A Model-Driven Approach for Monitoring ebBP BusinessTransactions

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

***Abstract*—ebXML BPSS (ebBP) is well-suited to specify Business-to-Business (B2B) interactions as choreographies of so-called BusinessTransactions. Web Services and WS-BPEL as dedicated interface technologies then can be used to provide the implementation of such choreographies. Tracking and ensuring the progress of choreographies calls for monitoring facilities that require gathering information from log data of the runtime systems that execute WS-BPEL processes. However, the information provided by WS-BPEL monitoring tools is fine-granular so that information about the actual progress in terms of choreographies must be extracted manually.**
Our approach streamlines the monitoring of ebBP BusinessTransactions leveraging model-driven engineering. First, hierarchical communicating automata are used to formalize BusinessTransactions. Second, WS-BPEL implementations of these automata are derived such that monitoring events are propagated to a monitoring service whenever a transition of the underlying automaton fires. Third, the monitoring service translates the monitoring events into choreography progress by visually highlighting the active and visited states within the hierarchical automata. It thus presents a user-friendly model that abstracts from the details of the implementing WS-BPEL processes. This makes tracking the current state of choreographies accessible to business users.

***Keywords*-Web Services Monitoring; Choreography; Orchestration; WS-BPEL; ebXML BPSS**

I. Introduction

ebXML Business Process Specification Schema (ebBP, [1]) as choreography format and the Web Services Business Process Execution Language (BPEL, [2]) as Web Services orchestration language have proven to be a fruitful toolchain for implementing Business-to-Business integration (B2Bi) scenarios [3], [4]. First, ebBP is used to specify the admissible sequences of business document exchanges as well as high-level security and reliability requirements. Second, BPEL is used to provide the implementation details based on Web Services. In order to guarantee the progress of interactions, monitoring the execution of distributed orchestrations is inevitable. While the granularity of ebBP is coarse and suited as communication means between software engineers or even business users, the granularity of BPEL is very fine and targeted at execution engines. This raises the question of how to monitor ebBP choreographies in a user-friendly way. Existing monitoring approaches for Web Services orchestrations only provide information at the fine-grained orchestration level.
Conversely, we provide an approach and a tool that offer monitoring information with choreography semantics accessible to business users. As ebBP choreographies are used for communication among humans, this approach promises better usability. As a first step, we concentrate on so-called BusinessTransactions as the core building block of ebBP and formalize these as hierarchical communicating automata. Using a model-driven approach, we derive and instrument BPEL implementations of the automata such that monitoring events reflecting state changes of the automata are delivered to a central monitoring service at runtime. This monitoring service then offers functionality to visualize the progress of BusinessTransactions by highlighting the currently active states and transitions of the underlying automaton formalization. Additionally, the execution history is traced by coloring and annotating the states and transitions that were traversed beforehand.

The paper proceeds as follows: Section II outlines the essentials of ebBP and section III introduces the monitoring concept that relies on representing ebBP BusinessTransactions as hierarchical communicating automata and on the approach for implementing these using BPEL. Section IV describes the monitoring architecture as well as the interplay of the BPEL orchestrations with each other and the monitoring service. In section V, the generation of the BPEL orchestrations using a model-driven approach is described. Section VI presents the user interface of our tool and validation results. Section VII discusses related work while section VIII concludes and points out directions for future work.

II. ebBP Basics

ebBP choreographies (denoted `Collaborations`) are composed from other ebBP choreographies and BusinessTransactions. In this paper, we concentrate on monitoring BusinessTransactions (BTs). A BT specifies the exchange of a request business document and an optional response business document between a requester (requesting) and a responding role. Accordingly, one-way and two-way BTs can be distinguished. The messaging details for each business document are specified in the so-called `RequestingBusinessActivity`

978-0-7695-4461-8/11 $26.00 © 2011 IEEE
DOI 10.1109/SERVICES.2011.41

or the `RespondingBusinessActivity`. Each `BusinessActivity` defines whether or not so-called `BusinessSignals` shall accompany business documents. `Receipt-` and `AcceptanceAcknowledgements` as well as corresponding exceptions can be used to signal the receipt of a legible business document or the successful import into a business application, respectively. Additionally, timeouts for the overall BT (`timeToPerform`, `toTTP`) and the exchange of `Receipt-`/`AcceptanceAcknowledgements` (`toRA`, `toAA`) can be defined using the ISO 8601 format for defining durations. So, if 'PT6M' is specified for `toRA` this means that a `ReceiptAcknowledgement` must be exchanged within a period of time of 6 minutes after the business document exchange. Further, a `retryCount` can be used to specify how many times a message exchange shall be retried in case of communication errors. Finally, reliability and security requirements such as `isAuthenticated` or `isConfidential` can be configured for the message exchanges.

