Data Perspective in Process Choreographies: Modeling and Execution

Andreas Meyer[1], Luise Pufahl[1], Kimon Batoulis[1], Sebastian Kruse[1], Thorben Lindhauer[1], Thomas Stoff[1], Dirk Fahland[2], and Mathias Weske[1]

[1] Hasso Plattner Institute at the University of Potsdam
{Andreas.Meyer,Luise.Pufahl,Mathias.Weske}@hpi.uni-potsdam.de
{Firstname.Lastname}@student.hpi.uni-potsdam.de
[2] Eindhoven University of Technology
d.fahland@tue.nl

Abstract. Process choreographies are part of daily business. While the correct ordering of exchanged messages can be modeled and enacted with current choreography techniques, no approach exists to describe and enact a choreography's *data perspective*. This paper describes an entirely model-driven approach for BPMN to include the data perspective while maintaining control flow aspects by utilizing a recent concept to enact data dependencies in internal processes. This work introduces few concepts that suffice to model data retrieval, data transformation, message exchange, and correlation. We present a modeling guideline to derive local process models from a given choreography; their operational semantics allows to correctly enact the entire choreography from the derived models only. We implemented our approach by extending the *camunda BPM platform* with our approach and show its feasibility by realizing all service interaction patterns using only model-based concepts.

Keywords: Process Modeling, Data Modeling, Process Choreographies, Process Enactment, BPMN, SQL

1 Introduction

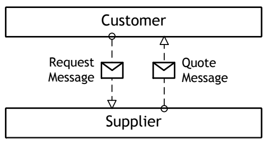


Fig. 1. Request for quote choreography.

In daily business, organizations interact with each other, for instance, concluding contracts or exchanging information. Fig. 1 describes an interaction between a customer and a supplier with respect to a request for a quote. The customer sends the *request* to a chosen supplier which internally processes it and sends the resulting *quote* as response which in turn is then handled internally by the customer. An interaction between business processes of multiple organizations via message exchange is called *process choreography* [30]. The industry standard BPMN (Business Process Modeling and Notation) [18] provides the following means and steps to model process choreographies. A *choreography diagram* describes the order of message *exchanges* between multiple participants from a global view, called *global choreography model*. The message exchanges are then refined into *send* and *receive activities* distributed over the different participants. This can be captured in *collaboration diagrams* describing how each participant's *public* process interacts with other participants [2], also called *local choreography model*. Deriving a local choreography from a global choreography is a non-trivial step; various techniques are required [7] including *locally enforcing* the order of globally specified message exchanges.

Typically, these two choreography models are used to globally agree on a contract about the messages exchanged and their order. In the request for quote example, both participants agreed that first the customer may send a request to the supplier which is then answered with a quote by the supplier. Based on the agreement, each participant has to implement its public process as a *private* process describing the executable part of this participant including the interactions with other participants as described in the choreography; this private process is called a *process orchestration* [13]. Existing approaches for deriving an orchestration for each participant from a choreography, such as the *Public-to-Private* approach [2], only cover the control-flow perspective of the contract: ensuring the correct *order* of messages. In the following, we address the correct *contents* of messages.

As messages are used to exchange data between the participants, the data perspective plays a crucial role for a successful process choreography realization. Generally, organizations store their data in local databases where other choreography participants do not have access to. These databases follow local data schemes which differ among the organizations. However, the interacting organizations want to communicate and therefore have to provide the information to be sent in a format which is understood at the receiving side. Thus, an agreed exchange message format has to be part of the global contract mentioned above. For a successful process choreography, it has to be ensured that messages to be sent are provided correctly and that received messages are processed correctly *based on the global contract*. In more detail, three challenges arise:

C1—Data heterogeneity. Interacting participants, such as our customer and supplier, each implement their own data schema for handling their private data. For sending a message to another participant, this local data has to be transformed into a message the recipient can understand. In turn, the received message has to be transformed into the local data schema to allow storing and processing by the recipient.

C2—Correlation. A participant may interact with *multiple instances* of another process at the same time. Therefore, messages arriving at the receiver side need to be correlated to the correct process instance to allow successful interaction.

C3—1:n communication. In choreographies, there may be multiple participants of the same type, e.g., multiple suppliers, a customer sends a request for quote to. Thus, individual processes need to communicate with a multitude of (external) uniform participants.

Current choreography modeling languages such as BPMN do not provide modeling concepts to solve C1-C3. Instead, each participant manually implements message creation and processing for their private process, which is error-prone, hard to maintain, and easily results in incompatibilities to other participants in the choreography.

In this paper, we describe a model-driven approach to include the data perspective within the process choreography modeling while maintaining existing control flow aspects to realize process choreographies. We utilize the industry standard BPMN and extend its choreography modeling by few but essential concepts for the data perspective. We describe a modeling guideline that shows how to utilize the new concepts for deriving the data perspective of a private orchestration model that is consistent to a public choreography model (the contract). We introduce operational semantics for the new modeling concepts which makes the orchestration models executable, and thus allows running the entire choreography purely model-based.

The remainder of this paper is structured as follows. Section 2 discusses the requirements derived from above challenges. Subsequently, we explain the modeling guideline in Section 3 followed by the operational semantics allowing to execute the modeled choreographies directly from process model information in Section 4. In Section 5, we discuss our implementation and its feasibility for implementing all service interaction patterns purely model-based [3]. Section 6 is devoted to related work and Section 7 concludes the paper.

2 Requirements

The challenges C1-C3 described above give rise to specific requirements for integrating the data perspective in process choreography modeling and execution. We discuss these requirements and their possible realization in the following.

R1—Content of message. Messages contain data of different types exchanged between participants. The involved participants have to commonly agree on the types of data and their format they want to exchange.

R2—Local storage. The participants create and process data used for communication with other participants in their private processes. This needs to be stored and made available in their local databases.

R3—Message provision. As the data provided in a message is local to the sender, the data must be adapted to the agreed format such that the recipient can interpret the message content.

R4—Message routing. Multiple parties may wait for a message at a certain point in time. This requires to route the message to the correct recipient.

R5—Message correlation. After being received by a participant, the message needs to be correlated to the activity instance which is capable to process the message content.

R6—Message processing. Activities receiving messages have to extract data from the message and to transform it into the local data format usable within their processes.

Requirements R1, R2, R3, and R6 are basic features to realize C1; R4 and R5 originate in C3; and R5 also addresses C2.

Languages such as WSDL [25] use data modeling to specify message formats; we adopt these ideas to address R1. Requirements R2, R3, and R6 concern the processing of data in an orchestration. The approach in [14] allows to model and enact data dependencies in BPMN processes for create, read, update, and delete operations on multiple data objects – even in case of complex object relationships. For this, annotations on BPMN data objects are automatically transformed into SQL queries (R2). Further, data querying languages such as XQuery [28] allow to implement data transformations between a message and a local data model. In the following, we combine these approaches to specify message extraction (R3) and message storage (R6) in a purely model-based fashion. Languages such as BPEL [17] and BPMN [18] correlate a message to a process instances based on key attributes in the message; we adopt this idea to address R5. The next sections describe how to model process choreographies including the data perspective so that data stored locally at the sender's side can be transmitted and stored in the receiver's local data model consistent with the global contract.

Requirement R4, the actual transmission of messages from sender to receiver, is abstracted from in choreography and process models and also not discussed in this paper. One can use standard technologies such as middleware or web services to realize the communication between the process engines of participants.

3 Modeling Guideline

This section introduces a few concepts that allow implementing the data perspective of a process choreography in an entirely model-based approach. We present these concepts embedded in a modeling guideline for devising private orchestration models consistent to a public choreography model; Section 4 presents the execution semantics for our choreography models.

Fig. 2 illustrates our modeling guideline which has a *global level*, where the public contract is defined, and a *local level*, where the local process implementations can be found. We assume that the choreography partners have already specified a collaboration diagram that shows how each participant's public process interacts with the other participants and ensures local enforceability of control-flow [2]; see Fig. 2 (top). To support data exchange between participants, we propose that this public contract is supplemented with a global data model in which the partners specify the business objects to be exchanged; see Fig. 2 (top middle). Next, we follow and extend the P2P approach [2] to move from the global to the local level: each participant separately defines a local data model and a schema mapping between their local and the global data model and implements the private process conforming to their public process in the global collaboration diagram. Next, we describe the details of the global contract followed by the local level both along our modeling guideline.

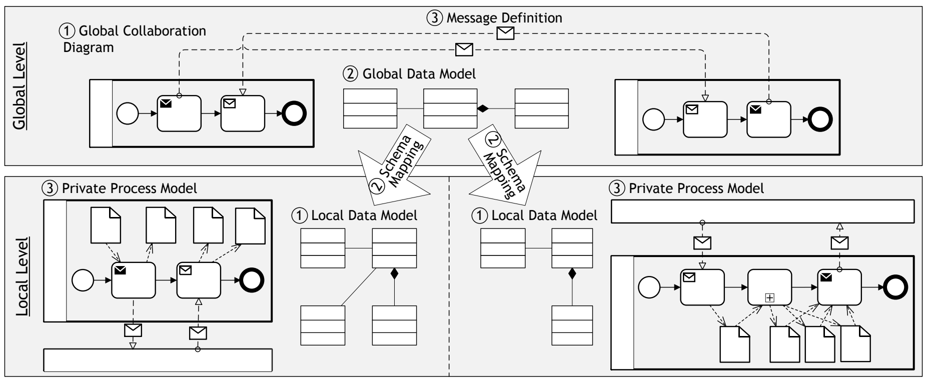


Fig. 2. Modeling guideline.

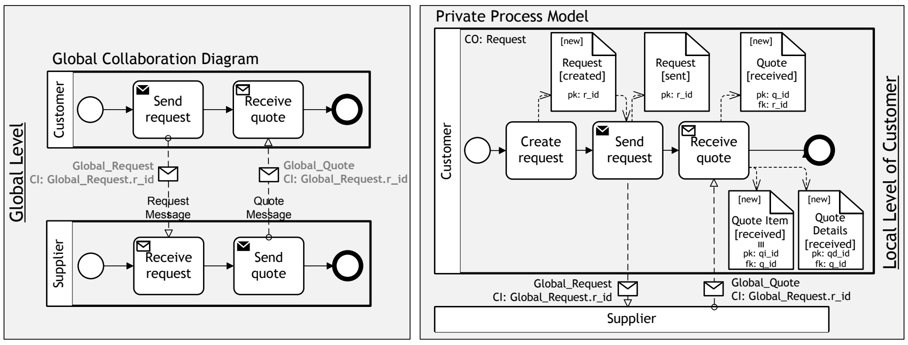


Fig. 3. Global choreography model and local process model of the customer.

On the global level, all choreography parties together define the following artifacts:

Global collaboration diagram: The global collaboration diagram describes the control flow layer of the choreography, i.e., it describes which messages are exchanged in which order on a conceptual level. Exemplary, the left part of Fig. 3 shows the collaboration diagram of the *Request for quote* choreography sketched in the introduction. It includes public processes with all necessary send and receive tasks for each participant, the customer and the supplier.

Global data model: Messages are used to exchange data. In choreography modeling languages such as WS-CDL [10] or BPEL4Chor [6], the data carried by a message is described technically by attribute names and data types for each message individually [25]. Instead, we propose that the interacting parties first agree on data objects they want to share and document this in a global data model, for instance using XSD (`http://www.w3.org/standards/xml/schema`). In our example, customer and supplier have agreed on three data objects, *Global_Request*, *Global_Quote*, and *Global_Articles*, as shown in the upper part of Fig.5. Each object has a unique identifier attribute (e.g., *r_id* for *Global_Request*) and some have a foreign key attribute (e.g., *r_id* for *Global_Quote*) to express relationships.

