Translating Shared State Based ebXML BPSS models to WS-BPEL

Andreas Schönberger
Distributed and Mobile Systems Group
University of Bamberg
Bamberg, Germany
andreas.schoenberger@uni-bamberg.de

Christoph Pflügler
Inter-organizational Systems Group
University of Augsburg
Augsburg, Germany
christoph.pfluegler@wiwi.uni-augsburg.de

Guido Wirtz
Distributed and Mobile Systems Group
University of Bamberg
Bamberg, Germany
guido.wirtz@uni-bamberg.de

ABSTRACT

Business-to-Business integration (B2Bi) as a core concept of Supply Chain Management (SCM) is a key success factor for enterprises today. Frequently, choreography models are used for agreeing about the overall message exchanges among integration partners while executable orchestration models derived from the choreography govern the local message flow of each individual participant. Today, ebXML BPSS (ebBP) as a dedicated B2Bi choreography language and WS-BPEL as the de-facto standard for Web service based orchestration modeling provide the technological basis for integrating choreographies and orchestrations in the B2Bi domain.
This paper introduces the concept of *partner-shared states* into ebXML BPSS (ebBP) choreography modeling in order to enable complex integration scenarios and shows how to implement these using Web services and WS-BPEL technology. *Shared states* explicitly represent the effect of business document exchanges, provide natural synchronization points for attaching admissible message exchange actions, and allow for controlling distributed timeouts as well as comprehensibly communicating the interaction's progress. We provide a workaround for modeling shared states in an ebBP schema compliant way as well as an ebBP schema extension that enables intuitive and straightforward models. A formalization of *shared state*-based ebBP models is introduced as concise basis for automatically translating extension-based ebBP models into workaround-based ebBP models. An operational semantics for *shared state*-based ebBP models using this formalization is utilized for comprehensibility because ebBP itself does not define clear semantics.
A prototypic realization of this semantics has been implemented by means of a translation tool generating a distributed WS-BPEL implementation of *shared state*-based ebBP models. The overall approach is evaluated using a RosettaNet PIP based use case.

iiWAS 2009 Kuala Lumpur - Special Issue

Keywords

B2Bi, ebXML BPSS, WS-BPEL, state based modeling, translation

1. INTRODUCTION

In today's competitive world, the success of an enterprise heavily depends upon effective integration along the enterprise's supply chain (cf. [19]). B2Bi as a core task of SCM (cf. [23]) therefore deserves special attention by industry and academia. [11] even find that *"The risk of not having IT-enabled SCM is enormous both in terms of survival and productivity of an organization"*. B2Bi particularly addresses the integration of processes crossing enterprise boundaries where central IT infrastructure for integration partners frequently is not available. Further, personnel from different enterprises with differing background and terminology is participating. According to [42], challenging requirements concerning various forms of *consistency* result from this situation, among others:

1. Heterogeneity between communication systems implementing message exchanges has to be overcome.
2. The state of each integration partner's systems has to be aligned in a transactional way at runtime.
3. Integration partners have to agree upon the type and order of business document exchanges.
4. Integration partners' local process definitions have to be compatible with respect to type and order of business document exchanges.

Requirements 1 and 2 frequently are implemented by using dedicated integration technologies, such as Web services, and by leveraging distributed computing standards like WS-ReliableMessaging [32] or WS-AtomicTransaction [30]. Requirements 3 and 4 can be addressed by introducing the

dichotomy of choreography and orchestration to the B2Bi domain (cf. [34]). While a choreography model of a B2Bi collaboration can be used to capture the publicly visible message exchanges from a global perspective, an orchestration model specifies the types and sequence of the messages a single integration partner is supposed to receive and send. This separation of concerns allows to concentrate on the purpose of collaborations at the choreography level (requirement 3) and to apply tooling for either generating choreography-compliant orchestrations or analyzing the compliance of existing orchestrations to the choreography model afterwards. According approaches have been proposed using different combinations of choreography and orchestration languages, e.g., Petri Nets for both, choreography and orchestration [45], Let's Dance and WS-BPEL [49], WS-CDL and BPEL [37], BPMN and BPEL4Chor [9] or BPMN, WS-CDL and BPEL [21].
In the B2Bi domain, the combination of ebXML BPSS (ebBP) [27] as choreography language and WS-BPEL (BPEL) [31] as orchestration language is particularly promising. ebBP offers domain-specific concepts in a declarative, technology-agnostic way (cf. section 2). For example, so-called BusinessTransactions can be used for specifying the exchange of a business document and an optional response document. Several parameters can be defined for BusinessTransactions such as so-called business signals for informing the sender of the message about the state of processing at the message receiver, or B2Bi-related Quality-of-Service (QoS) parameters like integrity or privacy. While ebBP can be used to demand the implementation of such features at the orchestration level, the details of implementation are deliberately left unspecified. This allows for a broader range of implementation technologies at the orchestration level than Web services only. In practice, technologies like ebMS [26, 29], AS2 [24] and others are used as well. Still, Web services as a dedicated interface technology are highly advantageous in terms of overcoming platform heterogeneity and loose coupling. While Web services once emerged as a stateless interaction paradigm with poor support for QoS, stateful interactions using BPEL and sufficient support of B2Bi-related QoS is available [39]. In consequence, the combination of ebBP or ebBP-like choreography models and BPEL as orchestration model already has been proposed in literature ([18], [12], [14], [15], [16], [43]).
Until now, the importance of state has not sufficiently been regarded in these tool chains. In prior work [41], we have proposed the concept of so-called *shared states* concertedly reached/left by integration participants for explicitly modeling the state changes realized by performing BusinessTransactions (business document exchanges). Explicit modeling of shared states is beneficial in different ways:

- **Shared states explicitly capture the effect of performing BusinessTransactions.**
  This is helpful for communication and agreement among personnel of different enterprises. For example, the sequences of BusinessTransaction executions that lead to a valid contract document can be deduced from analyzing the execution paths that lead to an explicitly modeled shared state *Contract*.

- **Shared states enable intelligible communication of progress.**
  Monitoring of B2Bi processes requires intelligible communication of progress. An executive overview of progress can be achieved more easily by signaling the current shared state, e.g., whether a *Quote* or *Contract* state has been reached, than by signaling the sequence of BusinessTransaction executions together with the business documents exchanged and the business document evaluation rules for assessing the implications of a business document's content.

- **Shared states allow for the specification of distributed timeouts.**
  B2Bi is the connection of business processes that run on scarce resources. Consistently, resource reservations frequently have a limited time horizon. Release of resources without requiring successful execution of BusinessTransactions can be specified by attaching timeouts to shared states. For example, a shared state *Quote* could be specified to be left after 3 days if no BusinessTransactions lead to a follow-on state.

- **Shared states provide natural synchronization points for specifying control flow.**
  The selection of BusinessTransactions that are admissible at a particular point in time typically depend on the integration partners' systems states, e.g., whether a valid contract document is available or not. It is easier to control the execution paths that emerge from an explicitly modeled shared state than controlling all execution paths that continue all possible execution paths that lead to a particular state.

This paper's research topic is the applicability of shared states to the ebBP-BPEL tool chain. As shared states are not naturally supported by ebBP, a workaround for representing shared states in an ebBP compliant way is presented first. In order to allow for more straightforward and intuitive modeling, an ebBP schema extension for modeling shared states is described as well. Based on the concept of shared states, a modeling approach for a restricted set of ebBP collaborations that enable real-world size B2Bi processes is developed. The class of collaborations that comply with our approach is concisely characterized by a formalization of shared state based ebBP models. This formalization also lays the foundation for describing the conversion of shared states modeled by means of our ebBP schema extension into ebBP compliant models, for precisely capturing the meaning of shared state based collaborations by means of an operational semantics, and for specifying a distributed BPEL-based implementation of ebBP models. The main characteristics of an integration architecture that allows for performing BPEL-based ebBP implementations are discussed and a prototypic translation engine that generates BPEL implementations from ebBP choreographies is presented. This prototype proves the realizability of the operational semantics defined and a real-world sized use case is applied for evaluating practical relevance.
This paper is an extended version of [35] and the main new material is the following:

- A formalization of shared-state based ebBP collaborations that precisely captures the set of valid models.

- An ebBP schema extension that allows for more straightforward and intuitive modeling of shared states than the workaround presented in [35].

- An algorithm for converting shared state-based collaborations modeled by means of the ebBP extension into ebBP compliant models.
- An operational semantics for shared-state based ebBP collaborations that facilitates comprehensibility and model interchange.
- A refined description of the algorithm for deriving BPEL implementations from ebBP collaborations.

The rest of the paper is organized as follows. A short introduction into ebBP and BPEL as the core technologies of this approach is given in section 2. Section 3 presents the use case for evaluating the work at hand and exemplifies the benefits of shared state based modeling. An ebBP schema compliant XML model of shared states as well as the according schema extension based model is presented in section 4. Also, an intuitive description of valid shared state-based collaborations is given that is formalized in section 5. The formalization is accompanied by an algorithm for converting shared state-based collaborations modeled by means of the ebBP extension into ebBP compliant models as well as the operational semantics of shared state-based collaborations. Section 6 describes an integration architecture for performing ebBP choreographies using distributed BPEL processes and section 7 shows how to automatically generate those BPEL processes. The evaluation of our approach is presented in section 8 by discussing the results of implementing the proposed integration architecture and an ebBP2BPEL translator, by analyzing the complexity of the algorithms presented, and by investigating the complexity reduction achieved by using our ebBP schema extension for shared states instead of the ebBP compliant workaround. Subsequently, section 9 discusses related work and section 10 concludes and points out directions for future work.

2. BASICS

ebBP and WS-BPEL are core to our approach and thus explained in more detail in this section. ebBP[27] is part of the modular ebXML standard suite aiming at creating a single global electronic market[1]. It allows for a modular, XML-based specification of choreographies based on the concept of *BusinessTransactions* (BT) exchanging business documents and respective signals. A typical ebBP choreography consists of declarations of *BusinessDocuments* and *Signals* (a) a specification of different *BusinessTransaction* types (b) which incorporate the declared business documents and signals and *BusinessCollaborations* (c) defining the actual choreography incorporating the previously defined transaction types.

(a) **Business Documents and Signals:** ebBP allows for the incorporation of XML-based business documents and signals specified using different technologies such as Document Type Definition, XML Schema Definition, or Schematron. It also allows for appending document specific information using, e.g., XPath expressions.

(b) **Business Transactions:** The ebBP specification offers several transactions patterns which have to be customized in order to meet specific needs. This customization consists of assigning business documents and signals, as well as the specification of quality of service (QoS) parameters. ebBP further provides facilities allowing for the definition of additional transaction patterns, which we take advantage of in our approach. Further, ebBP statically defines the *RequestingRole* and *RespondingRole* for BusinessTransactions. The *RequestingRole* always is the sender of a BusinessTransaction's first business document whereas the *RespondingRole* always is the receiver of that document.

(c) **Business Collaborations:** *BusinessCollaborations* (BC) with at least two roles (integration partners) are used to build complex processes using constructs like *Decision*, *Join* or *Fork* which link so-called *BusinessTransactionActivities* (BTA) or *BusinessCollaborationActivities* (CA). BTAs and CAs add execution parameters such as timeouts to BTs or BCs, and map the roles of the performing *BusinessCollaboration* to the roles of the performed activity.

WS-BPEL[31] is an XML-based, standardized workflow language used for the definition of executable (or abstract) processes composed by a series of incoming or outgoing Web Service calls. It allows for synchronous as well as asynchronous interactions using constructs like `invoke`, `receive` or `onMessage`. Constructs like `sequence`, `if` or `while` are used to specify the control flow between these interactions. WS-BPEL, usually along with extensions, has been incorporated into the middleware products of almost all major software vendors.

3. USE CASE

The use case for evaluating the work at hand is based on RosettaNet *Partner Interface Processes* (PIPs). RosettaNet is a non-profit standards organization dedicated to supporting B2B integration and endorsed by over 500 companies worldwide. RosettaNet defines business messages and rules for its electronic exchange. PIPs, classified in clusters like cluster 3 Order Management and segments like segment 3A Quote and Order Entry, describe the application context, the content and the parameters for the electronic exchange of one or two business documents. A use case consisting of nine shared states and nine PIPs has been created exemplifying shared state-based modeling. The RosettaNet document type definitions have been imported by means of ebBP *BusinessDocuments* and their flow has been remodeled using ebBP *BusinessTransactions*. The use case is taken from RosettaNet PIP segment 3A (*Quote and Order Entry*) and models a process for negotiating a contract. The size of standard processes as defined by the Northern European Subset[2] (NES) is comparable to our use case, so the use case's size can be considered to be realistic.

The use case represents a binary collaboration. The two business partners take the roles of *buyer* and *seller* throughout the whole collaboration. The overall goal of the composition is the negotiation of a contract and of contract changes. The collaboration terminates as soon as the buyer has received the goods and services he requested. The description of the use case starts with explaining the usage of the selected PIPs.

PIP 3A1: Request Quote

[1] http://www.ebxml.org/

[2] `http://www.nesubl.eu/`

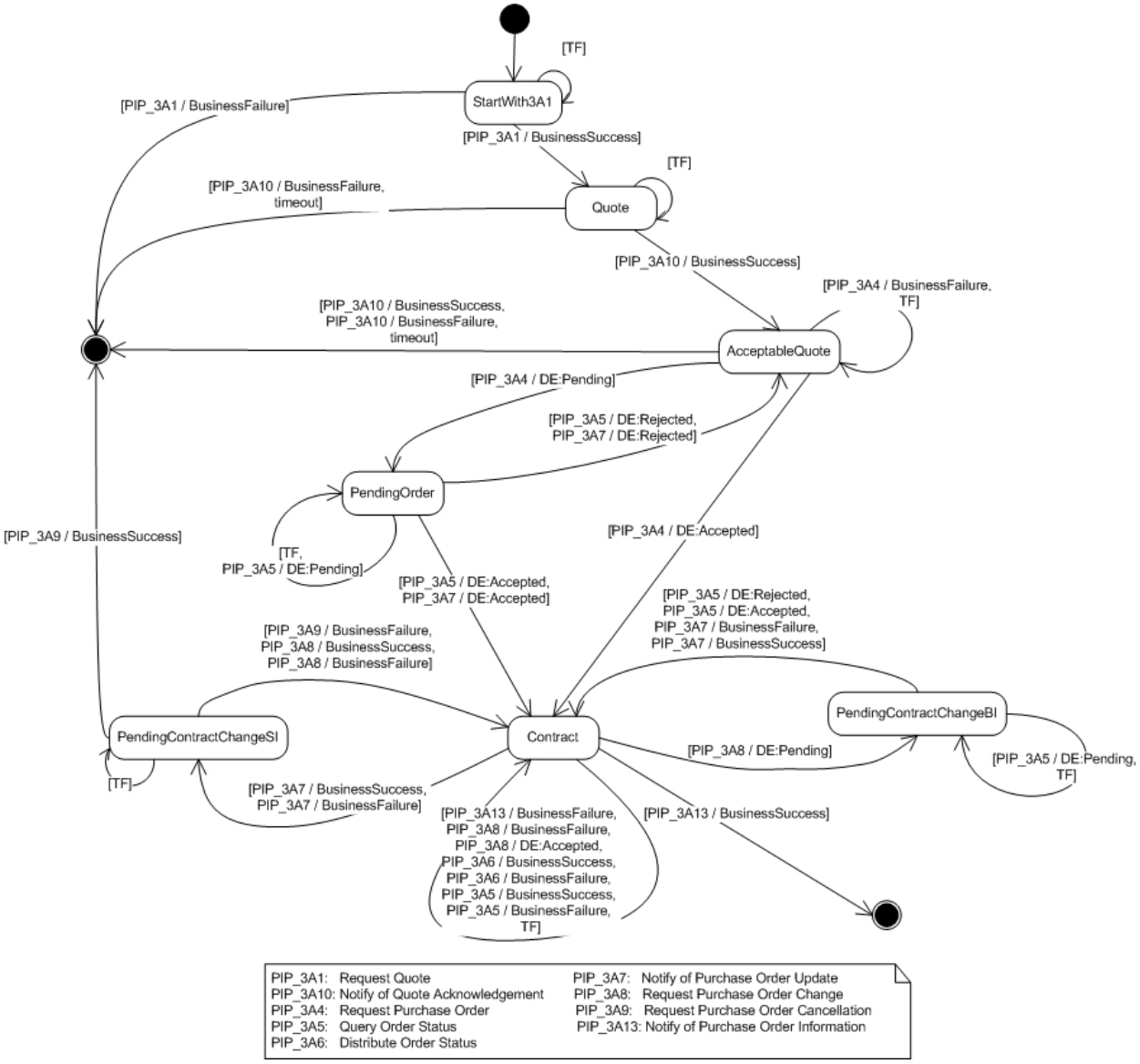

Figure 1: Use case for evaluating the work at hand

This PIP is used to start the collaboration. The buyer requests a quote for some particular goods and services. The seller answers with a response document either representing a *BusinessSuccess* or a *BusinessFailure*. (ebBP allows for associating the *BusinessSuccess* or *BusinessFailure* result values with types of response documents). In the former case the seller may reserve resources for the buyer, but the buyer is not obliged to accept the quote. In the latter case the collaboration is terminated immediately.

