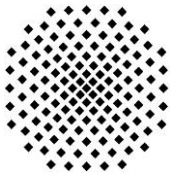

Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

Modeling Choreographies using the BPEL4Chor Designer: an Evaluation Based on Case Studies

Andreas Weiß, Vasilios Andrikopoulos,
Santiago Gómez Sáez, Dimka Karastoyanova,
Karolina Vukojevic-Haupt

Report 2013/03

Institut für Architektur von Anwendungssystemen

Universitätsstraße 38
70569 Stuttgart
Germany

CR: D.2.2

Contents

1 Introduction

This Technical Report shows a tool for modeling complex systems and it evaluation in the scope of case studies. The tool supports a top-down approach for modeling systems as choreographies in BPEL4Chor [DKLW07], [DKLW09], which can be transformed into executable BPEL processes and be executed on BPEL engines. The approach allows for a distributed enactment of a choreography. The approach has been introduced in [DKLW09]. The focus of this work is to demonstrate capabilities of the choreography modeling tool BPEL4Chor Designer, developed in the scope of our research work.

For this purpose we use two scenarios: the Taxi Application scenario [Hag11] depicted in Figure 4 and a scenario from the field of Smart Transportation, the FlexiBus scenario[1].

The remainder of this Technical Report is structured as follows: Section 2 shows an overview of the approach. Section 3 reviews the choreography language BPEL4Chor. In Section 4 the modeling of choreographies with our choreography editor is discussed and the state of the current implementation is analyzed. The transformations that are necessary to create executable processes for the participants of a choreography are covered in Section 5. In Section 6, we show by example how the generated processes can be refined manually with further non-communication process logic. Finally, in Section 7 we sum up the Technical Report and draw a conclusion.

2 Modeling Choreographies with the BPEL4Chor: Overall Approach

Figure 1 shows the steps of the overall approach starting with the domain problem definition and producing the executable processes in BPEL as a last step, as it was introduced by [DKLW09]. We assume that the domain problem description exists as plain text or a graphical process modeling language such as BPMN[2]. The domain problem is then modeled as a choreography using our Eclipsed-based choreography editor, the BPEL4Chor Designer [Son13] . The choreography model is an instance of the editor's choreography meta-model.

After modeling with the graphical choreography editor, described later in section 4, the choreography model can be transformed automatically into the choreography language BPEL4Chor [DKLW07], [DKLW09]. The generated BPEL4Chor artifacts can be used in a second transformation step to generate abstract BPEL processes and WSDL files. These WSDL files contain the technical information about the interfaces between the participants, i.e the port types, operations, messages, and partner links. Each previously modeled participant is transformed into exactly one abstract BPEL process. The abstract BPEL processes are input for the basic executable completion

[1] http://www.allow-ensembles.eu/

[2] http://www.bpmn.org/

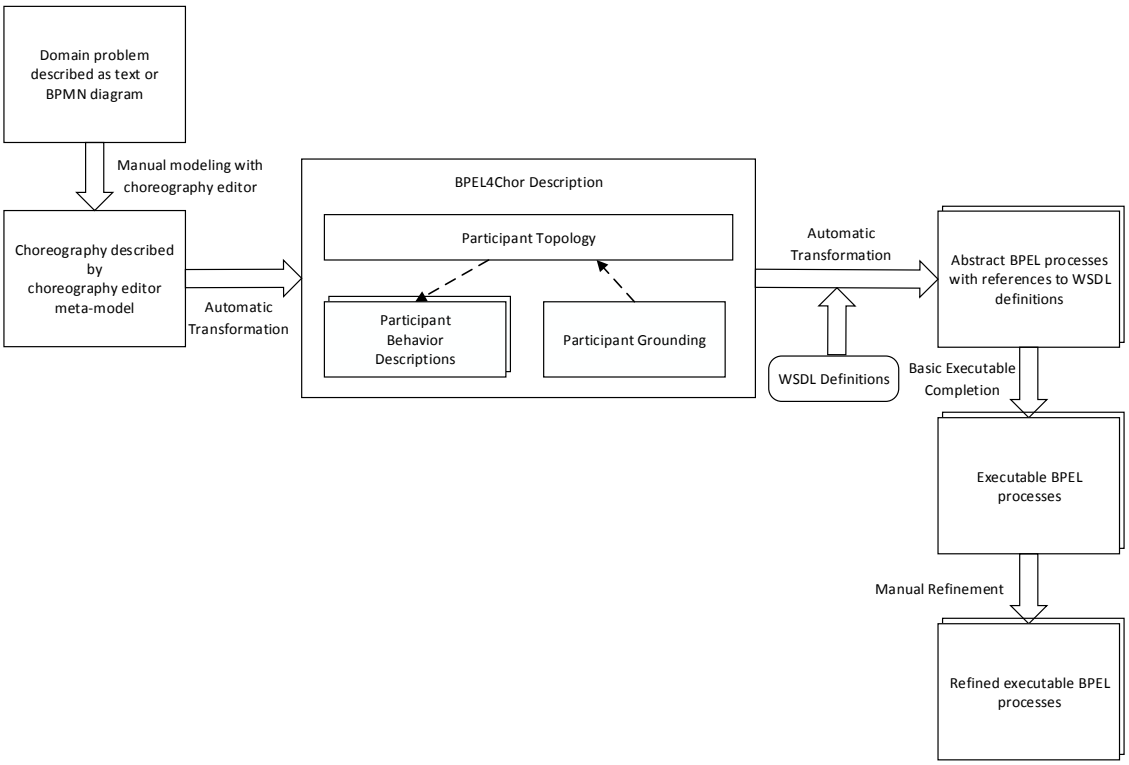

Figure 1: From a domain problem to executable BPEL processes. Adapted from: [DKLW09]

step, which transforms the abstract processes into executables ones. They can then be refined manually with the necessary domain logic that is not part of the communication behavior of the participants.

3 BPEL4Chor

BPEL4Chor [DKLW07] is a choreography language forming an additional layer on top of the BPEL standard [OAS07]. Whereas BPEL aims for orchestrating Web Services, BPEL4Chor specifies choreographies of orchestrated services, i.e. choreographies of workflows. The language follows the so-called *interconnected interface behavior model* approach. This approach avoids issues of *locally unenforceable* behavior that languages following the *interaction model* approach have. Interaction models comprise of request and request-response messages exchanges as basic interactions that can be grouped into more complex ones while behavioral dependencies are described between them. The dependencies are defined from a global perspective but possibly cannot be enforced. An example would be the sending of messages that depend on the sending of other

messages but it is left unspecified how the current sender can learn of the previous messages. Additional synchronization messages must be used to overcome this problem. The interconnected interface behavior models, e.g. BPMN and BPEL4Chor on the other hand prevent these issues by specifying the control flow of every participant. Nevertheless, deadlocks could arrive when a participant waits for a message from another participant in order to proceed but the message never arrives. Timeout mechanisms must be present to avoid deadlocks.

