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.

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 tools 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 conceptually reached/ left by integration participants for explicitly modeling the state changes realized by 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 commu-
• 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 these 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 Subset2 (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

1http://www.ebxml.org/

2http://www.nesubl.eu/
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 PendingContractChangeBI(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 indicating 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. 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
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 typeDataExchange 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

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

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 \textit{fromBusinessStateRef} and \textit{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 based ebBP collaborations we are proposing for B2Bi 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 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. Its 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 compliant 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 workarounds-type shared state-based ebBP BusinessCollaboration (WSTBC) is an extension of a directed graph defined as a 5-tuple \((R,N,G,\phi,\theta)\) consisting of the following elements:

- \(R = \{r_1,r_2\}\) is the set of collaboration participant roles where \(r_2\) is statically declared to be the leader of the collaboration.
- \(N = \{s_0\} \cup \text{FORK} \cup \text{JOIN} \cup \text{DEC} \cup \text{SBTA} \cup \text{SEBTA} \cup T\) 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 BusinessTransactionActivities.
– SEBTA is a set of EmptyBTAs as described in section 4.1.

\( G = (L \times E_l) \cup \{(XPath_1, ‘timeout’)\} \cup \{tt\} \) 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 \((XPath_1, ‘timeout’)\) for specifying a shared state’s timeout, or the boolean constant true (tt).

Out of the admissible ebBP expression languages we support the ConditionGuardValue (CGV) and DocumentEnvelope (DE) where CGV is an enumeration of generic ebBP protocol outcomes and DE is the set of ebBP DocumentEnvelopes defined for the directly preceding BusinessTransaction.

• The function \( \phi : \text{FORK} \rightarrow \mathbb{N}_0 \cup \{-1\} \) that assigns a timeout value to every Fork node. ‘-1’ is used for denoting an undefined timeout value.

\( \theta \) is a transition relation \( \theta : N \times 2^N \times N \) with the constraint that the union of the following components:

\[ -\text{Start} \subseteq \{s_0\} \times \{tt\} \times SEBTA \quad \text{and} \quad -\text{Start} \mid = 1. \]

\[ -\text{ST}_i \subseteq SEBTA \times \{tt\} \times JOIN \]

\[ -\text{ST}_i \subseteq JOIN \times \{tt\} \times SEBTA \]

\[ -\text{ST}_i \subseteq SEBTA \times \{tt\} \times FORK \]

\[ -\text{Terminal} \subseteq SEBTA \times \{tt\} \times T \]

\[ -\text{Trigger} \subseteq FORK \times \{tt\} \times SEBTA \]

\[ -\text{Eval} \subseteq SEBTA \times \{tt\} \times DEC \]

\[ -\text{Update} \subseteq SEBTA \times \{tt\} \times SEBTA \]

\[ -\text{Route} \subseteq DEC \times 2^N \times \{(XPath_1, timeout)\} \times SEBTA \]

\[ -\text{Timeout} \subseteq FORK \times \{(XPath_1, timeout)\} \times SEBTA \]

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

Two nodes \( n_k, n_i \) are directly connected in a WSTBC if there is a triple \((n_k, g, n_i)\) or \((n_i, g, n_k)\) ∈ \( \theta \). For \((n_k, g, n_i)\), \( n_k \) directly precedes \( n_i \) and \( n_i \) directly follows \( n_k \). Further, two nodes \( n_i, n_k \) are connected in a WSTBC if there is a triple \((n_k, g, n_i)\) or \((n_i, g, n_k)\) ∈ \( \theta \) where \( \theta \) is the reflexive-transitive closure of \( \theta \). A sequence of nodes \( \{n_1, \ldots, n_i\} \) 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 \( in(n_i) = \{x \mid \exists (x, g, n_i) \in \theta \} \).

• \( out : N \rightarrow 2^N \) computes the set of output nodes of a particular node \( n_i \) in WSTBC such that \( out(n_i) = \{x \mid \exists (n_i, g, x) \in \theta \} \).

• \( requestor : \text{SBTA} \rightarrow R \) determines which of the collaboration participant roles takes the ebBP RequestingRole of a particular BTA.

