Using the Business Process Model and Notation for Modeling Enterprise Integration Patterns

Daniel Ritter

HANA Platform, SAP AG
Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany
`daniel.ritter@sap.com`

Abstract. *Enterprise Integration Patterns* (EIP) are a collection of widely used stencils for integrating enterprise applications and business processes. These patterns represent a "de-facto" standard reference for design decisions when integrating enterprise applications.
For each of these patterns we present the integration semantics (model) and the conceptual translation (syntax) to the *Business Process Model and Notation* (BPMN), which is a "de-facto" standard for modeling business process semantics and their runtime behavior.

Keywords: Business Process Model and Notation (BPMN), Enterprise Integration Patterns, Message-based Integration, Middleware

1 Introduction

In the last decades, companies have developed a multitude of software applications. The interest of reusing them is not only a matter of convenience, but also of costs. Reuse of applications often occurs in connection with third parties–another department, or even another company. To allow applications from different domains–namely different data types and interfaces–to communicate, all parties have to integrate with a communication solution.

In their best practices book Enterprise Integration Patterns (EIP) [3] have collected a widely used and accepted collection of integration patterns allowing for easier implementation of a communication infrastructure. They identify four possible solutions for solving communication: file transfer, shared database, remote procedure invocation, and messaging. The patterns presented in the book are typical concepts used when implementing a messaging system and have proved to be useful during implementation. These concepts help to cope with the asynchronous nature of message exchange and the facts, that "networks are unreliable", "networks are slow", "any two applications are different", and "change is inevitable". On the other hand, the modular nature of patterns allows them to be used efficiently in new implementations. Subsequently, we present most patterns described by [3] plus well-known additional patterns (e. g., from [14] and [1]) and give a realization as Business Process Model and Notation (BPMN) [11]. Thus we have business process building blocks allowing us to model a messaging system in BPMN that is based on a design given by EIPs.

For instance, Figure 1 shows an integration scenario of a corporate with its bank and business monitoring via SAP Cloud for Cash, syntactically expressed in BPMN according to the definition proposed by this paper. The incoming message is of type "FSN" (short for Financial Services Network[1]), which has to be translated to its canonical data model incarnation "FSN:CDM" using a *Message Translator* pattern. Through an adapted *Claim Check* pattern, the message is stored for later use and handed over to the *External Service* pattern as request to the bank (no further translation required). On successful execution, the original message is restored from the claim check, translated to an ISO format "FSN-ISO" and send to the SAP Cash on Demand application[2] (ODC).
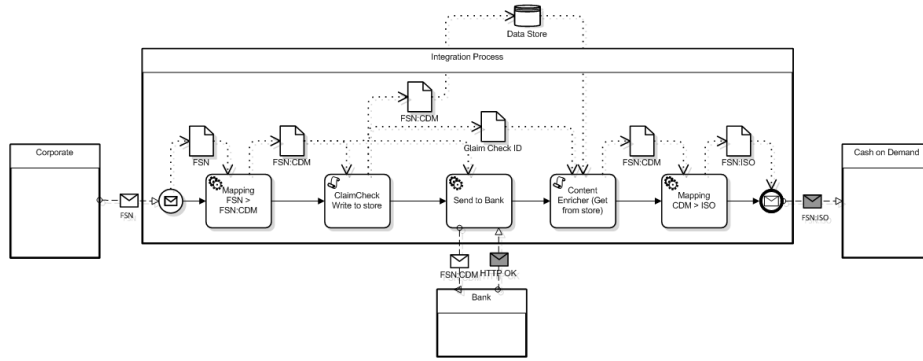

Fig. 1. Business Monitoring: Messages sent from Corporate to Bank are routed to SAP Cloud for Cash for business monitoring purpose.

The problem that we address in this paper is the lack of standardized syntactic and semantic formalization of the common integration patterns [3] and recent extensions (e. g., *Join Router*, *External Service* from [14], *Synch-Asynch/Asynch-Synch* bridges from [16], and *Multicast* from [1]). Our goal is to provide means for an implementation independent description (i. e., syntax) and semantics (i. e., model) of a messaging system. That means, only the modeling is considered and the concrete runtime implementations are neglected. Our BPMN realization provides a blue print for implementation that is proven to translate to existing messaging systems like Apache Camel [1], SAP HANA Cloud Integration[3], and SAP Process Integration[4].

The remainder of this paper is as follows: Section 2 discusses and manifests the contribution of the paper by putting it into context to related work. Then, integration semantics and the used subset of BPMN is defined in the context of

---

[1] http://scn.sap.com/docs/DOC-40696
[2] http://global.sap.com/germany/solutions/technology/cloud/business-by-design/highlights/index.epx
[3] http://help.sap.com/cloudintegration
[4] http://scn.sap.com/community/pi-and-soa-middleware

the EIPs, which leads to the definition of the syntax and model in Section 3. Section 4 concludes the discussion with a short summary.

2   Related Work

The patterns described by [3,14,16,1] are not building blocks of a modeling language, however, they describe typical concepts in designing a messaging system; thus they are an informal specification language. For that, there are elaborated modeling techniques like the *Business Process Model and Notation* (BPMN) [11] or the *Workow Patterns* defined by [18]. Enterprise Integration Patterns complement these notations by a set of typical designs found in a messaging infrastructure.

Our approach stresses on the control flow, data flow and modeling capabilities of BPMN as well as its execution semantics. Recent work on "Data in Business Processes" [6] shows that besides *Configuration-based Release Processes* (CORE-PRO) [10,8,9], which mainly deals with data-driven process modeling and (business) object status management, and UML activity diagrams, BPMN achieves the highest coverage in the categories relevant for our approach. Compared to BPMN and apart from the topic of "object state" representation, neither *Workflow Nets* [17] nor petri nets do support data modeling at all [6]. Based on that work, BPMN was further evaluated with respect to data dependencies within BPMN processes [5,7], however, not towards control and data flow as in our approach.

Our work combines this foundation with efforts on executable integrations patterns, their configuration and mapping to the *Web Services Business Process Execution Language*[5] (WSBPEL) proposed by [14] and leverages the work of [15,16] that started to map the EIPs to the BPMN syntax and some semantics by example. In this document, we provide a systematic continuation of [15,16] by defining a comprehensive syntax and model for widely-used patterns.

Since BPMN can be translated to *Petri Nets* [12,13], our approach can leverage the body of work from the BPMN verification domain. In particular, the area of verification of middleware designs [2] translated the EIPs from [3] into *Colored Petri Nets*. Since BPMN is closely related to petri nets, the syntactical translation is comparable to our approach. However, the concrete syntactical mapping of integration patterns and their corresponding model (semantics) that leverages the BPMN syntax with a partially changed execution model is novel.

3   Enterprise Integration Patterns

Before presenting the patterns we shortly discuss the BPMN syntax that is used to describe EIPs and the (process) model we use for the integration semantics. The model might slightly differ from the original BPMN runtime semantics.

