Towards Executing ebBP-Reg B2Bi Choreographies

Andreas Schönberger and Guido Wirtz
Distributed and Mobile Systems Group
Otto-Friedrich-University of Bamberg
Bamberg, Germany
{*andreas.schoenberger* | *guido.wirtz*}@*uni-bamberg.de*

Abstract

Applying choreography and orchestration technology to Business-to-Business integration (B2Bi) scenarios has become a popular technique for very good reasons. Choreography descriptions can be used to capture B2Bi scenarios from a global and abstract perspective while orchestrations then can be used to specify the local implementation of each integration partner. ebXML BPSS (ebBP) is a prominent B2Bi choreography standard with very helpful domain-specific concepts, but clear guidelines for creating executable choreographies are missing.
In order to create ebBP models that are both adequate and executable, expressiveness, comprehensibility and standard-conformance have to be weighed up. In this paper, we introduce ebBP-Reg as an ebBP modeling flavor that is designed such that ebBP-Reg choreographies are executable as WS-BPEL orchestrations. At the same time, ebBP-Reg models strictly conform to the ebBP standard and support concurrency and decomposition. We characterize syntactic validity of ebBP-Reg models by means of language production rules and show how instances of ebBP-Reg can be implemented using WS-BPEL.

Keywords: B2Bi, ebXML BPSS, WS-BPEL, choreography, orchestration

1. Introduction

In the B2Bi domain, integration partners typically first have to agree upon the types and sequence of message exchanges for connecting their information systems and then implement the agreed-upon process in a distributed manner. Applying choreography and orchestration technology to this task are a good fit because choreography languages offer a global and abstract view on system interactions while orchestration languages offer the functionality for implementing each integration partner's local behavior.
Considering concrete languages for a B2Bi choreography/orchestration tool chain, ebXML BPSS (ebBP) [1] and WS-BPEL [2] are a particularly good choice. ebBP offers domain-specific concepts like BusinessTransactions or B2Bi Quality-of-Service and allows for the technology-independent specification of B2Bi choreographies. Although technology-independence enables choosing multiple implementation technologies, ebBP must be mapped to an implementation technology at some point in time. WS-BPEL as the most important Web services orchestration standard is a natural fit for that due to its platform-interoperability. However, deriving WS-BPEL orchestrations from ebBP choreographies is not straight-forward because the semantics of ebBP is not precisely defined and, consequently, validity and implementability are not clear. Therefore, the class of processes to be considered for modeling ebBP choreographies must be carefully chosen in order to ensure validity and implementability. At the same time, expressiveness, comprehensibility and standard-conformance play an important role in tailoring an adequate class of valid and implementable ebBP models. In [3], ebBP-ST, ebBP-Reg and ebBP+ are proposed as three different ebBP modeling flavors that serve different requirements. In the work at hand, we present ebBP-Reg as the ebBP modeling flavor that strictly conforms to the ebBP XML schema and supports hierarchical decomposition as well as concurrency for binary collaborations. Standard-conformance and implementability while covering the majority of B2Bi scenarios have been the decisive requirements for shaping ebBP-Reg (cf. [3]). We are giving production rules for creating valid and implementable models and we informally define its operational semantics by mapping ebBP-Reg to WS-BPEL.
Section 2 introduces ebBP and section 3 defines valid and implementable ebBP-Reg models. Then, section 4 presents the use case that is used for validation and section 5 describes the mapping of ebBP-Reg to WS-BPEL. Finally, section 6 discusses related work and section 7 concludes and points out directions for future work.

2. Basics

The core concepts of ebBP are so-called BusinessTransactions (BTs) and BusinessCollaborations (BCs). BTs specify the exchange of a request document and an optional response document between the BT 'requester' role (request document sender) and the BT 'responder' role (request document receiver). B2Bi Quality-of-Service attributes as well as additional control messages signaling the progress of document processing are available as additional BT parameters. BCs define the choreography between at least two integration partner roles and are composed from BTs using the concept of BusinessTransactionActivities (BTAs). A BTA represents the execution of a BT and maps BC roles to the BT requester/responder roles. Additional execution parameters such as a 'TimeToPerform' can be declared for BTAs. Using BTAs, multiple executions of a BT within a BC can be specified. Similarly, a BusinessCollaborationActivity (BCA) specifies the execution of a BC within a second BC which enables hierarchical decomposition.

The control flow structure of a BC is an almost arbitrary graph that uses Forks, Joins, Decisions and Transitions to describe the flow between BTAs and BCAs. ebBP Forks can be of type 'OR' or 'XOR' and ebBP Joins carry a boolean 'waitForAll' attribute. We denote Joins with 'waitForAll' set to true as AND-Joins and with 'waitForAll' set to false as OR-Joins. Forks of type 'XOR' (XOR-Fork) are allowed to trigger exactly one successor whereas an arbitrary number of successors can be triggered by an Fork of type 'OR' (OR-Fork). If an OR-Fork is matched by an AND-Join then all Fork successors have to be triggered. We will denote the Fork of such a combination as AND-Fork. If an OR-Fork is matched by an OR-Join then an arbitrary number of Fork successors may be triggered, but the behavior of the OR-Join is not precisely defined. ebBP Decisions connect multiple BTAs/BCAs and choose between different paths by assigning guards to the according links whereas ebBP Transitions connect exactly two BTAs/BCAs and also allow for the specification of guards.

The reader is assumed to be familiar with WS-BPEL.

3. ebBP-Reg

First, a superset of the eligible ebBP-Reg processes is defined that is restricted by means of wellformedness rules afterwards.