• \( responder : \text{SBTA} \rightarrow R \) 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 \( ST_{\text{WSTBC}}(e_0, j, e_1, f, \theta_{ST}) \) such that:

• \( e_0 \in SEBTA \)

• \( j \in JOIN \)

• \( e_1 \in SEBTA \)

• \( f \in FORK \)

• \( \theta_{ST} = \{(e_0, \{tt\}, j), (j, \{tt\}, e_1), (e_1, \{tt\}, f)\} \subseteq \theta \)

\[ \theta(n_k, g, n_i) \in \theta \wedge n_k = j \vee n_i = f \]

A WSTBC is said to contain a \( ST_{\text{WSTBC}} \) if it conforms to the above definition. For dealing with STs, some additional functions are needed. Let \( SH_{\text{WSTBC}} \) be the set of all \( ST_{\text{WSTBC}}(e_0, j, e_1, f, \theta_{ST}) \) contained in a WSTBC. Then, the following functions are defined:

• \( \text{nodeb, nodej, nodei, nedef, trans, nodeset} \) are functions on \( SH_{\text{WSTBC}} \times N \) that compute the first, second, third, fourth, fifth or union of the first four components of a given \( ST_{\text{WSTBC}} \).

• \( \text{parentST} : N \rightarrow SH_{\text{WSTBC}} \cup \{\bot\} \) computes the \( ST_{\text{WSTBC}} \) for a given node (the existence of such a function follows from fact 5.1.2).

• \( \text{ctrlFlow} \) computes the control flow relation \( \vartheta \) of a WSTBC such that \( \vartheta = \theta \setminus \bigcup_{\text{trans}(st)} \).

We write \((n_k, g, ST)\) if there is a \((n_k, g, n_i) \in \theta \wedge n_i \in \text{nodeset}(ST)\) and

\((ST, g, n_i)\) if there’s a \((n_k, g, n_i) \in \theta \wedge n_k \in \text{nodeset}(ST)\).

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 \( st \in SH_{\text{WSTBC}} \), there are no elements in \( \vartheta \) that link to \( \text{nodej}(st) \) or \( \text{nodef}(st) \) or from \( \text{nodej}(st) \) Moreover.
• If for any ebta ∈ SEBTA,\[\exists(ebta, tt, j) ∈ \theta : j ∈ JOIN \implies \exists st ∈ SHWSTBC : ebta = nodeb(st)\]

• If for any ebta ∈ SEBTA,\[\exists(ebta, tt, f) ∈ \theta : f ∈ FORK \implies \exists st ∈ SHWSTBC : ebta = node1(st)\]

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

FACT 5.1.1 (No Joins/Forks outside STs).\[\forall n ∈ JOIN ∪ FORK : n ∈ \bigcup_{st ∈ SHWSTBC} nodeset(st)\]

**Proof.** 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 ∈ JOIN ∪ FORK, holds: e ∈ SEBTA. From restriction 5.1.1, we know that for all e ∈ SEBTA that directly precede a node n ∈ JOIN ∪ FORK: e ∈ \bigcup_{st ∈ SHWSTBC} 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 st₁, st₂ ∈ SHWSTBC such that st₁ ≠ st₂ holds: nodeset(st₁) ∩ nodeset(st₂) = \{\}

**Proof.** 5.1.2. Assume the opposite for st₁, st₂ ∈ SHWSTBC. If nodeb(st₁) ≠ nodeb(st₂) then nodeset(st₁) ∩ nodeset(st₂) = \{\} because of the definition of ST (def. 5.1.2) and \(\rightarrow_{ST1}, \rightarrow_{ST2}\) and \(\rightarrow_{ST3}\) being partial functions.

