S. P. Midkiff and D. A. Padua, *Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs*, in *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, *LNCS*, vol. 1366 (Springer, 1997), pp. 114–130, doi:10.1007/BFb0032687.
22. S. Moser, A. Martens, K. Görlach, W. Amme and A. Godlinski, *Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis*, in *Proceedings of IEEE International Conference on Services Computing (SCC 2007)* (2007), pp. 98–105, doi:10.1109/SCC.2007.22.
23. O. Kopp, R. Khalaf and F. Leymann, *Deriving Explicit Data Links in WS-BPEL Processes*, in *Proc. of the International Conference on Services Computing, Industry Track, SCC 2008* (IEEE Computer Society Press, Honolulu, Hawaii, USA, 2008), to appear.
24. O. Kopp, R. Khalaf and F. Leymann, *Reaching Definitions Analysis Respecting Dead Path Elimination Semantics in BPEL Processes*, Technical Report Computer Science 2007/04, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany (2007), URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2007-04&engl=1`.
25. V. Balasundaram and K. Kennedy, *A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations*, in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation* (Portland, Oregon, 1989), pp. 41–53, doi:10.1145/74818.74822.
26. K. Kennedy and N. Nedeljković, *Combining Dependence and Data-Flow Analyses to Optimize Communication*, in *Proceedings of the 9th International Parallel Processing Symposium* (Santa Barbara, CA, 1995), pp. 340–346, doi:10.1109/IPPS.1995.395954.
27. V. Sarkar, *Analysis and Optimization of Explicitly Parallel Programs Using the Parallel Program Graph Representation*, in *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing (LCPC '97)*, edited by Z. Li, P.-C. Yew, S. Chatterjee, C.-H. Huang, P. Sadayappan and D. C. Sehr, *LNCS*, vol. 1366 (Springer, 1998), pp. 94–113, doi:10.1007/BFb0032686.
28. R. Khalaf, O. Kopp and F. Leymann, *Maintaining Data Dependencies Across BPEL Process Fragments*, in *Service-Oriented Computing – ICSOC 2007*, edited by B. J. Krämer, K.-J. Lin and P. Narasimhan, *LNCS*, vol. 4749 (Springer, 2007), pp. 207–219,

doi:10.1007/978-3-540-74974-5_17.

All links were last followed on April, 14 2008.