Definition 3.1 (ebBP-Reg Process):
An ebBP-Reg process is a five-tuple RP (s_0,F,A,C,T) with the following elements:

- s_0 the start node.
- F a non-empty set of final states.
- A = SBTA ∪ SBCA with SBTA a non-empty set of BTAs and SBCA a set of BCAs.
- C = SXORF ∪ SANDF ∪ SANDJ with SXORF a set of XOR-Forks, SANDF a set of AND-Forks and SANDJ a set of AND-Joins.
- T the union of the following transition sets
  - $T^{start} = \{(s_0,true,e)\}$, $e \in A$
  - $T^{end} = A \times G \times F$
  - $T^{ctrlIn} = A \times G \times C$
  - $T^{ctrlOut} = C \times \{true\} \times A$
  - $T^{straight} = A \times G \times A$

  where $G = G^{bta} \cup G^{bca}$ a set of boolean guards defined on the results of BTAs and BCAs, respectively. □

Further, the following auxiliary functions are defined.

Definition 3.2 (Auxiliary Functions):

- .-notation/#-notation is used for accessing the components of a tuple by name/index.
- A path between two nodes a,b $\in \{s_0\} \cup A \cup C \cup F$ is a sequence of nodes a,$n_{1..x}$,b such that for all i=1...x-1, (n_i,g_i,n_{i+1}) and (a,g_a,n_1) and (n_x,g_x,b) ∈ T. Let Path(a,b) be the set of all paths between a and b. □

Considering the ebBP control flow constructs introduced in section 2, OR-Forks, OR-Joins and Decisions are missing in the definition of ebBP-Reg processes. The functionality of OR-Forks to perform an arbitrary selection of follow-on activities is not supported. Consequently, OR-Joins are not needed because any node $n \in A \cup C \setminus SANDJ$ then can serve as join node for an XOR-Fork and AND-Joins can be used for joining AND-Forks. The functionality of ebBP Decisions is provided using the guards on ebBP Transitions in order to circumvent referential constraint problems when linking from Decisions to other control flow nodes (cf. toBusinessStateRef/fromBusinessStateRef constraints in [1] sec. 3.8.2).

The following production rules describe how to create a syntactically valid ebBP-Reg model step by step. The production rules are chosen such that any syntactically valid ebBP-Reg model can be performed using WS-BPEL (cf. section 5).

Rule 3.1 (Elementary Process):
Any process rp = (s_0, {f}, {bta}, ∅, {(s_0, true, bta), (bta, true, f)}) is a valid RP. □

Rule 3.2 (Add BTA):
Let rp be a valid RP, bta a BTA ∉ rp.A.SBTA, and pred = (pred#1,pred#2,pred#3) ∈ rp.T^{end}.
Then rp'= (rp.s_0, rp.F, rp.A.SBTA ∪ {bta}, rp.C, rp.T ∪ {(pred#1, pred#2, bta), (bta, true, pred#3)} \ {(pred#1, pred#2, pred#3)}) is a valid RP. □

Rule 3.3 (Process Composition):
Let rp_1, rp_2 be valid RPs, pred ∈ rp_1.T^{end}, bca^{rp_2} be a BCA executing rp_2, and bca^{rp_2} ∉ rp_1.A.SBCA.
Then rp' = (rp_1.s_0, rp_1.F, rp_1.A.SBCA ∪ {bca^{rp_2}}, rp.C, rp.T ∪ {(pred#1, pred#2, bca^{rp_2}), (bca^{rp_2}, true, pred#3)} \ {(pred#1, pred#2, pred#3)}) is a valid RP. □

Rule 3.4 (Event-Based Choice):
Let rp be a valid RP and pred ∈ rp.T^{end}. Let ebc = (xorF, F, A, $T^{ctrlOut}$, T^{end}) be an 'event-based choice component' that chooses from a set of BTAs/BCAs at run-time as requested by a backend/partner process with:

- xorF an XOR-Fork.
- F = {f_1,...,f_n} a set of final states.
- A = {a_1,...,a_n} a set of BTAs and BCAs with at least one BTA.
- $T^{ctrlOut}$ = xorF × {true} × A.
- $T^{end} \subseteq A \times \{true\} \times F$ such that $A \times \{true\} \rightarrow F$ is a bijective function.

Further, let pred#3 ∈ ebc.F ∧ ebc.xorF ∉ rp.C ∧ (ebc.F \ {pred#3}) ∩ rp.F = ∅ ∧ ebc.A ∩ rp.A = ∅.
Then rp' = (rp.s_0, rp.F ∪ ebc.F, rp.A ∪ ebc.A, rp.C ∪ {xorF},

rp.T $\cup$ ebc.$T^{ctrlOut}$ $\cup$ ebc.T^{end} $\cup$ {(pred#1,pred#2,xorF)} $\setminus$ {(pred#1, pred#2, pred#3)}) is a valid RP. $\square$

Rule 3.5 (Parallel):
Let rp_x be a valid RP and pred $\in$ $rp_x.T^{end}$. Let rp_1,...,rp_n be a set of valid RPs and PCA = bca^{rp_1},...,bca^{rp_n} be a set of BCAs for executing each RP and for all bca^{rp_i} with i in [1;n] holds: $bca^{rp_i} \notin rp_x$.A. Let andF, andJ $\notin$ rp_x.C be an AND-Fork and an AND-Join and let T^{fork} = {andF} $\times$ {true} $\times$ PCA and T^{join} = PCA $\times$ {true} $\times$ {andJ}.