III. Monitoring Concept

For realizing monitoring with choreography semantics, we assume an integration architecture (cf. [3], [4], [5]) that accommodates typical B2Bi settings and strives for separating application logic from control flow logic. Existing business applications (backends) implement application logic such as the creation and validation of business documents or the detection of real-world events that trigger business document exchanges. This functionality is assumed to be available as Web Services and is used by so-called control processes that govern the actual cross-enterprise message exchange. It is the task of these control processes to enforce message exchange sequences that strictly comply to ebBP choreography definitions. Figure 1 visualizes the relation between control processes and backends. Note that each integration partner has a control process of its own. Backends do not interact directly with each other, but request the exchange of business documents at their associated control process. Control processes, in turn, cater for secure and reliable cross-organizational message exchanges and signal the receipt of business documents as well as the results of BTs to the backends.

For exemplifying the basic interaction (cf. [4]) between

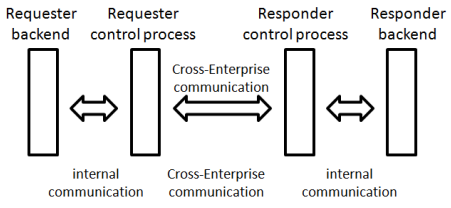

Figure 1. Basic Integration Architecture (from [4])

control processes and backends, assume that the requester backend of figure 1 detects the need to perform a BT and signals this need to the requester control process. The requester control process then coordinates the business document exchange with the responder control process. Both control processes interact with the according backend processes for fetching/delivering/creating business documents. At the end, the control processes deliver the result of the BT to the backend components.

Implementing control processes is a typical orchestration task and BPEL is a natural candidate for that. Consequently, the derivation of BPEL based control process implementations from ebBP choreographies has been proposed in [3], [4]. In this paper, we refine this concept by enhancing the BPEL processes with monitoring notifications that signal the progress of BTs to a central monitoring service that provides visualization features to human users. In section III-A, a communicating automata model for ebBP BTs is provided. The history and progress of BTs is tracked by highlighting the states and transitions of the automata. Subsequently, the realization of the model and the monitoring functionality based on BPEL is sketched in section III-B.

A. BusinessTransactions as Communicating Automata

The ebBP format provides parameterization options for BTs, but does only informally specify the control flow for the requester and responder role in a flow diagram (cf. [1], section 3.6.3). Before a precise model of control flow can be given, the influence of the different BT parameters on control flow has to be investigated. For this purpose, we reuse the analysis and model presented in [5]. Reliability and security features are recommended to be solved on the line level, e.g., using the WS-Security [6] and WS-ReliableMessaging [7] standards. Conversely, the number of BusinessActivities, BusinessSingals, the setting of timeout values as well as the retryCount hardly can be provided on the line level and therefore directly influence control flow.

In [5], a flat communicating automata model for the control processes of the integration architecture of figure 1 has been proposed. This model uniquely determines the set of admissible message exchange sequences by providing a formal operational semantics that is tailored to the characteristics of Web Services based B2Bi. By separating control flow from business logic and by using the automata model, control processes can be translated into fully executable BPEL processes. The isolation of control flow in control processes becomes evident in the choice of automata communication roles, i.e., in a `Partner`, a `Backend` and a `Collaboration` role. The `Partner` role is used for exchanging messages with the integration partner's control process, the `Backend` role represents interactions with existing business applications and the `Collaboration` role represents the superordinate process instance that governs the flow between subsequent BTs. The control flow then is specified by defining message send (role!msg) and