Figure 2 shows the artifacts the language consists of. The *Participant Behavior Descriptions* (PDB) use abstract BPEL processes to specify the communication behavior of choreography participants. Only activities that are allowed according to the *Abstract Process Profile for Observable Behavior* [OAS07] specified by the BPEL standard must be used for the description of the communication behavior. Furthermore, BPEL4Chor forbids partnerLink, portType and operation attributes. This helps to achieve the goal of decoupling the PBDs and the description of the message exchange from a concrete technical realization.

The structural aspects of a choreography are captured in the *Participant Topology*. It consists of *Participant Types*, *Participant References*, and *Message Links*. Participant Types are represented by the abstract BPEL processes of the Participant Behavior Description. Participant References are the BPEL4Chor representation of choreography participants. The Message Links indicate which participants can exchange messages with each other. Therefore, the participant topology connects the different Participant Behavior Descriptions in a choreography.

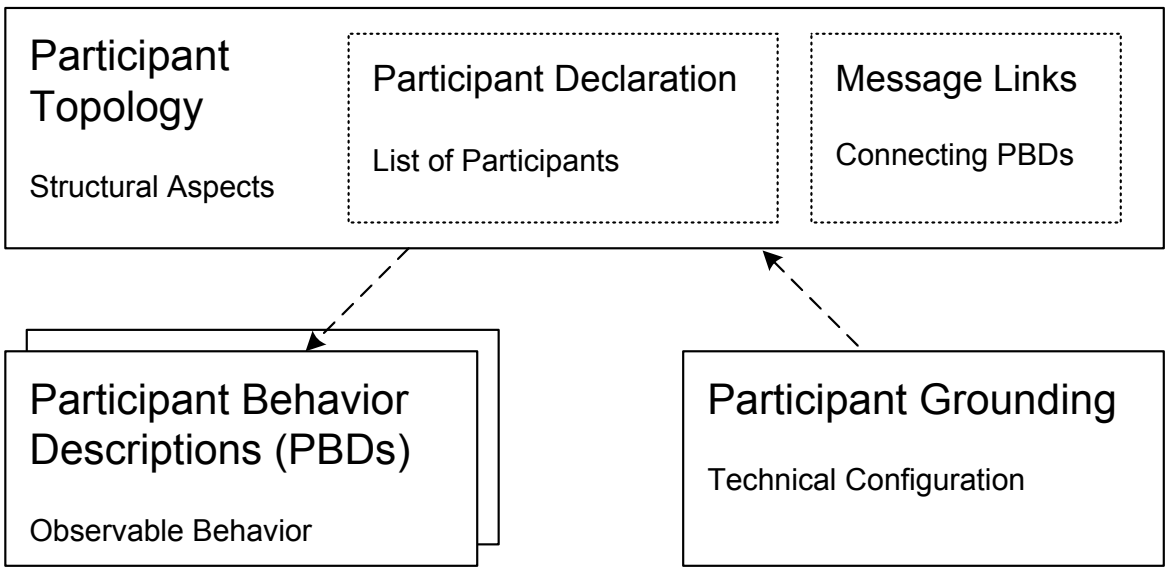

Figure 2: BPEL4Chor artifacts. Source: [DKLW09]

The third artifact in BPEL4Chor is the *Participant Grounding*. BPEL4Chor offers

the possibility to separate technical information such as the message format, the port types, and operations of processes from the abstract communication behavior, i.e. from the Participant Behavior Description and the Participant Topology. The technical information is specified in a grounding file.

4 The BPEL4Chor Designer

Our choreography editor, the BPEL4Chor Designer, was built using the Eclipse Platform in the context of the work of [Son13]. The Eclipse Modeling Framework (EMF)[3] is part of the Eclipse Platform and follows a model-driven-architecture[4] approach where code is generated from models. It provides facilities to build own meta-models with the meta-meta-model Ecore notation, which is similar to UML[5]. Furthermore, the BPEL4Chor Designer uses the Graphical Modeling Framework (GMF)[6] and the Graphical Editing Framework (GEF)[7].

The BPEL4Chor Designer's meta-model (called in the following Chor Model) specifies the diagram elements independently from existing choreography languages. Nevertheless, the meta-model is oriented on BPEL4Chor and BPEL concepts as they facilitate the description of choreographies and orchestrations, respectively. The meta-model consists of four Ecore models: Chor Model, PBD Model, Topology Model, and Grounding Model.

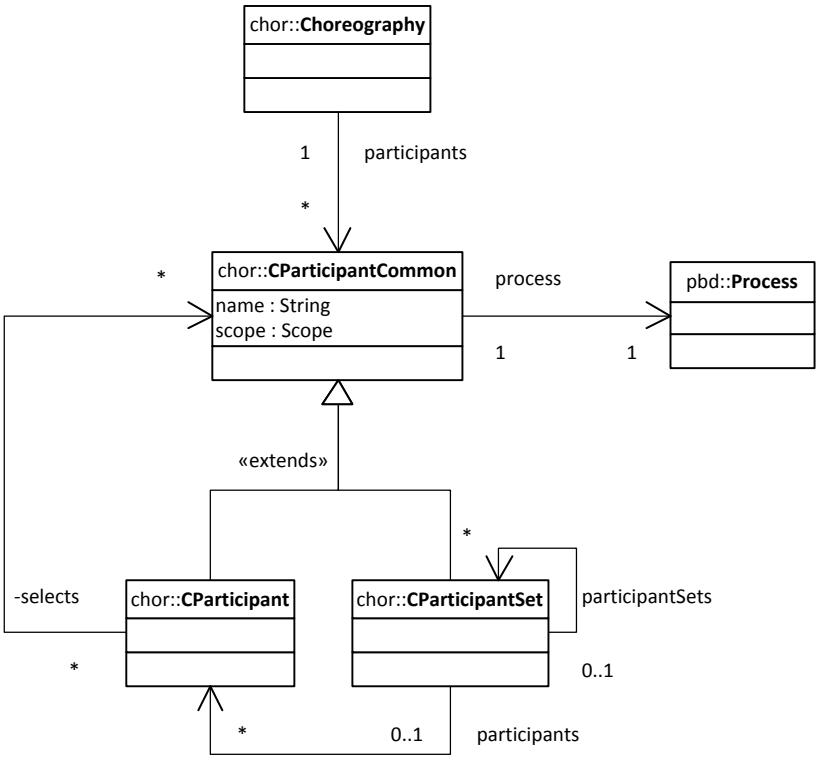

Figure 3: Chor Model with participants. Source: [Son13]

[3]http://www.eclipse.org/modeling/emf/
[4]http://www.omg.org/mda/
[5]http://www.uml.org
[6]http://www.eclipse.org/modeling/gmp/
[7]http://www.eclipse.org/gef/

The *Chor Model* specifies all elements that can be placed on the drawing area of the editor. Figure 3 shows an excerpt of the Chor Model in the EMF Ecore notation. The communication activities of a participant can be captured with the *PBD Model* and correspond to the notion of a Participant Behavior Description in BPEL4Chor. The participant classes in the Chor Model reference the Process class in the PBD Model, thus, a connection between both models is established. Further elements of the meta-model are the *Topology Model* and the *Grounding Model*. The Topology Model contains the participants, the types, and the message links. The Grounding Model describes the technical information that are captured for a choreography.