PIP 3A10: Notify of Quote Acknowledgement

If the buyer has received a valid quote she is obliged to use this PIP to inform the seller whether the quote is generally acceptable or not. If not, the collaboration is terminated immediately. Otherwise the seller extends the reservation of resources and waits for an order. The buyer is still not obliged to accept the quote.

PIP 3A4: Request Purchase Order

This PIP can either be used to start the collaboration or to sign a contract after having confirmed a quote to be acceptable with the help of PIP 3A10. The buyer sends a quote to the seller that can be answered with either an *Accepted*, a *Rejected* or a *Pending* message in corresponding ebBP *DocumentEnvelopes* (ebBP allows for capturing the result of a BTA by referring to the DocumentEnvelope types of the exchanged messages). If the answer is *Accepted* the parties have signed a legally binding contract. If the answer is *Rejected* the collaboration terminates immediately. If the answer is *Pending*, the buyer waits until the seller notifies her about the decision using PIP 3A7 or the buyer queries the decision with PIP 3A5.

PIP 3A5: Query Order Status

The buyer can use this PIP either if there is a valid contract, or if the decision of the seller about a quote (PIP 3A4) is still pending or if the decision of the seller about a contract change request (PIP 3A8) is still pending. The answer of the seller has to be evaluated depending on which of these

situations applies.
In the first case, the seller can either provide new information about order progress or just tell that no progress has been achieved. In case of the other two situations the seller can either send an *Accepted*, a *Rejected* or a *Pending* message. If an order has not yet been decided upon, a new contract is signed (*Accepted*), the collaboration is terminated immediately (*Rejected*) or the decision is further postponed (*Pending*). If a contract change request has not yet been decided upon, either the current contract is replaced by a new one (*Accepted*), the current contract remains valid (*Rejected*) or the decision is further postponed (*Pending*).
PIP 3A6: Distribute Order Status If there is a valid contract, the seller can use this PIP to communicate information about order progress to the buyer.
PIP 3A7: Notify of Purchase Order Update The seller must trigger this PIP if she has sent a *Pending* message in PIP 3A4 or 3A8 before. The seller may reply using an *Accepted* or a *Rejected* message. Analogously to PIP 3A5, a new contract is then signed (*Accepted*) or the collaboration is terminated/the current contract remains valid (*Rejected*). Further, the seller can use PIP 3A7 to request contract changes. As PIP 3A7 is a Single-Action Activity, i.e., only one business message can be exchanged, the buyer cannot directly answer such a request. To answer a contract change request, the buyer must either use PIP 3A8 or PIP 3A9.
PIP 3A8: Request Purchase Order Change. This PIP is usually used by the buyer to request a contract change. The seller can then either send *Accepted* to confirm the change, *Rejected* to keep the current contract or *Pending* to postpone the decision.
Further, this PIP is used to answer a contract change request initiated by the seller with PIP 3A7. If the buyer wants to reject the change request, she sends a *Purchase Order Change Request* message that exactly contains the data of the current contract. The current contract then remains valid no matter what the seller answers. To be concise, the seller should send a *Rejected* message. If the buyer is about to accept the change request of the seller (with modifications) then she sends a *Purchase Order Change Request* message (with modifications). The seller may then only respond with an *Accepted* message to sign a new contract or with a *Rejected* message to keep the current contract.
PIP 3A9: Request Purchase Order Cancellation
This PIP is usually used by the buyer to cancel a current contract. Moreover, the buyer can offer the cancellation of a contract instead of a contract change requested by the seller (PIP 3A7).
In both cases the seller can only send an *Accepted* message to rescind the contract or a *Rejected* message to keep the current contract.
PIP 3A13: Notify of Purchase Order Information
The buyer uses this PIP to notify the seller about processing updates or the fulfillment of the contract.

The use case is visualized in a state machine-like manner in figure 1. The start of the collaboration is represented by the unique start element. Each shared state is represented as a state and the executions of PIPs as BTAs are represented as transitions. The event part of a transition is used to name the BTA (PIP) to be executed and the guard part of a transition is used to capture the outcome of BTAs. As decisions are not explicitly visualized, there may be multiple transitions for the same shared state that are triggered by the same event. The condition guards of the particular transitions, however, are mutually exclusive. The permissible ebBP guard values for the use case are *AnyProtocolFailure* (denoted `TF`), *BusinessFailure* or *BusinessSuccess*. *AnyProtocolFailure* captures arbitrary technical problems during performing BTAs. If no such problems occur, *BusinessSuccess* indicates that integration partners did achieve their goals from a business point of view whereas *BusinessFailure* indicates they didn't. Finally, guard values based on DocumentEnvelopes (denoted with a leading `DE:`) that relate to the content of the latest business document exchanged using suitable XPath expressions are allowed as well. Two final states are used to represent an ebBP Failure state (on the left-hand side) and an ebBP Success state (on the right-hand side). Although the execution of `PIP_3A9` in state `PendingContractChangeSI(SellerInitiated)` may terminate with a *BusinessSuccess* guard value, it still represents a failure from the overall collaboration perspective.

Using this use case the benefits of shared state based modeling can easily be demonstrated:
Shared states explicitly capture the effect of performing BusinessTransactions.
Obviously, the business contents exchanged in BTAs govern the state alignment actions to be performed in participating integration systems. Shared states can be used to represent the result achieved by having exchanged particular content. For example, PIP 3A4's request document may be answered by different DocumentEnvelopes inidicating *Pending*, *Accepted* or *Rejected*. Using shared states, collaboration partners can express that a *Pending* message results in a *PendingOrder* state that does not automatically result in a *Contract*.
Shared states allow for intelligible communication of progress.
Process visibility and analysis necessitate communication of progress information about active business processes. As the effect of BTAs may depend on the actions taken previously, communicating the type and content of a BTA is not necessarily sufficient for uniquely determining progress. In the use case scenario, having performed PIP 3A5 with result *DE:Pending* may lead to shared states *PendingOrder* or *PendingContractChangeBI* depending on the previous state.
Shared states allow for the specification of distributed timeouts.
B2Bi collaborations may necessitate the reservation of resources, e.g., in shared states *Quote* and *AcceptableQuote* of the use case. By attaching timeout values to these shared states, a time limit for resource release may be defined in case collaboration partners do not trigger BTAs in a timely manner.
Shared states provide natural synchronization points for specifying control flow.
As BTAs depend on and modify system state, shared states are a natural way for specifying control flow, i.e., the sequences of business document exchanges that lead to the same state and the business document exchanges that require the same state as precondition. For example, assume that the collaboration depicted in figure 1 has progressed to state *Contract*. Then the sequence of PIP 3A8 and PIP 3A5 (change initiated by the buyer role) may lead to a new contract as well as the sequence of PIP 3A7 and 3A8 (change

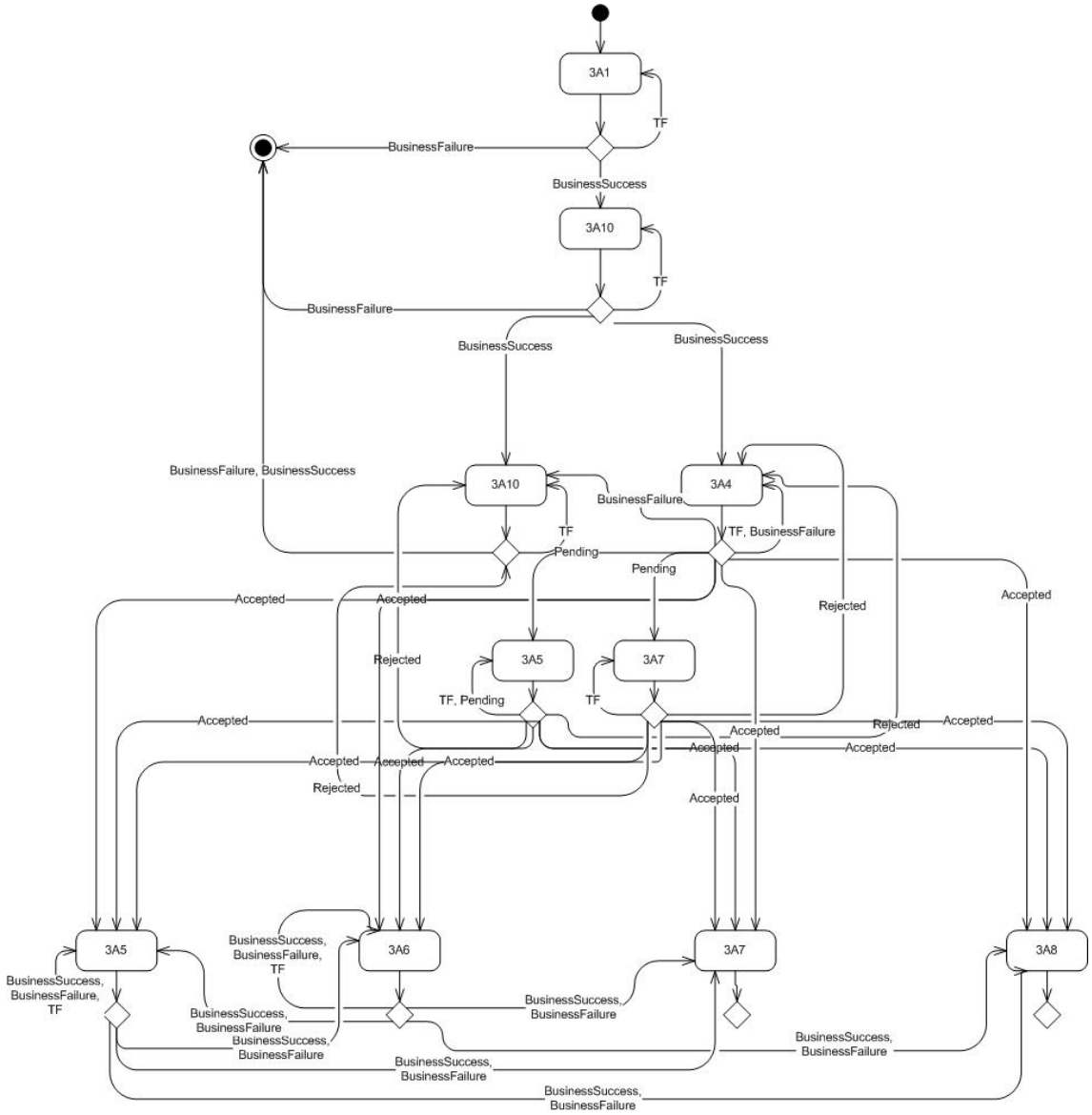

Figure 2: Alternative Modeling of an Excerpt of the Use Case Demonstrating Control Flow Explosion

initiated by the seller role). Conversely, PIP 3A5 is only admissible in state *PendingContractChangeBI* but not in *PendingContractChangeSI*.

In case such commonalities need to be expressed, alternative modeling approaches without constructs for joining/splitting control flow may lead to far more complex models. Figure 2 depicts an alternative model for the states *StartWith3A1*, *Quote*, *AcceptableQuote* and *PendingOrder* of figure 1 only. In figure 2, control flow is specified only by connecting BTAs (rounded boxes) and decision nodes (diamonds) and attaching guards to the transitions. The complexity explosion due to removing shared states can be explained by the fact that an extra transition must be introduced for each BTA that is allowed in a particular shared state. For example, BTAs 3A10 and 3A4 are admissible in *AcceptableQuote*. Therefore, after having performed 3A10 successfully (in shared state *Quote*), two paths have to be specified for allowing both 3A10 AND 3A4. Note that extra control flow links also are a result of different possible BTA results and BTAs that are applicable in different states. For example, there are two links from BTA 3A4 to a 3A7 node in figure 2. This is due to the fact that 3A7 may be performed in state *PendingOrder* (i.e., result *Pending* for 3A4) and in state *Contract* (i.e., result *Accepted* for 3A4).

An obvious way for working around this situation without shared states is using pure control flow pseudo-nodes like ebBP *Join* for merging alternative paths or ebBP *Fork* for splitting up alternative paths. Therefore, one would only have to create transitions from the ebBP *Decision* nodes to the matching *Join* nodes and then using *Fork* nodes as needed for splitting up control flow again. Unfortunately, this is not permitted by ebBP because transitions may only reference BTAs or CAs (cf. toBusinessStateRef/fromBusinessStateRef constraints in [27] sec. 3.8.2).

4. INFORMAL ebBP MODELS

A shared state of a B2Bi collaboration as introduced in [41] is a synchronization point that represents alignment of information items among integration partners and captures the progress of a collaboration. Integration partners use BTAs to consistently align information and thus concertedly leave and reach shared states. While the preceding sections motivated the use of shared states, this section discusses the modeling of shared states using ebBP.

4.1 ebBP Compliant Shared State Model

There is no ebBP construct that directly matches the concept of a shared state so these have to be emulated. Generally speaking, a state can be modeled with an ebBP *Join* construct followed by a *Fork* construct. However, ebBP prohibits directly linking *Joins* and *Forks* as the corresponding *fromBusinessStateRef* and *toBusinessStateRef* attributes may only reference BTAs or CAs (cf. [27] sec. 3.8.2). To overcome this constraint in a standard compliant manner, a workaround can be used. The concept of an *EmptyBTA* based on the extensible ebBP transaction type *DataExchange* is introduced that serves as a target for linking to a shared state and for connecting the *Join* and *Fork* of a shared state.

Listing 1 shows the ebBP representation of the shared state *Quote* (cf. above and figure 1). The *EmptyBTA* before the shared state's *Join* is used as target of ebBP *Decisions* that are not allowed to directly link to *Joins* (cf. [27] sec. 3.8.2). The shared state's *Join* links to another *EmptyBTA* that is connected to the shared state's *Fork*. This *Fork* then specifies a *ToLink* for every BTA that is permissible to be performed from this shared state. Shared state timeouts, i.e., the point in time when shared states should be left without performing a BTA, can also be specified on this *Fork*. In case such a timeout occurs, it has to be switched to the *EmptyBTA* defined in the *ToLink* that carries the according *timeout ConditionExpression*.

