An Execution Semantics for Mediation Patterns*

Michael Altenhofen[1], Egon Börger[2], and Jens Lemcke[1]

[1] SAP Research, Karlsruhe, Germany
{michael.altenhofen, jens.lemcke}@sap.com
[2] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it
On sabbatical leave at SAP Research, Karlsruhe, Germany
egon.boerger@sap.com

Abstract. This paper utilizes the abstract, formal specification of mediators as described by the Virtual Provider approach for an explicit specification of the execution semantics of workflow interaction patterns.

1 Introduction

Current process specification approaches like BPEL [3], WS-CDL [6], and RosettaNet [1] describe processes using different vocabularies. However, common interaction patterns can be identified throughout different languages [8,7]. We see these patterns as an abstraction from the underlying real workflow languages. But still, even using the same process language, a workflow can be expressed in different ways, i.e. by using different workflow pattern combinations performing an equivalent task.

To overcome the heterogeneity of interaction descriptions different applications use to specify their interfaces, process mediation is required. The goal of this paper is to prepare the ground for using the strategy pattern for the run-time selection of mediators, where any suitable provider interface is viewed as one of the implementations ("mediator orchestration") of a strategy pattern assigned to a requestor interface (see Fig. 1). Thus, one has to investigate the following questions:

- How to assure that a provider interface matches the strategy pattern of the requestor?
- How and starting from which information can one build automatically the strategy pattern implementations?

We first define an abstract, high-level execution semantics of process mediation. For that, we use the ASM[1] specification of "Virtual Providers" (VP) as defined in [2]. To simplify understanding, Sect. 2 contains a condensed overview of the VP concepts. In Sect. 3, we provide some sample refinements of the abstract VP covering the interaction patterns addressed in [5].

---

[1] Abstract State Machines (ASM): An introduction into the ASM method for high-level system design and analysis is available in textbook form in [4], but most of what we use below to model VP constructs is self-explanatory.
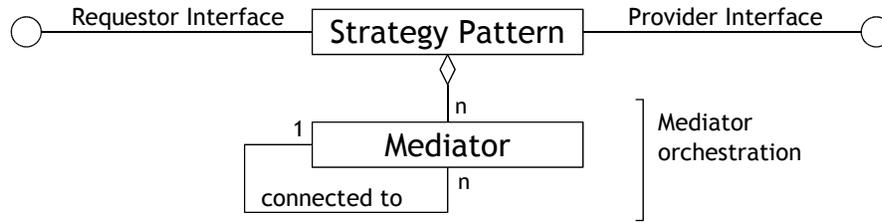
Fig. 1. Strategy Pattern

2   Virtual Providers

We see a Virtual Provider (VP) or mediator as an interface (technically speaking as an ASM module), which is defined by the following five methods (technically speaking ASMs):

MODULE VIRTUALPROVIDER =
    RECEIVEREQ   SENDANSW   PROCESS   SENDREQ   RECEIVEANSW

This view of a VIRTUALPROVIDER as a module (i.e. a collection of defined and callable machines, without a main machine that controls the execution flow) results from the decision to separate the scheduling of these components from their execution. The underlying architecture is illustrated in Fig. 2.
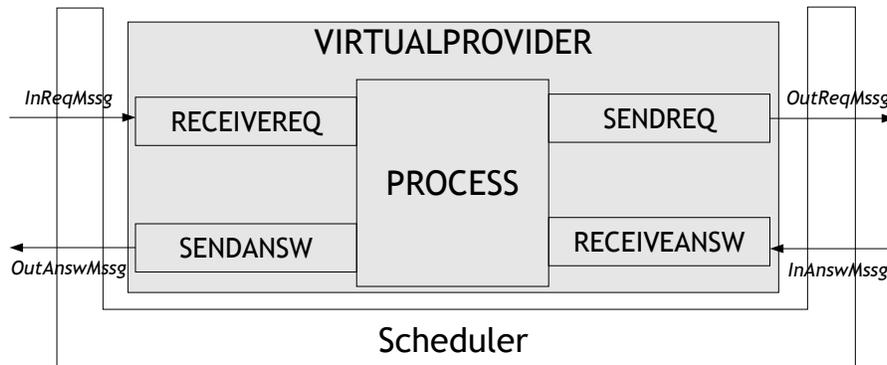

Fig. 2. Architecture

2.1   The Send/Receive Submachines of VirtualProviders

The interaction between a client and a virtual provider, which is triggered by the arrival of a client's request message via the message passing system, is characterized by creating at the VP a request object (a request ID, say element r of a set $ReqObj$ of currently alive request objects), which is appropriately initialized by recording in an internal representation the relevant data, which are encoded in the received request message. This includes decorating that object by an appropriate *status*, say $status(r) := started$, to signal to (the scheduler for) Process its readiness for being processed, and by other useful information.