Let for any rp_i with i in [1;n]: All constituent sets of rp_i and rp_x are pairwise disjoint.
Then, rp' = ($rp_x.s_0$, rp_x.F, rp_x.SBCA $\cup$ PCA, rp_x.C $\cup$ {andF, andJ}, rp_x.T $\cup$ T^{fork} $\cup$ T^{join} $\cup$ {(pred#1,pred#2,andF), (andJ,true,pred#3)} $\setminus$ {(pred#1, pred#2, pred#3)}) is a valid RP. $\square$

For the following rules, let rp.A.PCA denote the set of BCAs within any parallel structure of a given valid RP rp as defined in rule 3.5.

Rule 3.6 (Add Transition):
Let rp be a valid RP, d $\in$ (rp.A $\setminus$ rp.A.PCA) $\cup$ rp.C.SXORF $\cup$ rp.C.SANDF $\cup$ rp.F, p^{add} $\in$ rp.A $\setminus$ rp.A.PCA, $T_{p^{add}} \subseteq$ rp.T with: $\forall t \in T_{p^{add}}$: t#1 = p^{add}, and (p^{add}, g, f) $\in$ rp.T^{end} for some g and f.

Further, let newT = (p^{add},newG,d) be a transition, R be a function that computes a new set of transitions from $T_{p^{add}}$ by assigning a new valid guard to each element of $T_{p^{add}}$ such that (newG $\vee$ $\bigvee_{t \in R(T_{p^{add}})}$ t#2) evaluates to true and $\forall$ t1,t2 $\in$ newT$\cup$R($T_{p^{add}}$), t1 $\neq$ t2: (t1#2 $\wedge$ t2#2) evaluates to false.
Then, (rp.s_0, rp.F, rp.A, rp.C, (T $\setminus$ $T_{p^{add}}$) $\cup$ {newT} $\cup$ R($T_{p^{add}}$)) is a valid RP. $\square$
Note that this rule also is valid for d $\in$ rp.F and therefore also covers new transitions to final nodes.

Rule 3.7 (Add Final Node):
Let rp be a valid RP, p^{add} $\in$ rp.A $\setminus$ rp.A.PCA, $T_{p^{add}} \subseteq$ rp.T with: $\forall t \in T_{p^{add}}$: t#1 = p^{add}, and (p^{add}, g, f) $\in$ rp.T^{end} for some g and f.

Further, let newF $\notin$ rp.F be a final node, newT = (p^{add},newG,newF) be a transition, R be a function that computes a new set of transitions from $T_{p^{add}}$ by assigning a new valid guard to each element of $T_{p^{add}}$ such that (newG $\vee$ $\bigvee_{t \in R(T_{p^{add}})}$ t#2) evaluates to true, and $\forall$ t1,t2 $\in$ newT$\cup$R($T_{p^{add}}$), t1 $\neq$ t2: (t1#2 $\wedge$ t2#2) evaluates to false.
Then, rp' = (rp.s_0, rp.F $\cup$ newF, rp.A, rp.C, (T $\cup$ {newT} $\setminus$ $T_{p^{add}}$) $\cup$ R($T_{p^{add}}$)) is a valid RP. $\square$

Rule 3.8 (Remove Final Node):
Let rp be a valid RP, f $\in$ rp.F, $PRED_f \subseteq$ rp.T^{end} with: $\forall t \in PRED_f$: t#3 = f, and H_f be the set of sets of outgoing transitions of predecessors of f such that: $PRED_f \subseteq \bigcup_{h \in H_f}$ h $\wedge$ $\forall$ $h_{x,f} \in H_f$: ($\forall$ t $\in$ $h_{x,f}$: t#1 = x) $\wedge$ ($\nexists$ t $\in$ rp.T $\setminus$ $h_{x,f}$: t#1 = x) $\wedge$ $h_{x,f} \cap PRED_f \neq \emptyset$.

Further, let DEST $\subseteq$ (rp.A $\setminus$ rp.A.PCA) $\cup$ rp.C.SXORF $\cup$ rp.C.SANDF $\cup$ rp.F $\setminus$ {f}.

Further, let R be a function that computes a new set of transitions from each $h_{x,f} \in H_f$ by assigning to each t $\in$ $h_{x,f}$ a new valid guard and replacing t#3 with some d $\in$ DEST such that: $\bigvee_{t \in R(h_{x,f})}$ t#2 evaluates to true $\wedge$ $\forall$ t1,t2 $\in$ R($h_{x,f}$), t1 $\neq$ t2: (t1#2 $\wedge$ t2#2) evaluates to false, and for each $h_{x,f} \in H_f$: $\exists$ Path(x,e) $\neq \emptyset$ with e $\in$ rp.F $\setminus$ {f}.
Then, rp' = (rp.s_0, rp.F $\setminus$ {f}, rp.A, rp.C, (T $\setminus$ $\bigcup_{h \in H_f}$h) $\cup$ $\bigcup_{h \in H_f}$R(h)) is a valid RP. $\square$

4. Use Case

For giving an impression of ebBP-Reg's expressiveness and for validation purposes, the artificial RosettaNet[1] PIPs (conceptually equivalent to BTs) based purchasing use case depicted in figure 1 is used. The use case starts out with a parallel structure for concurrently exchanging purchase order requests (PIP 3A19) within two BCAs of the same type (BC-single3A19-1, BC-single3A19-2) that, again, specify seller and buyer roles. For BC-single3A19-1, the root level seller role takes the BC-single3A19 seller role and for BC-single3A19-2, the root level seller role takes the BC-single3A19 buyer role. After the concurrent purchase order requests, a single purchase order confirmation (PIP 3A20, denoted 'BT-3A20') is exchanged within a BTA. In case of a protocol failure (denoted 'P-F') or a negative confirmation message (denoted 'Stop'), the process is terminated. Otherwise, the process is continued and multiple actions may be taken. The root level buyer role may try to change (PIP 3A21) or cancel (PIP 3A23) the purchasing process or the root level seller role may try to finish the process by sending an invoice (PIP 3C3). Depending on whether or not these BTAs succeed ('[P-S]' transitions) or fail ('[P-F]' transitions) and depending on the result of additional BTAs (BT-3A22/3A24 for replying to change/cancellation requests) the process eventually terminates. The different final states denote the different overall results of the purchasing process.