---

[5] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

The main syntactical artifacts in BPMN conduct process steps, sequences and the representation of messages that are exchanged between processes during runtime. Subsequently, the BPMN elements relevant to this work are introduced. They are mainly taken from the *BPMN Collaboration Diagram* [11].

Process flow control The flow in a BPMN graph is controlled by flow objects that initiate, route and terminate the flow:

- Events: control the process flow. We consider start and end events. Start events initiate the process flow and thus only have outbound edges in the graph. Figure 2 shows a start event without defined event (first row, left) and one with a "message-arrives" semantic, which triggers the event (first row, second from left). Similarly, the end event terminates the flow and thus terminates a process or part of a process. The standard end event without defined result (first row, third from left) can also appear with message symbol with the semantic of sending a message to a participant before terminating the process (first row, forth from left). In addition, throwing/catching intermediate events are used to express message events (second row, two icons on the left), errors (second row, middle), and timers (second row, right).
- Activities/Tasks: are process steps that have to be finished, before a flow can proceed. Depending on the type of activity, a process step can be executed multiple times. The label specifies the operation that is executed by the process step (third row, left). If the activity is atomic (i. e., a task), the task types like *Send Task*, *Receive Task*, *Script Task* can be specified (not shown).
- Gateways: handle multiple process flows, where they work as logical cases that support simple logical operations like AND, OR, and more complex nested cases within flows (e. g., parallel processing, multiplexers). In this document, we mainly work with exclusive (third row, second from left), inclusive and parallel gateways where the label specifies the semantics of the case.
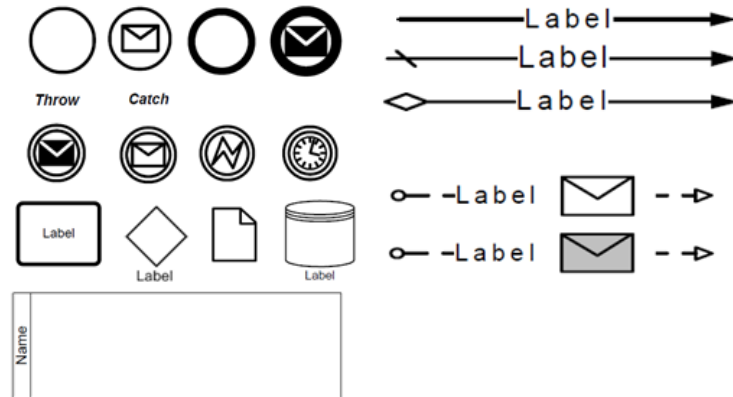

Fig. 2. Used BPMN Syntax.

Connections The connecting objects are the edges in the graph that differ by the transferred objects:

- Sequence Flows: sequences of process steps. The non-default, standard sequence flow has an unspecified "conditionExpression" (right, top). The default sequence flow again has an unspecified "conditionExpression" (right, second to top). A specified condition expression with non-default semantics is shown in Figure 2 (right, third to top).
- Message Flow: message exchange (e. g., process status information, error messages, data) between participants or participants and process elements. The initiating message flow is shown on the left, and the non-initiating message flow on the right.

Pools

- Pool/ Participant: a pool is a general graphical, grouping element, which represents a participant. It also acts as a graphical container for partitioning a set of Activities from other pools, usually in the context of B2B situations. A pool may have internal details, in the form of the process that will be executed. A pool that has no internal details is called "black-box" pool (Figure 2, fourth row).

Additional Artifacts

- Data Objects: represent provided or generated Data. Data objects are associated (cf. association) to another BPMN element and do not exist independently. Figure 2 shows data objects (third row, third from left). The data associations are not shown.
- Data Stores: represent a local or central storage of data (e. g., messages). Stores are associated to BPMN Flow Elements (e. g., Data Objects, Events, Activities, Gateways). The data store is shown in Figure 2 (third row, fourth from left).

Besides the syntax (i. e., graphical notation), BPMN specifies a (process) model (i. e., semantics), called execution semantics. The execution semantics define the meaning of the syntax and specify the behavior during execution. In this work, we remain conformant to the syntax, but define a slightly different process model for the execution semantics (basically a subset of the BPMN semantics with a stronger focus on the data flow).

Definition 1 (Process model). *A process model $M = (N, SF, DO, DF)$ consists of a finite non-empty set $N \subseteq A \cup G \cup E$ of nodes being activities A of types ServiceTask, ScriptTask and MessageTask, gateways G of types Exclusive-Gateway and ParallelGateway, and events E of types StartEvent, EndEvent and Intermediate Event, where A, G, E are pairwise disjunct.*

The finite non-empty set of SequenceFlow relations $SF \subseteq (N \setminus EndEvent) \times (N \setminus StartEvent)$ represents the control flow. The finite, non-empty set of data objects DO represents data associated to N and $DF \subseteq (N \cup DO) \times (DO \cup N)$ is the data flow relation.

For the Process and Sub-Process instantiation, a Message Start Event *or* Receiving Message Task *is required comparable to a* `constructor`. *The instances can be terminated by* (Message) End Events, *the* `destructor`. *An already instantiated process can be re-invoked using the BPMN correlation mechanism, similar to a* `factory pattern`.

In other words, a process is initiated by a start event, i.e., a message that contains data according to a specified format (e.g., XML Schema). Then a sequence flow is fired that moves the control to the next flow element in the process (e.g., acticity, gateway, end event). The data flow is handled by associated data objects from one element to the next one. The process ends through the invocation of a message end event that fires the outgoing message before the process context stops.

3.1 Integration Semantics

The integration patterns that are mapped to BPMN subsequently can be summarized by their integration semantics like (i) cardinality, (ii) message and meta-data creation, (iii) quality of service/ transactional behavior, (iv) logic, (v) lineage, and (vi) processing types.

The *cardinality* (i) can be determined on a message and a channel level. On the message level, the cardinality specifies the number of incoming and outgoing message instances of a pattern. For instance, if the original message is routed through an *Activity* representing a pattern, it has a cardinality of 1:1 and if a pattern splits the original message into n outgoing messages, then the cardinality is 1:n. Similarily, the definition of cardinality on a message channel level describes the incoming and outgoing channels (edges) of a pattern. Note that the *Splitter Pattern* has a message cardinality of 1:n, but a channel cardinatlity of 1:1 (i.e., 1:n if and only if the multiple output formats are defined).

Message Creation (ii) is a characteristic that defines the behavior of a pattern when processing the original message. If and only if the message cardinality is 1:1 and the identifier of the outgoing is the same as the incoming message instance (not necessarily the content or format), a pattern is *non-message generating*. Otherwise, the pattern is *message generating*. The message's meta-data is either represented by a *Message.ItemDefinition* assigned to the incoming and outgoing *MessageFlow* elements or a *DataObject.ItemDefinition.structureRef* within the *Process* and its *Sub-Process* definitions.