Employing two *EmptyBTAs* allows for different semantics when linking to a shared state with respect to timeouts: In case of linking to the *EmptyBTA* before a shared state, its timeout is reset whereas linking to the *EmptyBTA* within the shared state does not have this effect. The latter case is particularly useful if protocol failures occur during performing a subsequent BTA which means that the shared state actually has not been left. Note that ebBP *Joins* and *Forks* are only used for modeling states and are not allowed elsewhere in the collaboration description.

Listing 1: ebBP Compliant Model of a Shared State

```
1
2  <!-- State Quote -->
3  <BusinessTransactionActivity
       businessTransactionRef="empty"
4      nameID="empty_before_Quote">
5   <TimeToPerform></TimeToPerform>
6   <Performs currentRoleRef="Buyer"
        performsRoleRef="empty1"/>
7   <Performs currentRoleRef="Seller"
        performsRoleRef="empty2"/>
8  </BusinessTransactionActivity>
9
10 <Join waitForAll="false" nameID="Quote">
11  <FromLink fromBusinessStateRef="
        empty_before_Quote"/>
12  <FromLink fromBusinessStateRef="
        empty_before_Quote"/>
13  <ToLink toBusinessStateRef="
14      empty_in_Quote"/>
15 </Join>
16
17 <BusinessTransactionActivity
       businessTransactionRef="empty"
18     nameID="empty_in_Quote">
19  <TimeToPerform></TimeToPerform>
20  <Performs currentRoleRef="Buyer"
        performsRoleRef="empty1"/>
21  <Performs currentRoleRef="Seller"
        performsRoleRef="empty2"/>
22 </BusinessTransactionActivity>
23
24 <Fork nameID="fork_Quote" type="XOR">
25  <TimeToPerform duration="P3D"/>
26  <FromLink fromBusinessStateRef="
        empty_in_Quote"/>
27  <ToLink toBusinessStateRef="
        BTA_3A10_NotifyOfQuoteAck"/>
28  <ToLink toBusinessStateRef="
        BTA_3A10_NotifyOfQuoteAck"/>
29  <ToLink toBusinessStateRef="
        empty_before_FAILURE">
30   <ConditionExpression
31     expressionLanguage="XPath1"
32    expression="timeout"/>
33  </ToLink>
34 </Fork>
```

4.2 ebBP Extension for Shared States

The motivation of providing an ebBP extension and hence dropping compliance is complexity reduction. In the workaround presented in section 4.1, four different control flow nodes have to be used for specifying a shared state and its outgoing transitions. Using the proposed extension, the same information can be represented by only one node. The new `SharedState` construct (depicted in listing 2) that has been defined is similar to an ebBP *Fork* node with the *type* attribute set to *XOR*.

Listing 2: Extension-based Model of a Shared State

```
1
2  <!-- State Quote -->
3  <SharedState nameID="Quote">
4   <TimeToPerform duration="P3D"/>
5   <FromLink
6    fromBusinessStateRef="
         DECISION_3A10_NotifyOfQuoteAck"
7    stTimeoutReset="false">
8    <ConditionExpression
9     expressionLanguage="ConditionGuardValue"
10    expression="AnyProtocolFailure" />
11  </FromLink>
12  <ToLink toBusinessStateRef="
        BTA_3A10_NotifyOfQuoteAck" />
13  <ToLink toBusinessStateRef="
        Collaboration_FAILURE">
14   <ConditionExpression
15     expressionLanguage="XPath1"
16    expression="timeout" />
17  </ToLink>
18 </SharedState>
```

It reuses the ebBP definitions of *TimeToPerform*, *FromLink* and *ToLink*, but removes the cardinality constraints on the number of *FromLinks* and *ToLinks* completely. Thus, a logical link between any control flow node and a `SharedState` can syntactically either be specified within the `SharedState`, within the control flow node under consideration, or using a separate ebBP *Transition* element. For enabling the distinction between resetting a timer when linking to a shared state or not, the optional boolean flag `stTimeoutReset` has been added to *FromLinks* and *ToLinks*. Note that the proposed ebBP schema extension for representing share states does not render existing ebBP models obsolete. Only the ebBP

constraint requiring *fromBusinessStateRef* and *toBusinessStateRef* attributes to exclusively reference BTAs or CAs (cf. [27] sec. 3.8.2) has been dropped. ebBP neither describes the rationale behind that constraint nor defines a semantics that relies on that constraint. In section 5 we define a semantics for shared state based ebBP collaborations that works without that constraint.

4.3 Shared State-based Collaborations

This section informally describes the class of shared state-based ebBP collaborations we are proposing for B2Bi process specification. Basically, a shared state is entered by reaching the *EmptyBTA* before the shared state. The *Fork* of the shared state then links to all BTAs that are permissible for the respective shared state. Each of these BTAs (except *EmptyBTAs*) must be followed by an ebBP *Decision* that evaluates the outcome of the BTA. Predefined ebBP *ConditionGuardValues* and user-defined *DocumentEnvelopes* are used for determining the follow-on shared state of a *Decision*. In case an ebBP *AnyProtocolFailure* is detected, the *Decision* must link back to the *EmptyBTA* within the shared state the BTA to be evaluated was started from. Otherwise, it is linked to the *EmptyBTA* before the (same or another) shared state.
The restrictions chosen are aligned with two goals. First, a large part of real-world processes shall be representable in a straightforward manner. Second, distributed BPEL implementations shall be automatically derivable from the specified collaborations. In a multi-case study [36], Reijers and van der Aalst report the results from an investigation of 16 business processes from six Dutch organizations: *"One of the striking observations was that out of the 16 processes considered none of these processes incorporated concurrent behavior, i.e. parallel processing of single cases. Business processes turned out to be completely sequential structures. Their routing complexity was only determined by choice constructs and iterations."* This finding is further backed by the B2Bi models created for the eBIZ-TCF project (`http://www.moda-ml.net/moda-ml/repository/ebbp/v2008-1/en/`) that also do specify concurrent behavior. Therefore, the set of ebBP models that is considered for translation is a subset of the class of multi-transmission interactions as defined in [3] with the special restriction that only two collaboration partners are allowed. Informally, these restrictions are summed up as follows:

- A choreography is modeled as a single ebBP *BusinessCollaboration*. Hierarchical compositions are not supported.
- Only binary collaborations are supported, i.e., the number of integration partners within the collaboration is limited to two.
- A collaboration starts with an ebBP *Start* that immediately links to the initial shared state of the collaboration.
- ebBP *Decisions* are only allowed directly after BTAs.
- Alternative paths are realized by ebBP *Decisions* and by ebBP *Forks* used for representing shared states.
- Looping is realized by *Decisions* that link back to shared states that have been visited before.
- The only case in which a *Decision* branch does not link to a shared state is when process termination is detected. In this special case a *Decision* links to an EmptyBTA before an ebBP *Success* or *Failure* state.
- A choreography ends when a final state, i.e., an ebBP *Success* or *Failure* state is reached. Multiple *Success* and *Failure* states are allowed per collaboration. As multiple instances of BTAs are not allowed for and as state is synchronized after each BTA (there are only two participants), a choreography immediately ends when a final state is reached.
- At any point in time, there is at most one active BTA (no multiple instances). In order to ensure this, ebBP *Forks* have to set the *type* attribute to *XOR* and ebBP *Joins* must have the *waitForAll* attribute set to *false*.
- An EmptyBTA may only link to one single ebBP *Join* or *Fork* element.

This informal class of ebBP models is formalized in the next section.

5. FORMAL ebBP MODELS

The formalization of the informally defined class of shared state-based ebBP collaboration serves several purposes. Above all, it is used for precisely defining the set of valid models that is accepted for translation into distributed BPEL implementations. Further, it is used as concise basis for describing the translation of shared states modeled using our ebBP schema extension into ebBP compliant shared states. Therefore, the formalization closely reflects the main different ebBP element types for specifying *BusinessCollaborations*. In the following, 'STBC' will be used to abbreviate a *shared state based ebBP BusinessCollaboration*, and STBCs that contain shared states modeled by means of our schema extension will be denoted ESTBC (Extension-type STBC) whereas STBCs that employ ebBP complaint shared states will be denoted WSTBC (Workaround-type STBC).
Finally, as ebBP does not provide a formal semantics, another major motivation for providing a formalization is an unambiguous and comprehensible description of STBC semantics. Section 5.1 introduces the formalization of WSTBCs and section 5.2 presents the semantics. Section 5.3 then discusses the formalization of ESTBCs and an algorithm for translating ESTBCs to WSTBCs.

5.1 WSTBC

DEFINITION 5.1.1 (WSTBC).
A workaround-type shared state-based ebBP BsinessCollaboration *(WSTBC) is an extension of a directed graph defined as a 5-tuple (R,N,G,ϕ,θ) consisting of the following elements:*

- $R = \{r_1, r_2\}$ *is the set of collaboration participant roles where* r_1 *is statically declared to be the* leader *of the collaboration.*
- $N = \{s_0\} \cup FORK \cup JOIN \cup DEC \cup SBTA \cup SEBTA \cup T$ *is a set of nodes where the components of the union are pairwise disjoint.*
  - s_0 *is the initial node.*

  - *T being the non-empty set of terminal nodes.*
  - *FORK is a set of ebBP* Fork *elements with the* type *attribute set to 'XOR'.*
  - *JOIN is a set of ebBP* Join *elements with the* waitForAll *attribute set to 'false'.*
  - *DEC is a set of ebBP* Dec *elements.*
  - *SBTA is a set of ebBP* BusinessTransactionAcitivities.
  - *SEBTA is a set of* EmptyBTAs *as described in section 4.1.*
- $G = (L \times E_L) \cup \{(XPath1, \text{'timeout'})\} \cup \{tt\}$ *is a set of guards defined as either a pair of a language* $l \in L$ *and expression* $exp \in E_l$ *defined in l, the special purpose pair (XPath1, 'timeout') for specifying a shared state's timeout, or the boolean constant true (tt).*
  Out of the admissible ebBP expression languages we support ConditionGuardValue *(CGV) and* DocumentEnvelope *(DE) where* E_{CGV} *is an enumeration of generic ebBP protocol outcomes and* E_{DE} *is the set of ebBP* DocumentEnvelopes *defined for the directly preceding BusinessTransaction.*
- *The function* $\phi : FORK \rightarrow \mathbb{N}_0 \cup \{-1\}$ *that assigns a timeout value to every* Fork *node. '-1' is used for denoting an undefined timeout value.*
- θ *is a transition relation* $\theta : N \times 2^G \times N$ *with the constraint that* θ *is the union of the following components:*
  - $\rightarrow^{Start} \subseteq \{s_0\} \times \{tt\} \times SEBTA$ *and* $\left|\rightarrow^{Start}\right| = 1$.
  - $\rightarrow^{ST1} \subseteq SEBTA \times \{tt\} \times JOIN$
  - $\rightarrow^{ST2} \subseteq JOIN \times \{tt\} \times SEBTA$
  - $\rightarrow^{ST3} \subseteq SEBTA \times \{tt\} \times FORK$
  - $\rightarrow^{Terminal} \subseteq SEBTA \times \{tt\} \times T$
  - $\rightarrow^{Trigger} \subseteq FORK \times \{tt\} \times SBTA$
  - $\rightarrow^{Eval} \subseteq SBTA \times \{tt\} \times DEC$
  - $\rightarrow^{Update} \subseteq SBTA \times \{tt\} \times SEBTA$
  - $\rightarrow^{Route} \subseteq DEC \times 2^G \setminus \{(XPath1, timeout)\} \times SEBTA$
  - $\rightarrow^{Timeout} \subseteq FORK \times \{(XPath1, timeout)\} \times SEBTA$

  Note that all components of θ *except for* $\rightarrow^{Trigger}$ *are partial functions* $N \times 2^G \nrightarrow N$. *Further, a workaround-based shared state (cf. section 4.1) may result in two identical elements* $t_1 = t_2 = (n_1, \{tt\}, n_2)$ *that would have to be added to either* $\rightarrow^{ST1}$ *or* $\rightarrow^{Trigger}$. *In that case,* t_2 *is discarded.* □

Two nodes n_k, n_l are *directly connected* in a WSTBC if there is a triple (n_k, g, n_l) or $(n_l, g, n_k) \in \theta$. For (n_k, g, n_l), n_k *directly precedes* n_l and n_l *directly follows* n_k. Further, two nodes n_k, n_l are *connected* in a WSTBC if there is a triple (n_k, g, n_l) or $(n_l, g, n_k) \in \theta^*$ where θ^* is the reflexive-transitive closure of θ. A sequence of nodes $[n_1, .., n_x]$ is defined such that for any i, $1 \leq i < x$: $\exists (n_i, g, n_{i+1}) \in \theta$.

Moreover, we need several functions defined on the components of a WSTBC (for later use in the translation algorithms):

- $in : N \rightarrow 2^N$ computes the set of input nodes of a particular node n_i in WSTBC such that $in(n_i) = \{x | \exists (x, g, n_i) \in \theta\}$.
- $out : N \rightarrow 2^N$ computes the set of output nodes of a particular node n_i in WSTBC such that $out(n_i) = \{x | \exists (n_i, g, x) \in \theta\}$.
- $requestor : SBTA \rightarrow R$ determines which of the collaboration participant roles takes the ebBP *RequestingRole* of a particular BTA.
- $responder : SBTA \rightarrow R$ determines which of the collaboration participant roles takes the ebBP *RespondingRole* of a particular BTA.

In the definitions so far there is no shared state construct. That is due to the fact that the ebBP elements that make up a shared state are explicitly emulated. This is necessary for being able to provide a concise conversion algorithm from ESTBCs to WSTBCs. The following definition characterizes shared states within WSTBCs.

Definition 5.1.2 (Shared State).
A shared state *(ST) is defined for a WSTBC as 5-tuple* $ST_{WSTBC}(e_b, j, e_i, f, \theta_{ST})$ *such that*

- $e_b \in SEBTA$
- $j \in JOIN$
- $e_i \in SEBTA$
- $f \in FORK$
- $\theta_{ST} = \{(e_b, \{tt\}, j), (j, \{tt\}, e_i), (e_i, \{tt\}, f)\} \subset \theta$
- $\nexists (n_k, g, n_l) \in \theta \setminus \theta_{ST} : n_k = j \vee n_l = j \vee n_l = f$

A WSTBC is said to contain a ST_{WSTBC} if it conforms to the above definition. For dealing with STs, some additional functions are needed. Let SH_{WSTBC} be the set of all $ST_{WSTBC}(e_b, j, e_i, f, \theta_{ST})$ contained in a WSTBC. Then, the following functions are defined:

- *nodeb, nodej, nodei, nodef, trans, nodeset* are functions on $SH_{WSTBC} \times N$ that compute the first, second, third, fourth, fifth or union of the first four components of a given ST_{WSTBC}.
- $parentST : N \rightarrow SH_{WSTBC} \cup \{\perp\}$ computes the ST_{WSTBC} for a given node (the existence of such a function follows from fact 5.1.2).
- *ctrlFlow* computes the *control flow* relation ϑ of a WSTBC such that
  $\vartheta = \theta \setminus \bigcup_{st \in SH_{WSTBC}} trans(st)$.
- We write (n_k, g, ST) if there is a $(n_k, g, n_l) \in \vartheta \wedge n_l \in nodeset(\text{ST})$ and
- (ST, g, n_l) if there's a $(n_k, g, n_l) \in \vartheta \wedge n_k \in nodeset(\text{ST})$.

We now present the ebBP language restrictions that reflect the rationale of ST based modeling. The first restriction says that *EmptyBTAs* that link to a *Fork* or *Join* always are part of a shared state ST.