Note that the use case captures the main types of process components (parallel structures, loops, event-based choices) that can be created from the rules of the last section. For validating the mapping of the next section, the prototypic WS-BPEL implementation of the use case has manually been derived according to the translation rules of section 5.2. Thereby, the implementability of the translation rules has been checked as well. The ebBP-Reg specification as well as the WS-BPEL implementation of the use case are available under http://www.uni-bamberg.de/pi/ebBP-Reg-CtrlFlowUseCase.

5. WS-BPEL implementation

The description of how to use WS-BPEL for implementing ebBP-Reg choreographies is split up into two parts. Section

1. http://www.rosettanet.org

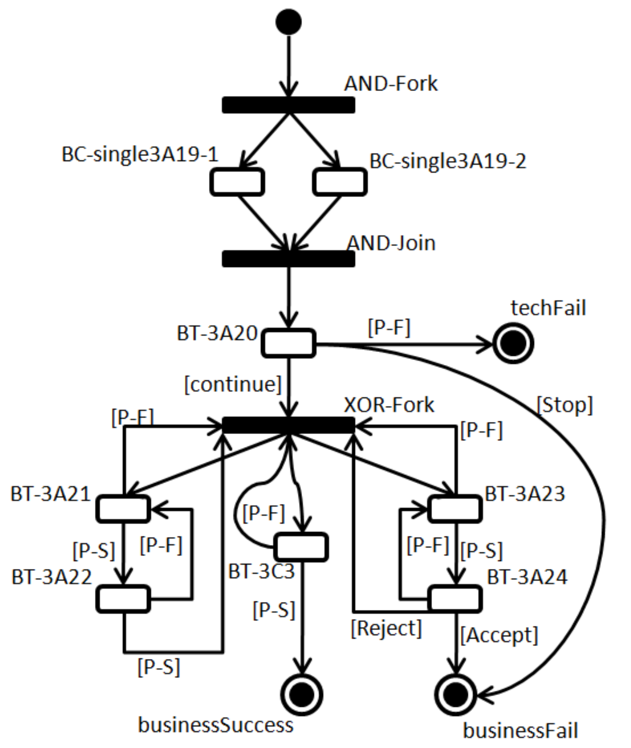

Figure 1. ebBP-Reg Use Case ([true] guards left out)

5.1 describes the organization of integration components and section 5.2 shows how to derive the components' WS-BPEL code from an ebBP-Reg model.

5.1. Integration Architecture

The integration architecture for performing ebBP-Reg choreographies refines the approach of a previous paper [4] that is based on the concept of control processes and reflecting ebBP's modular structure in the organization of integration components.

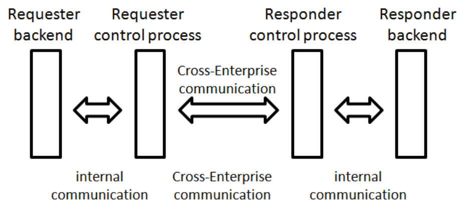
Requester backend
Requester control process
Responder control process
Responder backend
Cross-Enterprise communication
internal communication
Cross-Enterprise communication
internal communication

Figure 2. Basic Integration Architecture

A control process as depicted in figure 2 encapsulates an integration party's control flow logic and uses existing functionality (abstractly denoted 'backend' in figure 2) for including business logic such as the creation and validation of business documents and for being notified about real-world events such as that a new BC or BT have to be performed. Control processes implement the complete cross-enterprise communication and therefore relieve the burden of dealing with complex distributed computing issues from business applications. For exemplifying the basic interaction between control processes and backends assume that the requester backend of figure 2 detects the need to perform a BTA and signals this need to the requester control process. The requester control process then coordinates the business document exchange with the responder control process. Both control processes interact with the according backend processes for fetching/delivering/creating business documents. At the end, the control processes deliver the result of the BTA to the backend components. Reflecting ebBP's modular structure in the organization of control processes can be done by creating a separate pair of control processes for each BTA/BCA of a BC as proposed in [4]. We present the set of interfaces needed for modularizing control processes and specify which messages have to be exchanged via these interfaces for implementing ebBP-Reg choreographies (section 5.2).