Similarly, if nodeb(st₁) = nodeb(st₂) then nodeset(st₁) = nodeset(st₂). □

EmptyBTAs are used for solving eBP 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). ∀e ∈ SEBTA : \(\forall(e, \{tt\}, n_k), (e, tt, n_k) ∈ \theta : n_k = n_l\)

**Proof.** 5.1.3. Directly follows from the definition of \(\rightarrow_{ST1}, \rightarrow_{ST2}\) 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).**\[\forall(f,g) ∈ \theta : f = nodef(st₁), st₂ ∈ SHWSTBC, f ∈ FORK, e ∈ SEBTA holds: g = (XPath₁, timeout) \land (3(e, tt, t) \in \rightarrow_{Terminal} \lor (e ∈ nodeset(st₁), st₂ ∈ SHWSTBC \land st₂ \neq st₁))\]

The next language restriction makes clear that a particular BTA may not be triggered from different STs. If the same eBP 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).**\[\forall(f_s, tt, b), (f_t, tt, b) ∈ \rightarrow_{Trigger} : f_s = f_t\]

We use the function btaSrc : BTA → SHWSTBC to compute the shared state st with (nodef(st₁), tt, b) ∈ \(\rightarrow_{Trigger}\).

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 eBP 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).**\[\forall b ∈ BTA : (b, tt, n₁), (b, tt, n_l) ∈ \theta : n_k = n_l\]

**Proof.** 5.1.4. Directly follows from the definition of \(\rightarrow_{Eval}\) and \(\rightarrow_{Update}\). □

**Restriction 5.1.4 (Unprocessed BTA result).**\[\forall(b, tt, e) ∈ \rightarrow_{Update} : e ∈ nodeset(btaSrc(b))\]

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).**\[\forall(b₁, tt, d₁), (b₁, tt, d₁) ∈ \rightarrow_{Eval} : b₁ = b₁\]

**FACT 5.1.5 (Unique source ST of a DEC).**\[\forall(f_s, tt, b), (f_t, tt, b) ∈ \rightarrow_{Trigger} and (b₁, tt, d₁), (b₂, tt, d₂) ∈ \rightarrow_{Eval} holds: f_s = f_t\]
Proof 5.1.5. Directly follows from restrictions 5.1.3, 5.1.5 and fact 5.1.4. □

We use the function \( \text{decSrc} : \text{DEC} \rightarrow SH_{\text{WSTBC}} \) to compute the shared state \( s \) with \( (\text{node}(s), t, t, b) \in \rightarrow_{\text{Trigger}} \) and \( (b, t, d) \in \rightarrow_{\text{Eval}} \).

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).

\( \forall (d, g, e) \in \rightarrow_{\text{Route}} \) holds:
\[ g = (CGV, \exp) \land \exp \text{ 'indicates a protocol failure' } \Rightarrow e \in \text{nodeset}(\text{decSrc}(d)) \]

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 \( (b, t, d, d') \in \text{DEC} \), holds:
\[ \forall (d, g, e_1), (d, g, e_2) \in \rightarrow_{\text{Route}} : g = g_1 \Rightarrow e_1 = e_2 \]

Proof 5.1.6. Directly follows from the definition of \( \rightarrow_{\text{Route}} \) □

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 \( C \in N \times O \times V_1 \times V_{th} \) 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, \( V_1 \) 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 \( V_1 \) and \( V_{th} \) be \( \{0 \} \cup \{-1\} \). Note that although there are multiple timeouts defined (for different STs), at one point in time, there is at most one timer active. Let \( \chi \) be the function that computes the outcome of a BTA that has just finished. Further, \( \psi : 2^C \times N \times O \times V_1 \times V_{th} \rightarrow \{true, false\} \) 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, \( \psi([tt], C) = true \) for arbitrary \( C \). Any element \( t = (n_k, g, n_t) \in \theta \) is said to be enabled for a given \( C = (o_v, v_1, v_2, v_{th}) \) if \( n_k = n_o \land \psi(g, C) = true \). It directly follows that all \( t = (n_k, g, n_t) \in \rightarrow_{\text{Start}} \cup \rightarrow_{\text{ST1}} \cup \rightarrow_{\text{ST3}} \cup \rightarrow_{\text{Terminal}} \cup \rightarrow_{\text{Eval}} \cup \rightarrow_{\text{Update}} \) are always enabled once a configuration \( C \) contains \( n_o \) as the first component.