Message Definition: Then, message types are specified by *referring* to business objects defined in the global data model. We assume that each message carries exactly one global data object; nested objects allow placing complex data object hierarchies within one message. Further, we adopt key-based correlation [17, 18] for messages: each message contains a set of key/value pairs that allow identifying the correct process instance on the receiver side; each key is an attribute of some data object in the global data model. For example, *Request Message* of Fig. 3 (left) refers to the *Global_Request* object and *Quote Message* refers to *Global_Quote* which has multiple *Global_Article* objects. A *Quote Message* will contain a *Global_Quote* object and all its *Global_Article* objects. Both messages use attribute *r_id* of *Global_Request* as correlation key.

Altogether, a message can be declared as a tuple $m = (name, CI, d)$, where $name$ is the message type, the correlation information $CI \subseteq K \times V$ is a set of key/value pairs, and d is the actual data object in the message. To model this tuple, BPMN must be extended as shown in the UML class of Fig. 4. Originally, each

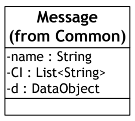


Fig. 4. Message class.

message contains a string identifying its name, i.e., the message type. We add correlation information as a list of strings, each denoting one key/value pair, and the payload as a data object.

Then, each participants locally creates the following artifacts, based on the global contract:

Local Data Model: Each participant defines a local data model which describes the classes of data objects handled by the private process. For example, the local data model of the *Customer* has four classes *Request*, *Quote*, *Quote Details*, and *Quote Item*; see Fig. 5 (bottom). We propose to also use the local data model to design the schema for the database where the objects are stored and accessed during the process execution. There are some requirements to the local data model wrt. the global data model as described next.

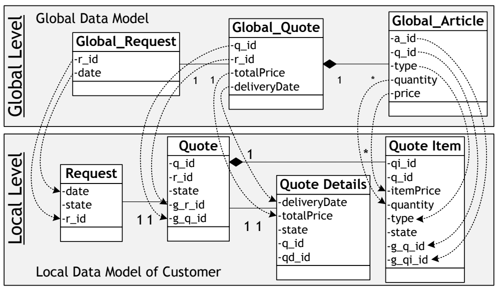


Fig. 5. Schema mapping for customer.

Schema Mapping: A schema mapping defines how attributes of local classes map to attributes of global classes, and allows to automate a data transformation between global objects contained in messages and local data objects. For this paper, we consider a simple attribute-to-attribute schema mapping which injectively maps each attribute of a global object to an attribute of a local object as shown in Fig. 5. Note that the attributes of object *Global_Quote* are distributed over objects *Quote* and *Quote Details*. The local implementation can hide private data in a local attribute by not mapping it to a global attribute (the mapping is not bijective), e.g., the *state* attributes of each local class. Local data model and schema mapping must ensure that primary and foreign keys are managed locally to avoid data inconsistency: when a local object can be created from a received global object, key attributes of the global object must map to non-key attributes of the local objects. For example, the local *Quote* shall be created from a *Global_Quote* object, thus *Quote* gets the attributes *g_q_id* and *g_r_id* to store the primary key *q_id* and the foreign key *r_id* of *Global_Quote* for local use. Typically, these keys are used for correlation.

Executable private process: Based on the global collaboration diagram, each participant designs their private process by enriching their public process with activities that are not publicly visible. In addition, each process (and each subprocess) gets assigned a *case object*; instantiating the process also creates a new instance of this case object that uses as primary key value the process instance id [14]. Fig. 3 (right) shows the private process model of the customer. First, activity *Create request* creates and prepares a new instance of the case object *Request* (see "CO" in the top left corner of the process). The schema mapping defines which local data objects are required to derive the payload d and the correlation information CI for a message to be sent; this is included in the process model by associating the required data objects as input to the send task. In our example in Fig.3, activity *Send request* creates a *Request Message* containing a *Global_Request*. The corresponding local *Request* object is associated to *Send request* as input. Correspond-

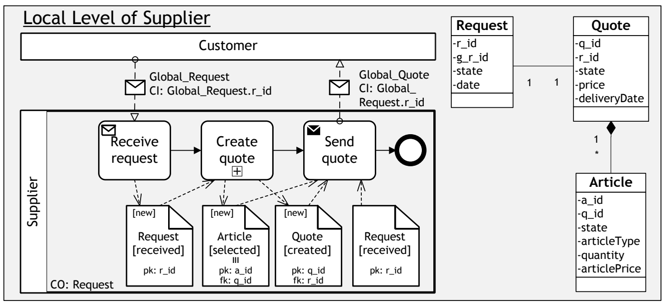


Fig. 6. Private process model and local data model of the supplier.

ingly, we associate the local data objects into which the payload of a received message can be transformed as output data objects of a receive task. The last activity modeled in the customer process receives the *Quote Message*. The payload of this message is transformed into data objects *Quote*, *Quote Details*, and the multi-instance data object *Quote Item* all being associated as output to the receive task. The process designer has to specify whether the receive task creates new or updates existing data objects. We use the data annotations described in [14] to express operations and dependencies of local objects. In the given example, the message payload is used to create new data objects only as indicated by the identifier *new* in the upper part of each object. Local data schema, schema mapping, and private process *together* define the local choreography of the participant.

Fig.6 shows the private process model and the local data model of the second participant – the *Supplier*. Here, each attribute of a local class directly maps to a corresponding attribute with an equivalent name in the corresponding global class. For instance, attribute *price* of class *Global_Article* maps to to attribute *articlePrice* of class *Article*, attribute *r_id* of class *Global_Request* maps to attribute *g_r_id* of class *Request*, and so on. The private process has three activities: After receiving the *Global_Request*, which is stored as *Request* object in state *received*, the supplier processes the request and creates the *Quote*. Sending the *Global_Quote* message requires data objects *Quote* and *Article* to set the payload and *Request* to set the correlation identifier *Global_Request.r_id*.

This modeling guideline proposes a logical order in which the artifacts should be created based on dependencies between them. However, situations may arise where a different order (or iterations) are required. In any case, by refining the public process into a private one and by defining local data model and schema mapping as described, a process modeler always obtains a local choreography that is consistent with the global contract. In the next section, we show how to make the local choreography executable, thus achieving a correct implementation by design.

4 Executing Data-annotated Process Choreographies

In the previous section, we showed how to model executable process choreographies with respect to the data layer. This section introduces the execution semantics to automatically generate as well as correlate messages and to persist them using the modeling concepts introduced in the previous section. First, we start with an overview based on our example before we dive into details in Sections 4.2 to 4.4.

4.1 Overview of Choreography Execution

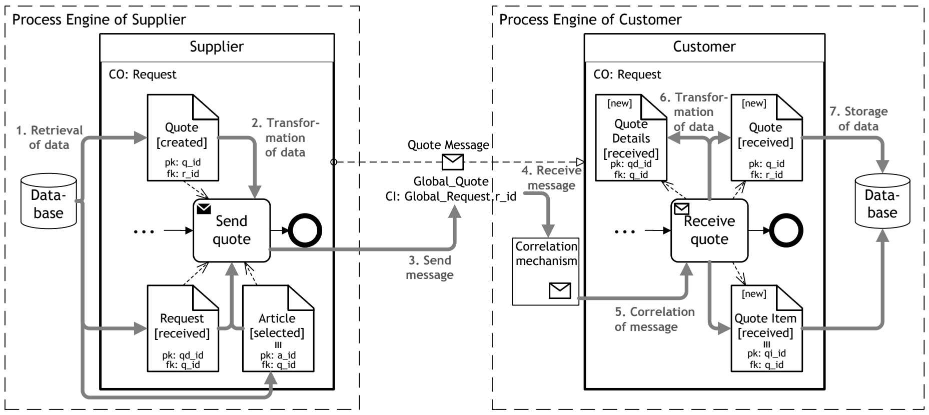
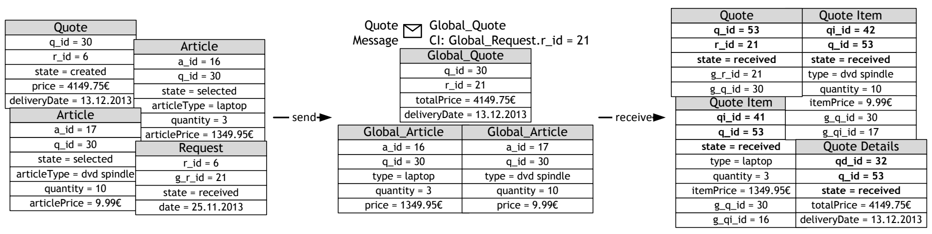


Fig. 7. Approach overview.

Fig. 7 shows the implementation of the second interaction between a supplier and a customer in which the supplier sends a quote to the customer. It comprises seven steps numbered accordingly in the figure and satisfying the requirements raised in Section 2: (1) The required data objects are retrieved from the supplier's database (satisfying requirement R2) and (2) transformed to the message (satisfying R1 & R3), which is (3) sent from the supplier and (4) received at the customer's side (satisfying R4). The received message is then (5) correlated to the corresponding activity instance (satisfying R5), where the message (6) gets transformed into data objects (satisfying R1 & R6) which are then (7) stored in the customer's database (satisfying R2 again).

The send task labeled *Send quote* creates and sends the message. As described in [14], the input data objects specify the data objects and their states required to start activity execution. In this paper, the input data objects to send tasks additionally describe the local data required to create the message to be sent. Therefore, in step 1, they are retrieved from the local database before step 2 transforms this information into the corresponding message based on the given schema mapping. Here, the objects *Quote* and *Article* (as multi instance data object) are utilized to create the message's payload *Global_Quote* which is a hierarchical object consisting of a number of *Global_Article*s (see global data model in Fig.5). Further, the specified correlation identifier *Global_Request.r_id* is added to message based on the input data object *Request*. This is needed by the customer to correlate the message to its correct scope instance (process or sub-process instance). After preparing the message, the actual sending to the recipient is executed by the send task (step 3). The execution of the send and getting the message to the correct recipient is done by inter-engine communication in lower layers, e.g., by web services or middleware, which is not discussed in this paper.

Analogously, the retrieval of the message at the recipients side, here the customer, is managed by the same underlying layer (step 4). Next, the message needs to be correlated



(a) Supplier database. (b) Message. (c) Customer database.

Fig. 8. Representation of on instance from the message flow shown in Fig. 7 where each presented object refers to one column in the corresponding database table named as the respecting object (cf. [14]).

to the corresponding scope instance in the correct process model (step 5), where it is assigned to the correct instance of the respective receive task. In our example, the response of the supplier is handled in the *Receive quote* activity. Again, as described in [14], the output data objects of an activity specify the data objects and their states expected to exist after activity execution. Therefore, step 6 transforms the received message's payload into the specified data objects following the given schema mapping. Here, data objects *Quote*, *Quote Details* and *Quote Item* (as multi instance data object) are derived from message's payload *Global_Quote*. Finally, execution of the receive task stores the derived data objects in the customer's local database (step 7). Because of following the given schema mapping for object derivation, the persistence step succeeds.

These seven steps can be summarized in four phases from which we discuss three in detail in the upcoming sections while the fourth phase utilizes solutions already existing as discussed above. These phases are (i) preparing to send the message (see Section 4.2), (ii) handling the received message (see Section 4.3), (iii) correlating the message to the correct scope instance of the receiver (see Section 4.4), and (iv) routing a message from the sender to each receiver.

4.2 Send

The preparation of a message to be sent consists of the retrieval of required data objects from the database (step 1) and their transformation accordingly to a given schema mapping (step 2). This section presents both steps in detail. During execution, each activity appears in various activity states depending on the current status of execution. With respect to the BPMN specification [18], an activity may be, among others, in states *inactive*, *ready*, *active*, *completing*, or *completed* in this order. Initially, since initialization of the respecting process instance, the send task *Send quote* is in state *inactive* and waits for its enablement. With arrival of the control flow, the send task changes into the *ready* state. There, data dependencies, i.e., the availability of the specified input data objects, are checked (cf. [14]).