Figure 3 shows the set of interfaces that can be used for implementing an ebBP-Reg choreography as an UML component diagram. The use case of section 4 is used as example. Each control process is depicted as a separate component with the stereotype <<CtrlProc>> and for each type of component a control process interacts with two interfaces are created, one interface per communication direction. The seller's control process *BC-controlFlowTestS* (the center component of figure 3) implements the top-level collaboration of the use case, i.e., it coordinates the sequence of executing the specified BCAs and BTAs. Using interfaces `cftS-BE-Client` and `cftS-BE-Callback`, the *BC-controlFlowTestS* receives the trigger for starting the overall collaboration as well as BTAs and it sends back acknowledgements to the backend once the requested BTAs have been started. Before such acknowledgements can be sent to the seller's backend, the *BC-controlFlowTestS* control process informs the buyer's top level control process *BC-controlFlowTestB* that a new BTA is needed, waits until *BC-controlFlowTestB* signals that a new instance of the requested BTA's buyer control process has been created, and then sets a up a new instance of the requested BTA's seller control process. For example, for starting the use case's BTA *BT-3A20*, *BC-controlFlowTestS* would wait until *BackendS* requests this BTA, request a new instance via interface `BC-controlFlowTestB` and await the acknowledgement via `BC-controlFlowTestS`, then inform the backend that the BTA control process has been set up and finally create a new instance of *BT-3A20Requestor*. Note that the seller's BTA control process indeed is created after informing the backend because the first message exchange between *BT-3A20Requestor* and *BackendS* is initiated by the BTA control process. Conversely, when a new BCA has to be started, e.g., *BC-single3A19-2* of the use case, the top level control processes would first start the BCA's control processes and then inform the backends because the first message exchange between backends and BCA control processes is initiated by the backend components.

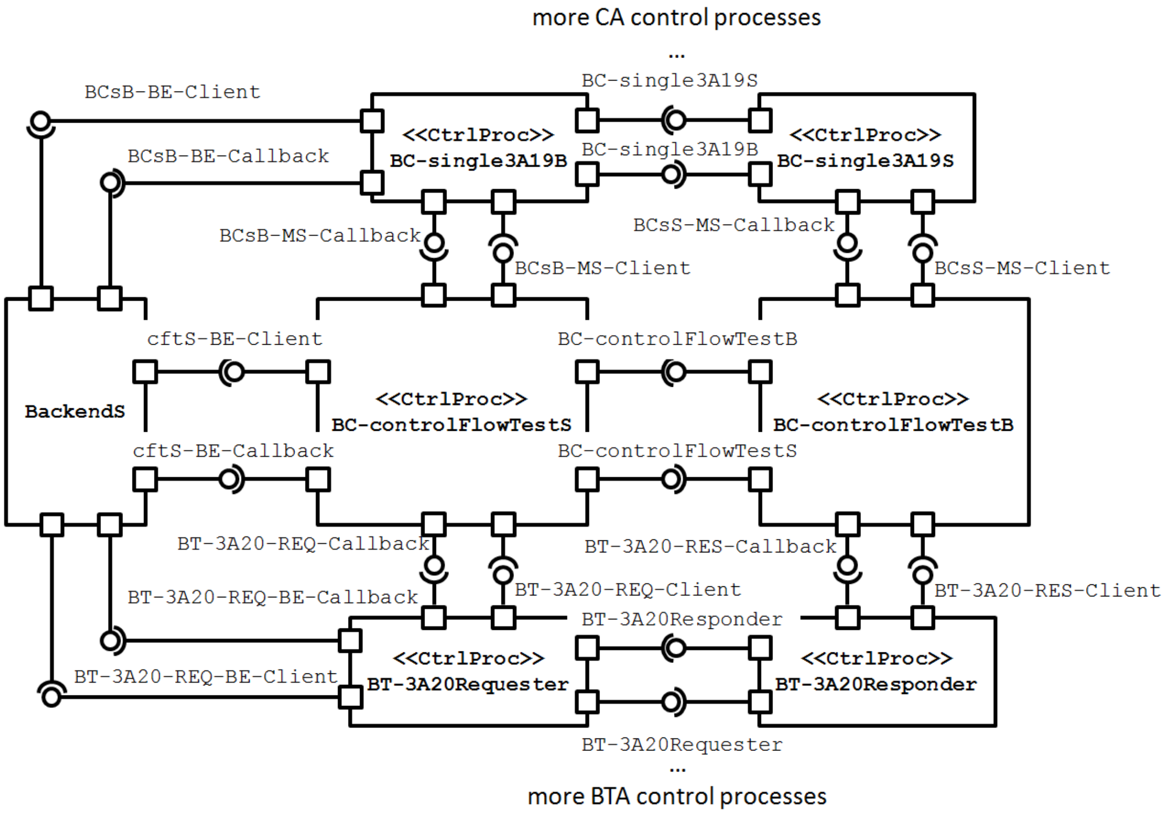

Figure 3. Interfaces for Interacting with the Main Control Process

Figure 3 only shows control processes for one type of BTA (BT-3A20) and BCA (BC-single3A19-2), but for any additional type of BTA/BCA additional control process components would have to be installed. Once the pair of control process instances has been created for a new BCA/BTA via the according *-Client interfaces, control is passed on to these control processes which, again, interact with each other for coordinating more deeply nested activities and with the respective backends for including business logic. Once the lower level control processes have performed the work, control is returned to the higher level control processes via the *-Callback interfaces. Note that figure 3 does not include the interfaces for interacting with the buyer's backend.

5.2. WS-BPEL mapping

WS-BPEL is a natural candidate for implementing the above integration architecture. One WS-BPEL process is used for implementing a particular <<CtrlProc>> component of figure 3 and each component interface (both consumed and offered) is described as a separate WSDL file which is bound to the WS-BPEL process via WS-BPEL's partnerLinkType construct. Message correlation is solved using WS-BPEL's correlation mechanism and by including the message header of listing 1 in every message exchanged between the components of figure 3.