4.1 Modeling With the BPEL4Chor Designer

The following subsection discusses the two case studies used to evaluate the BPEL4Chor Designer in this Technical Report: the Taxi Application scenario and the FlexiBus Scenario.

4.1.1 Taxi Application Scenario

In Figure 4 a simplified representation of the Taxi Scenario is shown. The BPMN diagram has three lanes depicting three participants: Customer, Taxi Company, and Taxi Service Provider of the Taxi Application choreography [Hag11]. If a Customer wishes to book a Taxi, he sends an initial request to the Taxi Company call center, which forwards it to the Taxi Service Provider. The Taxi Service Provider process determines the nearby available taxis and the contact information of the taxi drivers using Context Integration Processes (CIPs) [WKNL07]. The CIPs are not shown in the BPMN diagramm for brevity. Subsequently, a transport request is sent to each available taxi driver and the responses of the taxi drivers are collected for a specified duration. The gathered transport information is sent back to the Customer.

Figure 5 shows the Taxi Application as a choreography modeled with our BPEL4Chor Designer. Each process lane from Figure 4 is represented as participant in the choreography diagram of Figure 5. Figure 6 shows the complete Taxi Application scenario including the Context Integration Processes. The taxi drivers are represented by the Taxi Transmitters, devices carried by the drivers. The rectangular shapes in the editor view in Figure 5 stand for the choreography participants whereas the message links and their directions are depicted by labeled arrows. The editor also incorporates the a-priori unknown number of taxi drivers involved in the choreography, i.e. the Taxi Transmitter devices of the taxi drivers form a set of participants. This is reflected by the dashed boundary of that particular shape. Inside the participants the control flow regarding the communication behavior, represented by activities such as <receive> or <send>, is visible. After modeling a choreography like the Taxi Application scenario the editor offers the functionality to transform the choreography instance of the editor's meta-model into BPEL4Chor.

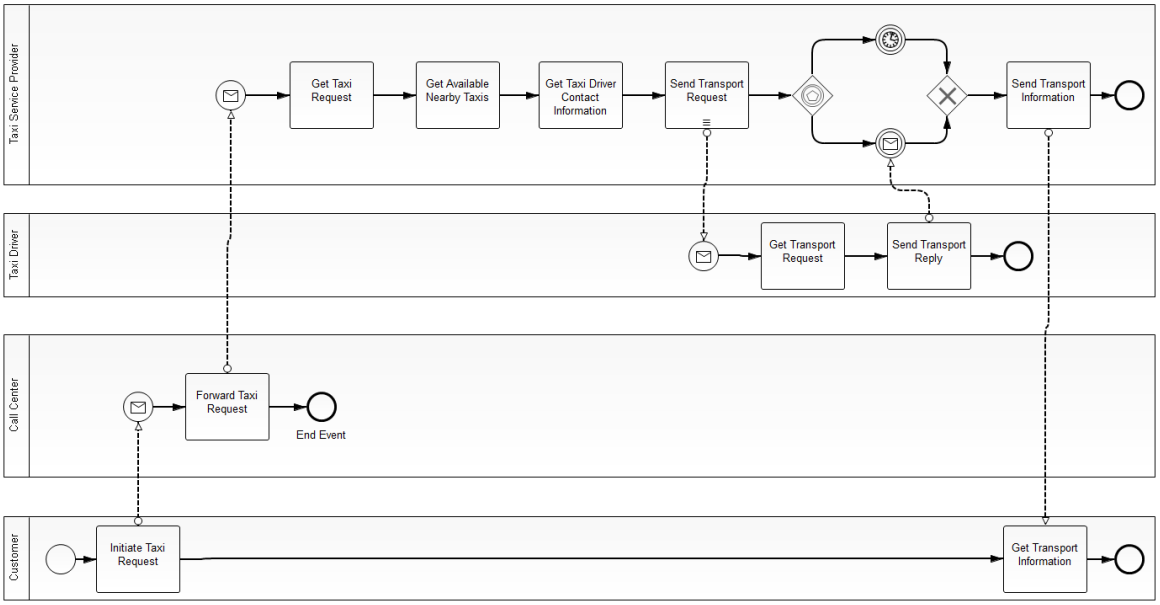

Figure 4: The BPMN diagram of the Taxi Application. Adapted from: [Hag11]

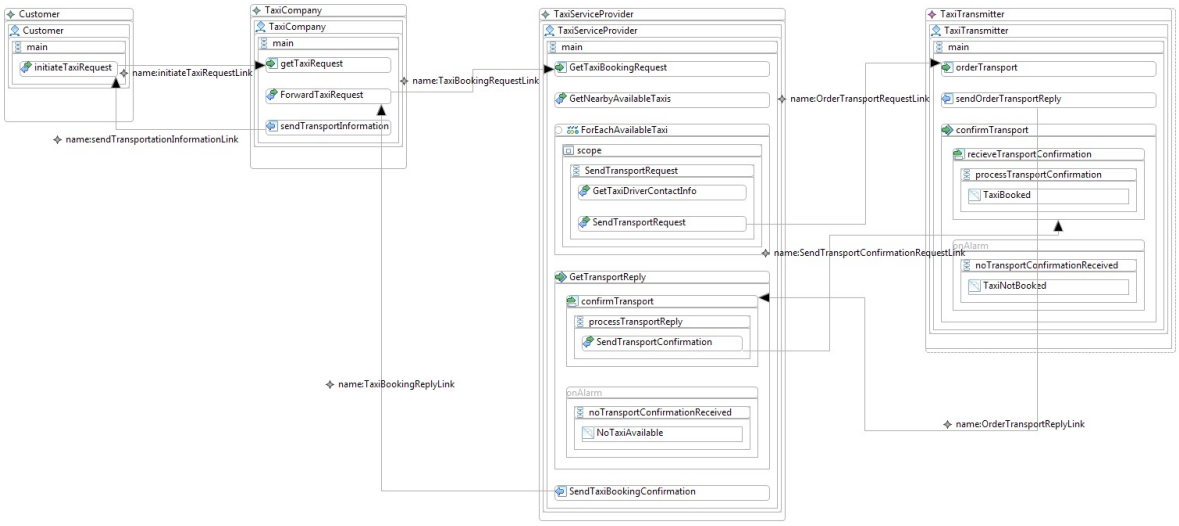

Figure 5: Excerpt of the Taxi Application choreography modeled with the BPEL4Chor Designer