The predicate *NewRequest* checks, when an *inReqMsg* is received, whether that message contains a new request, or whether it is about an already previously received request. In the first case, CreateNewReqObj as defined below is called. In the second case, instead of creating a new request object, the already previously created request object corresponding to the incoming request message has to be retrieved, using some function $prevReqObj(inReqMsg)$, to RefreshReqObj by the additional information on the newly arriving further service request. In particular, a decision has to be taken upon how to update the $status(prevReqObj(inReqMsg))$, which depends on how one wants the processing *status* of the original request to be influenced by the additional request or information presented through *inReqMsg*. Since we want to keep the scheme general, we assume that an external scheduling function *refreshStatus* is used in an update $status(r) := refreshStatus(r, inReqMsg)$. This leads to the following definition of ReceiveReq, where we skip notationally the set and function parameters $ReqObj, prevReqObj$:

> ReceiveReq($inReqMsg$) =
>   **if** $ReceivedReq(inReqMsg)$ **then**
>   **if** $NewRequest(inReqMsg)$ **then**
>   CreateNewReqObj($inReqMsg, ReqObj$)
>   **else let** $r = prevReqObj(inReqMsg)$ **in**
>   RefreshReqObj($r, inReqMsg$)
> **where**
>   CreateNewReqObj(m, R) =
>   **let** $r = New(R)$ **in**
>   Initialize(r, m)

2.2   The Processing Submachine of VirtualProviders

In our definition we want to abstract from the scheduling mechanism which calls Process for a particular current request object *currReqObj*. We therefore describe the machine as parameterized by such a $currReqObj \in ReqObj$, which plays the role of a global instance variable. The definition is given in terms of control state ASMs in Fig. 3, using the standard graphical representation of finite automata or flowcharts.

The definition in Fig. 3 expresses that each Processing call for a *started* request object *currReqObj* triggers to Initialize an iterative sequential subrequest
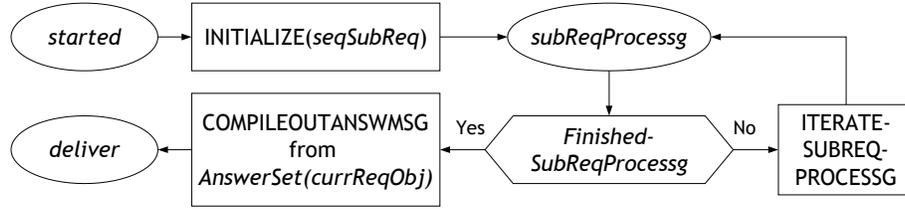
Fig. 3. Processing(*currReqObj*)

processing, namely of the immediate subrequests of this *currReqObj*, in the order defined by an iterator over a set *SeqSubReq*(*currReqObj*). This reflects the first part of the hierarchical view underlying our specification of VP request processing, namely that each incoming (top level) request object *currReqObj* triggers the sequential elaboration of a finite number of immediate subrequests, members of a set *SeqSubReq*(*currReqObj*), called for short sequential subrequests. We view each such sequential subrequest to trigger a finite number of further subsubrequests, which are sent to external providers where they are elaborated independently of each other, so that we call them parallel subrequests of the sequential subrequest. Process uses for the elaboration of the sequential subrequests of *currReqObj* a submachine IterateSubReqProcessg specified below.

$$
\begin{aligned}
&\text{Process}(currReqObj) = \\
&\quad \textbf{if } status(currReqObj) = started \textbf{ then} \\
&\quad\quad \text{Initialize}(seqSubReq(currReqObj)) \\
&\quad\quad status(currReqObj) := subReqProcessg \\
&\quad \textbf{if } status(currReqObj) = subReqProcessg \textbf{ then} \\
&\quad\quad \textbf{if } FinishedSubReqProcessg \textbf{ then} \\
&\quad\quad\quad \text{CompileOutAnswMsg from } currReqObj \\
&\quad\quad\quad status(currReqObj) := deliver \\
&\quad\quad \textbf{else} \\
&\quad\quad\quad StartNextRound(\text{IterateSubReqProcessg})
\end{aligned}
$$