Listing 1. Control Message Header

```
<xs:complexType name="commonMetaBlockType">
 <xs:sequence>
  <xs:element name="RootIdentifier"../>
  <xs:element name="ParentIdentifier"../>
  <xs:element name="InstanceIdentifier"../>
  <xs:element name="ProcessDepth"../>
 </xs:sequence>
</xs:complexType>
```

Message correlation is performed using either the `InstanceIdentifier` or the `ParentIdentifier` field depending on whether communication takes place between components at the same process depth or between parent and child control process. So, *BC-controlFlowTestS* would use the `InstanceIdentifier` field for binding messages from its partner top level control process *BC-controlFlowTestB* and `ParentIdentifier` field for binding messages from, for example, *BT-3A20Requestor*. For this mechanism to work, a BCA control process assigns a new id to the `InstanceIdentifier` field and its own process id to the `ParentIdentifier` field when sending a start message to a lower level control process.
The sequence of message exchanges for implementing an ebBP-Reg choreography is given by specifying the message exchanges for setting up a BCA control process and then defining how any of ebBP-Reg's language constructs would be translated into the control process. For solving concurrency issues, one out of the two control processes for coordinating the activities of a collaboration is assigned the

'leader' role. This only is a technical distinction and does not effect the business semantics of a collaboration. Therefore, an arbitrary algorithm for assigning the leader role could be used or the integration partners even could assign the leader role at build-time. Due to space limitations, we are only presenting the WS-BPEL mapping of a 'leader'-type BCA control process, but the rules for mapping 'non-leader'-type processes are symmetric. Further, the description of how to implement BTAs is not presented because a fully compatible solution is available in [5]. Note that the mapping of ebBP-Reg into WS-BPEL is designed such that it can easily be automated, although the prototype was created manually (cf. section 4). In that sense, the operational semantics of ebBP-Reg is given in terms of interacting WS-BPEL orchestrations.
The control messages to be exchanged are described in table 1.

Ctrl. Msg.	Explanation
cbStart	Starts a BCA control process.
initFail	Signals that starting a root level BCA control process failed.
reqAct	Requests an activity at the leader control process.
actChoice	Sent by a BCA control process to signal that a new BTA shall be started. If both BCA control processes may start a BTA, only the leader process is allowed to send this message.
initAct	Used to distribute the instance id of a new activity.
initAck	Used to acknowledge the start of an activity to the partner control process.
idReq	Requests a new id at the backend.
idRes	Result of an id request.
ttpReq	Requests the ebBP timeToPerform value for a new activity (if not statically assigned).
ttpRes	Result of a timeToPerform request.
actReady	Signals to the backend that a BCA control process has been set up or that a BTA control process immediately will be set up.
bcResult	Reports the result of a BCA.
txResult	Reports the result of a BTA.
txStart	Starts a BTA.

Table 1. Control Messages Overview

5.2.1. Setting up a collaboration level control process. Initially, a BCA control process either receives a `cbStart` message from a superordinate BCA control process or from the backend. In the latter case (which means the process under consideration is a root level process), the `cbStart` message is forwarded to the corresponding BCA control process which in turn informs its backend component about the new collaboration instance. Upon receipt of the `cbStart` message, the correlation sets for identifying the process instance are set up (cf. above). In case the partner BCA control process is not reachable an `initFail` message is returned to the initiator and the BCA control process is terminated. If the collaboration level ebBP timeToPerform value has been set to 'runtime' the BCA control process synchronously requests a `ttpRes` message from its backend using a `ttpReq` message. Then, the main WS-BPEL *scope* of the process is entered that carries a WS-BPEL `onAlarm` for controlling the collaboration's `timeToPerform` timer. If the timeout occurs, the main scope gets interrupted and the BCA control process sends a `bcResult` message to its initiator informing it about the technical failure. The main scope's content consists of a WS-BPEL `while` that continues until a final state is reached. Within this loop, a series of WS-BPEL `ifs` are used to determine the 'state' the collaboration currently is in. Any BCA, BTA, AND-Fork, XOR-Fork and final state that is not located between a matching AND-Fork and AND-Join is considered to be a state. Looking at the use case depicted in figure 1, the AND-Fork at the top or the BTA `BT-3A21` would be considered as states, but not BCAs `BC-single3A19-1/2`. In each of these states, i.e., within the WS-BPEL `if` tags, the code for the respective ebBP-Reg language constructs is mapped. Listing 2 exemplifies the concept referring to the use case of figure 1.

Listing 2. Main loop of a BCA control process
```
<while name="cbWhile">
  <condition>not($inEndState)</condition>
  <sequence name="cbSwitchSeq">
    <if name="cbSwitch-fk-parallel">
      <condition>$cbState = 'fk-parallel'</
          condition>
      <scope name="scope-fk-parallel">
       ... code implementing the parallel
           structure ...
      </scope>
    </if>
    <if name="cbSwitch-bta-bt-PIP3A20">
      <condition>$cbState = 'bta-bt-PIP3A20'
          </condition>
      <scope name="scope-bta-bt-PIP3A20">
       ... code implementing PIP 3A20 ...
      </scope>
    </if>
    ... switch across other 'states' ...
  </sequence>
</while>
```

Eventually, when a final state has been reached (businessSuccess, techFail or businessFail for the above use case) then the global loop condition is set to false, the calling BCA process and/or the backend process are informed about the result and the process is terminated.