Restriction 5.1.1 (ST linking). *From definition 5.1.2 it is already clear that, for a particular* $st \in SH_{WSTBC}$, *there are no elements in* ϑ *that link to* $nodej(st)$ *or* $nodef(st)$ *or from* $nodej(st)$. *Moreover:*

- *Iff for any ebta* $\in SEBTA$,
  $\exists(ebta, tt, j) \in \theta : j \in JOIN \Rightarrow \exists st \in SH_{WSTBC} : ebta = nodeb(st)$.
- *Iff for any ebta* $\in SEBTA$,
  $\exists(ebta, tt, f) \in \theta : f \in FORK \Rightarrow \exists st \in SH_{WSTBC} : ebta = nodei(st)$.

The following fact clarifies that *Forks* and *Joins* exclusively are used for modeling STs.

Fact 5.1.1 (No Joins/Forks outside STs).
$\forall n \in JOIN \cup FORK : n \in \bigcup_{st \in SH_{WSTBC}} nodeset(st)$

Proof 5.1.1. *From definition 5.1.1, and in particular the definition of* θ *it is clear that for all nodes e that* directly precede *a node* $n \in JOIN \cup FORK$, *holds:* $e \in SEBTA$. *From restriction 5.1.1, we know that for all* $e \in SEBTA$ *that* directly precede *a node* $n \in JOIN \cup FORK$: $e \in \bigcup_{st \in SH_{WSTBC}} nodeset(st)$. *The fact then follows from the definition of* θ *and the definition of STs (def. 5.1.2).* □

The next fact points out that STs in a WSTBC do not overlap, i.e., a structure as depicted in figure 3 is forbidden.

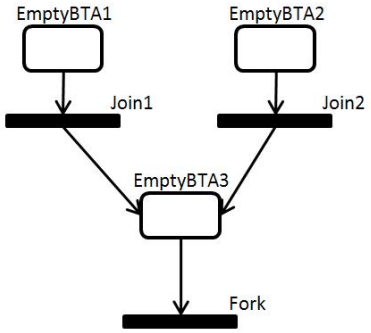

Figure 3: Invalid: Shared State Overlap

Fact 5.1.2 (Disjoint STs).
For any st_1, $st_2 \in SH_{WSTBC}$ *such that* $st_1 \neq st_2$ *holds:* $nodeset(st_1) \cap nodeset(st_2) = \{\}$

Proof 5.1.2. *Assume the opposite for* st_1, $st_2 \in SH_{WSTBC}$. *If* $nodeb(st_1) \neq nodeb(st_2)$ *then* $nodeset(st_1) \cap nodeset(st_2) = \{\}$ *because of the definition of ST (def. 5.1.2) and* $\rightarrow^{ST1}$, $\rightarrow^{ST2}$ *and* $\rightarrow^{ST3}$ *being partial functions.*
Similarly, if $nodeb(st_1) = nodeb(st_2)$ *then* $nodeset(st_1) = nodeset(st_2)$. □

EmptyBTAs are used for solving ebBP schema constraints only. This implies that *EmptyBTAs* shall not contain any business logic and therefore shall always link to exactly one successor node.

Fact 5.1.3 (No Logic in EmptyBTAs).
$\forall e \in SEBTA : \forall (e, \{tt\}, n_k), (e, tt, n_l) \in \theta : n_k = n_l$

Proof 5.1.3. *Directly follows from the definition of* $\rightarrow^{ST1}$, $\rightarrow^{ST3}$ *and* $\rightarrow^{Terminal}$ □

The following language restriction highlights that in case a ST can be left by a timeout then the following *EmptyBTA* shall precede a terminal node or be part of a different ST.

Restriction 5.1.2 (Leaving ST by timeout).
$\forall (f, g, e) \in \theta, f = nodef(st_x), st_x \in SH_{WSTBC}, f \in FORK, e \in SEBTA$ *holds:*
$g = (XPath1, timeout) \wedge (\exists (e, tt, t) \in \rightarrow^{Terminal} \vee (e \in nodeset(st_y), st_y \in SH_{WSTBC} \wedge st_y \neq st_x))$

The next language restriction makes clear that a particular BTA may not be triggered from different STs. If the same ebBP BusinessTransaction were to be admissible in different STs then multiple BTAs of that type would have to be specified. A model that contains a structure like that depicted in figure 4 is invalid.

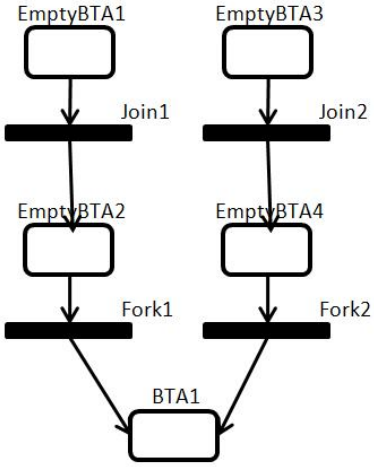

Figure 4: Invalid: BTA triggered from different STs

Restriction 5.1.3 (Unique source ST of a BTA).

$\forall (f_k, tt, b), (f_l, tt, b) \in \rightarrow^{Trigger} : f_k = f_l$

We use the function $btaSrc : \text{BTA} \rightarrow SH_{WSTBC}$ to compute the shared state st with $(nodef(st), tt, b) \in \rightarrow^{Trigger}$.

The result of a BTA determines the next ST of a collaboration. Consistently, the control flow routing decision for determining the next ST shall either be explicitly represented in the ebBP definition or the ST shall not be left. This is ensured by fact 5.1.4 and 5.1.5 as well as restrictions 5.1.4 and 5.1.5 that follow.

Fact 5.1.4 (Unique BTA result processing).
$\forall b \in BTA : \forall (b, tt, n_k), (b, tt, n_l) \in \theta : n_k = n_l$

Proof 5.1.4. *Directly follows from the definition of* $\rightarrow^{Eval}$ *and* $\rightarrow^{Update}$. □

Restriction 5.1.4 (Unprocessed BTA result).
$\forall (b, tt, e) \in \rightarrow^{Update} : e \in nodeset(btaSrc(b))$

In order to be clear which BTA's result a Decision node processes, the following restriction is defined.

Restriction 5.1.5 (Unique BTA reference).
$\forall (b_k, tt, d), (b_l, tt, d) \in \rightarrow^{Eval} : b_k = b_l$

Fact 5.1.5 (Unique source ST of a Dec).
$\forall (f_k, tt, b_m), (f_l, tt, b_n) \in \rightarrow^{Trigger}$ *and* $(b_m, tt, d), (b_n, tt, d) \in \rightarrow^{Eval}$ *holds:* $f_k = f_l$

PROOF 5.1.5. *Directly follows from restrictions 5.1.3, 5.1.5 and fact 5.1.4.* □

We use the function $decSrc : \text{DEC} \rightarrow SH_{WSTBC}$ to compute the shared state st with $(nodef(st), tt, b) \in \rightarrow^{Trigger}$ and $(b, tt, d) \in \rightarrow^{Eval}$.

Apart from explicit representation of routing rules in collaborations, the requirement of state alignment for reaching new STs is important. The next restriction says that in case an error is detected during performing a BTA, a new state may not be reached.

RESTRICTION 5.1.6 (NO ST EXIT WITHOUT ALIGNMENT).
$\forall(d, g, e) \in \rightarrow^{Route}$ *holds:*
$g = (CGV, exp) \wedge exp$ *'indicates a protocol failure'* $\Rightarrow$
$e \in nodeset(decSrc(d))$

Finally, if the result of a BTA has been agreed upon then there is no room for non-determinism to decide about the next ST to reach which is highlighted by fact 5.1.6.

FACT 5.1.6 (UNIQUE BTA RESULT).
For any result of a given BTA b *with* $(b, tt, d), d \in DEC$, *holds:*
$\forall(d, g_k, e_k), (d, g_l, e_l) \in \rightarrow^{Route}: g_k = g_l \Rightarrow e_k = e_l$

PROOF 5.1.6. *Directly follows from the definition of* $\rightarrow^{Route}$
□

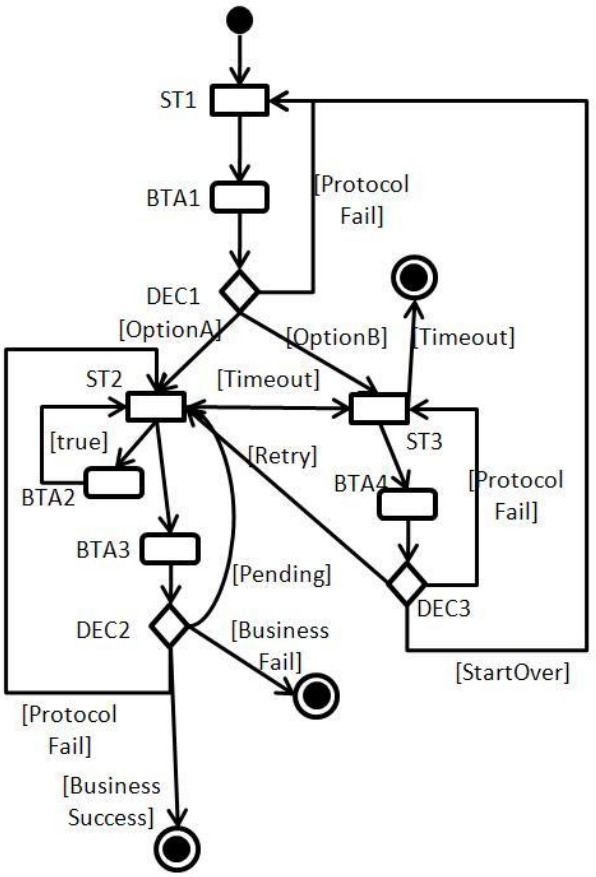

Figure 5: Valid Example WSTBC

Figure 5 visualizes the main control flow options for STBCs when respecting all restrictions. Now that the main syntactical constraints of WSTBCs have been defined, the WSTBC execution semantics can be made precise.

5.2 WSTBC Execution Semantics

ebBP does not define a formal semantics on its own. So, the execution of BTAs and CAs has to be defined. The details of BTA execution are not part of the presented formalization and so the assumption is used that performing a BTA takes some time and eventually either leads to an agreed-upon result or a protocol failure. The implementation of a BTA is described in section 6.1 and the following semantics essentially describes how to iteratively perform this implementation as defined by the links between shared states, BTAs and *Decisions*.

Let $\mathcal{C} \in N \times O \times V_t \times V_{th}$ be the configuration of a WSTBC where N is the set of nodes as in definition 5.1.1, O is the set of all possible outcomes of all BTAs of a WSTBC, V_t represents all possible timer values, and V_{th} represents all possible timer threshold values. As t is defined to be a discrete timer, let the domain of V_t and V_{th} be $\mathbb{N}_0 \cup \{-1\}$. Note that although there are multiple timeouts defined (for different STs), at one point in time, there is at most one timer active. Let χ be the function that computes the outcome of a BTA that has just finished. Further, $\psi : 2^G \times N \times O \times V_t \times V_{th} \rightarrow \{true, false\}$ is the function that evaluates a given set of guards under a given configuration to one of the boolean constants true or false. In particular, $\psi(\{tt\}, \mathcal{C}) = true$ for arbitrary $\mathcal{C}$. Any element $t = (n_k, g, n_l) \in \theta$ is said to be *enabled* for a given $\mathcal{C} = (n_c, o, v_t, v_{th})$ iff $n_c = n_k \wedge \psi(g, \mathcal{C}) = true$. It directly follows that all $t = (n_k, g, n_l) \in \rightarrow^{Start} \cup \rightarrow^{ST1} \cup \rightarrow^{ST3} \cup \rightarrow^{Terminal} \cup \rightarrow^{ST2} \cup \rightarrow^{Trigger} \cup \rightarrow^{Eval} \cup \rightarrow^{Update}$ are always enabled once a configuration $\mathcal{C}$ contains n_k as the first component.

The semantics is defined operationally by the relation $\vdash \subseteq (N \times O \times V_t \times V_{th}) \times (N \times O \times V_t \times V_{th})$. The initial configuration of a WSTBC is $\mathcal{C} = (s_0, \{\}, -1, -1)$, where -1 for the timer and timer threshold values indicates that there is neither a current timer nor an according threshold. All transitions in $t = (n_k, g, n_l) \in \rightarrow^{Start} \cup \rightarrow^{ST1} \cup \rightarrow^{ST3} \cup \rightarrow^{Terminal} \cup \rightarrow^{ST2}$ immediately fire once they are enabled, their processing is assumed to take zero time and the new configuration $\mathcal{C}'$ differs from the preceding $\mathcal{C}$ only in switching from n_k to n_l. This kind of $\vdash$ transitions reflects the fact that $\rightarrow^{Start} \cup \rightarrow^{ST1} \cup \rightarrow^{ST3} \cup \rightarrow^{Terminal} \cup \rightarrow^{ST2}$ have been introduced for creating ebBP conform models only and that there is no logic in empty BTAs. The remaining elements of $\vdash$ can be derived using the set of rules below that represent *triggering BTAs* ($\vdash^{\rightarrow^{Trigger}}$), *finishing BTAs* ($\vdash^{\rightarrow^{Eval}}$,$\vdash^{\rightarrow^{Update}}$), *evaluating BTAs* ($\vdash^{\rightarrow^{Route}}$), *leaving STs by timeout* ($\vdash^{\rightarrow^{Timeout}}$) and *the elapse of time* ($\vdash^{clock}$).

1: Trigger a BTA
$(n, o, v_t, v_{th}) \vdash^{\rightarrow^{Trigger}}$
(n', o, v_t, v_{th}) iff
$(n, g, n') \in \rightarrow^{Trigger} \wedge$
$\psi(g, (n, o, v_t, v_{th})) = true \wedge$
$((v_{th} = -1) \vee (v_t < v_{th}))$

2: Finish a BTA and start result evaluation
$(n, o, v_t, v_{th}) \vdash^{\rightarrow Eval}$
(n', o', v_t, v_{th}) iff
$(n, g, n') \in \rightarrow^{Eval} \wedge$
$\psi(g, (n, o, v_t, v_{th})) = true \wedge$
$o' = \chi n$

3: Finish a BTA and ignore result
$(n, o, v_t, v_{th}) \vdash^{\rightarrow Update}$
(n', o, v'_t, v'_{th}) iff
$(n, g, n') \in \rightarrow^{Update} \wedge$
$\psi(g, (n, o, v_t, v_{th})) = true \wedge$
$((n' = nodeb(btaSrc(n)) \wedge v'_t = 0 \wedge$
$v'_{th} = \phi(nodef(btaSrc(n)))) \vee$
$(n' = nodei(btaSrc(n)) \wedge v'_t = v_t \wedge v'_{th} = v'_{th}))$

4: Evaluate a BTA result
$(n, o, v_t, v_{th}) \vdash^{\rightarrow Route}$
(n', o', v'_t, v'_{th}) iff
$(n, g, n') \in \rightarrow^{Route} \wedge$
$\psi(g, (n, o, v_t, v_{th})) = true \wedge$
$o' = \{\} \wedge$
$((parentST(n') = \perp \wedge v'_t = v_t \wedge v'_{th} = -1) \vee$
$(parentST(n') \neq \perp \wedge n' = nodei(decSrc(n)) \wedge$
$v'_t = v_t \wedge v'_{th} = v'_{th}) \vee$
$(parentST(n') \neq \perp \wedge n' \neq nodei(decSrc(n)) \wedge$
$v'_t = 0 \wedge v'_{th} = \phi(nodef(parentST(n'))))$

5: Leave ST by timeout
$(n, o, v_t, v_{th}) \vdash^{\rightarrow Timeout}$
(n', o, v'_t, v'_{th}) iff
$(n, g, n') \in \rightarrow^{Timeout} \wedge$
$\psi(g, (n, o, v_t, v_{th})) = true \wedge$
$((v_{th} > -1) \wedge (v_t >= v_{th})) \wedge$
$((parentST(n') = \perp \wedge v'_t = v_t \wedge v'_{th} = -1) \vee$
$((parentST(n') \neq \perp) \wedge v'_t = 0 \wedge$
$v'_{th} = \phi(nodef(parentST(n'))))$

6: Elapse of time
$(n, o, v_t, v_{th}) \vdash^{clock}$
(n, o, v'_t, v_{th}) iff
$v'_t = v_t + 1$

Finally, we state that a WSTBC is valid if and only if every node n ∈ N is connected to the initial and one terminal node and there exists a configuration such that the terminal node is reachable by a sequence of transition steps as defined above.