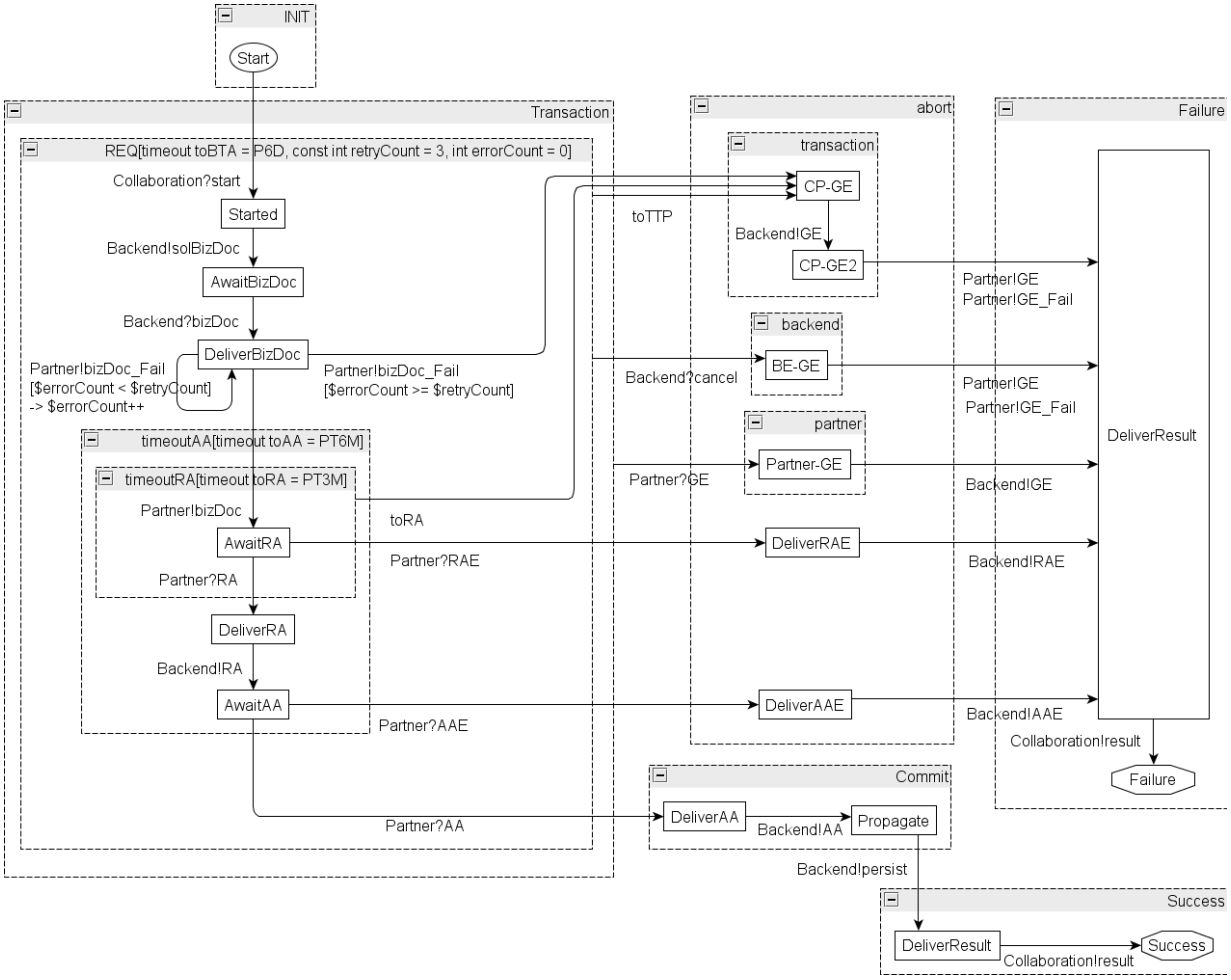

Figure 2. Sample Requester Automaton

receive (role?msg) events, by using states for capturing the effect of messaging events and by connecting these using transitions. The relevant messages are a `start` message, the business document to be exchanged (`bizDoc`) and the BusinessSignals Receipt-/Acceptance-Acknowledgement as well as Receipt-/Acceptance-AcknowledgementException (`ra`,`rae`,`aa`,`aae`). Additional control messages are `ge` for signaling general exceptions, `solBizDoc` for requesting the creation of a business document as well as `persist` and `result` for transaction demarcation and result propagation, respectively. The BT timeout definitions (toTTP, toRA, toAA; cf. section II) and messaging failure events (denoted as msg_Fail) are defined as events that are local to the partner. Finally, some helper variables for controlling the retryCount are used. As the automata model only acts as basis for monitoring BTs, please see [5] for the full formal model and semantics.

We basically reuse the flat automata model of [5], but transform it into hierarchical communicating automata for better visualization and for representing typical transaction phases such as `INIT`, `Commit` or `Failure`. Figure 2 shows the hierarchical automaton for the requester role of a One-Action BT with both BusinessSignals (`ra`, `aa`) specified and using all timeout values. Using this sample, we demonstrate the functional integrity of our approach and tool. All other configurations of BTs either are a subset of the automaton (when leaving out some BusinessSignal, timeout value or retryCount) or just require a duplication of the automaton (in case two business documents are to be exchanged within a Two-Action BT). For interpretation of the automaton, we comment the first few transitions: The requester role, which is the focal role of the automaton, starts out in the `Start` state and becomes activated by receiving the `start` message from the `Collaboration` role. Thereby, the `Started` state as well as the hierarchical states `Transaction` and `REQ` are entered. If a `ge` message is received from the `Partner` (defined on `Transaction`) the currently active sub-states