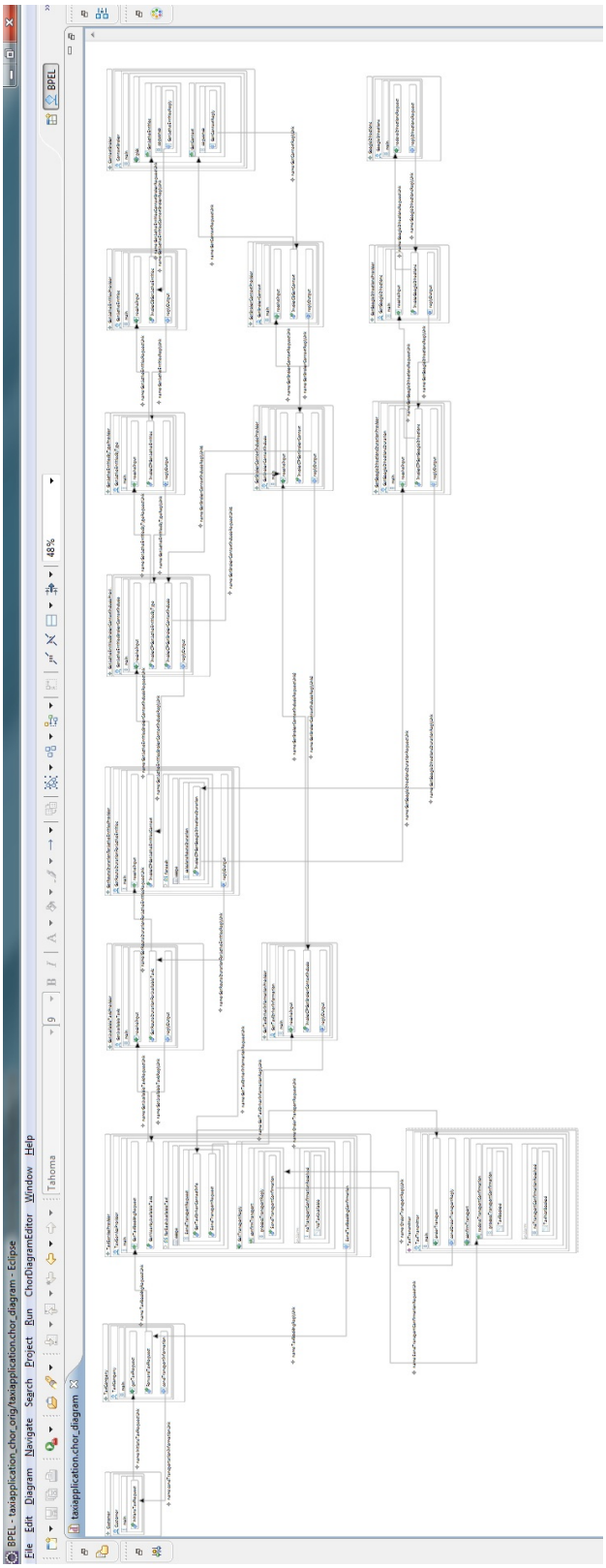

Figure 6: Complete Taxi Application choreography modeled with the BPEL4Chor Designer

4.1.2 FlexiBus Scenario

The second scenario used as case study for the evaluation of the BPEL4Chor Designer is the FlexiBus scenario from the Allow Ensembles project[8]. In this scenario, the description of the participants was available as plain text and non-standardized process diagrams. Figure 7 shows an excerpt of the FlexiBus scenario considering trip booking is modeled with the BPEL4Chor Designer as a choreography. Whereas buses in public transport have fixed routes and schedules, the FlexiBus is oriented by passenger needs and heads for pre-booked pick-up locations. The participants of the choreography are the Passenger, the Passenger Management System (PMS), the Route Planer, and the Payment Manager. A Passenger requests a trip from the PMS, which forwards the request to the Route Planer in order to obtain information about the particular route.

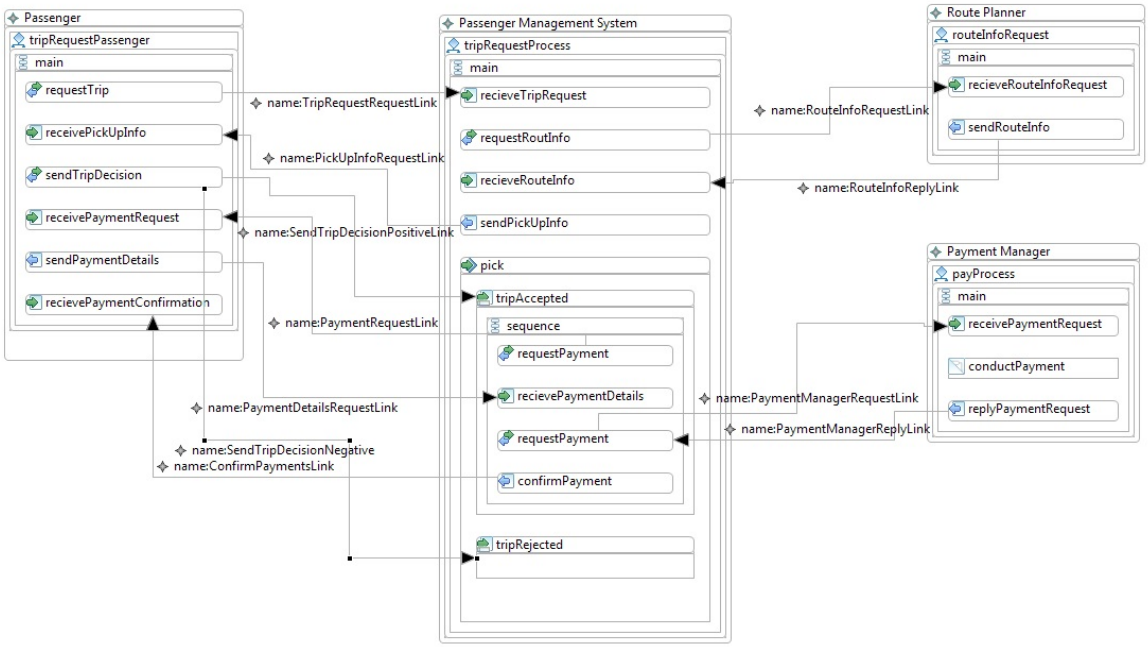

Figure 7: Trip Booking in the FlexiBus scenario modeled with the BPEL4Chor Designer

The Passenger Management System sends the information where the Passenger can be picked up for this particular route to the latter one. The Passenger can then decide about the trip and send the decision to the Passenger Management System. In case of acceptance of the trip by the Passenger the PMS requests and receives the payment details from the Passenger and contacts the participant Payment Manager to conduct the Payment. Finally, the Passenger receives a payment confirmation message.

[8]http://www.allow-ensembles.eu/

4.2 State Of The Implementation

The current implementation of the choreography editor offers the following base activities for modeling the *communication behavior* of choreography participants:

- <receive>
- <reply>
- <invoke>
- <opaqueActivity>

The variables of these activities can only be set to opaque.