The semantics is defined operationally by the relation \( \models \subseteq (N \times O \times V_1 \times V_{th}) \times (N \times O \times V_1 \times V_{th}) \). The initial configuration of a WSTBC is \( C = (s_0, \emptyset, -1, -1) \), where \(-1\) for the timer and timer threshold values indicates that there is neither a current timer nor an according threshold. All transitions in \( t = (n_k, g, n_t) \in \rightarrow_{\text{Start}} \cup \rightarrow_{\text{ST1}} \cup \rightarrow_{\text{ST3}} \cup \rightarrow_{\text{Terminal}} \cup \rightarrow_{\text{Eval}} \cup \rightarrow_{\text{Update}} \) immediately fire once they are enabled, their processing is assumed to take zero time and the new configuration \( C' \) differs from the preceding \( C \) only in switching from \( n_k \) to \( n_t \). This kind of \( \models \) transitions reflects the fact that \( \rightarrow_{\text{Start}} \cup \rightarrow_{\text{ST1}} \cup \rightarrow_{\text{ST3}} \cup \rightarrow_{\text{Terminal}} \cup \rightarrow_{\text{Eval}} \cup \rightarrow_{\text{Update}} \) have been introduced for creating ebBP conform models only and that there is no logic in empty BTAs. The remaining elements of \( \models \) can be derived using the set of rules below that represent triggering BTAs (\( \rightarrow_{\text{Trigger}} \)), finishing BTAs (\( \rightarrow_{\text{End}}, \rightarrow_{\text{Update}} \)), evaluating BTAs (\( \rightarrow_{\text{Route}} \)), leaving STs by timeout (\( \rightarrow_{\text{Timeout}} \)) and the elapse of time (\( \rightarrow_{\text{Clock}} \)).

1. Trigger a BTA
\[
(n, o, v_1, v_{th}) \models \rightarrow_{\text{Trigger}} \\
(n', o, v_1, v_{th}) \text{ iff} \\
(n, g, n') \in \rightarrow_{\text{Trigger}} \land \\
\psi(g, (n, o, v_1, v_{th})) = true \land \\
((v_{th} = 1) \lor (v_t < v_{th}))
\]

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.
2: Finish a BTA and start result evaluation
   \((n, o, v_i, v_{ih}) \rightarrow^{\text{Eval}} \) 
   \((n', o', v_i', v_{ih}') \) if \(f\) 
   \((n, g, n') \rightarrow^{\text{Eval}} \land \phi_n \) 
   \(\psi(g, (n, o, v_i, v_{ih})) = \text{true} \land \phi_n\) 
3: Finish a BTA and ignore result 
   \((n, o, v_i, v_{ih}) \rightarrow^{\text{Update}} \) 
   \((n', o', v_i', v_{ih}') \) if \(f\) 
   \((n, g, n') \rightarrow^{\text{Update}} \land \phi_n \) 
   \(\psi(g, (n, o, v_i, v_{ih})) = \text{true} \land \phi_n\) 
   \(n' = \text{nodei}((\text{btasrc}(n)) \land v_i' = 0 \land v_{ih}' = \phi(\text{nodef}(\text{btasrc}(n)))) \lor \) 
   \(v_i' = v_i \land v_{ih}' = v_{ih}'\) 
4: Evaluate a BTA result 
   \((n, o, v_i, v_{ih}) \rightarrow^{\text{Route}} \) 
   \((n', o', v_i', v_{ih}') \) if \(f\) 
   \((n, g, n') \rightarrow^{\text{Route}} \land \phi_n \) 
   \(\psi(g, (n, o, v_i, v_{ih})) = \text{true} \land \phi_n\) 
   \(n' = \text{nodei}((\text{btasrc}(n)) \land v_i' = 0 \land v_{ih}' = v_{ih}'\) 
5: Leave ST by timeout 
   \((n, o, v_i, v_{ih}) \rightarrow^{\text{Timeout}} \) 
   \((n', o', v_i', v_{ih}') \) if \(f\) 
   \((n, g, n') \rightarrow^{\text{Timeout}} \land \phi_n \) 
   \(\psi(g, (n, o, v_i, v_{ih})) = \text{true} \land \phi_n\) 
   \((v_{ih} > -1) \land v_i' = v_i \land v_{ih}' = v_{ih}'\) 
6: Elapse of time 
   \((n, o, v_i, v_{ih}) \rightarrow^{\text{clock}} \) 
   \((n, o, v_i, v_{ih}) \) if \(f\) 
   \(v_i' = v_i + 1\) 