In our example, the data objects *Request* and *Quote* as well as all *Article*s have to be available in their annotated states to activate the *Send quote* activity. Those data objects are necessary to provide the payload and correlation identifier of the message to be sent. The object *Quote* in Fig. 7 is expected in state *created* and has a primary key *q_id* and a foreign key *r_id* pointing to the *Request*. Based thereon, the following guard for checking the availability of object *Quote* is created: `(SELECT COUNT (q_id) FROM Quote WHERE r_id = $ID AND state = created)` $\geq$ `1`. This SQL query returns the number of *Quote* entries in the local database that are in state 'created' and related via foreign key *r_id* to the case object instance *Request* of the current process instance (identified by `$ID`); there has to be at least one [14]. In [14], the case object is introduced as the object which drives the execution of a process and all other data objects are related to this one. The case object relates to the process instance by its primary key.

In Fig. 8a, an extract of the supplier database is illustrated with each table representing one entry in the *Request*, *Quote*, and *Article* tables respectively. Assuming that the currently running process instance has the identifier `$ID = 6` and no other entry in the *Quote* table refers to this identifier, above SQL statement returns value 1 indicating availability of the required data object. Analogously, the other input data objects are checked for availability.

If all data dependencies are fulfilled, the message to be sent gets prepared by retrieving the required data objects from the local database followed by the transformation step building the actual message. For retrieval, we adapt the SQL statements from [15] by changing `SELECT COUNT` to `SELECT *` and removing the quantity expectation $\geq$ `1`. Thus, object *Quote* is retrieved by statement `SELECT * FROM Quote WHERE r_id = $ID AND state = created`. Each specified data object is retrieved analogously and then transformed into its global representation following the given schema mapping, e.g., object *Quote* is transformed into object *Global_Quote*. In our example, we utilize the schema mapping explained in Section 3. As object *Global_Quote* is a hierarchical data object, also all related *Article* objects are transformed into corresponding *Global_Article*s. Here, there are two such objects indicated by the foreign key `q_id = 30`. After transformation, all three global objects are added the payload of the message to be sent by the corresponding sent task. The correlation information *Global_Request.r_id* = 21 is taken from attribute *g_r_id* of the local object *Request* as specified in the schema mapping as well. After completing the message creation and adding the correlation identifier, the state of the send task changes from *ready* to *active* indicating processing of the activity. The work performed by a send task is to initiate the actual send of the prepared message shown in Fig. 8b.

4.3 Receive

After a received message has been correlated to the corresponding instance (see Section 4.4) it can be processed by basically reversing the two steps for sending a message. First, the objects in the message are transformed into the local data model (step 6 in Fig. 7) followed by storing them in the local database (step 7). A receive task can only receive a message while it is in state *active*. Succeeding with the message receipt triggers the change of activity state *active* to state *completing* for the receive task. In this activity state, steps 6 and 7 from Fig. 7 take place.

The transformation, again, follows the given schema mapping; see Fig. 5 for details. Thus, *Global_Quote* and its hierarchical depending *Global_Article*s, the payload of the message in our example, is mapped to the data objects *Quote*, *Quote Details*, and *Quote Item* (as multi instance data object) by filling the respecting attributes. The data objects to be used are specified by the output data objects to the receive task (likewise the input data objects for the send task). Thereby, primary keys and foreign keys of newly created objects as well as their states are not yet set (consider them empty for now). For instance, the local object *Quote* gets attributes *r_id* and *g_q_id* set to *30* and *21* respectively while attributes *state* and *q_id* are still undefined. This information is added to the data objects while persistence mechanisms take place, i.e., in step 7. Step 7 is only triggered upon completion of the transformation step comprising that all data objects were created or updated successfully with message information. Thereby, new information overwrites probably existing one.

Setting the missing information (primary and foreign keys as well as states) and storing the data objects in the local database utilizes the concepts from [14]. Thereby we differentiate between creating a new table entry (*INSERT*) and updating an existing one (*UPDATE*):

INSERT: Data objects, which representation in the process model contains a *[new]* annotation in the upper left corner, are supposed to be newly created. In the given example, all objects being output to the receive task shall be created. In [15], patterns and corresponding SQL queries are provided to insert a new data object into the local database. Here, only the information given in the data object representation, i.e., primary key, foreign key, and state, are regarded, e.g., , `INSERT INTO Quote (q_id, r_id, state) VALUES (DEFAULT, $ID, received)` for the *Quote* object with `$ID = 21` being the current process instance id. In this paper, we extend these queries by adding the information extracted from the received message based on the local data model such that the complete query for the *Quote* object looks as follows: `INSERT INTO Quote (q_id, r_id, state, g_q_id) VALUES (DEFAULT, $ID, received, 30).`

Analogously, the *INSERT*-statements are created for the other output data objects. For instance, object *Quote Details* gets the remaining information, *totalPrice* and *deliveryDate*, from the *Global_Quote* object. Fig. 8c visualizes the customer database after inserting all data object extracted from the received message. Please note, the order of storing the data objects into the local database is important as, for instance, one object may relate to another object via foreign key relationship. In this case, the second object must have been stored first to ensure that the key value is known to be added for the first object. In our example, object *Quote Details* has a foreign key relationship to object *Quote* such that is must be inserted after object *Quote*.

We assume that the foreign key relationships between the output data objects of a receive task form a directed acyclic graph over the respecting data objects. It implies that these relations have a partial order and that it is possible to insert referenced data objects before the ones that reference them. Then, this directed acyclic graph describes the insertion order from leaf to root node.so that first the *Quote* object and then the *Quote Details* and the *Quote Item* objects are inserted. When the graph is completely traversed, the receive task has finally reached the *completed* state.

UPDATE: The representation of an output data object to be updated has no additional annotation in the process model. Again, we utilize the SQL queries provided for various update patterns from [14] and extend them to update the information retrieved in the message as well. For each such data object, the local data model specifies the attributes to be updated, i.e., overwritten with the values provided in the received message. Assuming that object *Quote* is updated instead of inserted, then the following UPDATE statement would apply: `UPDATE Quote SET state = received, g_q_id = 30 WHERE r_id = $ID;`. As primary key and foreign key cannot be updated based on message information (see above), they are note included in the created SQL queries. In the given example, a specific update ordering is not necessary as the keys do not change. However, there do exist patterns where the foreign key is set by such query. In these cases, the ordering gets important and is enforced as discussed above for the *INSERT*-statements. In contrast to the *INSERT*-statements, updates cannot be applied to data collections, i.e., multi instance data objects, because in is not clear which information would belong to which object of the collection as they are not distinct in the process instance. Assigning explicit ids would solve this issue but is out of scope for this paper.

We also allow combinations of inserts and updates for one receive task, if the limitations of both operations are considered, i.e., the insertion order for the newly created data objects and no update on data collections.

4.4 Correlation

Before a message can be handled, it has to be assigned to its *receiving instance* which is also known as *correlation handling*. The standard approach is *key-based* correlation [17, 18], where some attributes of the data model are designed as *correlation keys*. An incoming message is correlated to a process instance when both store the same value for all correlation keys *in the message*; any two instances must be distinct on their correlation values. We first consider the case when an instance has all keys *initialized* already and then discuss *how* to initialize a key.

All keys initialized. Our approach refines key-based correlation by making correlation keys part of the global data model. On the one hand, each message $m = (name, CI, d)$ explicitly defines a number of correlations keys CI, where each key $d_2.a \in CI$ points to some attribute a of some global data object d_2 (not necessarily d). For example, the message of Fig. 8b has the correlation key *Global_Request.r_id* while its payload is of type *Global_Quote* (as specified in Fig. 3). On the other hand, each participant defines a local data model, where each correlation key attribute $d_2.a$ of m is mapped to a local attribute $f(d_2.a) = d_2'.b$ of some local data object d_2'. Each process instance `$ID` has its own case object instance and related object instances; message m correlates to `$ID` when the value of each $d_2.a \in CI$ matches the value of the corresponding $f(d_2.a)$ of some data object related to instance `$ID`. For example, the *Customer* maps *Global_Request.r_id* to *Request.r_id* (see Fig. 5). Thus, the message of Fig. 8b can be correlated to a process instance where the case object has *Request.r_id* $= 21$.

Formally, the correlation information of a message $m = (name, CI, d)$ is a set $CI = \{(k_1, v_1), \ldots, (k_n, v_n)\}$ of key/value pairs, where each key $k_i = d_i.a_i$ is an attribute a_i of a global data object d_i. A participant's schema mapping f maps each key to a local attribute $f(d_i.a_i) = d_i'.a_i'$. The *value* of the correlation attribute $d_i'.a_i'$ can be extracted with respect to the case object c of the receiving instance `$ID` as follows. Object d_i' relates to c via foreign key relations. Thus, we can build an SQL query joining the tables that store d_i' and c, select only the entries where the primary key of c equals

$\texttt{\$ID}$, and finally extract the value of attribute $d'_i.a'_i$; see [15]. Let $e(d'_i.a'_i, c, \texttt{\$ID})$ denote the results of this query. By ensuring that in the local data model the relations from c to d'_i are only 1:1, the extracted value $e(d'_i.a'_i, c, \texttt{\$ID}) = v$ is uniquely defined. Now, m *correlates* to an instance $\texttt{\$ID}$ of a process with case object c iff for each $(k_i, v_i) \in CI$ holds $e(f(k_i), c, \$ID) = v_i$. This definition can be refined to not only consider the case object of the entire process, but also the case object and instance id of the scope that encloses the *active* task that can receive m.

Initializing correlation keys. When sending a message m, then its correlation keys are automatically *initialized* by extracting for each global correlation attribute k_i the corresponding value $e(f(k_i), c, \$ID) = v_i$ from the sender's local data model. Technically, this can be done in the same way as extracting the payload of m, see Section 4.2. From this point on, all process instances receiving a message with correlation key k_i have to agree on the value v_i. The only exception is when $e(f(k_i), c, \$ID) = \perp$ is still *undefined* at the receiving instance. By initializing the local attribute $f(k_i)$ to value v_i, we can make $\texttt{\$ID}$ a matching instance for m. Thus, we generalize the above condition: m *correlates* to an instance $\texttt{\$ID}$ of a process with case object c iff for each $(k_i, v_i) \in CI$ holds if $e(f(k_i), c, \$ID) \neq \perp$ then $e(f(k_i), c, \$ID) = v_i$. When receiving m, the local key attribute $f(k_i)$ can be initialized for $\texttt{\$ID}$ to value v_i by generating an SQL update statement as discussed in Section 4.3.

5 Evaluation

We implemented our approach by extending the *camunda Modeler*, a modeling tool supporting BPMN, and the *camunda BPM Platform*, a process engine for BPMN process models. The modeling tool was extended with the annotations for messages and data objects described in Section 3; message types of the global data model are specified in XSD and a simple editor allows to create an attribute-wise schema mapping from the global to the local data model. Once a private choreography model has been completed, the user can automatically generate XQuery expressions (http://www.w3.org/TR/xquery/) at the send and receive tasks to transform between local and global data model (Section 4). The engine was extended with a messaging endpoint for sending and receiving messages in XML format to correlate messages, to read and write local data objects by generating SQL queries from process models, and to process messages as described in Section 4. As the concepts in this paper, also our implementation does not address R4 (message routing); in particular if the receiving task is not in state *active* to receive the incoming message, the message will be discarded. Making the process layer compatible with error handling of the message transport layer is beyond the scope of this paper.

To demonstrate the feasibility of our approach, we implemented the *service interaction patterns* [3] which capture basic forms of message-based interaction. In the following, we briefly describe each pattern and how it can be realized using the proposed approach. Thereby, we reuse the pattern classification into single-transmission bilateral, single-transmission multilateral, multi-transmission, and routing interaction patterns to structure this section.