of Transaction are interrupted and the Partner-GE state is reached for subsequently notifying the Backend and the Collaboration about the result. Similarly, the timeout toTTP defined on REQ leads to state CP-GE. In the 'intended' flow, the requester role sends a solBizDoc to the Backend role and receives the bizDoc in turn. After reaching DeliverBizDoc, the requester role tries to send the bizDoc to the Partner. As the interaction with the Partner crosses the boundary of the 'controlled' internal environment of the requester role (cf. figure 1), send failures must be considered. Therefore, the local event Partner!bizDoc_Fail captures such a send failure. Depending on the number of failed attempts and the retryCount setting, the BT is aborted or continued. The interpretation of the remaining automaton transitions does not deviate conceptually from the presented transitions.

B. Monitoring by Instrumenting BPEL

Our approach to monitoring ebBP BTs relies on generating BPEL based control process implementations from BT specifications using model-driven engineering. First, the automaton representation of the respective BT specification is loaded. Then, the automaton is translated into BPEL. The basic idea for monitoring is extending the generation process with a call to a central monitoring service whenever a transition of the automaton is taken. In [3] and [4], flat automata structures are mapped to BPEL by means of switching across the automata states within a while loop. We apply this concept to hierarchical automata by nesting such while loop constructs according to the nesting structure of the hierarchical automaton.

Listing 1 shows the basic structure of the BPEL process that implements the automaton depicted in figure 2. First, partnerLinks for each role of the communicating automaton are defined and a correlationSets declaration is used to configure a unique transaction identifier (initialized upon receipt of the start message). Then a first sequence and scope (named 'main') are defined for declaring global variables. Within the 'main' scope's sequence two further scopes ('INIT' and 'AUTOMATON') for dealing with the initialization of the control process and implementing the automaton states are defined. The 'INIT' scope shows the typical incorporation of monitoring events by defining dedicated monitoring scopes after receive or invoke activities. Within these scopes, usual Web Services calls are used to propagate the monitoring information to the central monitoring service. The 'AUTOMATON' scope shows the while for switching across the top-level states of the requester control process (except for the Start state that has been dealt with in the 'INIT' scope). The hierarchical top-level states then set up nested while loops for switching across their sub-states. Timeouts are implemented by means of a dedicated *timer* Web service that takes a duration and the type of timeout (toTTP, toRA, toAA) and calls back the BPEL process when the timer has run out. Accordingly, the monitoring event signaling a timeout is propagated to the monitoring service after receipt of a timeout callback. Message failure events are propagated to the monitoring service after catching the corresponding communication error in BPEL fault handlers. The full BPEL process specifications that are generated for a particular communicating automaton are made available for download whenever a user finishes the generation process for a particular BT in the tool (cf. section V).

Listing 1. Basic BPEL Structure for Requester Role of Figure 2

```
<process ..>
<partnerLinks../>
<correlationSets../>
<sequence><scope name="main">
 <variables../>
 <sequence>
  <scope name="INIT"> ..
   <receive name="start-INIT-Start" ../>
   <scope name="MONITOR-INIT-Start_1">
    .. <invoke name="monitor-start-INIT-Start_1"
        ../>
   </scope>..
  </scope>
  <scope name="AUTOMATON">
   ..
   <while name="SWITCH-BTA">
   <condition>not($isFinished)</..>
    <if name="BTA-Transaction">
    <condition>$BTA-Transaction</..>
     <scope name="SCOPE-BTA-Transaction">
      ..
      <scope name="SCOPE-BTA-Transaction-REQ">
       ..
       <while name="SWITCH-BTA-Transaction-REQ">
        ..REQ implementation..
       </while>
       ..
      </scope>
     </scope>
    </if>
    <if name="BTA-abort">
     ..BTA-abort implementation..
    </if>
    .. <if/> for BTA-Failure, BTA-Commit and BTA-
       Success..
   </while>
  </scope>
 </sequence>
</scope></sequence>
</process>
```

IV. Monitoring Architecture

In this section, the architecture of the tool is outlined. It consists of three components, namely the *Monitoring Engine*, the *Test Engine* and the *BPEL Engine* as shown in figure 3.

For the *BPEL Engine*, the Oracle BPEL Process Manager[1] has been chosen. As described in more detail in section V, the BPEL control processes are deployed onto this engine which instantiates and executes instances of the control processes.