The Quality of Service (QoS) and the transactional behavior are rather a measure on the whole message channel and can be determined through all involved EIPs. However, one characteristic that can be found in those contexts is the persistence requirement of a pattern (iii): all patterns that do not store whole messages or parts of them are *stateless* pattern (e.g., typically all filters, message translators, routing). If a pattern requires to persist data to fulfill its purpose, it is *stateful* (e.g., typically a *Message Store*, *Aggregator*, *Resequencer*). The characteristic of storing data in memory between invocations is not covered.

The very nature of a pattern is described by its specific processing logic (iv). This aspect of the integration semantics can be very specific to the underlying runtime implementation and is normally fully expressed in the available programming languages on the runtime platform.

The ability to follow a message from the sender to the receivers is very important. The information necessary for that is called lineage or message correlation (v). Patterns that change the original message or even generate new messages have to be able to link the (new) outgoing message(s) to the original message.

The processing types (vi) specify the runtime behavior of the pattern in terms of (a) sequential or parallel processing, (b) stream processing, (c) timeouts, and (d) behavior in exceptional situations. The capability to process messages in parallel (a) allows sending several messages to that pattern at a time (concurrently), while the sender still has to wait for the operation to be completed. When streaming is enabled, bigger data streams become tractable and the timeout (c) ensures that the processing stops after a configurable amount of time. If one of the patterns raises an exception during processing, its behavior shall be configurable (d) to either stop the processing of this pattern immediately or continue with other aspects of that pattern or even with their patterns.

3.2   Integration Pattern Syntax and Model

We now present the actual syntactic translation of the EIPs to BPMN as continuation of [15,16] and specify the integration semantics in an informal manner by combining the work from [3,14] with our experience. The subsequent descriptions are grouped by the different pattern categories and the numbers in brackets represent the pages for reference in [3]. We start by describing a *Message Channel* (60), the fundamental infrastructure of a messaging system. The message channel connects sender with receiver applications (i. e., participants) for message-based communication. A simple form of a message channel is the *Point-to-Point Channel*, which connects one sender with one receiver directly. The unidirectional P2P channel is technically a bucket. The sender puts data on the channel (i. e., a message), the receiver takes it from the channel. If multiple receivers are connected to the channel, the actual recipient is not necessarily determined. However, a particular message is taken only by one receiver. The order of messages is a matter of implementation. In existing messaging systems we can normally assume the channel to be a queue, thus the messages are received in order of their sending.

Alternatively, one to many channels like publish-subscribe, broadcast, multicast and bus-like communication can be realized with message channels. A message sent to such a channel can be received by multiple receivers. For n receivers, n copies of the original message have to be provided.

In general, channels have special non-functional qualities like QoS characteristics that have to be configurable: *Message Exchange Pattern* (MEP), *availability*, *security*, *transactional processing*, *Guaranteed Delivery* (122), and *maximal message size*. The MEP defines a quality of the channel that can be derived from the adapter (i. e., message flow) or component/processor (e. g., activities, events)

configurations. For instance, a file polling message flow acts one-way (InOnly), since it cannot handle response messages. On the other hand, most document message exchange works according to the *Request-reply Pattern*, which specifies a two way communication (InOut). In other words, channels can derive MEP constraints from its adapters and components or configure its behavior explicitly. A *Channel* acts as logical address, thus the actual receiver is determined by the messaging system. A single *Message* can transport a piece of data (i. e., *Document Message* (147)), a command for execution (i. e., *Command Message* (145)), or an event for logging (i. e., *Event Message* (151)). *Pipes and Filters* [4] as well as message routers are meant for influencing message content and direction. A filter can drop unwanted messages or content and a router can direct messages based on their content or a system's state to a particular destination, and a pipe as special form of message channel connects these components. A *Message Translator* converts messages like transforming from one data format into another or extracting only necessary parts of a message. Finally, a *Message Endpoint* (95) connects an application to a messaging system. In case of BPMN, the *MessageFlow* represents a message endpoint by specifying the message with its structure, operation and interface (e. g., WSDL) that can be routed to the message channel.

Special Messaging Channels Apart from the general channel specification there are some special channel types that are important. The *Datatype Channel* (111) is a channel that only transports messages of a specific type. The message cardinality is 1:1, while the outgoing is the same as the incoming message. For exceptional cases, a channel for the storage and re-sending of messages can be configured (e. g., invalid message channel, dead letter channel). An *Invalid Message Channel* (115) forwards messages, which could not be processed by the receiver. Its message cardinality is 1:0..1 depending on whether the outgoing message shall be stored or forwarded. In contrast, a *Dead Letter Channel* (119) handles exceptional messages that have not yet been sent to a receiver, however, have been expired. The expiration is not determined by the dead letter channel, but by the base channel patterns (e. g., P2P, Publish/Subscribe). The message cardinality is the same as in the latter case.

Message Construction When applications want to exchange data, they wrap it into a message. The message channels can only transport data wrapped in a message. The following patterns are subsequently discussed:

1. Request-Reply (154)
    - Synchronous Request-Reply
    - Asynchronous Request-Reply
    - Synch-Asynch Bridge
    - Asynch-Synch Bridge
2. Correlation Identifier (163)

Table 1. Request-Reply (154)