The following *structured activities* needed to model choreographies are implemented in the editor:

- <sequence>
- <scope>
- <pick>
- <flow>

Furthermore, the editor allows to set <correlationSet> and <messageExange> elements in the root element <process> of the abstract BPEL processes, which represent the PDB of a choreography. The non-parallel and parallel versions of the <forEach>activity are also part of the editor's functionality. Activities can be connected with <link> elements inside a <flow> activity.

The following activities are not implemented in the BPEL4Chor Designer:

- <while>
- <repeatUntil>
- <if>
- <assign>

However, the editor's EMF meta-model already incorporates all activities allowed by the BPEL4Chor language and the feature completion will be straightforward.

4.3 BPEL4Chor Designer: Evaluation Findings

Although choreographies can be modeled, there are still some issues that must be resolved. Most of them are not necessarily implementation errors but rather inconvenient for the users of the BPEL4Chor Designer. In the following the issues and possible workarounds are documented.

- It is not possible to put new activities between already modeled activities. Instead, the activities after the new activity to be inserted have to be deleted manually before another activity can be inserted there. Subsequently, the previously deleted activities have to be inserted anew.
- It is not possible to arrange <onMessage> branch in a <pick> activity next to each other as it is known from the Eclipse BPEL Designer. Instead, all <onMessage> branches are arranged beneath each other.
- The <onAlarm> branch of a <pick> activity is not specifiable in the editor. The duration or point in time of the alarm has to be specified in the manual refinement step after the transformation to BPEL.
- Deleting a participant shape that is connected to another participant via a message link causes the choreography editor to crash. The editor's data-model and the visualization are desynchronized by this action. To avoid this behavior the message links attached to the participant shape to be deleted have to be removed first.
- It is not possible to delete already inserted configuration in the BPEL4Chor groundings section. Two overcome this situation the relevant message link has to be deleted. However, this leads to another problem, which is stated below.
- If any shape in the editor's drawing area is altered, the grounding data-model is desynchronized with the shapes. Thus, the export to BPEL does not work any more with regard to this particular grounding. Only a newly created grounding can solve this issue.
- The message links contain a unnecessary `name`: label on the drawing area that cannot be removed.
- It is not possible to model communication behavior such as calling several different participants in a loop. The loops have to be added later during the manual refinement phase, although, we would normally only add non-communication process logic. The same is true for all other not yet implemented activities.

5 Transformation Steps

The following section discusses the different transformation steps necessary to obtain executable processes. First, the high-level architecture of transformation components is shown in Subsection 5.1. Subsection 5.2 discusses the transformation from an instance of the choreography editor's meta-model to the choreography language BPEL4Chor. The transformation from BPEL4Chor to abstract BPEL is shown in Subsection 5.3. Finally, Subsection 5.4 is concerned with the automatic steps that transform the abstract BPEL processes into executable ones and Section 5.5 discusses the limitations of the current implementation.

5.1 High-Level Architecture

Figure 8 shows the transformation from an architectural point of view considering the flow of documents between the components of the BPEL4Chor Designer and the dependencies between them.

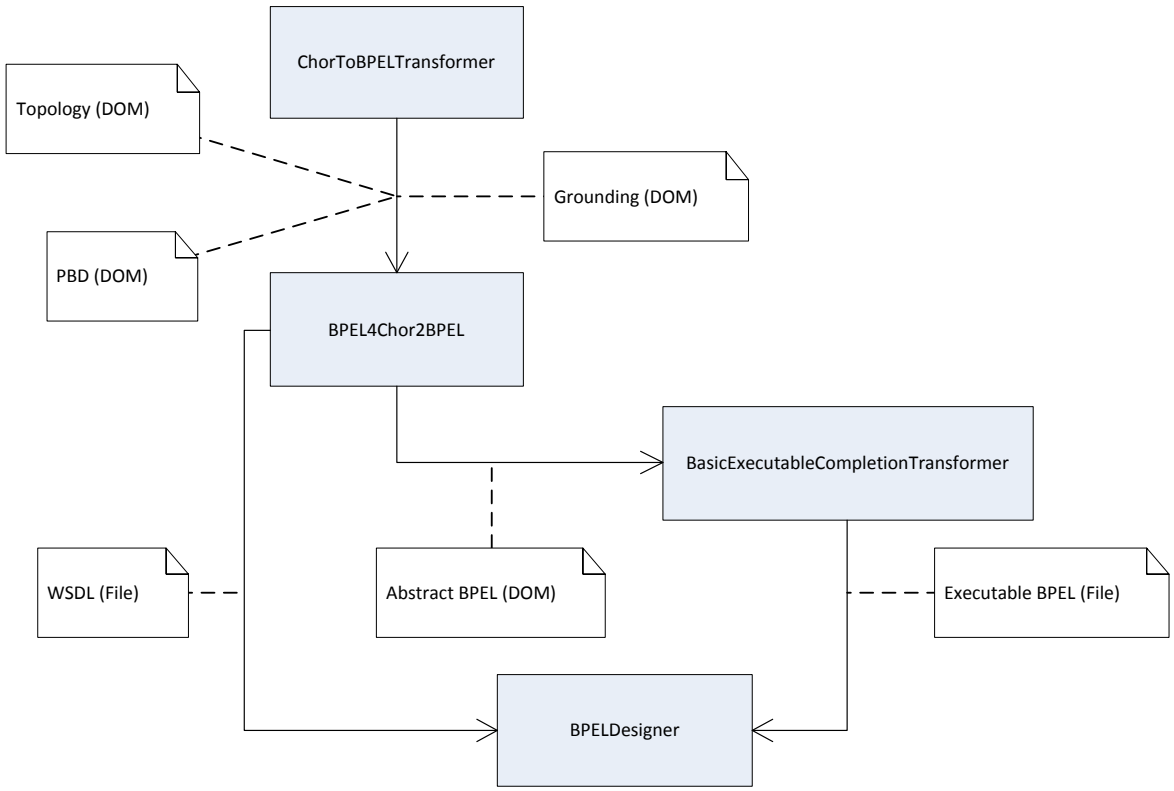

Figure 8: Document flow between the editor's components. Source: [Son13]

The components are represented by blue-colored rectangles whereas the documents are represented by rectangles with the snapped-off corners. The documents are attached

to the arrows between two components indicating on the one hand the output of a particular transformation performed by a particular component and on the other hand the input for the following component.

The component *ChorToBPEL* is responsible for the transformation from the editor's meta-model to the choreography language BPEL4Chor. The resulting documents are the Topology, the Participant Behavior Descriptions and the Grounding. All documents of this transformation are represented as in-memory Document Object Model (DOM) trees [WLHa00]. The component *BPEL4Chor2BPEL* is responsible for the transformation from BPEL4Chor to abstract BPEL processes and WSDL files. The WSDL files exist as physical files and are input for an editor capable of handling WSDL files such as our extended Eclipse BPEL Designer [SHK12]. The abstract BPEL process files represented as DOM trees are input for the component *BasicExecutableCompletionTransformer*. This component performs the *basic executable completion* described below. The result of this step are executable BPEL process files which can be opened and enriched in the Eclipse BPEL Designer.