The submachine to IterateSubReqProcessg is an iterator machine defined in Fig. 4. FeedSendReq elaborates simultaneously for each element of *ParSubReq*(*seqSubReq*(*currReqObj*)) an *outReqMsg*. As long as during *waitingForAnswers*, *AllAnswersReceived* is not yet true, ReceiveAnsw inserts for every *ReceivedAnsw*(*inAnswMsg*) the retrieved internal *answer*(*inAnswMsg*) representation into *AnswerSet*(*seqSubReq*) of the currently processed sequential subrequest *seqSubReq*, which is supposed to be retrievable as *requestor* of the incoming answer message.
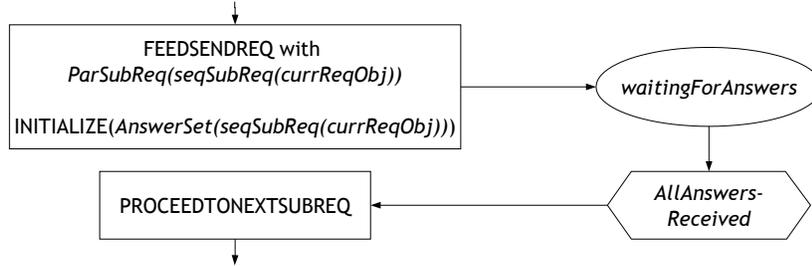
Fig. 4. IterateSubReqProcessg

IterateSubReqProcessg =
  if $status(currReqObj) = initStatus(\textsc{IterateSubReqProcessg})$ **then**
    FeedSendReq with $ParSubReq(seqSubReq(currReqObj))$
    Initialize$(AnswerSet(seqSubReq(currReqObj)))$
    $status(currReqObj) := waitingForAnswers$
  if $status(currReqObj) = waitingForAnswers$ **then**
    if $AllAnswersReceived$ **then**
      ProceedToNextSubReq
      $status(currReqObj) := subReqProcessg$

2.3 Composing VirtualProviders

We have formulated VirtualProvider in such a way that different instances $VP_1, \ldots, VP_n$ of it can be configured into say a sequence, leading to a virtual provider VP_1, which involves a subprovider VP_2, which involves its own subprovider VP_3 etc. For such a composition it suffices to connect the communication interfaces in the appropriate way, in the case of a sequence as follows:

- SendReq of VP_i with the ReceiveReq of VP_{i+1}, which implies that in the message passing environment, the types of the sets $OutReqMssg$ of VP_i and $InReqMssg$ of VP_{i+1} match (via some data mediation).
- SendAnsw of VP_{i+1} with the ReceiveAnsw of VP_i, which implies that in the message passing environment, the types of the sets $OutAnswMssg$ of VP_{i+1} and $InAnswMssg$ of VP_i match (via some data mediation).

Such a sequential composition allows one to configure VP (mediator) schemes where each element seq_1 of a sequential subrequest set $SeqSubReq_1$ of an initial request can trigger a set $ParSubReq(seq_1)$ of parallel subrequests par_1, each of which can trigger a set $SeqSubReq_2$ of further sequential subrequests seq_2 of par_1, each of which again can trigger a set $ParSubReq(seq_2)$ of further parallel subrequests, etc. This provides the possibility of unfolding arbitrary alternating seq/par trees, where at each level one has sequential subrequests each of which

branches into a subtree of parallel subsubrequests, each of which may have a subtree of other sequential subrequests, etc.

Obviously for VIRTUALPROVIDER even more complex composition schemes can be defined, if one wants to do this.

2.4 Defining Equivalence Notions for VIRTUAL PROVIDERs

To be able to speak about the relation between incoming requests and outgoing answers, one has to relate the elements of the corresponding sets *InReqMssg* and *OutAnswMssg*. Formally, this comes up to unfold the function *originator*, which for an *outAnswerMsg* yields the *inReqMsg* to which *outAnswerMsg* represents the answer. In fact this information is retrievable by COMPILEOUTANSWMSG from the *currReqObj*, if it was recorded there by CREATENEWREQOBJ(*inReqMsg*, *ReqObj*) as part of INITIALIZE.