|	
Pattern	Request-Reply (154)
Model	The *Request-Reply* pattern is a communication pattern allowing a MEP with two-way (InOut) communication between two applications (i. e., in contrast to asynchronous, one-way communication). In particular, one application (requester) sends a request on one channel and waits for the response. The second application (replier) receives the request message and replies on a separate channel. The invocation of the requester can be synchronous (blocking) or asynchronous (non-blocking). The request channel can be a *Point-to-Point Channel* (103) or a *Publish-Subscribe Channel* (105). The intend of the message exchange can be *Messaging RPC* (RPC with a *Command Message* request and a *Document Message* response), *Messaging Query* (remote query using a command message with the query and a document message with the result), or *Notify/Acknowledge* (the request is an Event Message and the reply a Document Message with the acknowledgement). Hereby, the request can be either *void* (e. g., notification), *result value* (single result object), or *Exception* (single exception object). The request can contain a return address and the reply should contain a *Correlation Identifier* to correlate to the request.
Syntax	The request-reply stands for a synchronous or asynchronous communication with an external application (participant). Figure 3 shows how a synchronous request is expressed by a *Service Activity* that initiates the message processing in the external processing upon activation and data input (i. e., association to *Data Object*). The activity waits until the external participant completed its processing and returns a message that is then assigned to the outgoing data object. In case of a *fault* is returned by the service, it is treated as *interrupting error* and the activity fails. The exception is processed in a separate channel. An asynchronous request-reply is shown in Figure 4. The *Send Activity* assigns the input from the *Data Object* to the message and sends it to the external receiver. The activity is finished and a *Receive Activity* is activated. The activity starts waiting for the associated message and the arriving message is assigned to the data object assigned to the activity. The correlation between the message sent to and the message received from the receiver have to be modeled in the definition of the *Message* element of the *Message Flow*. Frequently used combinations of the previously discussed variants is the synch-asynch bridge (Figure 5) and the asynch-synch bridge (Figure 6). While the asynch-synch bridge simply combines the shown syntax, the synch-asynch bridge in Figure 5 shows an allowed syntactic variation by expressing the request by a *Message Intermediate Event (throw)* that initiates the message processing in the external participant, while giving the control to a *Message Intermediate Event (catch)* (i. e., activates the event) that waits for the response from the external participant. If and only if the participant responds with a message, the active event starts working. The syntax for the bridge patterns shown in Figures 5 and 6 requires an additional participant.
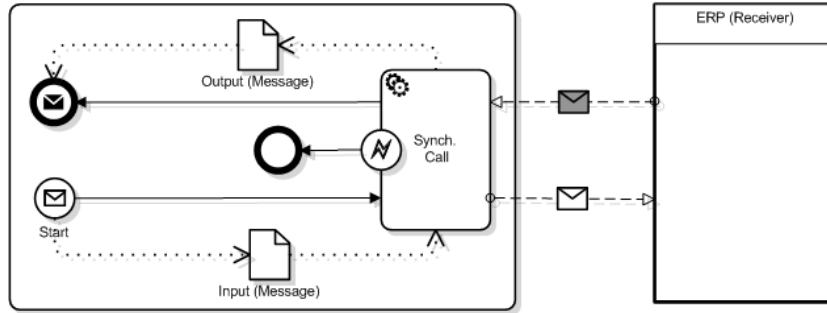
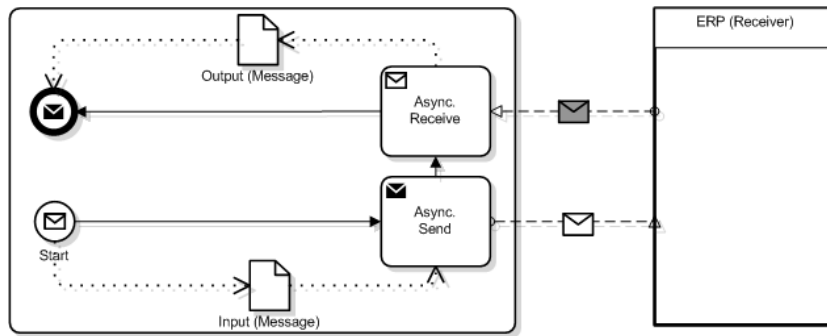**Fig. 3.** Request-Reply Pattern (synchronous).
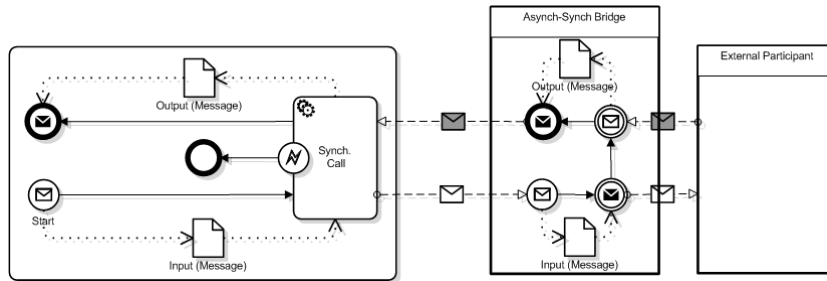
Fig. 4. Request-Reply Pattern (asynchronous).

Fig. 5. Synch-Asynch Bridge.

Table 2. Correlation Identifier (163)

Pattern Correlation Identifier (163)	
Model	The *Correlation Identifier* is used to correlate sent and received messages, when dealing with a multitude of messages. For instance, it can be used in a two-way message exchange for reply correlation to the request. The identifier mostly is a message header information.
Syntax	The correlation identifier is part of the message and the data objects that are part of the data flow. Hence there is no explicit syntax for it.
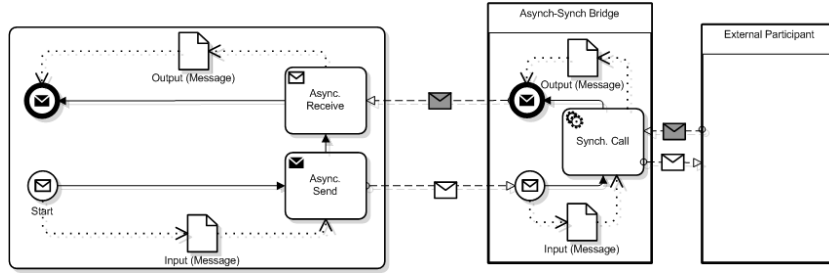
Fig. 6. Asynch-Synch Bridge.

Message Routing Messaging systems use message routing to decouple a message sender from the receiver of the message. In [3] different types of routers are categorized: *simple* (message routing from sender to one or more receivers), *composed* (combined simple routers), and *architectural patterns* (architectural styles based on routers). In this paper, we exclusively focus on simple routers and leave the task of composition up to the reader. The following patterns are subsequently discussed:

1. Content-based Router (230)
2. Message Filter (237)
3. Recipient List (249)
4. Splitter (259)
5. Aggregator (268)
6. Resequencer (283)
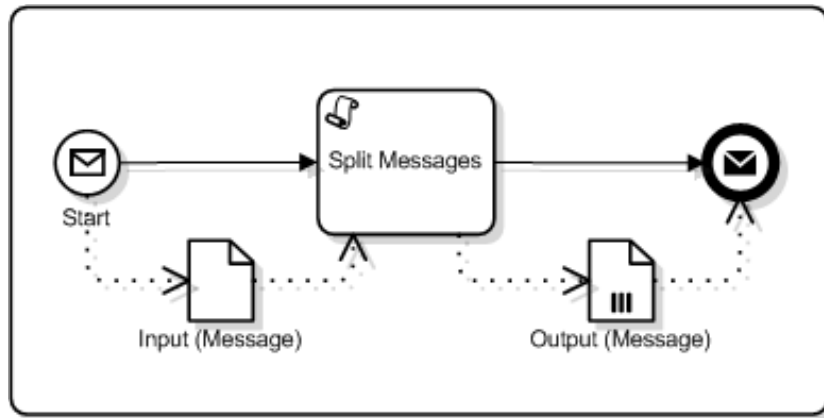
Fig. 7. Message Splitter Pattern.

Table 3. Content-based Router (230)

Pattern	Content-based Router (230)
Model	The *Content-based Router* determines the receiver of an incoming message based on its content. Dynamically forwarding the message to the correct receiver requires knowledge about all possible recipients. The router receives exactly one incoming message and routes it to one of multiple outbound channels. Neither the message nor its content is changed. The router is stateless.
Syntax	The content-based router decides on the route of the messages represented by a *Data Object* by evaluating a *Condition Expression* on the *Sequence Flow*. The sequence flow syntax reads as follows: if no condition (diamond shape) evaluates true, the default sequence flow is taken (diagonal dash; Figure 4). Since the pattern does not change the input message, the output data objects could be neglected (syntactically) during modeling.
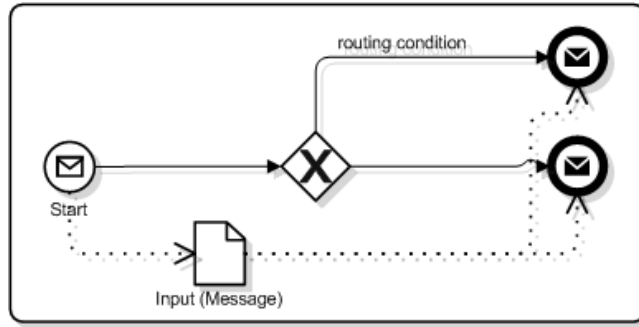
Table 4. Content-based Router Pattern.