5.2.2. Mapping ebBP-Reg's language constructs. Tables 2 and 3 sketch in itemized style the WS-BPEL mapping of the remaining ebBP-Reg language constructs assuming a 'leader'-type BCA control process. A detailed technical example is available via the prototype (see section 4). Remind that decisions and loops are implicitly covered by the evaluation of guards that may emerge from BTAs and BCAs. Also note that, by leveraging standard component interfaces as described in section 5.1, the resulting WS-BPEL processes are fully executable.

ebBP-Reg construct	WS-BPEL Mapping
R5.1) BTA requester role (i.e., the role taken by the BCA control process maps to the BTA requester role as specified in the ebBP-Reg model)	Await backend's `reqAct`; Request new id via `idReq`/`idRes` from backend; Inform partner control process via `actChoice`; Send BTA process id via `initAct` to partner; Await partner's `initAck`; Send `actReady` to backend (because first BTA message will be triggered by the control process); Start BTA control process with `txStart`; Await `txResult` from BTA control process; If ebBP-Reg model only specifies one outgoing transition for the BTA switch process state variable to the next value and continue the main loop; If multiple transitions are specified use multiple `ifs` to evaluate BTA result;
R5.2) BTA responder role	Await partner's `actChoice`; Await partner's `initAct`; Take over id from `initAct` and send actReady to backend and start BTA control process using `txStart`; Send `initAck` to partner control process; Await `txResult` from BTA control process and process it as described above;
R5.3) BCA	Immediately coordinate (leader control process) the execution of the BCA once the according 'state' of the BCA's main loop has been reached (cf. listing 2). Therefore, request new id via `idReq`/`idRes` from backend; If the choreography's timeToPerform value of the BCA has been set to 'runtime', request the timer value via `ttpReq`/`ttpRes` from backend; Inform partner control process via `actChoice`; Send new BCA's process id via `initAct` to partner; Await partner's `initAck`; Start BCA control process with `cbStart`; Send `actReady` to backend; Await `cbResult` from BCA control process; If ebBP-Reg model only specifies one outgoing transition for the BCA switch process state variable to the next value and continue the main loop; If multiple transitions are specified use multiple `ifs` to evaluate BCA result;

Table 2. WS-BPEL Mapping

ebBP-Reg construct	WS-BPEL Mapping
R5.4) Event-Based Choice	**1.** If all BTAs the XOR-Fork links to have associated the BTA requester role with the role of the BCA control process and the XOR-Fork does not link to BCAs then await backend's `reqAct`; Check the `reqAct` for the requested activity and then perform R5.1. **2.** If all BTAs the XOR-Fork links to have associated the BTA responder role with the role of the BCA control process and the XOR-Fork does not link to BCAs then await the partner's `actChoice`; Afterwards, perform R5.2. **3.** Otherwise, either integration partner is allowed to trigger one of the next BTAs/BCAs. Then, use a WS-BPEL `pick` to await either a backend's `reqAct` or a partner's `reqAct`. Perform R5.1, R5.2 or R5.3 depending on the type of selected activity.
R5.5) Parallel	Parallel structures are implemented in a threading like fashion; Start the BCAs the choreography's AND-Fork links to by iteratively applying R5.3 (without result collection); Then collect the results of the BCAs using multiple WS-BPEL `receives` in a WS-BPEL `flow`. In case some BCAs have exactly the same type then the according results have to be collected in sequence in order to avoid `conflictingReceives` (cf. [2], sect. 10.4); Afterwards, switch process state variable to the next value and continue the main loop (cf. listing 2);
R5.6) Terminal Node	Set the main loop's condition to false and prepare the result value of the collaboration, i.e., set the choreography's 'name' attribute of the reached terminal node to the result variable.

Table 3. WS-BPEL Mapping

6. Related Work

Before interaction style choreography languages such as ebBP were considered for specifying B2Bi processes, interconnection style languages as used in [6] or [7] were applied. Interaction choreographies apply a paradigm of a single process consisting of interactions only that then has to be dissected into the participants' orchestrations while interconnection choreographies consider local processes (or at least activity lanes) of the participants that then are connected (cf. [8]). This is a fundamental change in perspective.

Instead of ebBP, other choreography languages like WS-CDL [9] or BPEL4Chor [10] could be used, but these do not offer B2Bi specific concepts and are tied to Web services as implementation technology. However, Web services are not the only B2Bi communication technology.

UN/CEFACT's UMM [11] is an alternative, graphical B2Bi choreography standard that is tightly related to the textual ebBP standard. Giving up the benefits of visual modeling, we prefer to work with ebBP as a common B2Bi choreography interchange format that may be derived from various visual languages and seems to be more suitable for further handling by analysis, transformation and execution machinery.

There are many approaches that impose syntactic restrictions for ensuring validity and executability of models. Prominent representatives are [12] and [13] who structure control flow using pairwise corresponding control flow nodes. Our approach is different in only requiring pairwise corresponding control flow nodes for parallel structures and allowing arbitrary graphs for decision and loop structures.

Other publications focus on the translation of graph-oriented languages (such as ebBP) to block-oriented languages (such as WS-BPEL). For example, [14] and [15] both target at identifying a hierarchy of components within (almost) arbitrary graphs. While [14] identify components by looking for unique entry and exit edges, [15] do so by searching for unique entry and exit nodes. Our work is fundamentally different in translating ebBP into WS-BPEL by preserving a graph-like structure that uses threading-like process decomposition for solving concurrency issues instead of trying to identify block structures. The graph-like structure is preserved by using a 'go-to' programming style employing a global while loop for switching across the 'states' of the collaboration protocol which even allows for irreducible loops at the WS-BPEL level. Thus, our approach overcomes limitations concerning the handling of loops as contained in [7] and also avoids the use of WS-BPEL links (cf. [2] section 12.5.1) that are not supported by all contemporary WS-BPEL engines.