5.2 Transformation From Chor Model to BPEL4Chor

The Chor Designer has its own meta-model described by the Eclipse Modeling Framework (EMF). Figure 9 shows the high-level architecture of the components responsible for the transformation from the Chor Designer's meta-model to BPEL4Chor and the document flow between them. The depicted components are a detailing of the component *ChorToBPELTransformer* shown in Figure 8. As in Figure 8 the components are represented by blue-colored rectangles whereas the documents are represented by rectangles with the snapped-off corners. The documents are attached to the arrows between two components indicating on the one hand the output of a particular transformation performed by a particular component and on the other hand the input for the following component.

The architecture consists of two types of components: so-called *Builders* and *Transformers*. Builders create instances of the meta-model in-memory derived the graphical notation, whereas Transformer components are responsible for transforming these instances into BPEL4Chor artifacts represented by DOM trees. The document shapes are annotated with "Model" or "DOM" according to their representation in-memory. The component *ChoreographyTransformer* is the starting point of the Transformation. It receives a Chor Model as input, which is instantiated by using the Chor Designer and without need of a separate Builder component. The ChoreographyTransformer orchestrates the document flow to the Builder components *GroundingBuilder*, *TopologyBuilder*, and *FlowBuilder* which use information from the Chor and the PDB Model, respectively, to build the Grounding Model and the Topology Model. These model instances are then passed to the corresponding Transformer component to generate the BPEL4Chor artifacts as DOM trees.

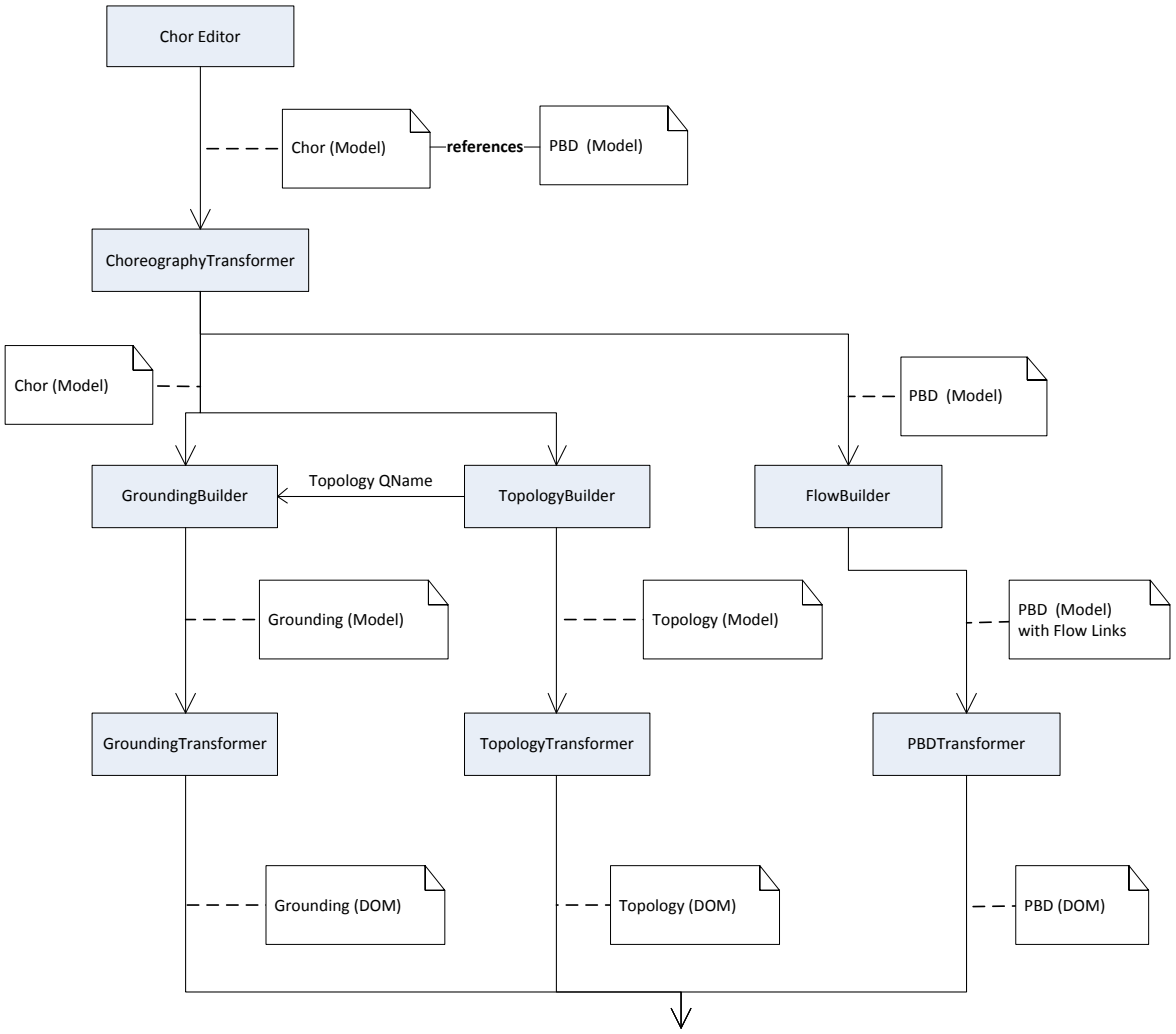

Figure 9: Document flow between Transformer and Builder components. Source: [Son13]

5.3 Transformation From BPEL4Chor To Abstract BPEL

The following subsection discusses the transformation from BPEL4Chor to abstract BPEL processes. In [Rei07] a formal, conceptual approach is presented to reach this goal. The current implementation is based on [Li10]. The transformation component is plugged-in into the Chor Designer. Figure 10 shows the input and the output the transformation needs and generates, respectively. On the left the input for the transformation is depicted: a choreography described with BPEL4Chor, and WSDL definitions. The WSDL files could also be generated during the transformation using the information modeled by the BPEL4Chor artifacts. However, this was not in the scope of the work of [Rei07]. Automatic generation of operations, port types and further WSDL properties, e.g. for correlation is possible but not yet implemented in the current

system. Message types cannot be determined automatically as there is no information present in the participant grounding which could represent the structure of messages. The current implementation generates the WSDL that are enriched with the partnerlink information during the transformation.