Summary 1:n message cardinality, 1:n channel cardinality (same outbound message formats), non-message generating, stateless, routing condition.
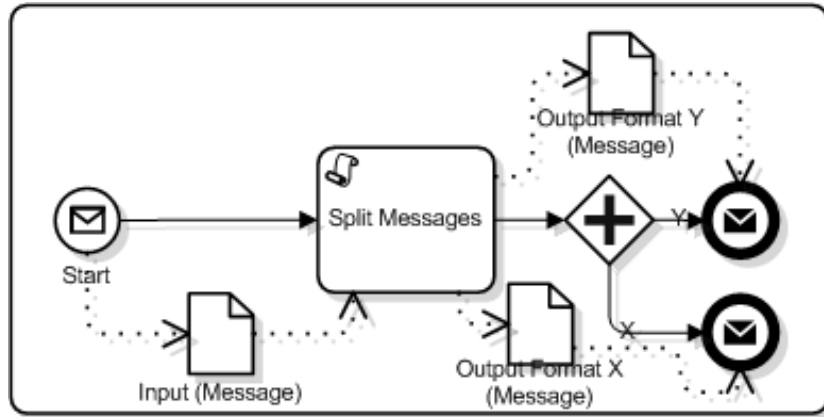

Fig. 8. Message Splitter Pattern with differing output messages.

Table 5. Message Filter (237)

Pattern	Message Filter (237)
Model	The *Message Filter* is a routing pattern with exactly one outbound channel. In case the content of the incoming message matches the criteria specified in the filter condition, the message is routed to the output channel. Else the message is discarded. The filter is stateless.
Syntax	The message filter receives one input message as *Data Object* and uses a static filter condition (again specified as *Script Task*) to emit only those messages sufficient to the condition. Figure 6 shows the pattern with a default message channel and a conditional one that does not pass the filtered messages further.
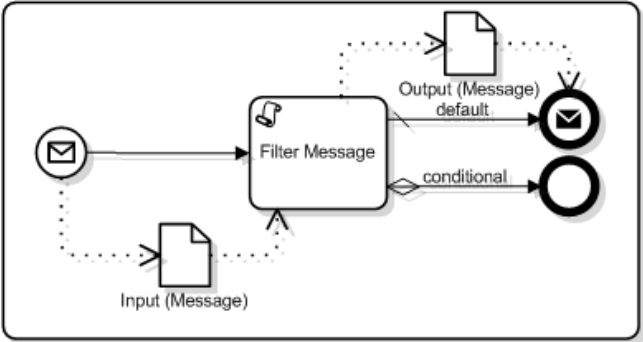
Table 6. Message Filter Pattern.

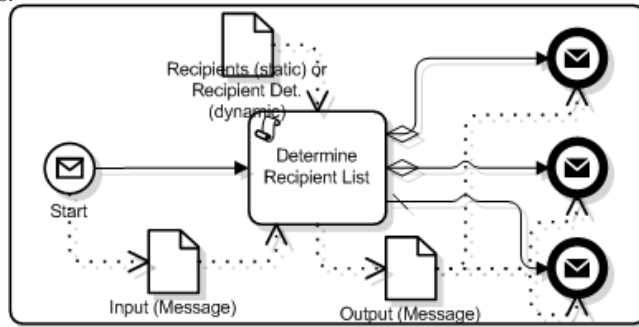Summary 1:1 message cardinality, 1:1 channel cardinality, non-message generating, stateless, filter expression.
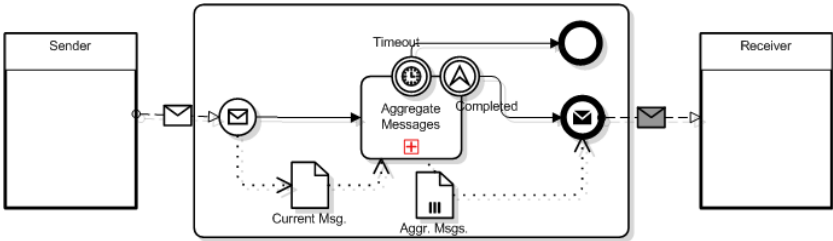
Fig. 9. Aggregator Pattern (collapsed).

Table 7. Recipient List (249)

Pattern	Recipient List (249)
Model	The *Recipient List* emits an incoming message to several receivers by copying in the original message. In contrast to the *Publish/Subscribe* pattern the recipient list controls the determination of the receivers. The recipient list consists of two configurable parts: the receiver determination and the distribution of the message copies to the receivers, while the message remains unchanged. The receiver determination can be statically defined, contained in the message or computed from the message. If the list of recipients is part of the message, it can be removed from the message to hide it from the receivers, which would be done by adding a *Message Translator* before the sending step. The recipient list processes outbound messages in order of the output channels (i. e., no parallel sending). With that characteristic, prioritized outbound processing can be modeled.
	For robustness reasons (e. g., transactional processing), the recipient list can be stateful (default: stateless). The pattern has to ensure that the incoming message is successfully delivered to all recipients. In other words this operation has to be atomic and restartable to complete all incomplete operations. To achieve that, the recipient list (a) can use transactional outbound channels on which the messages are put successfully before committing them, which guarantees an "all or nothing" behavior (*Single Transaction*), (b) can store the information about the sending status for the restart of incomplete messages after an incident (*Persistent recipient list*), or (c) can resend all messages– independent of their status, which requires all recipients to be *Idempotent Receivers*. In case of (c), the *Message Filter* pattern can be used to eliminate duplicates, which would make a receiver idempotent. For (b) the recipient list must be configured stateful.
Syntax	The recipient list emits copies of incoming messages (*Data Object*) to a list of statically or dynamically specified recipients. Figure 8 the pattern with a data object representing the static list of recipients and several conditional receivers.

Table 8. Recipient List Pattern.

Summary 1:n message cardinality, 1:n channel cardinality, message generating (copy), stateless (or stateful for robustness reasons), recipient list (incl. mapping to receivers) or receiver determination program, ordered processing.