As the instances of the BPEL control processes do not only communicate with the corresponding BPEL control

[1]See http://www.oracle.com/technetwork/middleware/bpel/

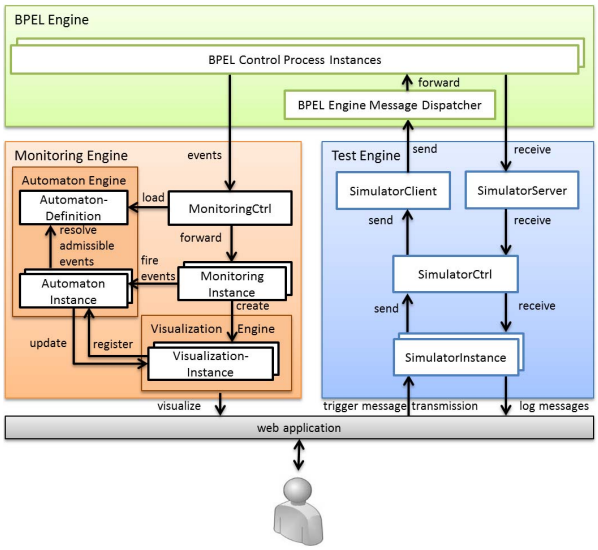

Figure 3. Monitoring and Simulation Architecture

process instances of their integration partner but also integrate the backend systems as well as additional services, stub implementations have to be provided. These stub implementations support the message property-based BPEL correlations mechanism for separating BPEL process instances. The *Test Engine* supports bidirectional communication with BPEL instances by providing `SimulatorCtrl`s with `SimulatorInstance`s which receive messages via a `SimulatorServer` and send messages to the control processes via the `SimulatorClient` interceptor. The `SimulatorCtrl` creates its `SimulatorInstances` which reuse the BPEL correlation information and act as a message sending and receiving proxy for the instances. The messages exchanged by each `SimulatorInstance` with its corresponding control process instance are stored and, therefore, are retrievable for the user. For the *Backend* and the *Collaboration* partner as well as for the *Timer* and the business document validation service, such a component has been created by means of inheritance. These components are able to create dummy messages which conform to the communication protocol without user interaction. The dummy messages are not automatically sent so that the user can decide when to trigger their transmission. Note that the *Test Engine* only has been created for validating our approach and would be replaced by the functionality of business applications in a real-world setting.

The main component is the *Monitoring Engine* onto which the hierarchical communicating automata have to be deployed. Each separate configuration of a BT, i.e., the selection of BusinessSignals, timeouts etc., is represented by an `AutomatonDefinition` while each instantiation of a BT is represented by an `AutomatonInstance`. For each `AutomatonDefinition` deployed, a `MonitoringCtrl` object is created which encapsulates the `AutomatonDefinition` and can receive monitoring events from the corresponding control process instances by providing a Web service endpoint. Received events are forwarded to the `MonitoringInstance` that correlates to the BPEL control process instance the event originated from. Each `MonitoringInstance` serves as a proxy for an `AutomatonInstance`. These `AutomatonInstances` are created by the *Automaton Engine* which allows to execute an `AutomatonDefinition` by firing events. The state information like the current state and the latest fired transition is stored in an `AutomatonInstance`. The `VisualizationInstances` of the *Visualization Engine* are created by the `MonitoringInstances` and are responsible for exporting snapshot images which can be viewed by the user. The visualizations and the resulting images are created with the yFiles for Java 2.8[2] library. This library also provides automatic layout algorithms which are applied during graph creation. The `VisualizationInstances` are connected to the `AutomatonInstances` via the Observer pattern which allows for constant updates upon state changes. The automaton visualizations are created as follows. First, the visual representation of the `AutomatonDefinition` is initialized by creating group nodes for composite states, nodes for states, and creating edges for transitions. Next, upon state change, the current state and latest transition are colored red while the already visited states and transitions are marked green. However, as loops are allowed and color alone cannot determine the unique execution path, transitions are labeled according to their occurrence in the execution history.

Both the *Test Engine* and the *Monitoring Engine* are implemented in Java and Groovy. They leverage the JAX-WS 2.2[3] API to create Web Services on demand at runtime by means of the `Endpoint` class as well as Web Services clients.