Finally, we state that a WSTBC is valid if and only if every node \(n \in N\) 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 \(t^*\) be the transitive closure of \(t\). A WSTBC is valid if and only if

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

\(ii\) \(\forall n \in N : \exists C = (n, o, v_i, v_{ih}) : (n, o, v_i, v_{ih}) \rightarrow^* (n, o, v_i, v_{ih}) \land n' \in T\)

### 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 \(S\) of an ESTBC is defined as the union \(\{s_0\} \cup S\) and for \(\cup S \subset \cup T\cup BTA\cup BTA\cup T\cup s\) that \(S\) 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. Classifying the WSTBC-

**Table 1: WSTBC/ESTBC Transition Relations**

<table>
<thead>
<tr>
<th>WSTBC</th>
<th>ESTBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-Start)</td>
<td>(-Start)</td>
</tr>
<tr>
<td>(-ST1)</td>
<td>No correspondence</td>
</tr>
<tr>
<td>(-ST3)</td>
<td>No correspondence</td>
</tr>
<tr>
<td>(-Terminal)</td>
<td>(-Route) \land (-Timeout)</td>
</tr>
<tr>
<td>(-ST2)</td>
<td>No correspondence</td>
</tr>
<tr>
<td>(-Trigger)</td>
<td>(-Trigger) \land ST \times {tt} \times SBTA</td>
</tr>
<tr>
<td>(-Eval)</td>
<td>Identical</td>
</tr>
<tr>
<td>(-Update)</td>
<td>(-Update) \land SBTA \times {tt} \times F \times ST</td>
</tr>
<tr>
<td>(-Route)</td>
<td>(-Route) \land \cup DEC \times 2^\cup \cup g^{fo} \times F \times ST</td>
</tr>
<tr>
<td>(-Timeout)</td>
<td>(-Timeout) \land \cup ST \times g^{fo} \times ST</td>
</tr>
<tr>
<td>(-Timeout)</td>
<td>(-Timeout) \land \cup \cup ST \times g^{fo} \times T</td>
</tr>
</tbody>
</table>

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 \(f\) in \(F = \{tt, ff\}\) 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 \(g^{fo}\) is used as abbreviation for \(\{\text{XPath1.timeout}\}\). Finally, the \(\phi\) function that maps every shared state to its corresponding timeout value is now defined on ST \(-\rightarrow N_0 \cup \{-1\}\) instead of FORK \(-\rightarrow N_0 \cup \{-1\}\).
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
variables : stMap<ST,SEBTA>; tMap<T,SEBTA>

Algorithm 1: ESTBC to WSTBC Conversion: part 1

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 n in ebc.T do
   9.   et = createEmptyNode();
   10.  tMap.add(⟨t, et⟩);
   11.  wbc.Terminal.add(⟨et, {t}, t⟩);
end

// Translate shared states
13. foreach s in ebc.ST do
14.   es = createEmptyNode();
15.   es = createEmptyNode();
16.   e = createJoinNode();
17.   f = createForkNode();
18.   f.th = st.th;
19.  wbc.SEPTA.add(⟨es⟩);
20.  wbc.SEPTA.add(⟨es⟩);
21.  wbc.JOIN.add(⟨e⟩);
22.  wbc.FORK.add(⟨f⟩);
23.  wbc.add(⟨es, (t), j⟩);
24.  wbc.add(⟨j, (t), e⟩);
25.  wbc.add(⟨e, (t), f⟩);
26.  stMap.add(⟨st, eb⟩);
end

// Translate Transitions
28. foreach ⟨n1, g, n2⟩ in ebc.−Start do
29.   e = stMap.get(n2);
30.  wbc.add(⟨n1, g, e⟩);
end

// ...continue

Algorithm 1: ESTBC to WSTBC Conversion: part 1

Algorithm 2: ESTBC to WSTBC Conversion: part 2

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-
migration 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 BusinessCollaborations 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 ReceiptAcknowledgements (RA) and AcceptanceAcknowledgements (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 role for the integration partner who sends the first business document of the reply business document. A WSTBC’s integration of ebBP and ebRP DocumentEnvelopes (DE), though for many business collaborations more sophisticated means will be necessary. A possible solution may be the definition of Schematron 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 or Web Services Transaction v1.2.