One can then define the *ServiceBehavior*(*VP*) of a virtual provider *VP* = VIRTUALPROVIDER as the correspondence between any *inReqMsg* and the *outAnswerMsg* related to it by the *originator* function:

$$originator(outAnswerMsg) = inReqMsg$$

Two virtual providers *VP*, *VP'* can be considered equivalent if an equivalence relation *ServiceBehavior*(*VP*) ≡ *ServiceBehavior*(*VP'*) holds between their service behavior. To concretely define such an equivalence involves detailing of the meaning of service 'requests' and provided 'answers', which comes up to further detail the abstract *VP* model in such a way that the intended 'service' features and how they are 'provided' by *VP* become visible. On the basis of such definitions one can then formally define different *VP*s to be alternatives for a stragey pattern for providing requested services.

Another interesting class of relations becomes rigorously analysable if we consider the pair *OutReqMssg* and *InAnswMssg* on the requestor side of a *VP*.

3 Mediation Patterns

In this section, we will describe how different interaction patterns can be realized by specializing the presented VP concept. In [5], the authors announce to support the basic pattern set *a)* to *e)* depicted in Fig. 5 by their Process Mediator (PM). We would like to demonstrate certain specializations of the abstract VP into some more concrete implementations that are able to handle these patterns. We thus create a set of new ASM MODULEs VPA, . . . , VPE, each handling a single one of the interaction patterns. In the given patterns, *data sets* are depicted by capital letters ("A" and "B"), *messages* by arrows.

In the VP, the whole de-compilation of incoming messages into sequential and parallel subrequest messages is carried out by the sets

$$SeqSubReq(currReqObj) \ \text{ and}$$
$$ParSubReq(seqSubReq)$$

for each value of the global PROCESS variable *currReqObj* and each value *seqSubReq* of the set *SeqSubReq*(*currReqObj*). Thus, one needs to define these sets *SeqSubReq*[2] and *ParSubReq* in order to realize specific pattern mappings. This is done for the five interaction patterns shown in Fig. 5 by defining these sets and related functions through a refinement of the INITIALIZE submachine of CREATENEWREQOBJ and the machine REFRESHREQOBJ as follows.
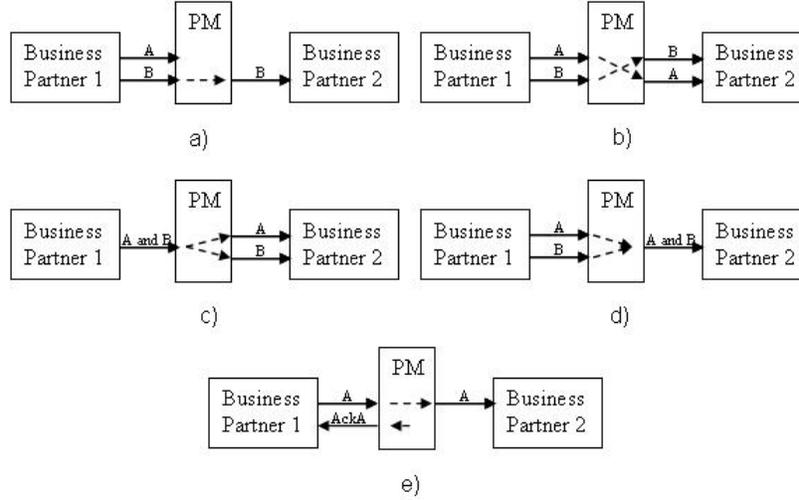
Fig. 5. Resolving Interaction Missmatches

Pattern a) Here, the data set "A" transported by the first message from "Business Partner 1" is discarded, only data set "B" transmitted by the second incoming message is being forwarded to "Business Partner 2". VPa is defined as VIRTUALPROVIDER instantiated as follows. The VP has to INITIALIZE the *newReqObj* by preventing its PROCESSing from being *started* yet. No data need to be recorded since only the data of the second message are relevant.

INITIALIZE(*newReqObj*, *inReqMsg*) =
    status(*newReqObj*) := *initStatus*(PROCESS).

The subsequently incoming message containing data set "B" first needs to be related to the same *newReqObj* by *prevReqObj*(*inReqMsg*) while evaluating *NewRequest*(*inReqMsg*) to false.[3] Second, the *dataSet* is extracted from the mes-

---

[2] Iterator functions *FstSubReq* and *NxtSubReq*, terminator symbol *Done* $\notin$ *SeqSubReq*.