5.1 Single-Transmission Bilateral Interaction Patterns

Patterns in this category describe the interaction of two participants A and B that each send/receive one message.

P1 Send and P2 Receive. Participant A sends a message which has to be received by participant B. The challenges are to generate and send a message, to correlate a message based on an initialized or uninitialized key, and to process a received message.

Fig. 9 shows the pattern for A sending a message to B. Thereby the local object *RequestA* is transformed into the global object *Request_P1* and the correlation key *Request_P1.request_id* is set from the primary key *requestID* of *RequestA*.

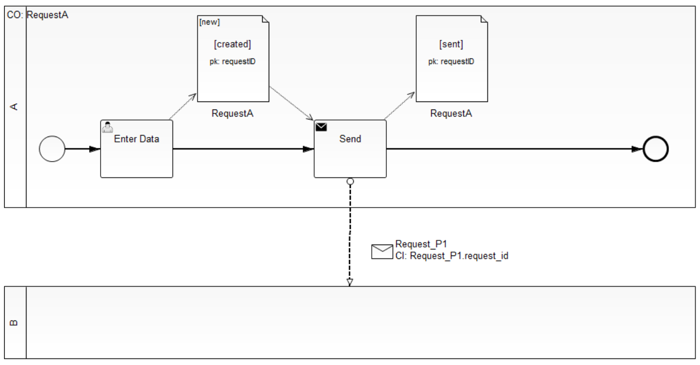


Fig. 9. Pattern P1: send.

Fig. 10 shows the pattern for B receiving the message from A. Thereby the message being received creates a new process instance; the global object *Request_P1* is mapped to the local object *RequestB* with its own primary key; the correlation information in *Request_P1.request_id* is mapped to an attribute *RequestB.requestID_fromA*. This correlation key is initialized upon receipt.

P3 Send/Receive. Participant A sends a request to B and receives a response. The challenge is to correlate the response message to the instance of A that sent the message.

Fig. 11 shows the pattern for A sending a message to B and then awaiting the corresponding response. Correlation of the response to the request is achieved by including in the response message the correlation information *Request_P3.request_id* that was sent to B in the request message. This way, only responses that match the request will be received. As the correlation key *Request_P3.request_id* is initialized from the primary key *RequestA.requestId*, the received response can be transformed into the local object *ResponseA* that has *requestId* as foreign key pointing to *RequestA*.

Fig. 12 shows the pattern for B receiving the message from A and producing a response. Thereby the message being received creates a new process instance; the global object *Request_P3* is mapped to the local object *RequestB* with its own primary key; the correlation information in *Request_P3.request_id* is mapped to an attribute *RequestB.requestID_fromA*. This correlation key is initialized upon receipt and used again when generating the response message *Response_P3* from the local object *ResponseB*.

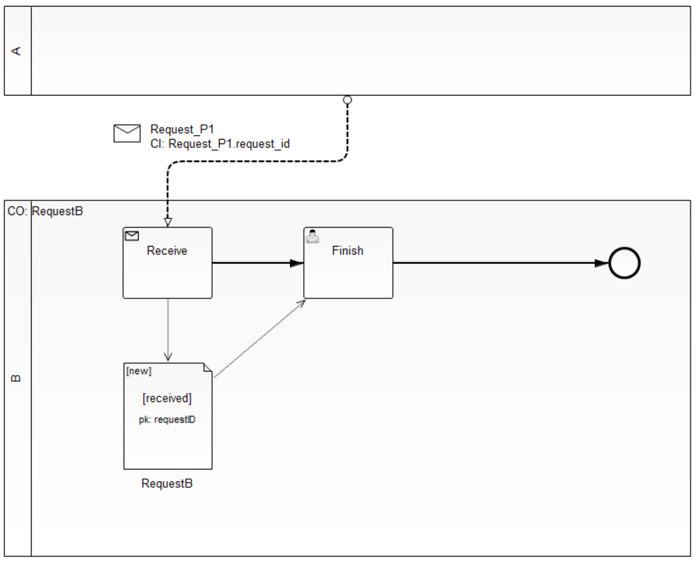


Fig. 10. Pattern P2: receive.

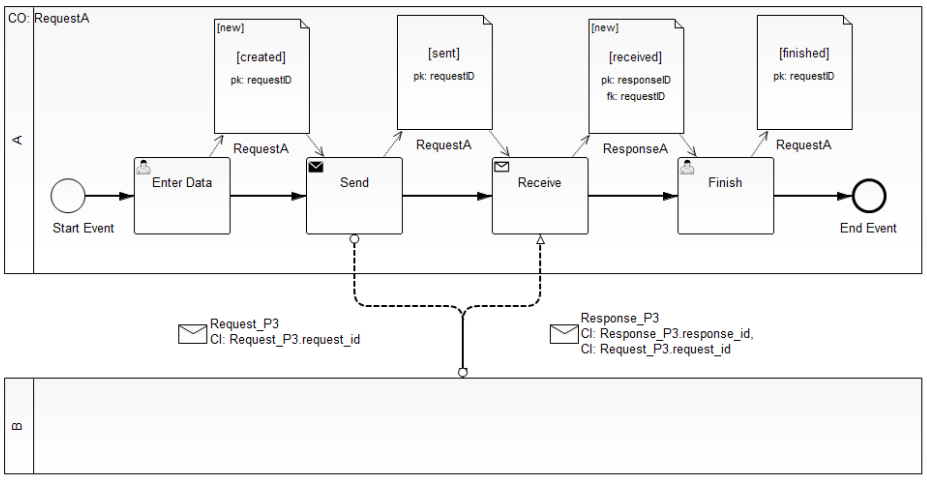


Fig. 11. Pattern P3: send/receive (Participant A).

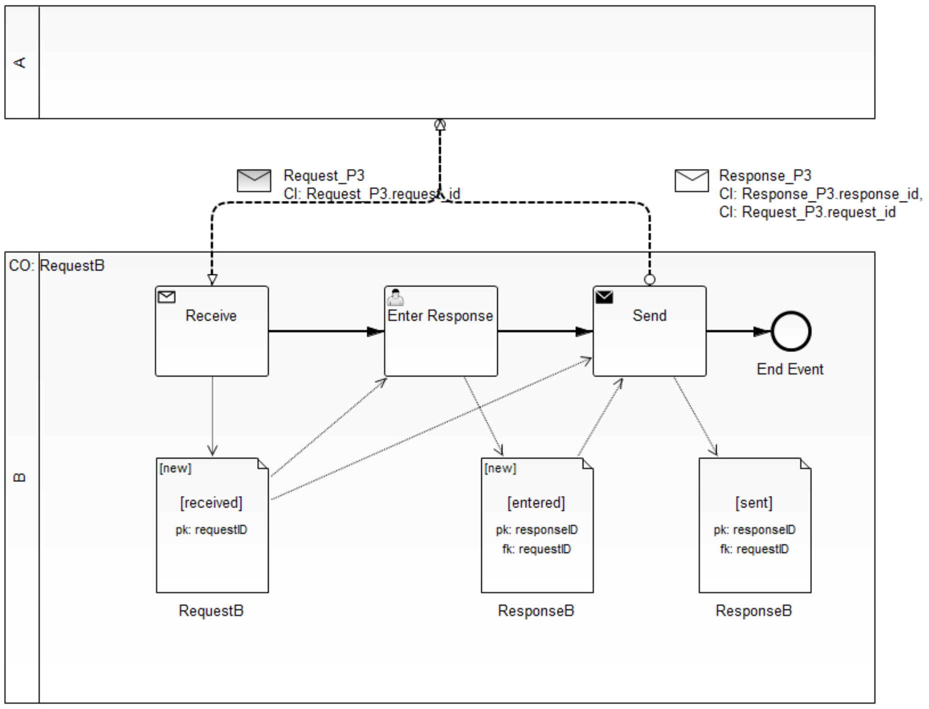


Fig. 12. Pattern P3: send/receive (Participant B).

5.2 Single-Transmission Multilateral Interaction Patterns

Patterns in this category describe the interaction of participants A with multiple participants B1, B2, ..., each sending/receiving one message.

P4 Racing Incoming Messages. Participant B expects one or more messages from participants A1, A2, ..., and will consume the first arriving message; messages arriving later may be discarded or queued. Optionally, a timeout is allowed in case no message arrives. The challenge is to ensure that B consumes exactly one message or the timeout occurs.

The behavior of A1, A2, ... is described in Fig. 9 (P1 send). Fig. 13 realizes the pattern using the event-based gateway that has two subsequent intermediate message events. Each awaits a message from a different participant (A or C in this case). Whichever message arrives first will be consumed and the corresponding path will be taken. The timer event after the event-based gateway implements the timeout mechanism. The model in Fig. 13 only implements the correlation handling: depending on which message is received first, the corresponding correlation property is set. The model does not implement transformation of the message into local data as the BPMN standard does not allows data handling at events [18].

Fig. 14 shows an alternative realization where the message events are replaced by receive tasks. By the BPMN standard [18], the event-based gateway will follow the sequence flow of the receive task which first has a message to consume. This also allows

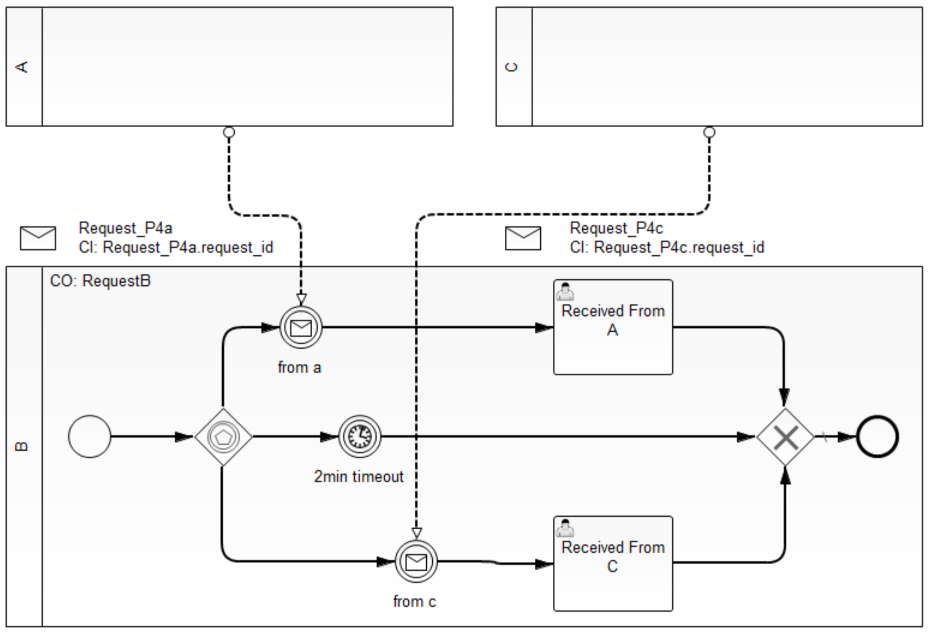


Fig. 13. Pattern P4: racing incoming messages (Participant B), with message events.

to transform data. However, by the time of our research the *camunda BPM platform* did not support receive tasks after and event-based gateway and thus this pattern could not be executed.

P5 One-to-Many Send, P7 One-to-Many Send/Receive. In P5, Participant A sends out a request to multiple participants B1, B2, ..., so that each participant receives one request. In P7, each participant B1, B2, ... then sends a reply to its request. The challenge is to generate multiple messages with different correlation information and to then correlate the incoming responses to the original request.