Table 9. Splitter (259)

Pattern	Splitter (259)
Model	A *Splitter* breaks one original message into multiple (smaller) messages. For that, the splitter creates as many new messages as the split results to. Common repeated elements (e. g., correlation identifiers, body elements) are duplicated and added to each new message. Per default, the resulting messages are ordered by the sequence they are processed. In general, the splitter can emit messages ordered or unordered. The default ordering is determined by the sequence of split operations. A sequence number can be added for later resequencing (i. e., Resequencer Pattern (283) or aggregation (i. e., Aggregator Pattern (268)). Adding the original message id to the produced messages as Correlation Identifier (163) could help tracing messages and is useful for Request-Reply pattern (154). A splitter that is able to split tree structures into sub-trees and for which the number of generated messages is unknown, but those messages always have the same structure, is called *Iterative Splitter* (Figure 7). In contrast, a *Static Splitter* splits a message structure into a fixed number of parts, while knowing the schema. The structure of the generated messages can differ (Figure 8). Originally, the splitter is a stateless pattern (i.e, the messages are not stored during processing). The splitter could be modeled as multi-cast channel followed by a set of content filters on the outgoing channels. In the case of a static splitter, the message structure of the outgoing messages can vary.
Syntax	The static and the iterative splitters break a message into smaller messages according to a split expression either provided as additional *Data Object* (not shown) or by using a *Script Task*. The script is platform/ engine specific and is executed when the task is performed. The structure of the input and output messages are again defined by a data object. If the message format is the same for all outgoing messages, the data object representing the output messages is configured as *Data Object Collection* (Figure 7). In contrast to [14], the number of output channels is assumed to be one, thus it cannot be parameterized. However, *Message End Events* in BPMN cannot emit messages of different type. In case of different message formats, a "Parallel Gateway" is used to emit the messages to different end events depending on its format. Therefore in our syntax, several end events and thus several outbound channels are addressable.
Summary	1:n message cardinality, 1:1 channel cardinality (1:n in case of differing outbound message formats), message generating, stateless, correlation identifier, split expression.

Table 10. Aggregator (268)

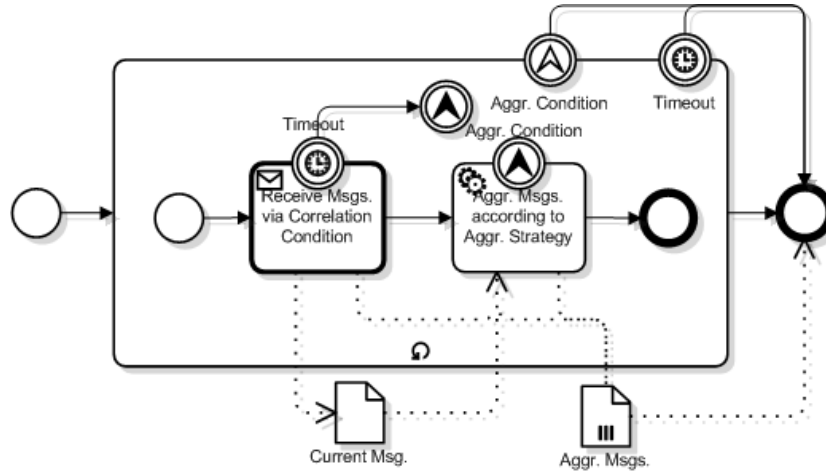| Pattern	Aggregator (268)
Model	An *Aggregator* combines single, but related, incoming messages to a new, complex message. For that, it receives a stream of messages and identifies correlated messages, until a complete set of correlated messages has received. The aggregator applies an aggregation algorithm and emits a single message. An aggregator is a stateful pattern (e. g., for checking the completeness condition), however, not all messages have to be stored completely, but only information relevant for the aggregation. The aggregator consists of at least the following parts: (i) the aggregation algorithm, (ii) the completeness condition, (iii) the correlation function with Correlation Identifier (163) in one or multiple fields in the body and header of the messages, and (iv) a aging/ removal strategy for the stored information. For the aggration (i), the aggragator holds a list of aggregates. New messages/ information is either added to existing aggregates or create new ones (no prior knowledge required: self-starting aggregator). If messages arrive that belong to an already closed aggregate and the opening of new aggregates shall be prevented, the closed aggregates have to be stored. The closed aggregates list should be purged periodically (should not grow infinitely). For instance, Message Expiration (176) [15] could be used for that. For the purge algorithm, not the complete messages have to be remembered, but only the close information. A Control Bus (540), could help to allow manual purging. The following types of aggregation algorithms have to be considered: select the best answer, condense data, and collect data for later evaluation (i. e., simply store data if the aggregator cannot make a decision). For the completeness condition (ii), the following *Aggregation Strategies* have to be considered: wait for all, timeout, first best, timeout with override (break faster and override), external event (e. g., Control Bus). A *Static Aggregator* always aggregates a fixed number of messages. In contrast to [2,15], only one incoming message channel is considered. The aggregator is the inverse of the Splitter Pattern (259)
Syntax	The high-level syntax of the aggregator pattern is shown in Figure 9. The first message that arrives instanciates an aggregator *Sub-Process* as *Activity* (collapsed). All subsequent messages are send to that sub-process directly. An *Escalation Intermediate Event* is used to send one aggregate to the receiver. The aggregation can timeout through a *Timer Intermediate Event*, which stops the execution. The expanded activity is shown in Figure 10. The "loop-wait" nature of the pattern is indicated by the "cyclic arrow" on the bottom of the sub-process. The structures of the incoming messages and the outgoing (aggregated) messages are specified via a *Data Object*. The correlation function and the correlation identifier (iii) to correlate different messages to one are defined by an *Activity* in the *Sub-Process* definition. The completeness conditions (ii) are represented by an associated *EventDefintion* of type *Interrupting Escalation StartEvent* (e. g., for wait for all, first best strategies) or of type *Interrupting Timer StartEvent* (e. g., timeout (with override), completion interval). The aggregation algorithm (i) is again denoted by an activity that specifies the named algorithms (e. g., a greater-equals operation on a special element of the message payload for a best-answer algorithm). In addition, a message translation algorithm can be specified to combine the incoming messages to one has to be provided. Open topics: external event and connection to the control bus.
Summary	n:1 messages (not channels), message generating (combines several original messages to one new message), stateful, aggregation algorithm, completeness condition, correlation function, correlation identifier, massege translation program.

Fig. 10. Aggregator Pattern (expanded).