[16] gives an overview of different translation strategies between graph-oriented and block-oriented languages. Com-

bining a graph-like structure and threading-like process decomposition at the WS-BPEL level as we do is not described. In [17], a different strategy for modeling and executing ebBP choreographies is presented. [17] introduces explicit ‘shared states’ into ebBP that represent global synchronization points. These models give up strict standard conformance, concurrency and process decomposition for the sake of simplicity and comprehensibility.
In [18] and [19], the translation of UMM models into WS-BPEL is proposed. A formalization of well-formedness rules for ensuring validity and executability of process models is not provided. Finally, there are several publications like [20] or [5] that only target at performing isolated ebBP BusinessTransaction like concepts as WS-BPEL orchestrations and leave out composition.

7. Conclusion and Future Work

This work describes how ebBP can be restricted to a class of binary choreography models (ebBP-Reg) that support process decomposition as well as concurrency and still are executable. Syntactic well-formedness rules that ensure executability as well as a mapping to fully executable WS-BPEL orchestrations are given as well.
Future work comprises adding convenience functionalities at the orchestration level such as allowing administrators to overrule the predefined control flow and providing a tool that automates the mapping of section 5.2. Moreover, the specification of formal ebBP-Reg execution semantics is planned in order to specify clear rules for non-WS-BPEL implementations and for reasoning on the correctness of models using formal methods. Finally, ebBP as a textual standard needs a mapping to visual languages. An assessment of visual representations as well as more real-world use cases is needed to further determine the usability and efficiency of ebBP-Reg.

References

[1] OASIS, *ebXML Business Process Specification Schema Technical Specification*, 2nd ed., OASIS, December 2006.

[2] ——, *Web Services Business Process Execution Language*, 2nd ed., April 2007.

[3] A. Schönberger, “The CHORCH B2Bi approach: Performing ebBP choreographies as distributed BPEL orchestrations,” in *Proc. of IEEE 2010 World Congress on Services (SERVICES), Miami, USA*.

[4] A. Schönberger and G. Wirtz, “Using variable communication technologies for realizing business collaborations,” in *Proc. of IEEE 2009 World Congress on Services (SERVICES PART II)*.

[5] A. Schönberger, G. Wirtz, C. Huemer, and M. Zapletal, “A composable, QoS-aware and web services-based execution model for ebXML BPSS businesstransactions,” in *Proc. of IEEE 2010 World Congress on Services (SERVICES), Miami*.

[6] W. M. P. van der Aalst and M. Weske, “The P2P approach to interorganizational workflows,” in *CAiSE ’01: Proc. of the 13th Int. Conf. on Advanced Information Systems Engineering*, London, UK, 2001, pp. 140–156.

[7] M. Benyoucef and S. Rinderle-Ma, “Modeling e-negotiation processes for a service oriented architecture,” *Group Decision and Negotiation*, vol. 15, pp. 449–467, 2006.

[8] G. Decker, O. Kopp, and A. Barros, “An introduction to service choreographies,” *Information Technology*, vol. 50, no. 2, pp. 122–127, 2008.

[9] W3C, *Web Services Choreography Description Language*, 1st ed., W3C, November 2005.

[10] G. Decker, O. Kopp, F. Leymann, and M. Weske, “BPEL4Chor: Extending BPEL for modeling choreographies,” in *Proceedings of the 2007 IEEE International Conference on Web Services (ICWS), Salt Lake City, USA*, 2007.

[11] UMM, *UN/CEFACT’s Modeling Methodology (UMM): UMM Meta Model - Foundation Module Version 1.0*, 1st ed., UN/CEFACT, 10 2006.

[12] B. Kiepuszewski, A. H. M. ter Hofstede, and C. Bussler, “On structured workflow modelling,” in *Proc. of the 12th International Conference on Advanced Information Systems Engineering (CAiSE)*. London, UK: Springer-Verlag, 2000.

[13] P. Dadam and M. Reichert, “The ADEPT project: a decade of research and development for robust and flexible process support,” *Computer Science - Research and Development*, vol. 23, no. 2, pp. 81–97, 2009.

[14] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, and W. M. P. van der Aalst, “From BPMN process models to BPEL web services,” in *ICWS ’06: Proc. of the IEEE Int. Conf. on Web Services*, Washington, DC, USA, 2006.

[15] J. Vanhatalo, H. Völzer, and J. Koehler, “The refined process structure tree,” *Data Knowl. Eng.*, vol. 68, no. 9, pp. 793–818, 2009.

[16] J. Mendling, K. B. Lassen, and U. Zdun, “On the transformation of control flow between block-oriented and graph-oriented process modelling languages,” *IJBPIM*, vol. 3, no. 2, pp. 96 – 108, 2008.

[17] A. Schönberger, C. Pflügler, and G. Wirtz, “Translating shared state based ebXML BPSS models to WS-BPEL,” *(to appear in) Int. Jour. of Business Intelligence and Data Mining*, vol. 5, no. 4, 2010.

[18] B. Hofreiter and C. Huemer, “Transforming UMM Business Collaboration Models to BPEL,” in *Proceedings of the OTM Workshop on Modeling Inter-Organizational Systems (MIOS 2004)*. Volume 3292 of LNCS: Springer, October 2004.

[19] ——, “A model-driven top-down approach to inter-organizational systems: From global choreography models to executable BPEL,” in *Joint Conference on E-Commerce Technology (CEC) and Enterprise Computing, E-Commerce, and E-Services (EEE)*. Crystal City, USA: IEEE, 2008.

[20] B. Hofreiter, C. Huemer, P. Liegl, R. Schuster, and M. Zapletal, “Deriving executable BPEL from UMM business transactions,” in *IEEE Conference on Services Computing (SCC)*, 2007.