Fig. 15 realizes the behavior of A for P7 (and thus also for P5). First, we generate a separate instance of data object *SubRequestA* for each request that is going to be sent. The number of requests to be generated is set in the process variable *numSubRequests*. Then, for each instance of *SubRequestA*, we create a new instance of the multi-instance subprocess; each handling one instance of *SubRequestA* as case object. This case object is mapped to the global data object *Request_P3*; the primary key *subRequestID* of *SubRequestA* is mapped to the correlation information *Request_P3.request_id*. Thus, each message carries a different correlation information. This correlation information is also set for the subprocess instance from which the message is sent making the subprocess instances distinguishable. Participant B can handle the message as described in Fig. 12 and send the response. The response is correlated by A to the receive activity in the subprocess instance which has the matching correlation information. The received global *Response_P3* is transformed to the local object *ResponseB* which is related to the top-level case object *RequestA*.

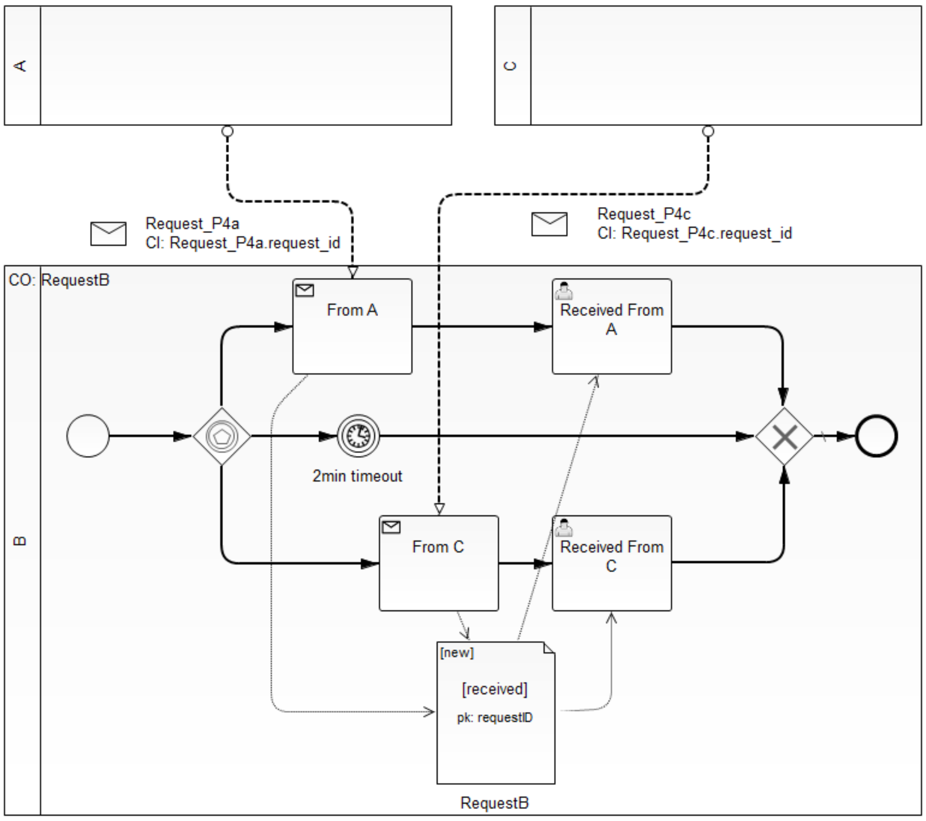


Fig. 14. Pattern P4: racing incoming messages (Participant B), with receive tasks.

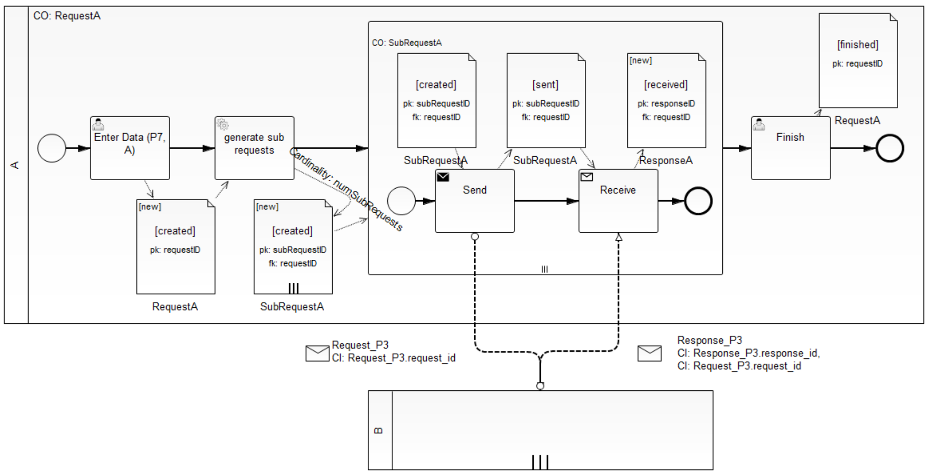


Fig. 15. Pattern P5 and Pattern P7: one-to-many send/receive.

This pattern also shows that the global data model used for Pattern P3 using *Request_P3* and *Response_P3* can be implemented in very different ways. In P3, Participant A sends just one message, whereas in P7, Participant A sends multiple messages to different recipients. The recipient process B (Fig. 12) cannot distinguish these two implementations as each request is handled by a different instance of B.

P6 One-from-Many Receive. In P6, participant A receives from an unknown number of participants B1, B2, B3, ... one message per participant. The sent messages logically correspond to each other and thus have to be correlated to the same instance of A. The challenge is to dynamically let the first message set the correlation information based on which the other messages are correlated to that instance. A message with a different correlation information has to be correlated to a different instance. Pattern P6 is not covered by Fig. 15 as there the correlation information is set by A, whereas in P6, the correlation information is distributedly set by B1, B2, etc.

The model in Fig. 16 realizes P6 for instances of A that are already running. The process uses *DocumentA* as case object. The subprocess is used to receive multiple messages containing a global object *Message_P6*. The contents of this global object is transformed to the local *DataObjectA*; the correlation information *conversation_number* is mapped to the attribute *DocumentA.number* of the case object. For each receive message, a new instance of *DataObjectA* is created. The first received message will initialize the correlation key *DocumentA.number* for the entire process. All subsequent messages that have the same key will be correlated to that instance. The subprocess has two termination criteria: receiving a certain number of messages and a timeout condition. The criterion on the received number of messages had to be implemented manually.

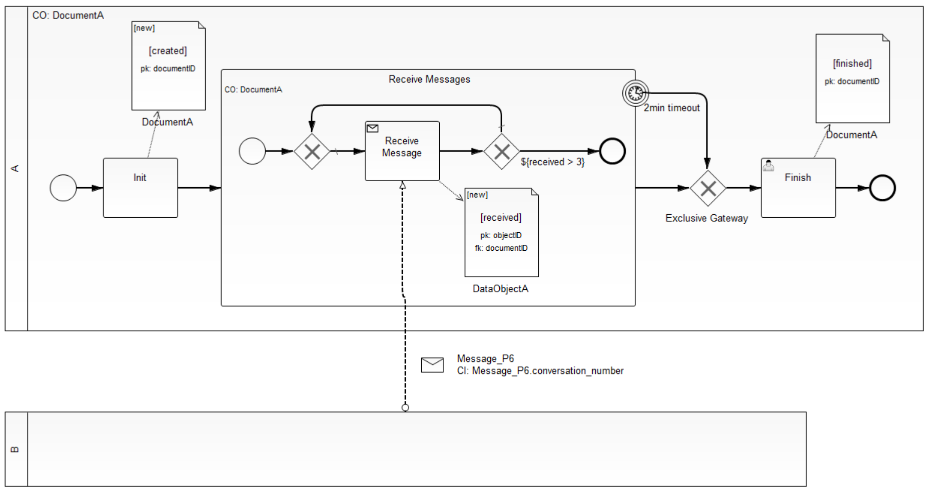


Fig. 16. Pattern P6: one-from-many receive (running instance).

The model in Fig. 16 realizes P6 for the situation when a new instance of A has to be created to receive the messages. Both receive activities can receive the same kind of messages. The first incoming message will be consumed by the instantiating receive task which also sets the correlation information. All subsequent messages that have the same correlation information will be routed to that instance. A message with a different correlation information causes the creation of a new process instance.

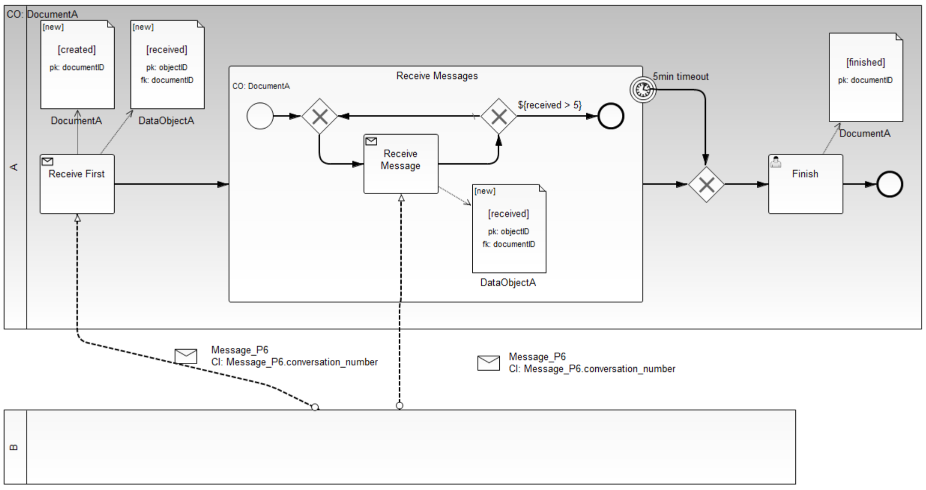


Fig. 17. Pattern P6: one-from-many receive (create instance)

5.3 Multi-Transmission Interaction Patterns

Patterns in this category describe scenarios where participant A directly exchanges multiple messages with one or more participants B1, B2, ...

P8 Multi-Responses. In P8, participant A sends a request to participant B and then receives one or more responses from B until a certain condition (based on received data or a timeout) holds. The challenge is to correlate each response of B to the instance of A that sent the request.

Fig. 18 and 19 realize Pattern P8 for participants A and B, respectively. A creates a global *Request_P8* object from their local *RequestA* object; the primary key *RequestA.requestID* serves as correlation key.

The message is received by B (Fig. 19) which can generate one or more responses in a loop. Each response carries again *RequestA.requestID* as correlation identifier; its value is retrieved from the local object *RequestB* that was created when receiving *Request_P8*. When the response arrives at A, the correlation key only matches the correlation information of the sending instance that receives multiple messages until a timeout occurs (or an upper bound of messages has been received). Note that the upper bound of messages is not derived from the model shown in Fig. 18, but implemented manually.

In general, P8 allows that B may respond with different message types. This can be achieved for B by replacing in the model of Fig. 19 the send activity with a block of alternative send activities (a pair of XOR-gateways enclosing one send activity for

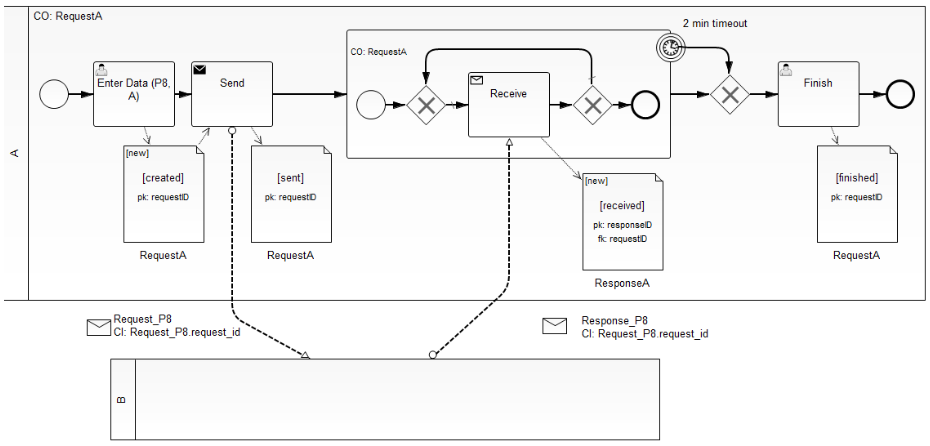


Fig. 18. Pattern P8: multi-responses (Partner A).