Definition 5.2.1 (Validity).
Let $\vdash^*$ be the transitive closure of $\vdash$. A WSTBC is valid iff

(i) Each $n \in N$ is connected *to both $\{s_0\}$ and at least one $t \in T$.*

(ii) (i) and $\forall n \in N : \exists \mathcal{C} = (n, o, v_t, v_{th}) :$ $(n, o, v_t, v_{th}) \vdash^* (n', o', v'_t, v'_{th}) \wedge n' \in T$

5.3 ESTBC

The core idea of the extension-type STBC (ESTBC) model is replacing the 4 nodes that represent a shared state in the WSTBC model by a single special-purpose node. This is a change at the syntactical level and instead of rephrasing large parts of the WSTBC definitions, we present the core differences between ESTBC and WSTBC models. Language restrictions, facts and semantics then hold analogously.
The node set N of an ESTBC is defined as the union $\{s_0\} \cup$ ST $\cup$ DEC $\cup$ SBTA $\cup$ T where ST denotes a set of shared states as introduced in section 4.1. Compared to the definition of WSTBC, the node sets FORK, JOIN and SEBTA are missing as these are needed only for representing a shared state in an ebBP compliant way. Conflating the WSTBC-based shared state components to a single element also influences some other elements. In particular, ebBP's *toBusinessStateRef/fromBusinessStateRef* constraint(cf. [27] sec. 3.8.2) has been dropped in order to allow for linking to shared states. Consistently, empty BTAs are not needed for linking to final states (T) any more. Instead, terminal nodes are reached in ESTBCs directly from STs or *Decision* nodes. Also, as the reset of timeout values cannot be deduced any more from whether the first or third node of a shared state component is the target of a link, this information is explicitly encoded into the links. Some elements therefore have an additional flag f ∈ F = $\{tt, ff\}$ that indicates whether the final state targeted at shall reset its timer or not. Table 1 compares WSTBC and ESTBC transition relations where the left column names WSTBC relations and the right column describes the corresponding ESTBC definition The relation names have not been changed for emphasizing the semantic similarity and 'g^{to}' is used as abbreviation for $\{(\text{XPath1}, timeout)\}$. Finally, the ϕ function that maps every shared state to its corresponding timeout value is now defined on ST $\rightarrow \mathbb{N}_0 \cup \{-1\}$ instead of FORK $\rightarrow \mathbb{N}_0 \cup \{-1\}$.

WSTBC	ESTBC
$\rightarrow^{Start}$	$\rightarrow^{Start} \subseteq s_0 \times \{tt\} \times$ ST and $\left\vert\rightarrow^{Start}\right\vert = 1$
$\rightarrow^{ST1}$	No correspondence
$\rightarrow^{ST3}$	No correspondence
$\rightarrow^{Terminal}$	via $\rightarrow^{Route'}$ and $\rightarrow^{Timeout'}$
$\rightarrow^{ST2}$	No correspondence
$\rightarrow^{Trigger}$	$\rightarrow^{Trigger} \subseteq$ ST $\times \{tt\} \times$ SBTA
$\rightarrow^{Eval}$	Identical
$\rightarrow^{Update}$	$\rightarrow^{Update} \subseteq$ SBTA $\times \{tt\} \times$ F $\times$ ST
$\rightarrow^{Route}$	$\rightarrow^{Route} \subseteq$ DEC $\times 2^G \setminus g^{to} \times$ F $\times$ ST
$\rightarrow^{Route}$	$\rightarrow^{Route'} \subseteq$ DEC $\times 2^G \setminus g^{to} \times$ T
$\rightarrow^{Timeout}$	$\rightarrow^{Timeout} \subseteq$ ST $\times g^{to} \times$ ST
$\rightarrow^{Timeout}$	$\rightarrow^{Timeout'} \subseteq$ ST $\times g^{to} \times$ T

Table 1: WSTBC/ESTBC Transition Relations

The translation of ESTBCs into WSTBCs is presented using a pseudo-algorithm (algorithm objects 1 and 2). The basic idea of the algorithm is first translating the input ESTBC's shared states and terminal nodes and associating these with the *EmptyBTAs* that have been generated during translation. During translation of the transition relation elements, references to the input ESTBC's shared states and terminal nodes then are mapped accordingly.

input : A valid ESTBC *ebc* to be transformed
output : A valid WSTBC *wbc*
`// map data structures for already mapped nodes`
variables : $stMap$<ST,SEBTA>; $tMap$<T,SEBTA>
algorithm:

```
// copy components that remain unchanged
1  wbc.R = ebc.R;
2  wbc.s0 = ebc.s0;
3  wbc.DEC = ebc.DEC;
4  wbc.SBTA = ebc.SBTA;
5  wbc.T = ebc.T;
6  wbc.G = ebc.G;
7  wbc.→^Eval = ebc.→^Eval;
// Create empty BTAs before terminal nodes
8  foreach t in ebc.T do
9  |  et = createEmptyNode();
10 |  tMap.add(t,et);
11 |  wbc.→^Terminal.add((et,{tt},t));
12 end
// Translate shared states
13 foreach st in ebc.ST do
14 |  eb = createEmptyNode();
15 |  ei = createEmptyNode();
16 |  j = createJoinNode();
17 |  f = createForkNode();
18 |  f.th = st.th;
19 |  wbc.SEBTA.add(eb);
20 |  wbc.SEBTA.add(ei);
21 |  wbc.JOIN.add(j);
22 |  wbc.FORK.add(f);
23 |  wbc.→^ST1.add((eb,{tt},j));
24 |  wbc.→^ST2.add((j,{tt},ei));
25 |  wbc.→^ST3.add((ei,{tt},f));
26 |  stMap.add(st,eb);
27 end
// Translate Transitions
28 foreach (n1,g,n2) in ebc.→^Start do
29 |  eb = stMap.get(n2);
30 |  wbc.→^Start.add((n1,g,eb));
31 end
// continue...
```

Algorithm 1: ESTBC to WSTBC Conversion: part 1

```
// ...continue
1  foreach (n1,g,n2) in ebc.→^Trigger do
2  |  f = nodef(parentST(stMap.get(n1)));
3  |  wbc.→^Trigger.add((f,g,n2));
4  end
5  foreach (n1,g,r,n2) in ebc.→^Update do
6  |  if r == true then
7  |  |  e = nodeb(parentST(stMap.get(n2)));
8  |  else
9  |  |  e = nodei(parentST(stMap.get(n2)));
10 |  end
11 |  wbc.→^Update.add((n1,g,e));
12 end
13 foreach (n1,g,r,n2) in ebc.→^Route do
14 |  if r == true then
15 |  |  e = nodeb(parentST(stMap.get(n2)));
16 |  else
17 |  |  e = nodei(parentST(stMap.get(n2)));
18 |  end
19 |  wbc.→^Route.add((n1,g,e));
20 end
21 foreach (n1,g,n2) in ebc.→^Route' do
22 |  e = tMap.get(n2);
23 |  wbc.→^Route.add((n1,g,e));
24 end
25 foreach (n1,g,n2) in ebc.→^Timeout do
26 |  f = nodef(parentST(stMap.get(n1)));
27 |  e = nodeb(parentST(stMap.get(n2)));
28 |  wbc.→^Timeout.add((f,g,e));
29 end
30 foreach (n1,g,n2) in ebc.→^Timeout' do
31 |  f = nodef(parentST(stMap.get(n1)));
32 |  e = tMap.get(n2);
33 |  wbc.→^Timeout.add((f,g,e));
34 end
35 return wbc;
```

Algorithm 2: ESTBC to WSTBC Conversion: part 2

6. INTEGRATION ARCHITECTURE

This section describes the proposed integration architecture for realizing B2Bi because it provides a proper basis for the derivation of BPEL orchestration models from ebBP choreographies. The application of one BPEL process per integration partner, as opposed to applying a single central BPEL process, is proposed because central IT infrastructure is assumed not to be available by integration partners or simply not intended. According to [44] this solution (apparently) scales better than using one single BPEL process and therefore seems to support a broader range of B2Bi scenarios. Further, B2Bi projects usually have to consider the investments of integration partners in existing IT infrastructure and therefore have to address the problem of interfacing with existing systems. If a B2Bi project simply automates an existing process then there is a high probability that integration partners already have systems in place for evaluating business documents, taking business decisions and capturing real-world events such as "a new order has to be placed".
Therefore, the application of so-called *control processes* that separate the message flow of a collaboration from the actual business logic is proposed. It is the control processes' task to ensure that the message flow at runtime conforms to the choreography defined. The actual business logic is encapsulated in so-called backend services that wrap existing systems. This separation of concerns is also advantageous in terms of software lifecycle management because the inte-

gration partners' processes can be generated such that they do not have to be adapted after generation. This approach is also applicable for an integration partner that does not yet have systems implementing business logic. Note, that this work focuses on the message flow among the control processes and backend services while, clearly, there's much more to a B2Bi project, e.g., data mappings and adaptations of business functions.

6.1 Message Flow

The core task in describing the message flow between control processes and backend services is the mapping of BTAs. The flow of *B*usinessCollaborations can then be derived by repeating the message flow of the respective BTAs according to the ebBP choreography.

Figure 6 uses a UML sequence diagram to show an idealized flow of a BTA that exchanges two documents and employs both ebBP *R*eceiptAcknowledgements (RA) and *A*cceptanceAcknowledgements (AA) as accompanying business signals. BTAs that only exchange one business document or do not employ business signals can be mapped analogously. Figure 6 distinguishes between the *requesting*

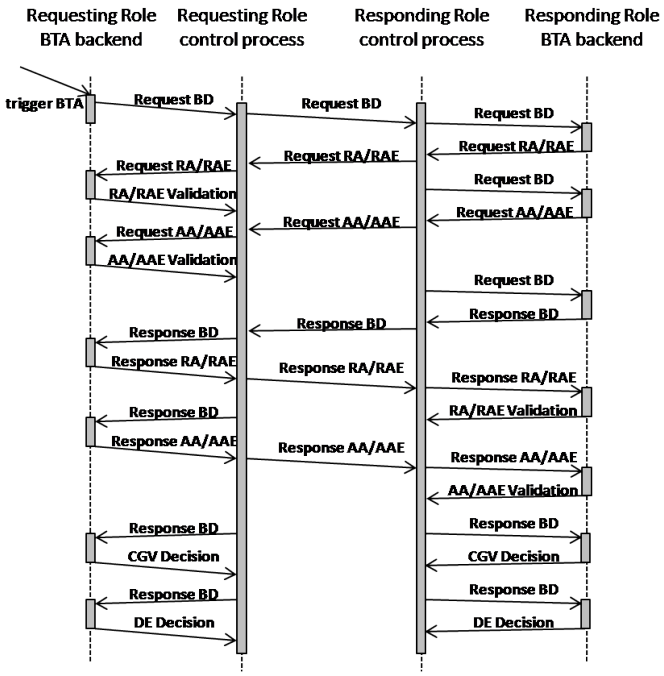

Figure 6: Idealized message flow of a BTA

role for the integration partner who sends the first business document of the BTA and the *responding role* for the sender of the reply business document. A WSTBC's integration partner roles can be mapped to these stereotypic BTA roles by means of the *requestor* and *responder* functions defined in section 5 The message flow of the BTA starts out with the backend of the requesting role capturing the real world event that a new BTA has to be performed and thus sends the request *B*usinessDocument (BD) to the requesting role's control process. The latter then passes the Request BD on to the responding party's control process that subsequently sends the BD to the responding party's backend services for obtaining a RA and an AA or the corresponding exceptions (RAE and AAE). These business signals are then sent back to the requesting role's control process for indicating that the request BD is readable and has been accepted for business processing (cf. [27]). The same procedure is afterwards performed for the response BD using exchanged roles.

After having exchanged all business documents and business signals, both control processes call their backend services for evaluating the outcome of the BTA according to the messages exchanged. Clearly, each integration partner has to apply the same evaluation rules agreed upon in the ebBP choreography. Therefore, the work at hand employs ebBP *C*onditionGuardValues (CGV) and ebBP *D*ocumentEnvelopes (DE), though for many business collaborations more sophisticated means will be necessary. A possible solution may be the definition of Schematron[3] files and thus it is assumed that such an agreement can be made.

Apart from deciding which shared state to switch to after a BTA, state changes have to be performed. We propose that such state changes are not performed until the end of a BTA. In order to perform state changes that are consistent among integration partners, distributed agreement has to be achieved. The realization of a BTA makes a step towards distributed agreement by applying business signals for excluding some error cases, but some business scenarios may require true distributed commitment. Though this is not yet implemented there are solutions to this problem available. One solution is to simply map the well-known Two-Phase-Commit protocol (2PC) to Web Services where the subject of agreement would be that all BTA messages have been exchanged (see [40] for details). Alternative solutions could be based on standards such as WS-ReliableMessaging v1.2[4] or Web Services Transaction v1.2[5].

6.2 BPEL and WSDL Artifacts

As pointed out above, this work proposes the generation of BPEL processes for implementing control processes and WSDL interfaces for encapsulating business logic. Especially the WSDL files for backend services may contain sensitive information, e.g., endpoint references, that should be hidden from the integration partner. Therefore, the structure of BPEL and WSDL files as depicted in figure 7 is proposed. Horizontal gray bars represent WSDL file types, the black squares in such a gray bar represent multiple copies of the same WSDL file. The vertical bars without filling show WSDL-files grouped together in sub packages, either for the purpose of providing a backend interface or a control process.

An ebBP business collaboration results in one BPEL process (RoleX.bpel) per participating party and several WSDL interfaces. common_msg_state.wsdl contains the definition of the collaboration's shared states and defines a WSDL `message` for communicating these. stateReceiverX.wsdl imports common_msg_state.wsdl and moreover defines the WSDL portType as well as the service definition and partnerLinkType of the Web Service (one per participating party) used for notifying the backend about the current process state. In figure 7, the two grey bars denoted stateReceiverX.wsdl represent the same WSDL file except for the port addresses that are used for signaling process states which are partner specific.