### 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 WS DL 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
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-BPEL 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 $r \in R$ 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 timeouts` 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.s0)))). 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

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

After initializing the two WS-BPEL processes, the shared state which can be reached from the initial state is trans-
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 BTAs with attached Decisions (cf. listing 5) admissible from the respective shared state are inserted. Finally, a so-called state epilog (cf. listing 6) closes all remaining open tags related to the shared state WS-BPEL code.

Listing 4: BPEL output of writeBPELStateProlog function

```xml
<condition>$processState_internal = '{state nameID}'</condition>
<scope name="{state nameID}.Timeout_Scope">
<faultHandlers>
  <catch faultName="StateTimeout">
    <empty />
  </catch>
</faultHandlers>
<assign>
  <from>
    <literal>
      $processState_internal
    </literal>
  </from>
  <to>
    {state inner entry nameID}
  </to>
</assign>
<sequence>
  <assign>
    <copy>
      $processState_internal
    </copy>
  </assign>
  <literal>
    $processState_internal
  </literal>
  <from>
    <literal>
      {state nameID}
    </literal>
  </from>
  <to>$processState_internal</to>
</sequence>
```

Algorithm 3: WSTBC to WS-BPEL translation algorithm

![Algorithm 3: WSTBC to WS-BPEL translation algorithm](image)

1. BPELprocesses<writeBPELHeader(wbc.r1.bpel, parentST( out(wbc.s0))), writeBPELHeader(wbc.r2.bpel, parentST( out(wbc.s0))))>
2. BPELprocesses<writeBPELStateProlog(wbc.r1.bpel, parentST(out(wbc.s0), st)), writeBPELStateProlog(wbc.r2.bpel, parentST(out(wbc.s0), tt))>
3. foreach link in out(nof( parentST( out(wbc.s0)))) do
4.   BPELprocesses<wbc.s (cf. listing 4)>
5. end
6. BPELprocesses<writeBPELStateEpilog(wbc.r1.bpel), writeBPELStateEpilog(wbc.r2.bpel)>
7. foreach st in wbc.ST | st ≠ parentST( out(s0)) do
8.   BPELprocesses<writeBPELStateProlog(wbc.r1.bpel, st, ff), writeBPELStateProlog(wbc.r2.bpel, st, ff)>
9. end
10. foreach t in wbc.T \{stk, tt\} in ctrlFlow(wbc) \{n\} in ctrlFlow(wbc) do
11.   BPELprocesses<writeBPELTerminalCode(wbc.r1.bpel, t), writeBPELTerminalCode(wbc.r2.bpel, t)>
12. end
13. return BPELprocesses<wbc.r1.bpel, wbc.r2.bpel>

input: A valid WSTBC to be transformed
output: Two orchestrated BPEL processes BPELprocesses<wbc.r1.bpel, wbc.r2.bpel>
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 nameIDs of all shared states of a business collaboration which constitute all possible values of processState, processState_internal also includes the nameIDs 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 scopes 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 in the specified respective timeout linkTo is assigned to processState_internal and a fault terminating all scopes not handling it is thrown. The outcome 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+DEC.placeholder function

```xml
<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

```xml
</pick>
</sequence>
</while>
</sequence>
</scope>
</scope>
</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

```xml
<condition>$processState_internal = '\{ terminal state nameID\}'</condition>
<sequence>
<assign>
<from>
<wslDoc:stateType>{terminal state nameID}</wslDoc:stateType>
</literal>
</from>
to>$processState</to>
</assign>
<invoke operation="dropProcessState" .. inputVariable="processState" />
<exit/>
</sequence>
</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