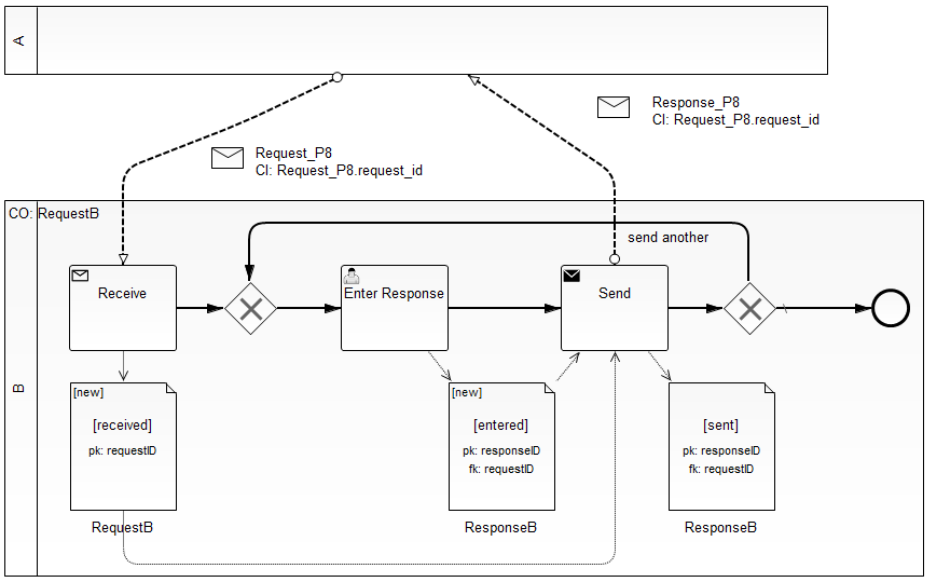


Fig. 19. Pattern P8: multi-responses (Partner B).

each message type). The XOR-gateway chooses the corresponding type by following a specific path based on the kind of information entered. For A, replace in Fig. 18 the receive activity with an event-based gateway followed by an intermediate message event or receive activity for each message type as shown in Fig. 13 and Fig. 14.

P9 Contingent Requests. In P9, participant A sends a request to participant B who shall send a response. If B's response does not arrive on time, then A will resend the request to another participant C, now expecting a response from C and discarding any response from B. If C's response does not arrive on time, the pattern is iterated with another participant D and so on until some response is received.

The challenges in P9 are (1) to pick a different recipient each time a request is sent and (2) to ensure that at any point in time only the response to the latest request is considered as valid. The model in Fig. 20 realizes this pattern as follows. Regarding (1), activity *Pick Recipient* stores the recipient's endpoint URL in a process variable; this variable is read when sending a message. Regarding (2), the case object *RequestA* has a single child object *SubRequestA* that is the case object of the subprocess. *SubRequestA* is mapped to the global object *Request_P3* and the primary key *subRequestID* is mapped to the correlation identifier *Request_P3.request_id*. The response *Response_P3* uses the same correlation identifier. For instance, the process of Fig. 12 can receive the request and send a corresponding response.

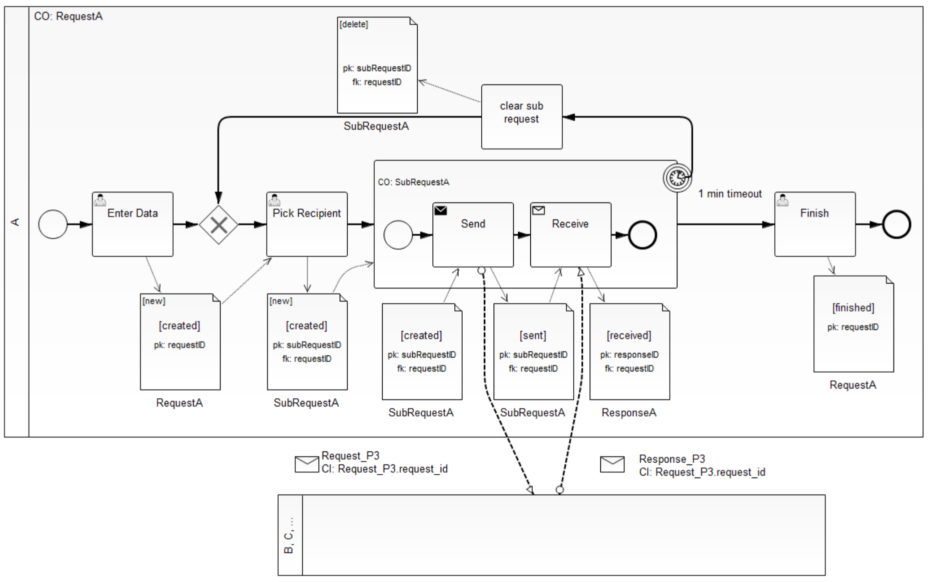


Fig. 20. Pattern P9: contingent requests (Partner A).

As *SubRequestA* is the case object of the subprocess, the correlation key is only valid for the instance of the subprocess from which the message was sent. When the timeout occurs, the instance terminates and all its correlation information is removed. Then, the *SubRequestA* instance for this request is deleted by task *clear subrequest* before creating a new one. This ensures that at any point in time *RequestA* has a unique child *SubRequestA*.

The new child is used in the next iteration of the send/receive until a response arrives. As the correlation information is attached to the subprocess, only responses arriving during the lifetime of the sending subprocess instance will be correlated to the process.

P10 Atomic Multicast Notification. In P10, participant A sends a request to multiple participants B1, B2, ..., Bm; of these between $\mathrm{n_{min}}$ and $\mathrm{n_{max}}$ participants have to respond within a certain time interval. If less than $\mathrm{n_{min}}$ participants or more than $\mathrm{n_{max}}$ participants respond, then all participants of B1, B2, ..., Bm who already did respond have to be notified, e.g., by a cancellation message. In other words, this pattern has conditional transactional properties: from all the participants that do respond to A, either all continue successfully, or all are notified with a cancellation message. Which case occurs depends on the total number of responses received by A.

The challenges in this pattern are to compute the number of received responses and depending on the outcome to either succeed or to notify all participants who did respond with a cancellation message.

The models in Fig. 21 and 22 realize this pattern. In Fig. 21, activity *Enter Data* generates the local *RequestA* data object. The subsequent service task then generates multiple *SubRequestA* objects from the contents of *RequestA* (the corresponding handling of attributes has been implemented manually). For each *SubRequestA*, an instance of the first multi-instance subprocess is created in which the *SubRequestA* is transformed into a global *Request_P10* object with *SubRequestA.requestID* as correlation identifier. When A receives a response, the *SubRequestA* object moves to state *received*; the contents of the response is stored in the object *SubResponseA*. When A did *not* receive a response until the timeout occurs, then task *clear request* deletes the *SubRequestA* object for which there was no response. The multi-instance subprocess completes when for each *SubRequestA* either the response arrived (and hence *SubRequestA* is in state *received*) or the timeout occurred (and hence *SubRequestA* has been deleted).

Thus, when reaching task *evaluate*, the case object *RequestA* of A has exactly one *SubRequestA* object instance for each received response. The task itself executes an SQL query to retrieve the number n of *SubRequestA* object instances that are associated to the case object and sets the process variable *sendCancel* to *false* iff $n_{\min} \leq n \leq n_{\max}$. If *sendCancel* is *false*, the pattern terminates (or could be extended to interact with the responding partners). If *sendCancel* is *true*, the second multi-instance subprocess is started creating one instance for each *SubRequestA* object instance associated to the case object. The subprocess instance carries the correlation key of the *SubRequestA* object, i.e., *Request_P10.request_id* which is mapped to *SubRequestA.subRequestID*. This correlation key is used in the cancellation message which can then be correlated exactly the instance that responded to the original request.

Participant B shown in Fig. 22 generates a response for the request using the same correlation information as in the response. After that, B waits at the event-based gateway for either the cancellation message to arrive or a timeout to occur after which no cancellation message from A will arrive.

Note that Fig. 21 and Fig. 22 realize P10 using model-based concepts only, except for generating contents of the subrequests, and for aggregating the number of received responses into a variable. These had to be defined manually.

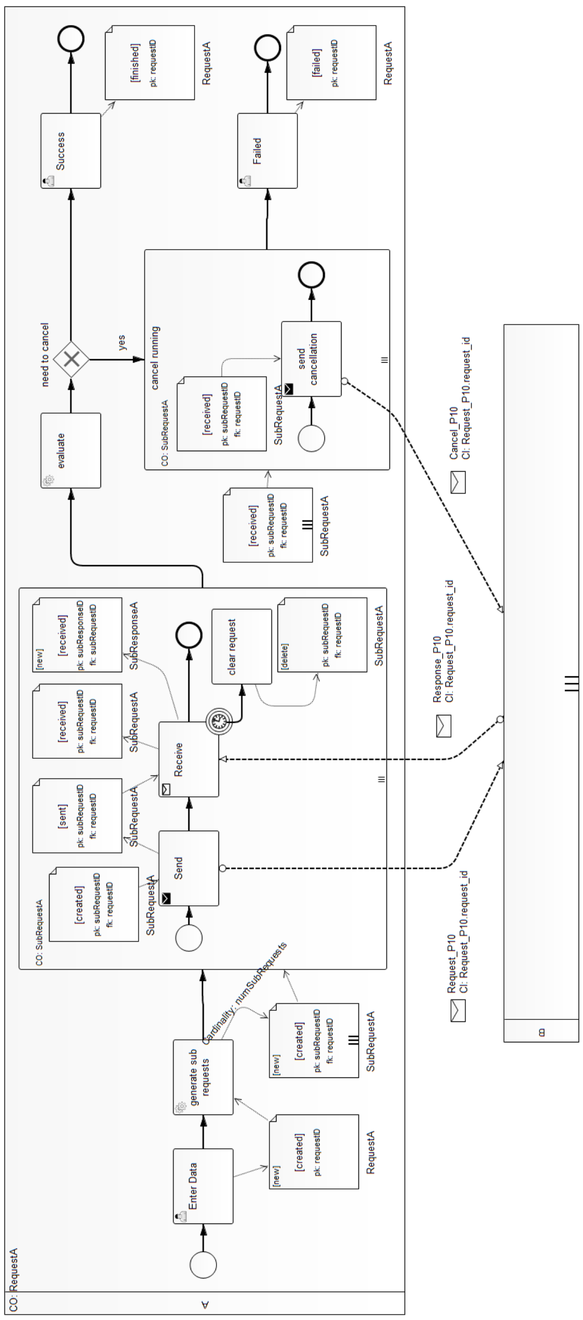


Fig. 21. Pattern P10: atomic multicast notification (Partner A).