Table 11. Resequencer (283)

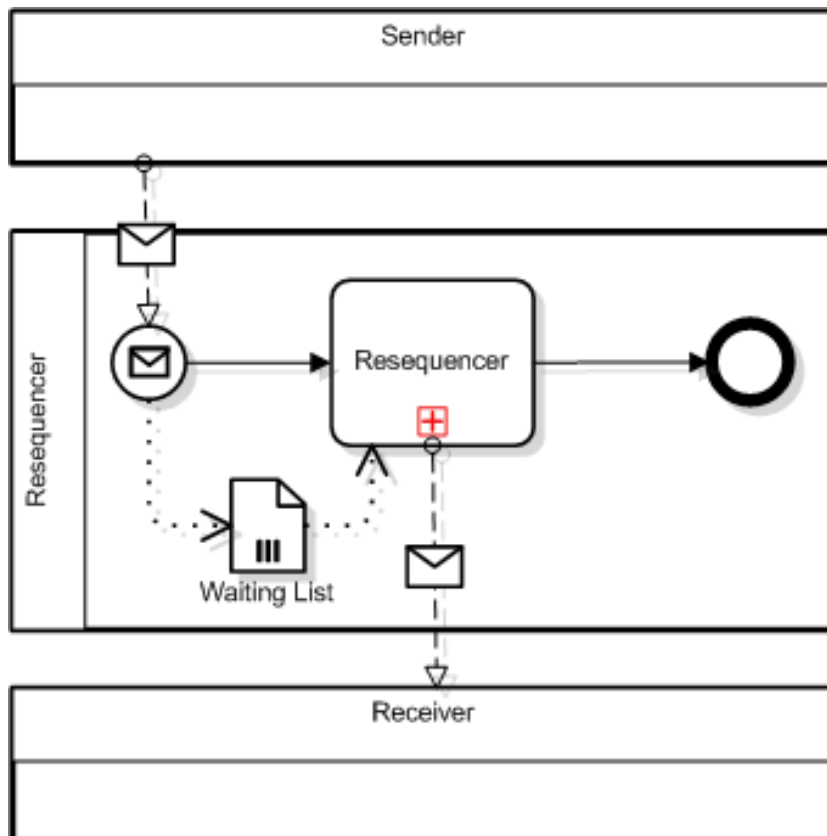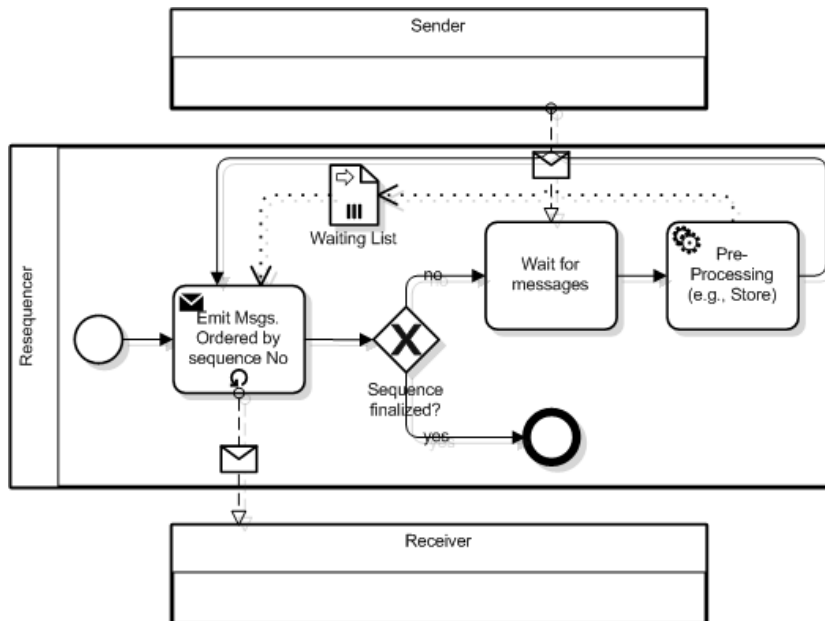| Pattern	Resequencer (283)
Model	The *Resequencer* orders a stream of un-ordered messages according to a sequence number and emits the messages to the output channel in the proper sequence. The concept of a unique sequence number has to be attached to each message. Hence the in-order sequence numbers distinguish from *Correlation Identifiers* and message identifiers. To handle cases, in which messages arrive unordered, the resequencer has to be stateful. These stored messages have to be checked for the completeness of the sequence to decide on when to order and emit the messages. When the ordered set of messages was successfully sent, the stored messages have to be removed from the resequencer's message store.
Syntax	The resequencer takes a set of messages as *Data Object Collection* and orders them according to a specified sequence that is emitted represented by a *Data Object*. Figure 11 shows the resequencer as collapsed *Sub-Process*. When expanding the sub-process (Figure 12), an activated *Activity* receives the messages and stores them in a store (*Service Activity* associated to a *Data Object*). When the complete sequence has been received, another activated activity takes the list of messages and emits them in order. In contrast to the other patterns, the resequencer in BPMN has an impact on the complete syntax in one participant. If the integration logic does not only re-orders messages, other steps have to be added as pre-or post processing to Figure 12.
Summary	1:1 messages (different order), non-message generating, stateful, sequence number, sequence complete condition, order operation.

Fig. 11. Resequencer Pattern (collapsed).

Fig. 12. Resequencer Pattern (expanded).

Message Transformation Since sender and receiver applications rarely agree on a common data format, messages have to be changed within the messaging (channel). The following patterns are subsequently discussed:

1. Content Enricher (336)
2. Content Filter (342)
3. Claim Check (346)

Table 12. Message Translator (85)

Pattern	Message Translator (85)
Model	A *Message Translator* converts an incoming message (format) into a data format expected by the corresponding receiver of the message. The concrete transformation logic is represented by a mapping definition that allows different levels of transformations, at least: (a) data structure (e. g., n:m structure mappings like projections), (b) data (e. g., value mapping), (c) data types (e. g., date/time conversion), (d) data representation (e. g., encryption, compression, data formats like XML, JSON), and (e) the transport protocol (e. g., TCP/IP). The translator is stateless and does modify the original message.
Syntax	The incoming message (format), outgoing message (format) as well as the mapping logic are specified by a *Data Object* (Figure 13).
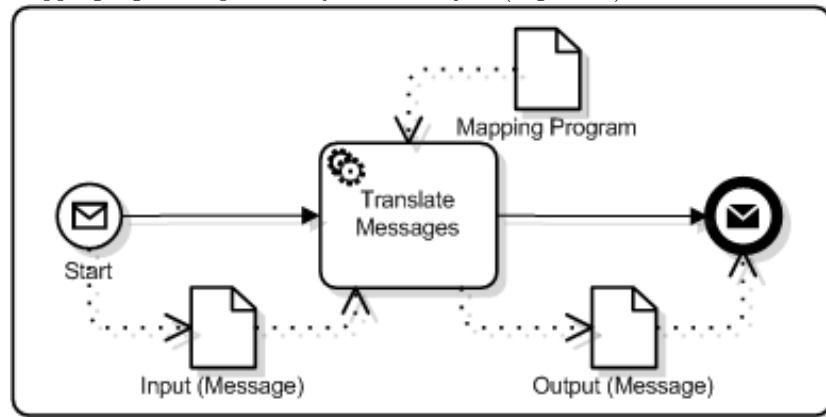
Table 13. Message Translator Pattern.

Summary 1:1 message cardinality, non-message generating, stateless, mapping program.