The user can interact with both, the *Test Engine* and the *Monitoring Engine*, via an integrated Web interface. The Web interface has been created using the Google Web Toolkit (GWT)[4] which is connected to the *Monitoring* and *Test Engine* via GWT Remote Procedure Calls (RPC). It aggregates all information for a specific ebBP BusinessTransaction definition and its instances by showing monitoring events and their visualization as well as the messages exchanged. Additionally, it allows to send dummy messages to the control processes via the *Test Engine*. Although the web interface aggregates information of the *Monitoring Engine* and the *Test Engine*, both engines run independently. This is important because, in a real world

[2]See http://www.yworks.com/en/products_yfiles_about.html
[3]See http://www.jcp.org/en/jsr/detail?id=224
[4]See http://code.google.com/webtoolkit/

scenario, the *Monitoring Engine* would be integrated into corporate monitoring tools while the *Test Engine* would be substituted by company-specific business applications. The *Test Engine* solely exists to simulate and test the execution with dummy messages. In section VI, the usage of the web interface is exemplified.

V. Generation Process

The model-driven generation process automatically derives the automata and the control processes from a given BT. The required steps are shown in figure 4 and are described in the following.

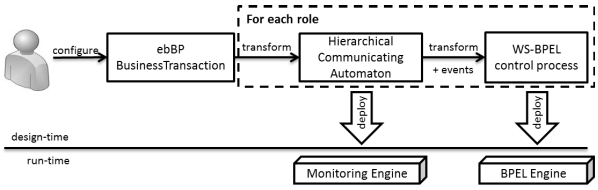

Figure 4. The Generation Process with the Created Artifacts

The process is triggered by the user who has to provide parameters to configure the desired ebBP BT. After validating the user parameters, an ebBP document containing an ebBP choreography definition with a single BT skeleton is loaded. This skeleton is altered according to the user parameters and validated against the ebBP XSD schema. The tool supports the configuration of one-way BTs with optional ReceiptAcknowledgement and AcceptanceAcknowledgement as well as corresponding timeouts.

Next, the model-driven generation process is forked into the process for the requesting role and the responding role of the BT. Each generation thread uses the configured ebBP BT along with the role they target at and transform the given BT to a hierarchical communicating automaton. This is done by employing predefined mappings from ebBP BTs to hierarchical communicating automata. While the different options for selecting BusinessSignals and timeouts are reflected in different mappings, the concrete values for timeouts and retryCount are used as parameters of the mappings.

The derived hierarchical communicating automaton is then transformed into its corresponding BPEL control process. The transformation maps each composite state, state and transition with their events to specific BPEL activities. The target BPEL structure is exemplified in listing 1. Along this transformation, the dependent WSDL files for the required Web Services interfaces as well as the required XSD files for the data types are created. Additionally, required component descriptors are generated allowing for packaging the BPEL process as a Composite Application[5] which can be deployed to the selected *BPEL engine*.

In order to represent ebBP BTs as hierarchical communicating automata and BPEL control processes in the model-driven process, *Ecore* meta-models based on the Eclipse Modeling Framework (EMF)[6] have been created. The EMF framework itself provides facilities to load and store meta-models as well as languages to transform models to other models or to text. Correspondingly, Ecore models for representing ebBP BT configurations, BT automaton representations and an abstract representation of BPEL have been created. The generation of BPEL processes then is performed in three steps. First, model-to-model transformations for translating ebBP configurations into automata and automata into abstract BPEL models, respectively, based on Eclipse QVTO (Query/View/Transformation Operation Mappings) are applied. Finally, a model-to-text transformation based on Eclipse Acceleo is applied to generate actual BPEL code from the abstract BPEL representation.

VI. Demonstration and Validation Results

The created tool allows two points of interaction with the user. Both are explained in the following.

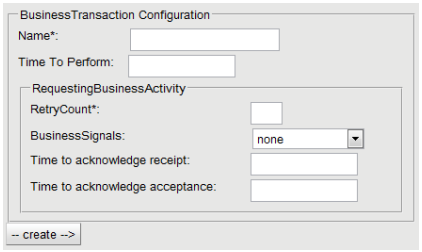

Figure 5. BusinessTransaction Configuration Form

Configure BusinessTransaction: In the first step, the user has to configure a BT by providing a name, timeouts, the retryCount as well as the selection of BusinessSignals. This is done by using the web-based form shown in figure 5 which guides the user by means of tool-tips as well as real-time validations. By clicking on the button *generate*, the generation process described in section V starts while the user waits for the process to complete. After completion, the created hierarchical communicating automata are deployed automatically to the *Monitoring Engine*, the BPEL control processes to the *BPEL Engine* and the partner simulation for this ebBP BT is started within the *Test Engine*. When all artifacts are deployed, the user is forwarded to the monitoring GUI for monitoring and simulating the execution of the generated ebBP BT.