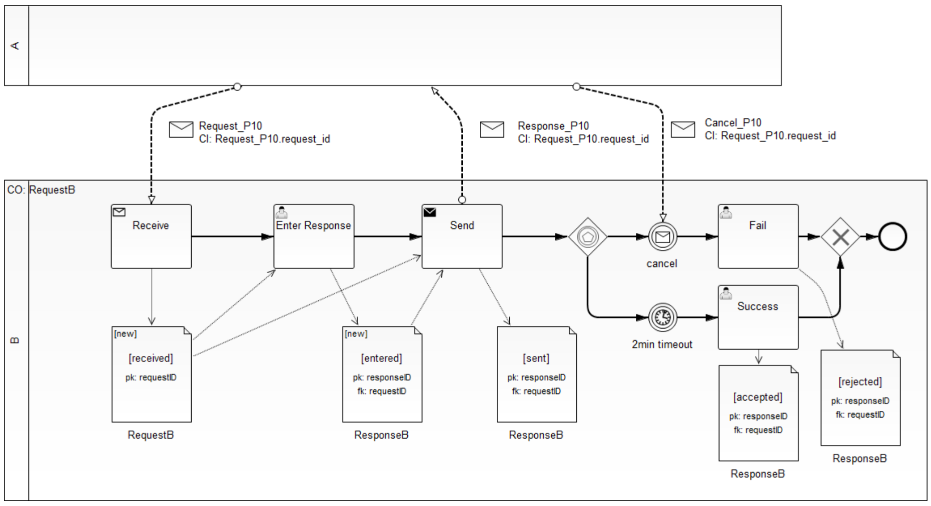


Fig. 22. Pattern P10: atomic multicast notification (Partner B).

5.4 Routing Interaction Patterns

Patterns in this category describe scenarios where participant A sends messages to participants it does not know yet via an intermediate participant B; Participant B routes messages received from A to participants C,D,...

P11 Request with Referral. In P11, Participant A sends B a request that contains the address of a participant it would like to contact. Participant B takes the recipient information from this message and forwards the request to the right recipient. The challenges in this pattern are to forward a message to another recipient and to set the recipient's address from data in the message.

The processes in Fig. 23, 24, and 25 realize this pattern. In Fig. 23, A generates the local *RequestA* which also contains an attribute *endPoint* to which the request shall finally be routed; the local object *RequestA* is transformed into the global *AtoB_P11* which is sent to B.

In Fig. 24, B receives the global object *AtoB_P11* and stores it in the local object *RequestB* including the attribute *endPoint*. The subsequent service task retrieves the value of *RequestB.endPoint* and stores it in a local variable *endPoint*. The subsequent send task generates the global object *BtoC_P11* from the stored *RequestB* and sends it to the URL in the variable *endPoint*.

The process at that URL finally receives the request as shown in the process model in Fig. 25.

The models can be extended to not only send a single endpoint URL but a list of URLs which B can then process one-by-one. Note that setting the process variable *endPoint* from the attribute of *RequestB* is not supported by our approach and had to be implemented manually.

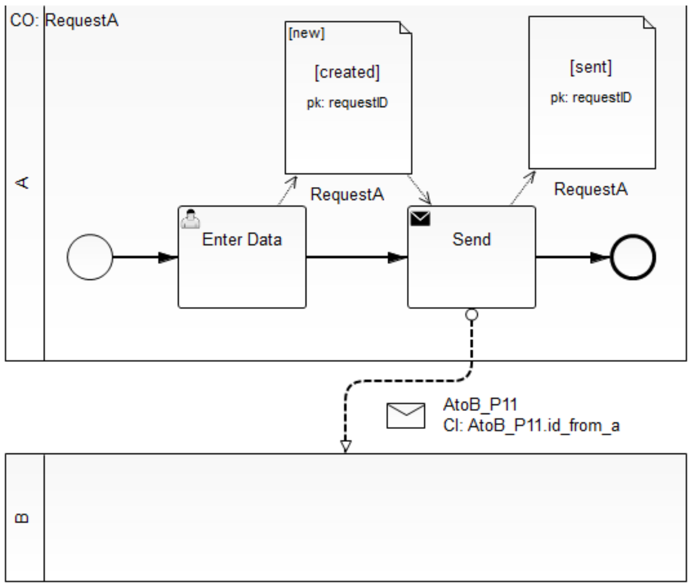


Fig. 23. Pattern P11: request with referral (Partner A).

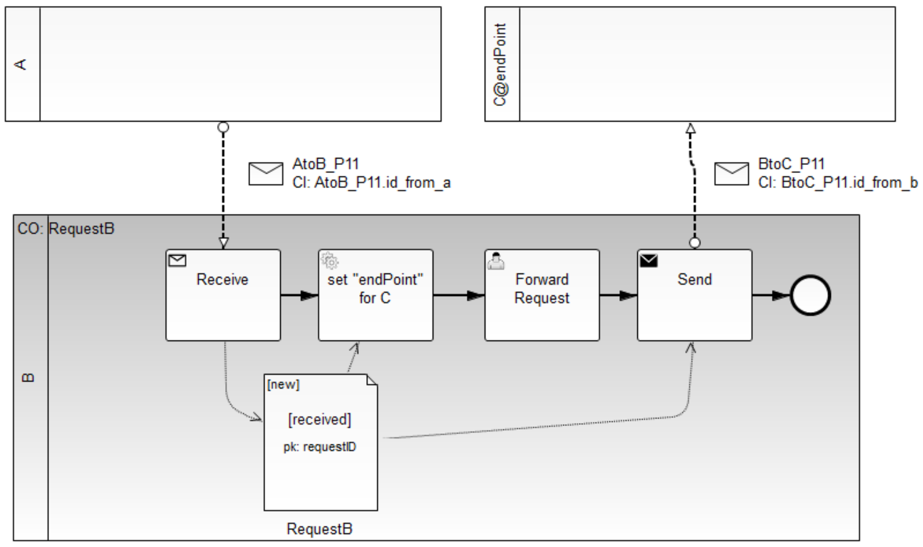


Fig. 24. Pattern P11: request with referral (Partner B).

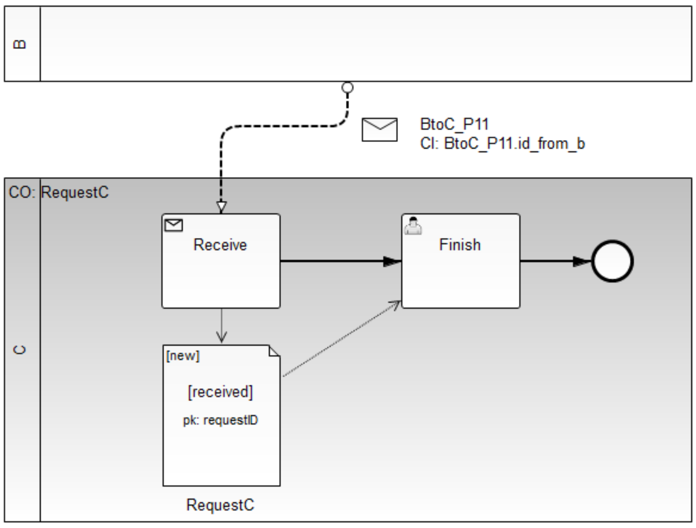


Fig. 25. Pattern P11: request with referral (Partner C).

P12 Relayed Request. In P12, participant A sends a request to B which is forwarded to C; the response of C is then sent directly to A and not via B. The challenge in this pattern is to provide C with the right correlation information so that the response from C can be correlated to the right instance of A.

The models in Fig. 26, 27, and 28 realize this pattern. Here, participants A, B, and C first agreed that the message from A to B carries correlation information to identify the sending instance of A (via correlation key *AtoB_P12.id_from_a*). This correlation information is included in the message sent from B to C so that C can use the information in the final response *CtoA_P12*. The control and message flow are then straightforward. In Fig. 26, participant A sends the request with the correlation key *AtoB_P12.id_from_a* that has been set from the local attribute *RequestA.requestID* and expects a response message *CtoA_P12* having the same correlation information. In other words, the value of *AtoB_P12.id_from_a* in *CtoA_P12* has to match the attribute *requestID* of the case object *RequestA*.

In Fig. 24, participant B receives the request from A and stores it in the local *RequestB*; in particular, attribute *AtoB_P12.id_from_a* is stored in the attribute *RequestB.idFromA*. The entire request is sent to C in the global object *BtoC_P12* which has the attributes *id_from_a* (mapped from *RequestB.idFromA*) and *id_from_b* (mapped from *RequestB.requestID*). Thus, the correlation information of A is forwarded to C although the message *BtoC_P12* only carries *BtoC_P12.id_from_b* as correlation key.

In Fig. 28, participant C receives the forwarded request from B and stores it in the local object *RequestC*; the attribute *BtoC_P12.id_from_a* is mapped to *RequestC.idFromA*. Then C generates a response which is transformed to the global object *CtoA_P12*. The message also gets the correlation key *AtoB_P12.id_from_a* for which the value is mapped from *RequestC.idFromA*. Thus the final response contains the expected correlation information for the sending instance.

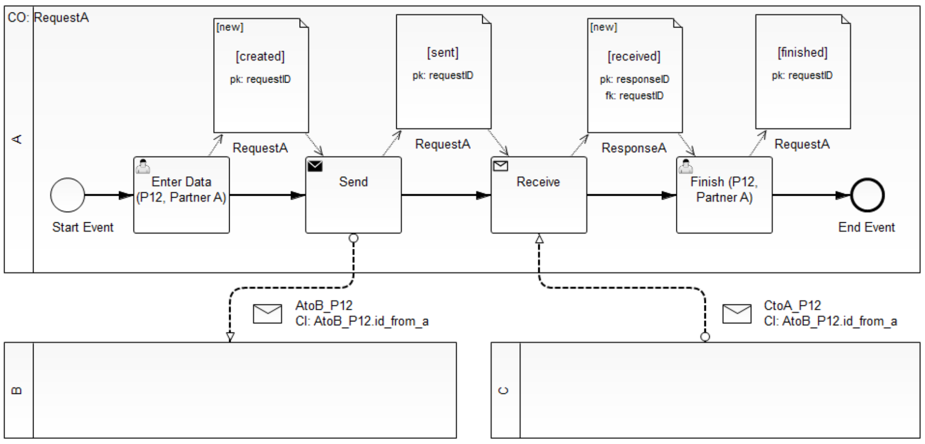


Fig. 26. Pattern P12: relayed request (Partner A).

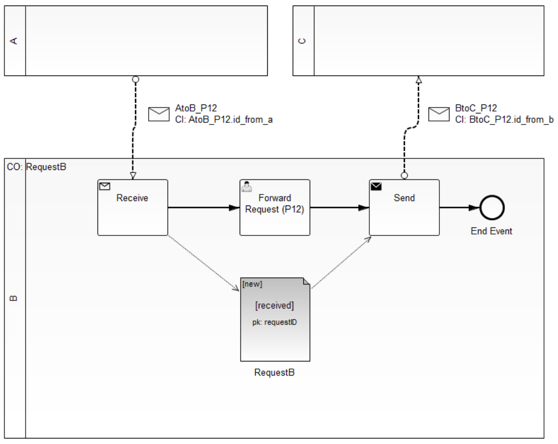


Fig. 27. Pattern P12: relayed request (Partner B).

P13 Dynamic Routing. In P13, Participant A sends a request to B which forwards the request to participant C or D depending on the contents of the message. This pattern differs from P11 in that C and D are known to B at design time and that the condition to whom to send the message is defined within B and not by A. The challenge here is to make an internal choice in B based on the contents of a received message.

The models in Fig. 29, 30, and 31 realize this pattern. In Fig 29, Participant A generates a *RequestA* which is transformed into the global *AtoB_P13* and then sent to B.

Participant B receives the message and transforms *AtoB_P13* into the local *RequestB* which has an attribute *requestText* (mapped from *AtoB_P13.request_text*). The subsequent user task does not do anything. The XOR gateway's outgoing arcs are annotated with a condition comparing attribute *requestText* of the case object *RequestB* to a value. Depending on the value, a different path is taken leading to an activation of a different send task to which *RequestB* is forwarded.