```xml
</sequence>
</while>
</sequence>
</scope>
</process>
```

Before ending the translation procedure, all placeholders for BTAs and attached Decisions 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 BTAs and Decisions is depicted more briefly in the rest of this section.

<table>
<thead>
<tr>
<th>Role</th>
<th>BPEL Process Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiator</td>
<td>- enclosing onMessage, receiving a partner or backend system</td>
</tr>
<tr>
<td>+</td>
<td>- enclosing scope for complete BTA</td>
</tr>
<tr>
<td>Responder</td>
<td>- all variables required for the ebBP Decision</td>
</tr>
<tr>
<td>Requesting-/RespondingBA and the ebBP Decision</td>
<td>- catch blocks for all ebBP failure types containing the corresponding reaction as specified in the ebBP BusinessCollaboration</td>
</tr>
<tr>
<td>-catchAll block containing reaction as specified in ebBP BusinessCollaboration for AnyProtocolFailure</td>
<td>- onAlarm to implement the TimeToPerform parameter specified for the BTA</td>
</tr>
<tr>
<td>-sequence containing the BPEL code for the ebBP constructs (in this order): RequestingBA, RespondingBA, Decision. For the respective production rules see tables 3/4 and 5.</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Role</th>
<th>BPEL Process Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiator</td>
<td>- enclosing while for trying to get a valid RA/RAE until ebBP retryCount is exceeded</td>
</tr>
<tr>
<td></td>
<td>- scope to encapsulate RA/RAE handling</td>
</tr>
<tr>
<td></td>
<td>- invoke to forward RA validity using the backend system</td>
</tr>
<tr>
<td></td>
<td>- throw to throw ebBP AnyProtocolFailure in case no valid RA was received and ebBP retryCount is exceeded</td>
</tr>
<tr>
<td></td>
<td>- throw to throw ebBP RequestReceiptFailure in case of a valid RA</td>
</tr>
<tr>
<td></td>
<td>- catchAll block for technical failure (TF) handling</td>
</tr>
<tr>
<td></td>
<td>- rthrow to forward TF to enclosing scope if ebBP retryCount is exceeded</td>
</tr>
<tr>
<td></td>
<td>- onAlarm to implement ebBP timeToAcknowledgeReceipt</td>
</tr>
<tr>
<td></td>
<td>- throw ebBP AnyProtocolFailure if ebBP retryCount is exceeded</td>
</tr>
<tr>
<td></td>
<td>- invoke to forward BusinessDocument (BD) to get a RA/RAE from Responder</td>
</tr>
<tr>
<td></td>
<td>- invoke to check RA validity using the backend system</td>
</tr>
<tr>
<td></td>
<td>- onAlarm to implement ebBP timeToAcknowledgeReceipt</td>
</tr>
<tr>
<td></td>
<td>- throw ebBP RequestReceiptFailure in case of a RA</td>
</tr>
<tr>
<td></td>
<td>- catch block for RA handling</td>
</tr>
<tr>
<td></td>
<td>- reply construct to forward RA to Initiator</td>
</tr>
<tr>
<td></td>
<td>- throw ebBP RequestReceiptFailure in case of a RA</td>
</tr>
<tr>
<td></td>
<td>- onAlarm to implement ebBP timeToAcknowledgeReceipt</td>
</tr>
<tr>
<td></td>
<td>- invoke to forward BD to and get a RA/RAE from backend system</td>
</tr>
<tr>
<td></td>
<td>- reply to forward RA to Initiator</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Role</th>
<th>BPEL Process Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiator</td>
<td>- enclosing while for trying to get valid AA/AAE</td>
</tr>
<tr>
<td></td>
<td>- scope to encapsulate AA/AAE handling</td>
</tr>
<tr>
<td></td>
<td>- catch block to forward ebBP RequestAcceptanceFailure faults to enclosing scopes</td>
</tr>
<tr>
<td></td>
<td>- catchAll block to handle TF</td>
</tr>
<tr>
<td></td>
<td>- empty to wait for an AA/AAE despite of TFS</td>
</tr>
<tr>
<td></td>
<td>- onAlarm to implement ebBP timeToAcknowledgeAcceptance</td>
</tr>
<tr>
<td></td>
<td>- pick to receive either AA or AAE</td>
</tr>
<tr>
<td></td>
<td>- invoke to check AA/AAE validity using the backend system</td>
</tr>
<tr>
<td></td>
<td>- throw ebBP RequestAcceptanceFailure in case of a valid AA</td>
</tr>
</tbody>
</table>

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

a valid RA/RAE by sending the corresponding BD to the process of the integration partner until the specified ebBP retryCount is exceeded or a timeout occurs. Further, if both signal types are used, it waits to receive a valid AA/AAE until the occurrence of a timeout. At the end of the BTA mapping, an ebBP Decision 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.

<table>
<thead>
<tr>
<th>Role</th>
<th>BPEL Process Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiator</td>
<td>- invoke to send BD of RespondingBA to backend system in order to get an evaluation</td>
</tr>
</tbody>
</table>

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