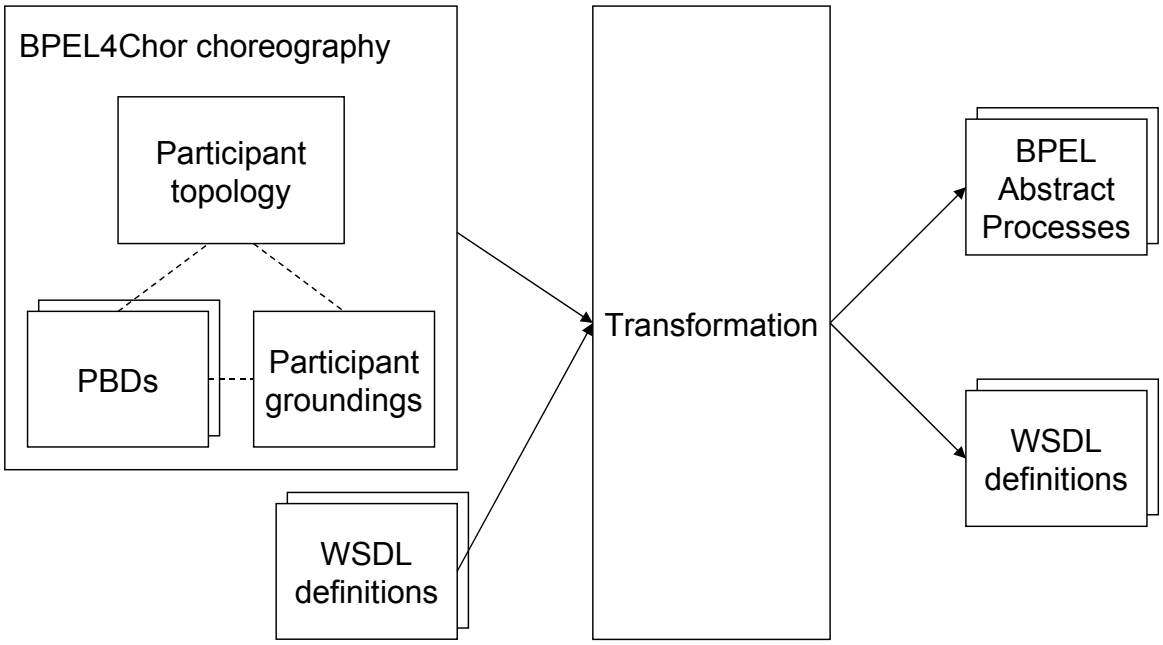

Figure 10: Input and output of the BPEL4Chor to abstract BPEL transformation. Source: [Rei07]

The output of the transformation, shown on the right side in Figure 10, are abstract BPEL processes and enriched WSDL files. However, the current implementation only considers the generation of <partnerLinkType> elements.

5.4 Basic Executable Completion

After the transformation to abstract BPEL processes the editor performs the so-called basic executable completion [OAS07]. The editor provides a component called *BasicExecutableCompletionTransformer* [Son13] that conducts the executable completion in the following way:

- The abstract namespace [9] is replaced by the executable namespace [10] of BPEL.
- The abstractProcessProfile is removed from the BPEL process.

[9] http://docs.oasis-open.org/wsbpel/2.0/process/abstract
[10] http://docs.oasis-open.org/wsbpel/2.0/process/executable

- <opaqueActivity> activities are replaced by <empty> activities.
- Creation of new variables for every opaque variable located in a <invoke>, <receive>, and <reply> activity/ element.
- The name of opaque input and output variables in <invoke>, >receive>, <reply>, and <onMessage> is set to "portType_operation_message".

Furthermore, the BPEL4Chor to abstract BPEL transformer component [Li10] inserts the following elements:

- New <PartnerLink> elements are generated for <invoke>, >receive>, <reply>, and <onMessage> activities.
- New variables are generated for every <forEach> iterating over a <ParticipantSet>.

5.5 Transformation Steps: Evaluation Findings

The following subsection discusses the issues the transformation steps have in the current state of implementation. Furthermore, possible workarounds to overcome these issues are addressed:

- The targetNamespace is not set correctly in the WSDL file that is generated for each participant in the choreography. The targetNamespace has to be corrected manually.
- The name attribute of the WSDL file is not set correctly. The attribute has to be corrected manually.
- The transformation step from BPEL4Chor to abstract BPEL includes all namespaces that are part of the choreography and not only the necessary ones for a particular participant. Furthermore, all namespaces are included twice, each time with a different prefix. The unnecessary namespaces have to be removed manually.
- The targetNamespace is not set correctly in the generated BPEL files of the choreography participants. The targetNamespace attribute of each BPEL process has to be corrected manually.
- The prefixes for the <partnerLinks> elements inserted in the basic executable completion step in are not set correctly. They have to be corrected manually.
- In the BPEL process files the imports for the WSDL files needed by the declared <partnerLinks> are not generated completely. The missing import statements have to be added manually.

- The namespaces of SOAP and XML schema are not declared automatically in the generated WSDL files. They have to be added manually.
- The BPEL4Chor2BPEL component does not generate <message>, <portType>, <binding>, and <service> elements in the WSDL file. These elements have to be created manually.
- The naming scheme of the resulting artifacts, i.e. the WSDL file and the BPEL process file for every participant of the choreography is not deterministic. If the transformation is rerun, a different number is assigned to the file names.
- A synchronous sending of a messages between two participants is modeled by two messages links: one from the <invoke> activity of the sending participant to the <receive> activity of the receiving participant and a second back from the <reply> activity of the receiving participant to the <invoke> activity of the sending participant. The transformation from BPEL4Chor to abstract BPEL generates the <partnerLinkTypes> elements twice in the resulting WSDL files. This has to be corrected manually.

6 Manual Refinement

The following section discusses the manual refinement that is necessary to add process logic to the already generated, executable BPEL processes. The modeled choreography in the BPEL4Chor Designer only captures the process logic that is necessary for the communication behavior of the participants. Therefore, the executable BPEL processes generated after the transformation steps do not yet contain any process logic besides the communication logic, e.g. for the manipulation of process variables or other necessary activities for implementing the internal behavior of the participant. Figure 11 depicts the communication activities of the Participant *GetAvailableTaxis* of the Taxi Application scenario. The process is visualized in the graphical notation of the BPEL Designer [SHK12]. The communication activities are a <receive> activity, an <invoke> activity, and a <reply> activity.

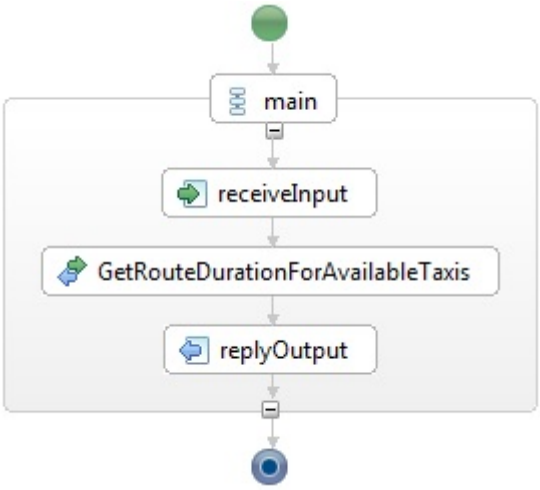

Figure 11: The executable process GetAvailableTaxis with the communication activities

Figure 12 depicts the same participant GetAvailableTaxis refined with additional activities that realize the non-communication logic of the process. <assign> activities are used to copy values between variables using XPath expressions [11]. An <if> activity is used to decide how often the <forEach> activity must iterate over a set of retrieved taxis. This are examples of process logic that is independent of the communication behavior of the process. The refinement of the BPEL process with additional activities in order to reach its desired functionality is a manual step and cannot be automated.