Simulate and Monitor the Execution of BusinessTransaction: Simulating and monitoring the execution of BTs is controlled using the web-based interface shown in figure 6. Recall that a control process does not contain any business logic, but only *controls* the exchange of messages between backend and the integration partner's control process. In consequence, the exchange of messages has to be triggered using the simulator functionality outlined in section IV. For

[5]See http://www.oasis-opencsa.org/sca

[6]See http://www.eclipse.org/modeling/emf/

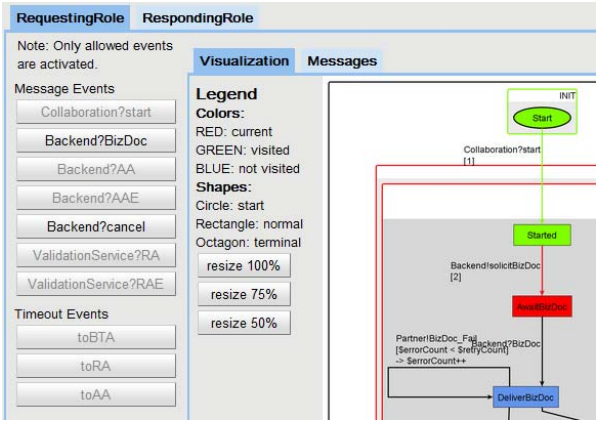

Figure 6. Monitor and Simulate the Execution of a BusinessTransaction

triggering a BT run, the user has to create an execution instance by clicking the *new instance* button which creates a UUID that is used for message correlation. This execution instance corresponds to a `SimulatorInstance` of the *Test Engine*. Next, the user can send preconfigured messages to the control processes of the requesting role or the responding role by using the controls shown at the top of figure 6. The labels of the controls correspond to the message exchange events as defined for the communicating automata in section III-A. Note that the interface depicted in figure 6 allows for sending messages **to** both control processes of a BT implementation for providing a test facility that spans the complete integration architecture. Only the messages that are sent **from** control processes cannot be triggered by the interface. The exchanged messages of an execution instance can be inspected for each participating role in the tab named *messages* while the visualization of the progress is shown in tab *visualization*.

Using the tool's functionality, the different permutations of BT control flow configuration options could successfully be performed, i.e., the happy path of the interactions could be performed. Due to dependencies between the configuration options, e.g., a toAA does not make sense for a BT without AcceptanceAcknowledgement, the number of permutations amounted to 13. The reaction to erroneous behavior, e.g., not providing an AcceptanceAcknowledgement on time or sending a BusinessSignal that has not been configured, has not been tested for each permutation. Instead, the reaction to each type of error has been tested for the BT configuration that uses all available BusinessSignals, timeouts and the retryCount. Based on these results, we are confident to provide a fully functional prototype implementation of the integration architecture of figure 1. Please note that the prototype cannot simulate message sending failure events as this would require the interception of messages.

VII. Related Work

The proposed monitoring approach reuses several existing concepts and combines them. The authors of [8] proposed a model-driven process for creating BPEL orchestrations from ebBP artifacts. However, neither a formalization of valid interactions nor an implementation of the BPEL generation are provided. The idea of instrumenting BPEL processes for monitoring aspects is stated in [9] while the concept of having monitor classes with its corresponding monitor instances is proposed in [10]. A formal basis for the communicating automata for ebBP BTs is given in [5] on which the hierarchical communicating automata have been created. The ideas and concepts have been combined and extended to design and create the approach and tool presented in this paper.

In several approaches, the execution of ebBP BT-like concepts such as UMM BusinessTransactions or RosettaNet PIPs is proposed [8], [11], [12], [13], [14]. However, monitoring and visualization features as presented in this paper have not been reflected in the corresponding implementation concepts so far.

VIII. Conclusion and Future Work

This paper presents an approach and tool for monitoring the progress of ebBP BTs by incorporating monitoring notifications into the implementing BPEL processes. Monitoring information is tied to hierarchical communicating automata as a B2Bi choreography-level formalization of BTs for ensuring usability. Using a model-driven approach, the BPEL-based BT implementations that include monitoring events whenever a choreography-level state transition is taken are automatically generated. The conceptual foundation for being able to derive fully executable BPEL processes automatically is an integration architecture that separates business logic from control flow. The BPEL processes focus on implementing control flow while the use case specific business logic is abstracted by a set of standard interfaces. The feasibility of the approach is demonstrated by providing a prototypic implementation that supports all essential BT configuration options that influence control flow.

Future work targets at supporting a larger subset of ebBP choreographies and at generalizing the underlying model for visualization.

Concerning user-defined ebBP choreographies, two ebBP dialects for composing BTs are available [3], [4] that can be interpreted as state machines. Hence, reapplying the automata model by interpreting BTs as sub-states of larger ebBP choreographies seems to be applicable. Special focus will be put on the constraints for folding such automata for maintaining usability.