Further, for each BPEL process, RoleX.wsdl and RoleX_back

[3] `http://www.iso.org/PubliclyAvailableStandards`
[4] `http://www.oasis-open.org/committees/ws-rx/`
[5] `http://www.oasis-open.org/committees/ws-tx`

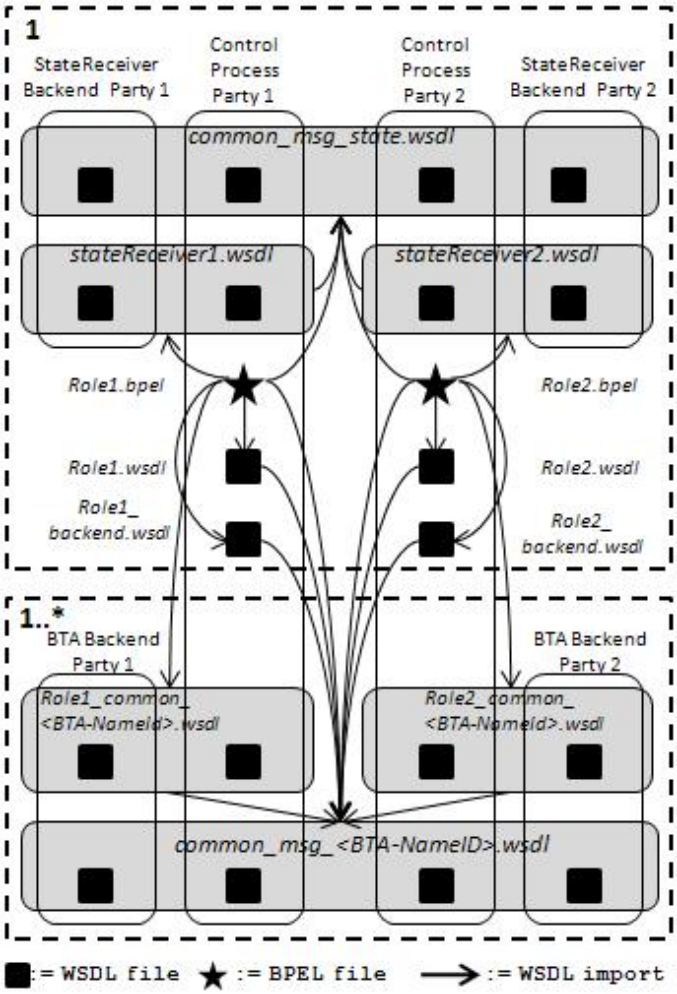

Figure 7: WSDL import relations

end.wsdl are defined. RoleX.wsdl contains all portTypes, bindings and service definitions required for inter-process communication, while RoleX_backend.wsdl contains components for communication that is triggered by the backend system. Furthermore, these WSDL-files contain all related partnerLinkTypes, bindings, service definitions and variable properties. Both import the common message WSDL file (common_msg_<BTA-NameID>.wsdl) generated from every BTA in the business collaboration to be implemented. If a participating party never is the initiator of a BTA throughout a complete collaboration, the RoleX_backend.wsdl only contains the WSDL `definitions` tag without further content or document imports.

A BTA results in three different WSDL files. Two RoleX_common_<BTA-NameID>.wsdl files that contain portType, binding, service definition, partnerLinkType and variable properties and import the aforementioned common_msg_<BTA-NameID>.wsdl containing common Types and Messages. Exactly the same common_msg_<BTA-NameID>.wsdl file is distributed over the entire process to ensure seamless message routing while hiding system internal knowledge like endpoint references (in RoleX_common_<BTA-NameID>.wsdl) from the business partners. Together, RoleX_common_<BTA-NameID>.wsdl and common_msg_<BTA-NameID>.wsdl form the interface for a role specific backend Web Service (per BTA), indicated by vertical bars without filling.

Altogether, each BPEL process imports common_msg_state.wsdl as well as the party specific WSDL files stateReceiverX.wsdl, RoleX.wsdl and RoleX_backend.wsdl. Moreover, the party specific WSDL interface generated for each BTA (RoleX_common_<BTA-NameID>.wsdl, common_msg_<BTA-NameID>.wsdl) is imported.

7. ebBP TO BPEL TRANSLATION

A WSTBC is translated into two orchestrated WS-BPEL processes using a two-stage procedure. Firstly, the control flow of a WSTBC is translated, whereas a placeholder for each *BTA* with its subsequent *Decision* is inserted. Then, each of these placeholders is replaced with the respective WS-PBEL code. The overall procedure is depicted in algorithm 3 and described in more detail in the following paragraphs. Thereby, each *write** function takes the reference to the file containing the WS-BPEL process of a role r∈R as first parameter. All other parameters are function specific. Note that each of the *write** functions appends its WS-BPEL code to the end of the WS-BPEL process definition file specified in the first parameter.

The translation algorithm begins with the insertion of the WS-BPEL code required before being able to insert the WS-BPEL translation of the actual control flow. This code includes the import of required WSDL interfaces (c.f. subsection 6.2), the specification of `partnerLinks`, the declaration of global `variables` such as the ones containing the process state, as well as the specification of a business document independent `correlationSet`. Further, *collaboration timeout*s are handled and the so-called *internal process state* is initialized with the *nameID* of the shared state which can be reached from the initial state (*nameID* of $parentST(out(wbc.s_0))$). Finally, the central `while` loop switching over all shared states is prepared. Listing 3 shows a corresponding WS-BPEL snippet.

Listing 3: BPEL output of writeBPELHeader function

```
<process ... name="UseCase">
 <!-- WSDL imports here -->
 <!-- parterLinks here -->
 <!-- variables here -->
 <!-- correlation set here -->
  <scope name="UseCase">
 <eventHandlers>
  <onAlarm>
   <!-- collaboration timeout handling -->
  </onAlarm>
 </eventHandlers>
 <sequence>
  <assign>
   <copy>
    <from>
     <literal>
      <wsdlDoc:stateType>{first state nameID}
       </wsdlDoc:stateType>
     </literal>
    </from>
    <to>$processState_internal</to>
   </copy>
  </assign>
  <while>
   <condition>'true'</condition>
   <sequence>
```

After initializing the two WS-BPEL processes, the shared state which can be reached from the initial state is trans-

input : A valid WSTBC *wbc* to be transformed
output : Two orchestrated BPEL processes $BPELprocesses<wbc.r_1$.bpel, $wbc.r_2$.bpel>
algorithm:

1 $BPELprocesses<$`writeBPELHeader(`$wbc.r_1.bpel$, `parentST( out(`$wbc.s_0$`)))`, `writeBPELHeader(`$wbc.r_2.bpel$, `parentST( out(`$wbc.s_0$`)))`$>$;
2 $BPELprocesses<$`writeBPELStateProlog(`$wbc.r_1.bpel$, `parentST(out(`$wbc.s_0$`))`, tt`)`, `writeBPELStateProlog(`$wbc.r_2.bpel$, `parentST( out(`$wbc.s_0$`))`, tt`)`$>$;
3 **foreach** *link* **in** `out(nodef( parentST( out(`$wbc.s_0$`))))` **do**
4 | $BPELprocesses<$`writeBTA+DECplaceholder(`$wbc.r_1.bpel$, $link$, `out(`$link$`))`, `writeBTA+DECplaceholder(`$wbc.r_2.bpel$, $link$, `out(`$link$`))`$>$;
5 **end**
6 $BPELprocesses<$`writeBPELStateEpilog(`$wbc.r_1.bpel$`)`, `writeBPELStateEpilog(`$wbc.r_2.bpel$`)`$>$;
7 **foreach** st **in** $wbc.ST \mid st \neq$ `parentST( out(`s_0`))` **do**
8 | $BPELprocesses<$`writeBPELStateProlog(`$wbc.r_1.bpel$, st, ff`)`, `writeBPELStateProlog(`$wbc.r_2.bpel$, st, ff`)`$>$;
9 | **foreach** *link* **in** `out(nodef(`st`))` **do**
10 | | $BPELprocesses<$`writeBTA+DECplaceholder(`$wbc.r_1.bpel$, $link$, `out(`$link$`))`, `writeBTA+DECplaceholder(`$wbc.r_2.bpel$, $link$, `out(`$link$`))`$>$;
11 | **end**
12 | $BPELprocesses<$`writeBPELStateEpilog(`$wbc.r_1$`)`.bpel, `writeBPELStateEpilog(`$wbc.r_2.bpel$`)`$>$;
13 **end**
14 **foreach** t **in** $wbc.T \mid \exists(n_k, tt, t) \in ctrlFlow(wbc) \wedge \exists(n_l, g, n_k) \in ctrlFlow(wbc)$ **do**
15 | $BPELprocesses<$`writeBPELTerminalCode(`$wbc.r_1.bpel$, t`)`, `writeBPELTerminalCode(`$wbc.r_2.bpel$, t`)`$>$;
16 **end**
17 $BPELprocesses<$`writeBPELProcessEpilog(`$wbc.r_1.bpel$`)`, `writeBPELProcessEpilog(`$wbc.r_2.bpel$`)`$>$;
18 $BPELprocesses<$`replaceBTA+DECplaceholders(`$wbc.r_1.bpel$`)`, `replaceBTA+DECplaceholders(`$wbc.r_2.bpel$`)`$>$;
19 **return** $BPELprocesses<wbc.r_1$.bpel, $wbc.r_2$.bpel>;

Algorithm 3: WSTBC to WS-BPEL translation algorithm

lated first before translating all remaining ones. The only difference between the translation of the first and the remaining shared states is that the execution of one of the BTAs admissible from the first shared state creates a new process instance (`createInstance="yes"`). The translation of a shared state is carried out in three steps: Firstly, a so-called state prolog providing the functionality for state timeout handling is added (listing 4). Secondly, the placeholders for all *BTA*s with attached *Decision*s (cf. listing 5) admissible from the respective shared state are inserted. Finally, a so-called state epilog (c.f. listing 6) closes all remaining open tags related to the shared state WS-BPEL code.

Listing 4: BPEL output of writeBPELStateProlog function

```
<if>
 <condition>$processState_internal = '{state
     nameID}'</condition>
 <scope name="{state nameID}_Timeout_Scope">
  <faultHandlers>
   <catch faultName="StateTimeout">
    <empty />
   </catch>
  </faultHandlers>
  <scope name="{state nameID}_Scope">
   <eventHandlers>
    <onAlarm>
     <for>'P6D'</for>
     <scope>
      <sequence>
       <assign>
        <copy>
         <from>
          <literal>
           <wsdlDoc:stateType>{nameID of state
                reached in case of state
               timeout}</wsdlDoc:stateType>
          </literal>
         </from>
         <to>$processState_internal</to>
        </copy>
       </assign>
       <throw faultName="StateTimeout" />
      </sequence>
     </scope>
    </onAlarm>
   </eventHandlers>
   <sequence>
    <assign>
     <copy>
      <from>
       <literal>
        <wsdlDoc:stateType>{state inner entry
            nameID}</wsdlDoc:stateType>
       </literal>
      </from>
      <to>$processState_internal</to>
     </copy>
    </assign>
    <while>
     <condition>$processState_internal =
      '{state inner entry nameID}'
      </condition>
     <sequence>
      <assign>
       <copy>
        <from>
         <literal>
          <wsdlDoc:stateType>{state nameID}
           </wsdlDoc:stateType>
         </literal>
        </from>
        <to>$processState</to>
       </copy>
      </assign>
      <invoke operation="dropProcessState" ..
```

```
            inputVariable="processState" />
58        <!-- createInstance="yes", if second
              parameter of function is true,
              createInstance="no" otherwise -->
59        <pick createInstance="yes">
```

In order to implement the shared state timeout behaviour described in section 5 in a WS-BPEL standard compliant manner, two distinct variables for the process state are necessary. The `processState_internal` variable is used for internal purposes only and governs the execution of the WS-BPEL processes according to the ebBP process definition. The second one, `processState`, is used to communicate the state of a collaboration instance to the collaboration partners. Note that the state is assigned to `processState` only after a WS-BPEL process already is in the respective state. Besides the *nameID*s of all shared states of a business collaboration which constitute all possible values of `processState`, `processState_internal` also includes the *nameID*s of all timer-preserving *nodei*(*st*) of all shared states *st* as possible values. It is important to know that in WS-BPEL, a `scope` in which a `fault` occurred is considered to have completed unsuccessfully [25]. Throwing a `fault` terminates all `scope`s this `fault` is thrown in or passed through until it is handled in some `scope`. Hence, if a state timeout is reached, the *nameID* of the state specified in the respective timeout *linkTo* is assigned to `processState_internal` and a fault terminating all `scopes` not handling it is thrown. The outmost scope of the WS-BPEL code for a shared state handles the fault and subsequently also terminates. Thus, the process switches to the shared state specified in the timeout *linkTo*. If a *Decision* attached to an admissible *BTA* of a shared state links to *nodei*(*st*) of that shared state *st*, the process remains in that state without resetting the timer due to the `while`-loop. In contrary, if that *Decision* links to *nodej*(*st*) of the shared state *st*, the next iteration of the `while`-loop switching over all shared states is triggered, and the timer of the shared state is reset consequently.

Listing 5: BPEL output of writeBTA+DECplaceholder function

```
1 <empty name="{BTA nameID}###{DECISION NameID}"
      />
```

Once the state prolog for a state is appended, a placeholder for each *BTA* with attached *Decision* admissible from the respective shared state is added. These placeholders are replaced by WS-BPEL code later in the translation procedure. The state epilog finally closes all open WS-BPEL tags corresponding to the translation of a shared state.

Listing 6: BPEL output of writeBPELStateEpilog function

```
1       </pick>
2      </sequence>
3     </while>
4    </sequence>
5   </scope>
6  </scope>
7 </if>
```

After translating all shared states, all terminal states need to be translated as well. Thereby, the WS-BPEL code depicted in listing 7 is inserted for each terminal state referenced by a *Decision* or a timeout *linkTo* of a shared state. As in every shared state, the process state is pushed to the backend system using the `invoke` construct.

Listing 7: BPEL output of writeBPELTerminalCode function

```
1 <if>
2  <condition>$processState_internal = '{
       terminal state nameID}'</condition>
3  <sequence>
4   <assign>
5    <copy>
6     <from>
7      <literal>
8       <wsdlDoc:stateType>{terminal state
            nameID}<wsdlDoc:stateType>
9      </literal>
10    </from>
11    <to>$processState</to>
12   </copy>
13  </assign>
14  <invoke operation="dropProcessState" ..
       inputVariable="processState" />
15  <exit/>
16 </sequence>
17 </if>
```

In the next step, all open WS-BPEL tags of the entire process (c.f. listing 8) need to be closed.

Listing 8: BPEL output of writeBPELProcessEpilog function

```
1     </sequence>
2    </while>
3   </sequence>
4  </scope>
5 </process>
```

Before ending the translation procedure, all placeholders for *BTA*s and attached *Decision*s need to be replaced by the respective WS-BPEL code. As the work at hand focuses on the introduction of shared states in modeling ebBP business collaborations, the translation of *BTA*s and *Decision*s is depicted more briefly in the rest of this section.

Role	BPEL Process Elements
Initiator + Responder	- enclosing `onMessage`, receiving a triggering message from integration partner or backend system - enclosing `scope` for complete BTA - all `variables` required for the ebBP *R*equesting-/RespondingBA and the ebBP *D*ecision - `catch` blocks for all ebBP failure types containing the corresponding reaction as specified in the ebBP *B*usinessCollaboration - `catchAll` block containing reaction as specified in ebBP *B*usinessCollaboration for *A*nyProtocolFailure - `onAlarm` to implement the *T*imeToPerform parameter specified for the BTA - `sequence` containing the BPEL code for the ebBP constructs (in this order): *R*equestingBA, *R*espondingBA, *D*ecision. For the respective production rules see tables 3/4 and 5.