After the manual refinement the BPEL Processes representing the participants in a choreography can be deployed and executed on any BPEL-Engine.

[11] http://www.w3.org/TR/xpath/

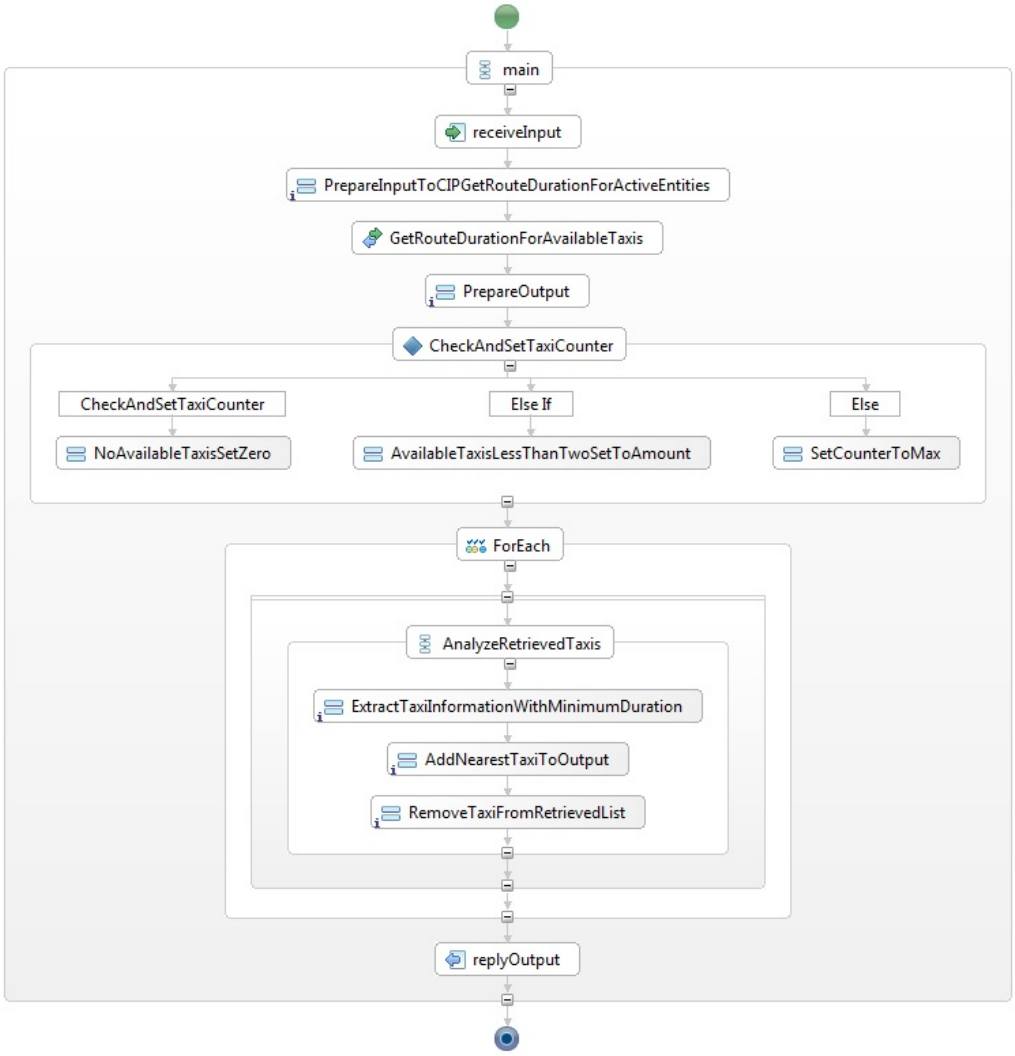

Figure 12: The executable process GetAvailableTaxis with the communication activities

7 Conclusion

This Technical Report showed in detail our approach from modeling complex systems using choreographies to making them executable. Two scenarios, the Taxi Application scenario and the FlexiBus scenario, are used to evaluate the approach and the state of the current implementation. We first modeled both scenarios in our choreography editor, the BPEL4Chor Designer, which provides a meta-model for choreographies, independent of any choreography language. The modeled choreography is then transformed into an instance of the choreography language BPEL4Chor. BPEL4Chor itself is not directly executable, so the model is transformed to abstract BPEL, made executable using basic executable completion and finally enriched with non-communication process logic.

Furthermore, we have analyzed the state of implementation of the BPEL4Chor Designer and the components responsible for the different transformation steps. Although, we can model choreographies and transform them to an executable representation, our system still lacks features such as missing communication activities. We have presented workarounds to cope with these issues. Currently, work is conducted to implement the missing basic features of the BPEL4Chor Designer and the transformation components.

References

[DKLW07] G. Decker, O. Kopp, F. Leymann, M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *Proceedings of the IEEE 2007 International Conference on Web Services (ICWS)*. 2007.

[DKLW09] G. Decker, O. Kopp, F. Leymann, M. Weske. Interacting services: from specification to execution. *Data & Knowledge Engineering*, 68(10):946–972, 2009.

[Hag11] R. Hagin. *Enabling Integration and Aggregation of Context Information into WS-BPEL Processes*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2011.

[Li10] C. Li. *An Editing Environment for BPEL4Chor Cross-Partner Scopes*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2010.

[OAS07] OASIS. *Web Services Business Process Execution Language Version 2.0 – OASIS Standard*, 2007.

[Rei07] P. Reimann. *Generating BPEL Processes from a BPEL4Chor Description*. Studienarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, 2007.

[SHK12] M. Sonntag, M. Hahn, D. Karastoyanova. Mayflower - Explorative Modeling of Scientific Workflows with BPEL. In *Proceedings of the Demo Track of the 10th International Conference on Business Process Management (BPM 2012), CEUR Workshop Proceedings, 2012*, pp. 1–5. CEUR Workshop Proceedings, 2012. URL `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2012-29&engl=0`.

[Son13] O. Sonnauer. *Modellierung von Scientific Workflows mit Choreographien*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2013.

[WKNL07] M. Wieland, O. Kopp, D. Nicklas, F. Leymann. Towards Context-aware Workflows. In *CAISE´07 Proceedings*. 2007.

[WLHa00] L. Wood, A. Le Hors, et al. Document Object Model (DOM) Level 1 Specification (Second Edition). Technical report, W3C, 2000. URL `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/`.

All links were last followed on September 30, 2013.