[3] We do neither specify an implementation for *prevReqObj*(*inReqMsg*) nor *NewRequest*(*inReqMsg*) here, since their behaviour depends on the real messaging system used. This aspect is an abstract parameter of the specification of this abstract interaction pattern.

sage and stored as the only element of $SeqSubReq(prevReqObj)$, to be forwarded as the only parallel subrequest in $ParSubReq(dataSet)$ to "Business Partner 2". Then, VPa is *started* without having to send or to receive any answer message. For brevity, we write $prevReqObj$ for $prevReqObj(inReqMsg)$.

> REFRESHREQOBJ$(prevReqObj, inReqMsg) =$
>   **let** $dataSet = extractDataSet(inReqMsg)$ **in**
> $SeqSubReq(prevReqObj) := \{ dataSet \}$
> $Done(\{ dataSet \}) := nil$
> $FstSubReq(\{ dataSet \}) := dataSet$
> $NxtSubReq(\{ dataSet \}, \_\_, \_\_) := nil$
> $ParSubReq(dataSet) := \{ dataSet \}$
> $ToBeAnswered(\{ dataSet \}) := \emptyset$
> $AnswToBeSent(prevReqObj) := false$
> $status(prevReqObj) := started$

The thus instantiated VP machine VPa starts in status *started* after having received the two incoming messages and eventually stays in status *deliver* after having sent the data set "B" to the mailer SENDREQ from where it is sent to "Business Partner 2".

Pattern b) This interaction pattern can be seen as an extension of *Pattern a)*. In addition to just forwarding data set "B" to "Business Partner 2", it INITIALIZEs by also storing data set "A", without triggering VP to get *started*[4], for sending it after "B":

> INITIALIZE$(newReqObj, inReqMsg) =$
> $status(newReqObj) := initStatus(\text{PROCESS})$
> $dataSet_A(newReqObj) := extractDataSet(inReqMsg)$
> REFRESHREQOBJ$(prevReqObj, inReqMsg) =$
>   **let** $dataSet_B = extractDataSet(inReqMsg)$ **in**
> $SeqSubReq(prevReqObj) := \{ dataSet_B, dataSet_A(prevReqObj) \}$
> $Done(\{ dataSet_B, dataSet_A(prevReqObj) \}) := nil$
> $FstSubReq(\{ dataSet_B, dataSet_A(prevReqObj) \}) := dataSet_B$
> $NxtSubReq(\{ dataSet_B, dataSet_A(prevReqObj) \},$
> $dataSet_B, \_\_) := dataSet_A(prevReqObj)$
> $NxtSubReq(\{ dataSet_B, dataSet_A(prevReqObj) \},$
> $dataSet_A(prevReqObj), \_\_) := nil$
> $ParSubReq(d) := \{ d \}$
> $ToBeAnswered(d) := \emptyset,$ for $d = dataSet_A(prevReqObj), dataSet_B$
> $AnswToBeSent(prevReqObj) := false$
> $status(prevReqObj) := started$

---

[4] Note: $initStatus(\text{VP}) \neq started$.

Pattern c) In this pattern, "Business Partner 1" sends both data sets "A" and "B" in the same message, whereas "Business Partner 2" expects to receive them in two different messages. VP does not need to wait for any subsequent incoming messages. It therefore INITIALIZEs by saving both data sets for their later sequential sending and directly enables the PROCESSING getting *started*:

> INITIALIZE($newReqObj$, $inReqMsg$) =
> **let** $dataSet_i = extractDataSet_i(inReqMsg)$ for $i = $ A, B **in**
> $\quad SeqSubReq(newReqObj) := \{\ dataSet_A, dataSet_B\ \}$
> $\quad Done(\ \{\ dataSet_A, dataSet_B\ \}\ ) := nil$
> $\quad FstSubReq(\ \{\ dataSet_A, dataSet_B\ \}\ ) := dataSet_A$
> $\quad NxtSubReq(\ \{\ dataSet_A, dataSet_B\ \}, dataSet_A,\ \_\_) := dataSet_B$
> $\quad NxtSubReq(\ \{\ dataSet_A, dataSet_B\ \}, dataSet_B,\ \_\_) := nil$
> $\quad ParSubReq(d) := \{\ d\ \}$
> $\quad ToBeAnswered(d) := \emptyset,\ $ for $d = dataSet_A, dataSet_B$
> $\quad AnswToBeSent(newReqObj) := false$
> $\quad status(newReqObj) := started$