Note that ConditionGuardValues are evaluated before DocumentEnvelopes.

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®). 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 JAXB7. 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](image1)

![Figure 9: Seller entering state Contract](image2)

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-*\(^6\). 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 Sandesha\(^7\)) and WS-Security (using Apache Rampart\(^8\)) 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\(^9\) [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., \(O(n)\). 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.

---

6http://jcp.org/en/jsr/detail?id=173
7http://jcp.org/en/jsr/detail?id=222
8http://ws.apache.org/sandesha/sandesha2/
9http://ws.apache.org/rampart/
10https://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 \(O(n)\) 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 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 \(3*|SH_{WSTBC}| / (|SBTA| + 2*|SH_{WSTBC}|)\).

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 \((4*|SH_{WSTBC}| + |T|) / (1 + 3*|SBTA| + 5*|SH_{WSTBC}| + T + 2*|DEC|)\). 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., \((1 + |SH_{WSTBC}| + |JOIN| + |SBTA| + 3*|DEC| + (|BTAs|-|DEC|))\). The number of ToLinks in Fork elements is captured by \(|SBTA|\) and \(|SBTA|\) 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 BusinessTransactions 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 any graph to a suitably restricted form.

<table>
<thead>
<tr>
<th>Metric</th>
<th>WSTBC</th>
<th>ESTBC</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>787</td>
<td>591</td>
<td>~ 0.249</td>
</tr>
<tr>
<td>BTA</td>
<td>31</td>
<td>15</td>
<td>~ 0.516</td>
</tr>
<tr>
<td>Fork</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Join</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Decision</td>
<td>13</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>Success</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Failure</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ST</td>
<td>0</td>
<td>7</td>
<td>undefined</td>
</tr>
<tr>
<td>Sum of nodes</td>
<td>60</td>
<td>37</td>
<td>~ 0.383</td>
</tr>
<tr>
<td>ToLink</td>
<td>67</td>
<td>58</td>
<td>~ 0.134</td>
</tr>
<tr>
<td>FromLink</td>
<td>38</td>
<td>15</td>
<td>~ 0.605</td>
</tr>
<tr>
<td>Sum of links</td>
<td>105</td>
<td>73</td>
<td>~ 0.305</td>
</tr>
</tbody>
</table>

Table 6: WSTBC/ESTBC Complexity Comparison (Use Case)
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 are then 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 theoretical 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 Baltan, 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 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 EDD). 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 BusinessTransactions 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-
11. REFERENCES


(IEEE'08), 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

executing UMM business transactions. In Proceedings
of the 2007 Inaugural IEEE International Conference
on Digital Ecosystems and Technologies (IEEE DESt
IEEE Computer Society.

[16] R. Khalaf. From RosettaNet PIPs to BPEL processes:
a three level approach for business protocols. Data &

[17] B. Kiepuszewski, A. H. M. ter Hofsteede, and
C. Bussler. On structured workflow modelling. In
CAiSE '00: Proceedings of the 12th International
Conference on Advanced Information Systems
Springer-Verlag.

choreography to a BPEL-based implementation.

chain management. Industrial Marketing Management,

Analyzing interacting WS-BPEL processes using
flexible model generation. Data Knowl. Eng.,

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.

choreography to BPEL process orchestration.
Journal of Enterprise Information Management (JEIM),
Special Issue on MIOS Best Papers, 2006.

G. Wirtz. QoS-enabled business-to-business integration using
WS-CDL and WS-BPEL. In EDOC '07: Proceedings
of the 11th IEEE International Enterprise Distributed
Object Computing Conference, page 15, Washington,

Implementation Framework: Core Specification,
v02.00.01 edition, March 2002.

Webservice Choreography and Orchestration Perspectives to
International Conference on Software Engineering
Research and Practice (SERP’06), June 26-29 2006.

Choreography and Orchestration Perspectives to
International Conference on Software Engineering
Research and Practice (SERP’06), June 26-29 2006.

[27] 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 (SC4(B2B), Bangalore, India. IEEE, September