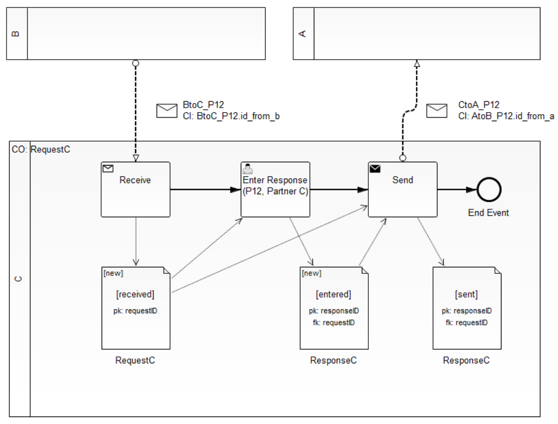


Fig. 28. Pattern P12: relayed request (Partner C).

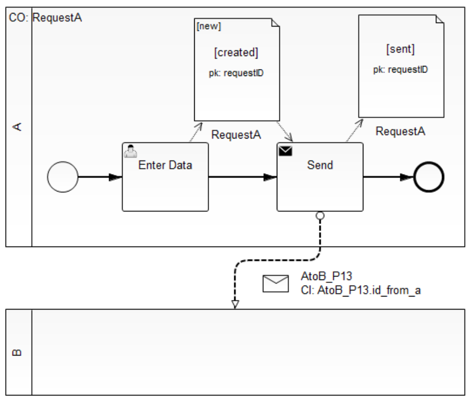


Fig. 29. Pattern P13: dynamic routing (Partner A).

This matches pattern A2 of [15] which translates to the following behavior: when reaching the XOR gateway, an SQL query `SELECT requestText FROM RequestB WHERE requestID = $ID` is generated to retrieve the value of the attribute mentioned at the outgoing arc(s) in order to evaluate the guard expressions. Technically, we store the result of the query in a local process variable which is then used to evaluate the expression.

The participant C or D (shown in Fig. 31) to which the request was sent by B receives the forwarded request and processes it locally.

Our implementation and the implemented patterns are available at *http://bpt.hpi.uni-potsdam.de/Public/BPMNData*.

6 Related Work

In this section, we briefly discuss approaches related to the concepts presented in this paper. Thereby, we focus on communication between distributed partners, the transformation of data, and message correlation. The service interaction patterns discussed in [3]

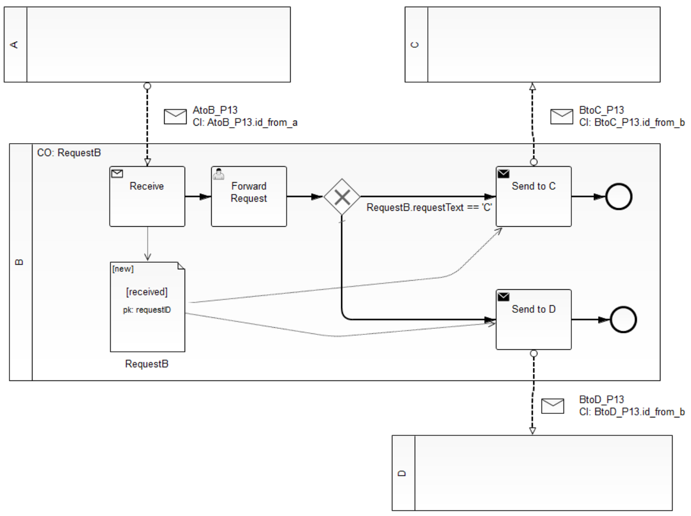


Fig. 30. Pattern P13: dynamic routing (Partner B).

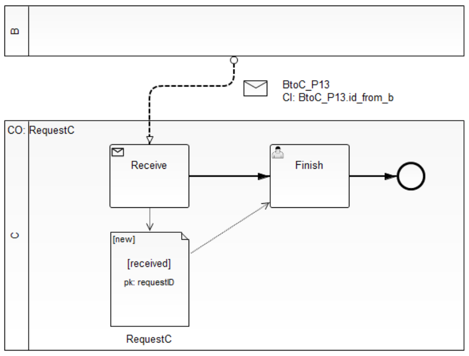


Fig. 31. Pattern P13: Dynamic Routing (Partner C and D).

describe a set of recurrent process choreography scenarios occurring in industry. Therefore, they are a major source to validate choreography support of a modeling language. Besides BPMN [18] as used in this paper as basis, there exist multiple solutions to cope with process choreographies. Most prominent are BPMN4Chor [5], Let's Dance [33], BPEL4Chor [6], and WS-CDL [27]. From these, only BPEL4Chor and WS-CDL realize operational semantics to handle message exchange by reusing respectively adapting the concepts defined in BPEL [17]. However, message transformation to achieve interoperability between multiple participants is done with imperative constructs meaning that the process engineer has to manually write these transformations and that she has to ensure their correctness. Additionally, BPEL4Chor and WS-CDL are not model-driven as the approach introduced in this paper.

Apart from process and service domains, distributed systems [22] describe the communication between IT systems via pre-specified interfaces similar to the global contract discussed in this paper. Usually, the corresponding data management is done by distributed databases [19] and their enhancements to data integration systems [12, 23] as well as parallel database systems [8] or is done by peer-to-peer systems [9, 24]. The database solution allows many participants to share data by working with a global schema which hides the local databases, but unlike our approach, the participants work on the same database or some replication of it. Peer-to-peer systems take the database systems to a decentralized level and include mechanisms to deal with very dynamic situations as participants change rapidly. In process choreographies, the participants are known and predefined such that a centralized solution as presented in this paper saves overhead as, in the worst case, the decentralized approach requires a schema mapping for each communication between two participants instead of only one mapping per participant to the global schema. The transformation of data between two participants can be achieved via schema mapping and matching [20, 21], a mediator [31], an adapter [32], or ontology-based integration [4, 16, 29]. For instance, [4] utilizes OWL [26] ontologies, which are similar to our global data model, and mappings from port types to attributes via XPath expressions to transform data between web services. In this paper, we utilize schema matching due to the close integration of database support for data persistence.

Returning to the process domain, there exist fundamental works describing the implementation of process choreographies [1, 7] with [1] ensuring correctness for inter-process communication. These works only describe the control flow side although the data part is equally important as messages contain the actual artifacts exchanged. [11] introduces a data-aware collaboration approach including formal correctness criteria. They define the data perspective using data-aware interaction nets, a proprietary notation, instead of a widely accepted one as BPMN, the industry standard for process modeling, used as basis in this paper.

7 Conclusion

In this paper, we presented an approach allowing to model and enact the data perspective of process choreographies entirely model-driven. Thereby, we utilized the industry standard BPMN and extended the model-driven data dependency enactment approach catered for process orchestrations with concepts for process choreographies. Based on challenges of data heterogeneity, correlation, and 1:n communication, we identified a set of six requirements covering the retrieval of data from the sender's local database, the data transformation into a global data schema all participants agreed upon, the correlation of a message to the correct activity instance, the transformation from the global to the receiver's local database schema, and the storage of the data there. The message routing between participants is out of scope of this paper by adapting existing technologies as, for instance, web services. In this paper, we describe a modeling guideline with the artifacts required to automatically execute the mentioned steps from these only. Furthermore, we describe the corresponding operational semantics and provide details about our implementation. Our approach has been implemented; we could implement all service interaction patterns of [3] except for dynamically setting URLs of recipients and evaluating data conditions over aggregations of data objects; both are outside the scope of this paper and deserve future work. Also the integration of process layer and message transport layer (in particular wrt. handling message transport errors) is outside the scope of this paper. The current approach only utilizes tasks to send and receive messages. However, BPMN also supports *message events* to which the discussed concepts could be applied as well by overcoming BPMN's limitation that events cannot transform and correlate data objects. In future work, we plan to provide an integrated formal verification technique for model-driven data enactment in process orchestrations and choreographies.

References

1. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: Multiparty contracts: Agreeing and implementing interorganizational processes. Comput. J. 53(1), 90–106 (2010)
2. van der Aalst, W.M.P., Weske, M.: The p2p approach to interorganizational workflows. In: CAiSE. pp. 140–156. Springer (2001)
3. Barros, A., Dumas, M., ter Hofstede, A.H.M.: Service interaction patterns. In: Business Process Management. pp. 302–318. Springer (2005)
4. Bowers, S., Ludäscher, B.: An ontology-driven framework for data transformation in scientific workflows. In: Data Integration in the Life Sciences. pp. 1–16. Springer (2004)
5. Decker, G., Barros, A.: Interaction modeling using bpmn. In: BPM Workshops. pp. 208–219. Springer (2008)
6. Decker, G., Kopp, O., Leymann, F., Weske, M.: Bpel4chor: Extending bpel for modeling choreographies. In: ICWS. pp. 296–303. IEEE (2007)
7. Decker, G., Weske, M.: Interaction-centric modeling of process choreographies. Information Systems 36(2), 292–312 (2011)
8. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Communications of the ACM 35(6), 85–98 (1992)
9. Halevy, A.Y., Ives, Z.G., Suciu, D., Tatarinov, I.: Schema mediation in peer data management systems. In: Data Engineering. pp. 505–516. IEEE (2003)
10. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., Barreto, C.: Web services choreography description language version 1.0. W3C candidate recommendation 9 (2005)
11. Knuplesch, D., Pryss, R., Reichert, M.: Data-aware interaction in distributed and collaborative workflows: Modeling, semantics, correctness. In: CollaborateCom. pp. 223–232. IEEE (2012)
12. Lenzerini, M.: Data integration: A theoretical perspective. In: Symposium on Principles of Database Systems. pp. 233–246. ACM (2002)
13. Mendling, J., Hafner, M.: From ws-cdl choreography to bpel process orchestration. Journal of Enterprise Information Management 21(5), 525–542 (2008)
14. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and Enacting Complex Data Dependencies in Business Processes. In: BPM. pp. 171–186. Springer (2013)
15. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and Enacting Complex Data Dependencies in Business Processes. Tech. Rep. 74, HPI at the University of Potsdam (2013)
16. Noy, N.F.: Semantic integration: a survey of ontology-based approaches. ACM Sigmod Record 33(4), 65–70 (2004)
17. OASIS: Web Services Business Process Execution Language, Version 2.0 (April 2007)
18. OMG: Business Process Model and Notation (BPMN), Version 2.0 (January 2011)
19. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Springer (2011)
20. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. The VLDB Journal 10(4), 334–350 (2001)
21. Shvaiko, P., Euzenat, J.: A survey of schema-based matching approaches. Journal on Data Semantics IV 3730, 146–171 (2005)

22. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms. Prentice Hall (2006)
23. Tomasic, A., Raschid, L., Valduriez, P.: Scaling access to heterogeneous data sources with disco. Knowledge and Data Engineering, IEEE Transactions on 10(5), 808–823 (1998)
24. Valduriez, P., Pacitti, E.: Data management in large-scale p2p systems. In: High Performance Computing for Computational Science (VECPAR), pp. 104–118. Springer (2005)
25. W3C: Web Services Description Language (WSDL) 1.1 (March 2001)
26. W3C: OWL Web Ontology Language (February 2004)
27. W3C: Web Services Choreography Description Language, Version 1.0 (November 2005)
28. W3C: XQuery 1.0: An XML Query Language (Second Edition) (December 2010)
29. Wache, H., Voegele, T., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H., Hübner, S.: Ontology-based integration of information-a survey of existing approaches. In: IJCAI workshop: ontologies and information sharing. pp. 108–117 (2001)
30. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Second Edition. Springer (2012)
31. Wiederhold, G.: Mediators in the architecture of future information systems. Computer 25(3), 38–49 (1992)
32. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Transactions on Programming Languages and Systems (TOPLAS) 19(2), 292–333 (1997)
33. Zaha, J.M., Barros, A., Dumas, M., ter Hofstede, A.H.M.: Let's dance: A language for service behavior modeling. In: OTM Conferences, pp. 145–162. Springer (2006)