Table 2: BPEL production rules for ebBP BusinessTransactionActivity

Tables 3/4 and 5 give an overview of the most important WS-BPEL elements used to translate a BTA with an at-

tached Decision and indicate the purpose of their particular usage. The elements are listed in the order of their occurrence in the WS-BPEL process. The table content corresponds to a BTA that contains a *ReceiptAcknowledgement* as well as an *AcceptanceAcknowledgement* signal. Further, all timing parameters offered for a BTA by the ebBP specification [27] are set. If only some or none of the signals for and parameters of a BTA are specified, the BPEL translation is a corresponding subset of the depicted one. As the mapping of a *RespondingBusinessActivity* is analogous to a *RequestingBusinessActivity*, only a *RequestingBusinessActivity* is described here. An occurrence of an `onAlarm` based timeout in combination with throwing a `fault` terminates the BTA and is handled by the *faultHandlers* of the `scope` enclosing the BPEL code of a BTA. *ReceiptAcknowledgement/-Exception* (RA/RAE) and *AcceptanceAcknowledgement/-Exception* (AA /AAE) are processed concurrently as suggested by the ebBP specification ([27], sec. 3.4.9.3.3). A process tries to get

Role	BPEL Process Elements
Initiator + Responder	- enclosing scope - enclosing flow for concurrent processing of *R*A/RAE and *A*A/AAE
RA / RAE	
Initiator	- enclosing while for trying to get a valid *R*A/RAE until ebBP *r*etryCount is exceeded - scope to encapsulate *R*A/RAE handling - catch block for *R*AE handling - invoke to check *R*AE validity using the backend system - throw to throw ebBP *A*nyProtocolFailure in case no valid *R*AE was received and ebBP *r*etryCount is exceeded - throw to throw ebBP *R*equestReceiptFailure in case of a valid *R*AE - catchAll block for technical failure (TF) handling - rethrow to forward TF to enclosing scope if ebBP *r*etryCount is exceeded - onAlarm to implement ebBP *t*imeToAcknowledgeReceipt - throw ebBP *A*nyProtocolFailure if ebBP *r*etryCount is exceeded - invoke to forward Business Document (BD) to and get a *R*A/RAE from Responder - invoke to check *R*A validity using the backend system
Responder	- enclosing scope - catch block for *R*AE handling - reply construct to forward *R*AE to Initiator - throw ebBP *R*equestReceiptFailure in case of a *R*AE - onAlarm to implement ebBP *t*imeToAcknowledgeReceipt - invoke to forward BD to and get a *R*A/RAE from backend system - reply to forward *R*A to Initiator

Table 3: BPEL production rules for ebBP RequestingBusinessActivity (1)

a valid *R*A/RAE by sending the corresponding BD to the process of the integration partner until the specified ebBP

Role	BPEL Process Elements
AA / AAE	
Initiator	- enclosing while to wait for valid *A*A/AAE - scope to encapsulate *A*A/AAE handling - catch block to forward ebBP *R*equestAcceptanceFailure faults to enclosing scopes - catchAll block to handle TF - empty to wait for an *A*A/AAE despite of TFs - onAlarm to implement ebBP *t*imeToAcknowledgeAcceptance - pick to receive either *A*A or *A*AE - invoke to check *A*A/AAE validity using the backend system - throw ebBP *R*equestAcceptanceFailure in case of a valid *A*AE
Responder	- enclosing scope - catch block for handling *A*AE - invoke to forward *A*AE to Initiator - throw ebBP *R*equestAcceptanceFailure in case of an *A*AE - onAlarm to implement ebBP *t*imeToAcknowledgeAcceptance - invoke to get *A*A/AAE from backend system - invoke to forward *A*A to Initiator

Table 4: BPEL production rules for ebBP RequestingBusinessActivity (2)

*r*etryCount is exceeded or a timeout occurs. Further, if both signal types are used, it waits to receive a valid *A*A/AAE until the occurrence of a timeout. At the end of the BTA mapping, an ebBP *D*ecision is realized by using an invoke for querying the backend services for the evaluation of the latest business document exchanged, the processState_internal variable is then set accordingly. As described earlier, this may lead to a switch to another shared state within the next iteration of the use case's while loop.

Role	BPEL Process Elements
Initiator + Responder	- invoke to send BD of *R*espondingBA to backend system in order to get an evaluation - if no *R*espondingBA exists, BD of *R*equestingBA is used - if and assign statements to determine and switch to next process state Note that *C*onditionGuardValues are evaluated before *D*ocumentEnvelopes.

Table 5: BPEL production rules for ebBP Decision

8. EVALUATION

The evaluation of this work concerns three main areas. Most important is the practical feasibility of the ebBP-2-BPEL translation algorithm assuming the integration architecture introduced in section 6. Further, the computational complexity of the ESTBC-WSTBC conversion and ebBP-2-BPEL translation algorithms is analyzed. Finally, the possible reduction of extensional complexity using ESTBCs instead of WSTBCs is discussed.

For evaluating the feasibility of the proposed ST-based ebBP-BPEL translation approach a translator has been written in the Java language; the main API used for that was the Streaming API for XML (StAX[6]). StAX does not require to load the complete XML document to be processed into memory and therefore it is theoretically possible to process arbitrarily large ebBP choreographies. In practice, this is only helpful in case the generated BPEL processes can still be executed on BPEL engines. On the other hand, the application of StAX is more laborious than using DOM based APIs like JAXB[7]. Approximately 14000 method lines of code have been written to implement the translator. Less code may have been needed using other libraries like DOM or other technologies like XSLT. For the approach presented here, the choice of technology for implementing the translator is of minor importance.

As regards the feasibility of translating arbitrary valid WSTBCs, please note that directed graphs without concurrency basically are state machines with multiple types of nodes. A valid WSTBC only contains *Fork* nodes of type 'XOR' and therefore is a directed graph without concurrency. The implementing BPEL processes use a global while loop for switching across a WSTBC's STs using a global variable for determining the current ST. Once a timeout occurs or a BTA has been performed that results in a ST change, this global variable is assigned accordingly and the new ST is reached in the next iteration of the global while loop. Such a BPEL process essentially amounts to the implementation of a state machine. The use case of this work (section 3) can be translated in full and produces fully BPEL compliant process descriptions. Translating the use case takes approximately 45 seconds using a Centrino duo 1,8GHz and 2GB RAM. Note that the produced BPEL processes are immediately ready for deployment as the incorporation of business logic is allowed for by predefined interfaces (cf. section 6.1). The BPEL processes created have been tested using the Apache ODE 1.2 BPEL engine and the Apache Axis2 1.4 Web Service stack. For the backend services described above dummy Web Services have been implemented that emulate business logic by forwarding decisions to the user. Figure 8 shows the *Seller* role deciding whether to accept an order in full (Accepted) or to defer its decision (Pending). Accordingly, figure 9 shows the notification that ST *Contract* has been entered after the user selected the *Accepted* option. The

Figure 8: Seller deciding upon quote request

use case from section 3 could not be performed on the selected platform in full due to an ODE bug in handling whiles in combination with picks which resulted in the situation that a shared state's while element can only be entered once. Thus, though every shared state of the use case could be

[6] http://jcp.org/en/jsr/detail?id=173
[7] http://jcp.org/en/jsr/detail?id=222

Figure 9: Seller entering state *Contract*

reached there were two states that could not be followed-on with a "normal" termination of the process.

In practice, an important question is the validity of the defined assumption that performing a BTA *"either leads to an agreed-upon result or an protocol failure"* defined in section 5. This is problematic because integration partners are assumed to perform BTAs collaboratively using separate BPEL processes. This introduces the problem of communication over unreliable media. Even worse, BTAs may require the realization of several B2Bi-related security features. Clearly, this issue could be attacked at the application level by applying distributed commit protocols and using digital encryption/signature facilities, but leveraging communication infrastructure facilities is more desirable. In the Web services arena, a set of QoS add-ons has emerged that frequently is denoted as WS-*. In [1] and [5], it has been proved that the combination of WS-ReliableMessaging [32] and WS-Security [28] (and some more related standards) can be used to implement fundamental security and reliability features in a mutual way, i.e., that both integration partners determine the same result of message exchanges. Agreement on BTA results then can be derived from that.

For the selected platform, the use of WS-ReliableMessaging (using Apache Sandesha2[8]) and WS-Security (using Apache Rampart[9]) has been considered for implementing QoS features. Though it was possible to offer BPEL processes as secure and reliable Web Services, invoking other Web Services from BPEL processes in a reliable and secure manner was not possible. So the application of these standards has been canceled. In different work, the application of WS-* to similar use cases has successfully been tested using the available WS-* implementations for GlassFish[10] [39]. In particular it has been shown, that the use of WS-* standards can be integrated into the ebBP-BPEL translation approach. In so far, the assumption of mutually agreed upon BTA results using Web services can be assumed to be realistic.

The main algorithms proposed in this work are the ESTBC-WSTBC (section 5.3) conversion algorithm and the ebBP-BPEL translation algorithm (section 7). As regards the ESTBC-WSTBC algorithm, the computational complexity is fairly low. If a hash-map data structure which provides linear time access can be used and the number of nodes of a ESTBC is n then the complexity of translating a ESTBC into a WSTBC is linear, i.e., $\mathcal{O}(n)$. In practice the complexity of parsing an input XML file and writing an output XML file must be considered as well. The first one can easily be done using standard tools. The second one is trivial as well because the ebBP schema does not impose any restrictions

[8] http://ws.apache.org/sandesha/sandesha2/
[9] http://ws.apache.org/rampart/
[10] https://wsit.dev.java.net/

on the ordering of *Forks*, *Joins*, *BTAs*, *Decisions* and terminal nodes. The ebBP-BPEL translation algorithm also scales well. As *Joins* and *Forks* are used for the representation of shared states only, the identification of shared states takes at most one iteration across a WSTBC's nodes. For creating each resulting BPEL's global while loop (lines 2-13 of algorithm 3) each ST is then visited once. Afterwards, the placeholders for BTAs and the according *Decisions* can be processed in the order they have been written to the BPEL output processes and looking up the matching BTAs and *Decisions* of the input WSTBC takes linear time as well if these have been stored in the first iteration in a hashmap-like data structure. This means that algorithm 3 also has linear time complexity $\mathcal{O}$(n) in terms of the number of nodes n.

Finally, the complexity reduction achievable by using our ebBP schema extension is assessed. At first sight, replacing the ebBP schema compliant representation of a ST with the more compact extension-based model as described in section 4.2 obviously leads to much more compact XML code. Table 6 summarizes the reduction of extensional complexity for this work's use case. The lines of code (LOC) metric refers to the representation of the ebBP *BusinessCollaboration* element and disregards the declaration of DocumentEnvelope definitions as well as BusinessTransaction type definitions that is the same for both types of ST representation. A more valid approach to measure complexity that is not as dependent on formatting is counting the ebBP elements needed for representing nodes and transitions. The first column of table 6 names the complexity metric/ebBP element considered and the other columns contain the measured values for this work's use case as well as the ratio of reduction. Note

Metric	WSTBC	ESTBC	Reduction
LOC	787	591	$\sim$ 0.249
BTA	31	15	$\sim$ 0.516
Fork	7	0	1
Join	7	0	1
Decision	13	13	0
Success	1	1	0
Failure	1	1	0
ST	0	7	undefined
Sum of nodes	60	37	$\sim$ 0.383
ToLink	67	58	$\sim$ 0.134
FromLink	38	15	$\sim$ 0.605
Sum of links	105	73	$\sim$ 0.305

Table 6: WSTBC/ESTBC Complexity Comparison (Use Case)

that the element reduction ratio depends on the collaboration's process structure. Considering the selected elements above, the node reduction ratio can exactly be calculated in terms of node sets via (3*|SH_{WSTBC}| + |T|) / (|SBTA| + |SEBTA| + 2*|SH_{WSTBC}| + |T| + |DEC|). In consequence, the reduction ratio depends on the ratio of |SBTA| to |SH_{WSTBC}|. In the worst case, if there is only one ST component and a very high number of BTAs that all are admissible for that ST and either link back to that ST or to a terminal node then the reduction ratio tends to 0. In the best case, if there is only one BTA for some initialization and a very high number of STs that are reachable via timeouts then the reduction ratio tends to (3*|SH_{WSTBC}|) / (|SEBTA| + 2*|SH_{WSTBC}|).

The reduction ratio of *ToLinks/FromLinks* cannot exactly be calculated in terms of node sets because *Decisions* may have a varying number of branches and a ST may or may not have a timeout configured. Also, for STs without timeout and with only one admissible BTA the *ToLink* to the following BTA must be configured twice to conform with ebBP schema restrictions. Removable links can be calculated as (4*|SH_{WSTBC}| + |T|). For the calculation of the overall number of links assume that every ST has a timeout configured and that each *Decision* has three branches. Due to ebBP's toBusinessStateRef/fromBusinessStateRef restriction every ToLink/FromLink must reference a BTA for WSTBCs. The number of FromLinks then is (|SBTA| + |SEBTA| + |SH_{WSTBC}|) because in every ST the first component is connected twice. The number of ToLinks can be derived by considering the different types of elements that link to BTAs or EmptyBTAs, i.e., (1 + |SH_{WSTBC}| + |JOIN| + |SBTA| + 3*|DEC| + (|BTA|-|DEC|)). The number of ToLinks in *Fork* elements is captured by |SBTA| and (|SBTA|-|DEC|) captures the number of BTAs that directly link back to the source ST. Considering the relation between STs and its components, the link reduction ratio can be calculated as (4*|SH_{WSTBC}| + |T|) / (1 + 3*|SBTA| + 5*|SH_{WSTBC}| + T + 2*|DEC|). The best case/worst case analysis then is similar as for the node reduction ratio.

9. RELATED WORK

In general, this work is about implementing B2Bi using interacting partner processes. In so far, the work of standardization institutions that specify how to perform *B*usinessTransactions or similar concepts at runtime is related to our work. For example, the so-called RosettaNet Implementation Framework [38] specifies rules which define how to perform PIPs. These standards, however, frequently do not allow for modeling complex processes and typically do not offer the generation of executable implementations of control processes as we do with our translator.

Modeling, analyzing and executing complex processes is a core issue of workflow research. Frequently, analysis is performed based on directed graphs or variants thereof. These results are relevant because ebBP *BusinessCollaborations* can be interpreted as directed graphs. Executing arbitrary directed graphs is not easy as unsound models may lead to deadlocks (e.g. an OR-Fork followed by an AND-Join) or may not be executable by process engines that frequently do not accept arbitrary graphs (e.g. BPEL is a block-structured language). One way to attack this problem is imposing restrictions on directed graphs. In [17] so-called *structured workflows* are described that are composed from pair-wise matching control nodes, e.g., an AND-Fork with two branches is matched by an AND-JOIN and control flow does not cross the control flow branches. Similar approaches are also taken by different research projects such as ADEPT [8]. Imposing restrictions on graphs ensures soundness properties like absence of deadlocks and enables straightforward execution by many process engines. If such restrictions are not acceptable, analysis algorithms must be employed to decompose

graphs. [33] and [46] both target at identifying a hierarchy of components within (almost) arbitrary graphs. While [33] identify components by looking for unique entry and exit edges, [46] do so by searching for unique entry and exit nodes. If all elements of a graph can be associated with *structured* components, the mapping to BPEL (which is considered in both publications) is straightforward. Things become more complex when a component's structure cannot be decomposed strictly hierarchically. For mapping such *unstructured* components to BPEL, [33] suggests the use of *event-action rules* that allow for directly representing arbitrary graphs by means of performing calls of a BPEL process instance to itself that then are processed by BPEL event handlers.

The work at hand follows the approach of imposing limitations on the graphs that can be processed, most notably concurrency is not supported. While this may be a significant limitation from a theoretic point of view, it enables us to directly translate a large set of real-world processes (cf. section 4) to our BPEL-based interaction architecture. Further, banning concurrency may have a positive effect on comprehensibility and agreement in the face of B2Bi scenarios with typically different personnel from different integration partners. This research question still has to be investigated. Further, the work at hand is different by not targeting at arbitrary workflows but B2Bi processes that are to be implemented by local partner processes respecting B2Bi QoS requirements like reliability and encryption.

The issue of using partner local processes for realizing collaborative processes also has been investigated in workflow research. Issues like the conformance of local processes to global process descriptions have been analyzed, among others, in [45] from a conceptual point of view. While van der Aalst and Weske [45] use Petri net technology to perform analysis, other researchers like Bultan, Xu and Fu or Benatallah, Casati and Toumani use state machines [7] [4] or collaboration diagrams [6].