Concerning generalization of the approach, alternative choreography languages as well as orchestration languages are to be considered:
For choreography languages, the distinction between interconnection choreographies and interaction choreographies [15] seems to be pivotal. While interconnection choreographies focus on the local send and receive actions of

individual partners as well as the interconnection of corresponding send/receive actions, interaction choreographies treat corresponding send and receive events as atomic actions and define sequences of these actions. ebBP corresponds to the class of interaction choreographies and provides BTs as atomic building blocks. Other interaction choreographies such as WS-CDL [16] or Let's Dance [17] seem to be accessible to hierarchical graph structures as well. The challenge for these languages is the deviating semantics of transitions between graph elements which may require additional logic for interpreting and visualizing monitoring events. Interconnection choreography languages such as BPEL4Chor [18] or Inter-Organizational Workflow Nets [19] frequently cannot be decomposed hierarchically and therefore require fundamentally different visual representations.

For orchestration languages, alternative implementation languages such as the Windows Workflow Foundation 4[7] and abstract orchestration characterizations modeled in more high-level languages are worth being investigated. Those high-level orchestration languages such as Yet Another Workflow Language[8] [20] or the Business Process Model And Notation 2.0[9] [21] also offer a visual model and therefore may be an option for visualizing interconnection choreographies.

REFERENCES

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

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

[3] A. Schönberger, C. Pflügler, and G. Wirtz, "Translating shared state based ebXML BPSS models to WS-BPEL," *International Journal of Business Intelligence and Data Mining*, vol. 5, no. 4, pp. 398 – 442, 2010.

[4] A. Schönberger and G. Wirtz, "Towards executing ebBP-Reg B2Bi choreographies," in *Proceedings of the 12th IEEE Conference on Commerce and Enterprise Computing (CEC'10), Shanghai, China*. IEEE, November 10-12 2010.

[5] A. Schönberger, G. Wirtz, C. Huemer, and M. Zapletal, "A composable, QoS-aware and web services-based execution model for ebXML BPSS businesstransactions," in *Proc. of the 6th 2010 World Congress on Services (SERVICES2010), Miami, Florida, USA*. IEEE, July 2010.

[6] OASIS, *Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)*, OASIS, February 2006.

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

[8] J.-H. Kim and C. Huemer, "From an ebXML BPSS choreography to a BPEL-based implementation," *SIGecom Exch.*, vol. 5, no. 2, pp. 1–11, 2004.

[9] J. Schiefer, H. Roth, and A. Schatten, "Auditable WSBPEL: Probing and monitoring of business processes with web services," *International Journal of Business Process Integration and Management, Inderscience*, vol. 2, no. 1, pp. 60–73, 2007.

[10] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti, "Runtime monitoring of instances and classes of web service compositions," in *IEEE International Conference on Web Services 2006 (ICWS '06), 2006*, 2006, pp. 63–71.

[11] R. Khalaf, "From RosettaNet PIPs to BPEL processes: A three level approach for business protocols," *Data & Knowlegde Engineering*, vol. 61, no. 1, pp. 23–38, 2007.

[12] A. Schönberger and G. Wirtz, "Realising RosettaNet PIP Compositions as Web Service Orchestrations - A Case Study," in *The 2006 International Conference on e-Learning, e-Business, Enterprise Information Systems, e-Government, & Outsourcing (CSREA EEE), Las Vegas, Nevada, USA*, 2006.

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

[14] B. Hofreiter, C. Huemer, P. Liegl, R. Schuster, and M. Zapletal, "Deriving executable BPEL from UMM business transactions," in *Proceedings of the IEEE International Conference on Services Computing (SCC)*, 2007, pp. 178–186.

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

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

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

[18] G. Decker, O. Kopp, F. Leymann, and M. Weske, "BPEL4Chor: Extending BPEL for modeling choreographies," in *Proceedings of the 2007 IEEE International Conference on Web Services (ICWS), July 9-13, 2007, Salt Lake City, Utah, USA*, 2007, pp. 296–303.

[19] W. M. P. van der Aalst and M. Weske, "The P2P approach to interorganizational workflows," in *CAiSE '01: Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, 2001, pp. 140–156.

[20] W. M. P. van der Aalst and A. H. M. ter Hofstede, "Yawl: yet another workflow language," *Information Systems*, vol. 30, no. 4, pp. 245 – 275, 2005.

[21] OMG, *Business Process Model and Notation, v2.0*, OMG, January 2011.

[7] See http://msdn.microsoft.com/en-us/library/dd489441.aspx

[8] See http://www.yawlfoundation.org/

[9] See http://www.bpmn.org/