Pattern d) The specialization for this pattern can be easily adapted from *Pattern b)* by just *combin*ing the received data sets into one single message to be sent to "Business Partner 2". We therefore indicate only the changes in the macro REFRESHREQOBJ:

> $FstSubReq(\ \{\ \{\ dataSet_A(prevReqObj), dataSet_B\ \}\ \}\ ) :=$
> $\quad \{\ dataSet_A(prevReqObj), dataSet_B\ \}$
> $NxtSubReq(\ \{\ \{\ dataSet_A(prevReqObj), dataSet_B\ \}\ \},$
> $\quad \{\ dataSet_A(prevReqObj), dataSet_B\ \},\ \_\_) := nil$
> $ParSubReq(\ \{\ dataSet_A(prevReqObj), dataSet_B\ \}\ ) :=$
> $\quad \{\ combine(dataSet_A(prevReqObj), dataSet_B)\ \}$

Pattern e) In this interaction pattern, a save medium for transmitting messages from the VP to "Business Partner 2" is apparently assumed. After receipt of the message carrying data set "A", it is saved and forwarded without waiting for further messages. After *FinishedSubReqProcessg*, the VP creates an appropriate *outAnswer*($newReqObj$) containig data set "AckA" to be returned to "Business Partner 1":

> INITIALIZE($newReqObj$, $inReqMsg$) =
> **let** $dataSet = extractDataSet(inReqMsg)$ **in**
> $\quad SeqSubReq(newReqObj) := \{\ dataSet\ \}$
> $\quad Done(\ \{\ dataSet\ \}\ ) := nil$
> $\quad FstSubReq(\ \{\ dataSet\ \}\ ) := dataSet$
> $\quad NxtSubReq(\ \{\ dataSet\ \},\ \_\_,\ \_\_) := nil$
> $\quad ParSubReq(dataSet) := \{\ dataSet\ \}$

$ToBeAnswered(\ \{\ dataSet\ \}\ ) := \emptyset\ ^5$
$AnswToBeSent(newReqObj) := true$
$status(newReqObj) := started$

4 Conclusions and Future Work

By utilizing the abstract, high-level specification for mediators presented in [2], this paper addresses two objectives: On the one hand, it provides an explicit and unambiguous means for defining patterns of mediators. Thus, it is able to build a base for *"communicating and documenting design ideas"*, e.g. of mediation patterns. Furthermore, it supports *"an accurate and checkable overall understanding"* yielding the ability for proving, that e.g. two patterns (or pattern combinations) perform equivalent transformations.

On the other hand, the ASM methodology facilitates to *"isolate the hard part of a system"* [4, p. 14-15], and thus enabling to view a system from different levels of abstraction. Based on this capability, the ASM model allows the future definition of the run-time selection of mediators by the strategy pattern (see Fig. 1). The sample interaction patterns presented in Sect. 3 can be expressed by a single VP for each pattern. However, as described in Sect. 2.3, by building more complex patterns through chaining multiple VPs, certain specific strategy patterns can be implemented.

References

1. Rosettanet. http://www.rosettanet.org/.
2. M. Altenhofen, E. Börger, and J. Lemcke. A high-level specification for mediators. *submitted to 1st International Workshop on Web Service Choreography and Orchestration for Business Process Management*, 2005.
3. A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu. Oasis — web services business process execution language — working draft, 27 February 2005. http://www.oasis-open.org/committees/download.php/11601/wsbpel-specification-draft-022705.htm.
4. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.
5. E. Cimpian and A. Mocan. D13.7 v0.1 process mediation in wsmx – wsmx working draft, 7 March 2005. http://www.wsmo.org/TR/d13/d13.7/v0.1/.
6. N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web services choreography description language version 1.0 — w3c working draft, 17 December 2004. http://www.w3.org/TR/ws-cdl-10/.
7. W. M. van der Aalst, A. Barros, A. ter Hofstede, and B. Kiepuszewski. Advanced workflow patterns, 2000.
8. W. M. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns, 2000.

---

[5] This realizes the assumption of a reliable transmission from VP to "Business Partner 2".