In recent years, researchers also adopt more and more the dichotomy of choreography and orchestration to describe and analyze collaborative processes. For example, Zaha et al. [50] propose *Let's Dance* as a language for modeling both, choreographies and orchestrations. These more conceptual approaches differ from our work in not using dedicated B2Bi standards like ebBP and BPEL.

Several more technology driven approaches like [40] and [16] derive BPEL from compositions of BTA-like interactions but do not offer a B2Bi standards based choreography model.

As regards the use of standards for representing choreographies and orchestrations, there are several proposals for mapping WS-CDL choreographies to BPEL, e.g., [22] and [47]. These approaches differ from ours in using WS-CDL instead of ebBP. While WS-CDL may be a good choice for many choreographies due to its tight relationship with WSDL as opposed to ebBP, we claim that ebBP is particularly useful for B2Bi due to its better support for specifying QoS features.

In [10], Decker et al. motivate the use of BPEL4Chor for describing choreographies. In particular, 10 requirements to be fulfilled by choreography languages are postulated. These requirements mainly target at control flow definition concepts, e.g., *multi-lateral interactions*. ebBP is blamed of supporting bi-lateral interactions only. While this is true for single BusinessTransactions, ebBP *BusinessCollaborations* allow for defining more than two integration partners and therefore ebBP supports multi-lateral interactions. ebBP is also criticized for missing integration with orchestration languages. For a restricted class of ebBP choreographies, integration with BPEL is presented in the work at hand. Apart from that, BPEL4Chor and ebBP are not directly comparable. Both languages enable choreography definitions, but while BPEL4Chor assumes service-based interactions ebBP is a dedicated B2Bi standard based on the concept of BusinessTransactions. BPEL4Chor is tightly related to BPEL and thus ties a choreography definition to BPEL-based execution. We consider BPEL to be a major implementation technology for implementing BusinessCollaborations but at the level of BusinessTransactions there should be a choice among different implementation technologies like AS2 [24] or ebMS [26, 29]. In practice, business document exchanges are not always based on BPEL and Web services due to implementation restrictions (existing implementations and integration partner systems) as well as legal restrictions (e.g., some customs authorities require the use of EDI). Tying a B2Bi choreography to BPEL-based execution therefore is not acceptable in many B2Bi scenarios. Moreover, ebBP allows for the definition of B2Bi related QoS attributes. All in all, we consider BPEL4Chor to be a promising proposal for service-based choreographies and service-based B2Bi interactions. For B2Bi interactions in general, ebBP seems to be better suited.

Analyzing compatibility of interacting BPEL processes is related to the work at hand as automatically deriving BPEL processes from common ebBP choreographies is one way of ensuring compatibility. While this is a constructive approach, the problem may be tackled in an analytical way as well, e.g., [20] analyze compatibility by means of defining a BPEL semantics in terms of Petri nets. Related to this are approaches like [48] and [2] that focus on analyzing conformance of orchestration models to choreography models in an analytical way. Note that more general findings from workflow research in general (cf. [45]) apply as well.

There are several contributions that translate dedicated B2Bi or Web service choreography languages to BPEL as we do. Most notably, Huemer and Hofreiter investigated this research area. [18] also propose the translation of ebXML BPSS to BPEL. Apart from being designed for BPSS 1.1, [18] is different from our work in not applying a shared state based modeling approach to ebBP and in not reporting on a fully working translator. In [14], UMM BusinessTransactions which are tightly related to ebBP BusinessTransactions are translated into BPEL and in [15] state machines for performing UMM BusinessTransaction are specified. In both contributions, support for BusinessCollaborations is not available. An interesting approach is presented in [13] that proposes an UML profile for modeling the orchestration of multiple UMM *B*usinessTransactions of one integration partner. Such a model can then be transformed to BPEL processes. This differs from the work at hand in actually transforming a *UML orchestration* (sometimes also denoted as *local choreography*) into a BPEL orchestration. The *UML orchestration*, in turn, is derived from one (or more) UMM choreographies. Thereby, more partner-specific logic is added to the *UML orchestrations*. In contrast to our work, this constitutes an extra step in deriving BPEL processes from choreographies. Interestingly, [13] also captures collaboration *state* for routing the choreography but incor-

porates it in the model by using transition guards. Last, [13] assumes UMM and consequently UML for modeling choreographies whereas we expect the textual format ebBP. We do not consider ebBP to be an alternative to visually modeling business collaborations but rather as a common 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.

10. CONCLUSION AND FUTURE WORK

The main goal of this paper, i.e., showing that *shared state* based ebBP models can be created in a standard compliant manner and subsequently translated into BPEL orchestrations has been achieved by describing a suitable modeling concept and by implementing a prototypic ebBP to BPEL translator. For usability, an ebBP schema extension for shared states as well as a formalization of valid shared state-based ebBP collaborations and their semantics has been developed. *Shared states* support the agreement function of choreography models and allow for the definition of state specific timeouts. They are beneficial for creating choreography as well as orchestration models by offering natural synchronization points and, finally, define the basis for signaling the collaboration's progress to process stakeholders.
Apart from introducing *shared states* into ebBP to BPEL translations, an integration architecture for using the generated BPEL processes has been proposed which does not require BPEL processes to be edited after generation. Comparing the size of the use case to the NES standard processes it can further be stated that even collaborations of real-world size can be processed. Tests using the BPEL engine Apache ODE showed that the generated BPEL processes can be executed to a large extent. Though BPEL engines and Web Service standards addressing QoS features have not yet reached their full potential, tests are promising that the approach proposed may be applicable for real world B2Bis in the future.
Nonetheless, there are limitations to the approach presented. Extended control flow capabilities for multi-party interactions, concurrency and hierarchical decomposition are desirable from a theoretic point of view. Future work therefore comprises combining shared states with these concepts as well as the assessment of what kind of models better suits user needs. Also, modularization of resulting BPEL processes for better maintainability and integration of non Web service communication technologies as described in [43] is needed. The same holds true for supporting QoS features like reliability or security which are indispensable in the B2Bi area. Results from [39] already have shown that this is possible. Furthermore, a process model for applying the approach within B2Bi projects should be defined, in particular when it comes to handling changes in the choreography. As the practical findings of this work encourage the use of ebBP to BPEL translations, more formal analysis and validation features should be applied. In particular, the conformance of BPEL orchestrations to ebBP choreographies and the compatibility between interacting BPEL processes are to be investigated. Existing semantics for BPEL and the ebBP semantics defined in this work provide the formal basis for rigorous analysis.

11. REFERENCES

[1] M. Backes, S. Moedersheim, B. Pfitzmann, and L. Vigano. Symbolic and cryptographic analysis of the secure WS-ReliableMessaging scenario. In *Proceedings of Foundations of Software Science and Computational Structures (FOSSACS)*, volume 3921 of *Lecture Notes in Computer Science*, pages 428–445. Springer, March 2006.

[2] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In M. Bravetti, L. Kloul, and G. Zavattaro, editors, *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 257–271. Springer, 2005.

[3] A. Barros, M. Dumas, and A. H. M. T. Hofstede. Service interaction patterns. In *Proceedings of the 3rd International Conference on Business Process Management (BPM), Nancy, France*, pages 302–318. Springer Verlag, 2005.

[4] B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols. *Data Knowl. Eng.*, 58(3):327–357, 2006.

[5] K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. *ACM Trans. Inf. Syst. Secur.*, 10(2):8, 2007.

[6] T. Bultan and X. Fu. Specification of realizable service conversations using collaboration diagrams. *Service Oriented Computing and Applications*, 2(1):27–39, 2008.

[7] T. Bultan, J. Su, and X. Fu. Analyzing conversations of web services. *IEEE Internet Computing*, 10(1):18–25, 2006.

[8] 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*, 23(2):81–97, 2009.

[9] G. Decker, O. Kopp, F. Leymann, K. Pfitzner, and M. Weske. Modeling service choreographies using BPMN and BPEL4Chor. In *CAiSE '08: Proceedings of the 20th international conference on Advanced Information Systems Engineering*, pages 79–93, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] G. Decker, O. Kopp, F. Leymann, and M. Weske. Interacting services: From specification to execution. *Data & Knowledge Engineering*, 68(10):946 – 972, 2009.

[11] A. Gunasekaran and E. W. T. Ngai. Information systems in supply chain integration and management. *European Journal of Operational Research*, 159(2):269–295, December 2004.

[12] 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)*, pages 507–519, Volume 3292 of Lecture Notes in Computer Science, October 2004. Springer.

[13] B. Hofreiter and C. Huemer. 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'08) and Enterprise Computing, E-Commerce, and E-Services*

(EEE'008), Crystal City, Washington D.C., USA, 7 2008. IEEE.

[14] B. Hofreiter, C. Huemer, P. Liegl, R. Schuster, and M. Zapletal. Deriving executable BPEL from UMM business transactions. In *IEEE SCC*, pages 178–186. IEEE Computer Society, 2007.

[15] C. Huemer and M. Zapletal. A state machine executing UMM business transactions. In *Proceedings of the 2007 Inaugural IEEE International Conference on Digital Ecosystems and Technologies (IEEE DEST 2007)*, Cairns (Australia), 2007. IEEE Computer Society, IEEE Computer Society.

[16] R. Khalaf. From RosettaNet PIPs to BPEL processes: A three level approach for business protocols. *Data & Knowlegde Engineering*, 61(1):23–38, 2007.

[17] B. Kiepuszewski, A. H. M. ter Hofstede, and C. Bussler. On structured workflow modelling. In *CAiSE '00: Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, pages 431–445, London, UK, 2000. Springer-Verlag.

[18] J.-H. Kim and C. Huemer. From an ebXML BPSS choreography to a BPEL-based implementation. *SIGecom Exch.*, 5(2):1–11, 2004.

[19] D. M. Lambert and M. C. Cooper. Issues in supply chain management. *Industrial Marketing Management*, 29(1):65 – 83, 2000.

[20] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing interacting WS-BPEL processes using flexible model generation. *Data Knowl. Eng.*, 64(1):38–54, 2008.

[21] M. Madiesh and G. Wirtz. A top-down method for B2B process design using SOA. In *Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008, July 14-17, 2008, Las Vegas Nevada, USA*, pages 444–450, 2008.

[22] J. Mendling and M. Hafner. From WS-CDL choreography to BPEL process orchestration. *Journal of Enterprise Information Management (JEIM). Special Issue on MIOS Best Papers*, 2006.

[23] J. T. Mentzer, W. DeWitt, J. S. Keebler, S. Min, N. W. Nix, C. D. Smith, and Z. G. Zacharia. Defining supply chain management. *JOURNAL OF BUSINESS LOGISTICS*, 22(2):1–26, 2001.

[24] D. Moberg and R. Drummond. *MIME-Based Secure Peer-to-Peer Business Data Interchange Using HTTP, Applicability Statement 2 (AS2)*. The Internet Engineering Task Force (IETF), July 2005.

[25] OASIS. Web services business process execution language (wsbpel).

[26] OASIS. *ebXML Message Service Specification*. OASIS, 2.0 edition, April 2002.

[27] OASIS. *ebXML Business Process Specification Schema Technical Specification*. OASIS, 2.0.4 edition, December 2006.

[28] OASIS. Web Services Security v1.1, February 2006.

[29] OASIS. *ebXML Messaging Services Version 3.0: Part 1, Core Features*. OASIS, October 2007.

[30] OASIS. Web Services Atomic Transaction (WS-AtomicTransaction) version 1.1, July 2007.

[31] OASIS. *Web Services Business Process Execution Language*, 2.0 edition, April 2007.

[32] OASIS. *Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.2*. OASIS, February 2009.

[33] 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: Proceedings of the IEEE International Conference on Web Services*, pages 285–292, Washington, DC, USA, 2006. IEEE Computer Society.

[34] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[35] C. Pflügler, A. Schönberger, and G. Wirtz. Introducing partner shared states into ebBP to WS-BPEL translations. In *Proc. iiWAS2009, 11th International Conference on Information Integration and Web-based Applications & Services, 14.-16. December 2009, Kuala Lumpur (Malaysia)*. ACM, December 2009.

[36] H. A. Reijers and W. M. van der Aalst. The effectiveness of workflow management systems: Predictions and lessons learned. *International Journal of Information Management*, 25(5):458 – 472, 2005.

[37] F. Rosenberg, C. Enzi, A. Michlmayr, C. Platzer, and S. Dustdar. Integrating quality of service aspects in top-down business process development using WS-CDL and WS-BPEL. In *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, page 15, Washington, DC, USA, 2007. IEEE Computer Society.

[38] RosettaNet, www.rosettanet.org. *RosettaNet Implementation Framework: Core Specification*, v02.00.01 edition, March 2002.

[39] A. Schönberger, T. Benker, S. Fritzemeier, M. Geiger, S. Harrer, T. Kessner, J. Schwalb, and G. Wirtz. QoS-enabled business-to-business integration using ebBP to WS-BPEL translations. In *Proceedings of the IEEE SCC 2009 International Conference on Services Computing, Bangalore, India*. IEEE, September 2009.

[40] A. Schönberger and G. Wirtz. Realising RosettaNet PIP Compositions as Web Service Orchestrations - A Case Study. In *The 2006 International Conference on e-Learning, e-Business, Enterprise Information Systems, e-Government, & Outsourcing (EEE'06)*, June 26-29 2006.

[41] A. Schönberger and G. Wirtz. Using Webservice Choreography and Orchestration Perspectives to Model and Evaluate B2B Interactions. In *The 2006 International Conference on Software Engineering Research and Practice (SERP'06)*, June 26-29 2006.

[42] A. Schönberger and G. Wirtz. Taxonomy on consistency requirements in the business process integration context. In *Proceedings of 2008 Conf. on Software Engineering and Knowledge Engineering (SEKE'2008)*, Redwood City, California, USA, 1.-3. July 2008. Knowledge Systems Institute.

[43] A. Schönberger and G. Wirtz. Using variable communication technologies for realizing business collaborations. In *Proceedings of the 5th 2009 World Congress on Services (SERVICES 2009 PART II), International Workshop on Services Computing for B2B (SC4B2B), Bangalore, India*. IEEE, September

2009.

[44] C. Schroth, T. Janner, and V. Hoyer. Strategies for cross-organizational service composition. In *MCETECH '08: Proceedings of the 2008 International MCETECH Conference on e-Technologies*, pages 93–103, Washington, DC, USA, 2008. IEEE Computer Society.

[45] W. M. P. van der Aalst and M. Weske. The p2p approach to interorganizational workflows. In *CAiSE '01: Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, pages 140–156, London, UK, 2001. Springer-Verlag.

[46] J. Vanhatalo, H. Völzer, and J. Koehler. The refined process structure tree. *Data Knowl. Eng.*, 68(9):793–818, 2009.

[47] I. Weber, J. Haller, and J. A. Mülle. Automated derivation of executable business processes from choreograpies in virtual organizations. In *In: F. Lehner, H. Nösekabel, P. Kleinschmidt (eds.): Multikonferenz Wirtschaftsinformatik 2006 (MKWI 2006), Band 2, XML4BPM Track, GITO-Verlag Berlin*, pages 313–327, Mar. 2006.

[48] W. L. Yeung. A formal basis for cross-checking ebxml bpss choreography and web service orchestration. In *APSCC '08: Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference*, pages 524–529, Washington, DC, USA, 2008. IEEE Computer Society.

[49] J. Zaha, M. Dumas, A. ter Hofstede, A. Barros, and G. Decker. Bridging global and local models of service-oriented systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 38(3):302 –318, may 2008.

[50] J. M. Zaha, A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Let's dance: A language for service behavior modeling. In *Proceedings of the 14th international conference on cooperative information systems (CoopIS'06)*, pages 145–162, Montpellier, France, 10 2006.