System Management The operation of a messaging system is equally challenging as its implementation itself. For that, some patterns have been compiled by [3], which are discussed subsequently. The following patterns are subsequently discussed:

1. Wire Tap (547)
2. Message Store (555)

Table 14. Content Enricher (336)

Pattern	Content Enricher (336)
Model	The *Content Enricher* allows to add information to an incoming message. The additional content can be assigned locally, i.e., determined within the pattern, or can be requested from external participants. Hereby, the locally assigned information can come from a computation on the current message or from the runtime environment. The content enricher is stateless and generates a new message from the aggregation of the input message and the additional information. The message cardinality is 1:1, while the outgoing message might have a different format than the incoming message.
Syntax	The content enricher uses the synchronous *Request-Reply* pattern (i.e., using a *Service Task*) to fetch data from an external participant and merges it with the incoming message represented as *Data Object* (Figure 15). Hereby, the service task specifies the operation that is used to create the request (incl. mapping) and the assigned *Message Flow* specifies the (remote) endpoint (e.g., transport-, message protocol).
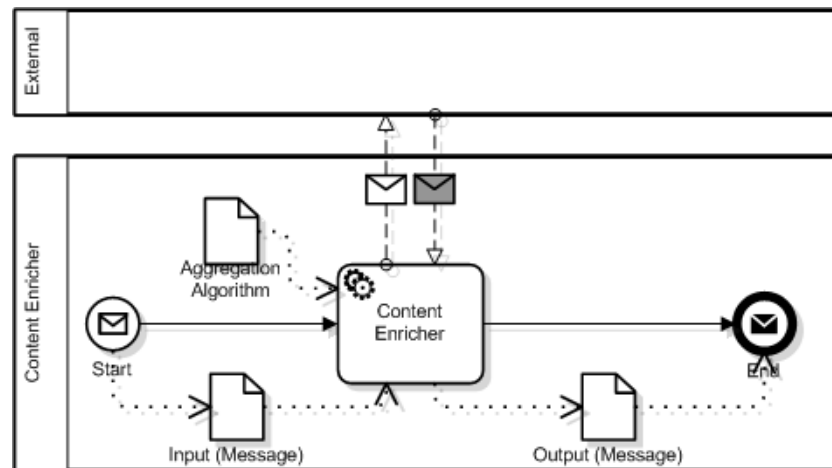
Table 15. Content Enricher Pattern.

Summary 1:1 message cardinality (format might change) and message channels within the process and 1:1 message flow channel to the external service, message generating (passes response), stateless, request mapping program, response aggregation/ enrichment.

Table 16. Content Filter (342)

Pattern	Content Filter (342)
Model	The *Content Filter* removes parts of the message not required by the recipient, or it simplifies the data structure through, e. g., thinning, flattening complex structures via projection according to its filter condition. Alternatively, the filter condition can come from an *External Service*. A filter is a stateless pattern, which works on the original (incoming) message (i. e., does not generate new messages). A content filter is a special variant of a Message Translator, while not translating the whole message, but only the required parts. A sequence of content filters can be used as static splitters.
Syntax	The content filter (shown in Figure 17) specifies a filter condition, the structure of the incoming and the outgoing message as *Data Object*.
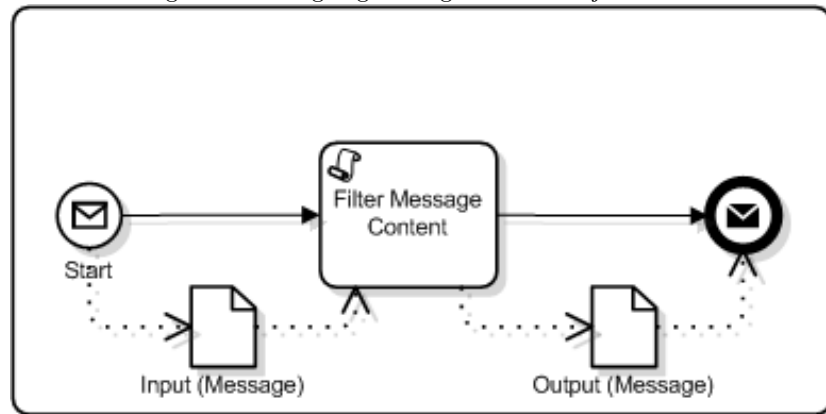
Table 17. Content Filter Pattern.

Summary 1:1 messages, non-message generating, stateless, filter condition.

Table 18. Claim Check (346)

Pattern	Claim Check (346)
Model	The *Claim Check* allows to model a "call by reference" for the complete payload of a message or only parts of it. The data from the message payload is removed temporarily from the message (e. g., if it is not relevant for the next-n processing steps) and can be added to the message later. For that, a filter condition is used to extract the data from the message and a unique key is generated under which the data is stored in a data store. The key is added to the message (e. g., as header field) and the message is sent to the channel. Later the key is used to "claim" the data from the data store and a local *Content Enricher* is used to add the data to the message. The content enricher has to take care that the "claim check" key is removed from the message afterwards. This pattern can be seen as the counterpart to a *Content Enricher*.
Syntax	The claim check frees an incoming message (*Data Object*) from its payload by storing the payload in a *Data Store*, while only passing the message without its payload and a claim check as data object to the next flow element (Figure 19). To retrieve the payload, a local *Content Enricher* as *Script Task* (from store; normally *Service Activity*) gets the claims check and the corresponding payload from the data store.
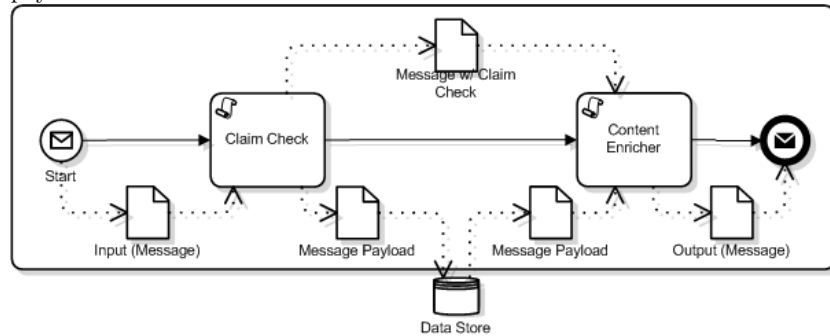
Table 19. Claim Check Pattern.

Summary 1:1 messages, non-message generating, stateful, filter condition, local *Content Enricher* (from store).

Table 20. Wire Tap (547)

Pattern	Wire Tap (547)
Model	The *Wire Tap* allows access to messages in a point-to-point channel. The wire tap is comparable to a static recipient list with two recipients and two outgoing message channels. While the first channel is the normal message channel, the second channel transports a copy of the original message for analysis or storage. Typically the extra channel is a point-to-point channel that persists the message in a *Message Store*.
Syntax	Figure 21 shows the usage of a wire tap with a point-to-point channel that persists the message in a message store. The input, output and copy of output messages are specified by a *Data Object*. The static routing is defined by an *Parallel Gateway*, which is a variant to the syntax shown for the recipient list pattern. In the example, the channel the data flow ends with the storage of the message.
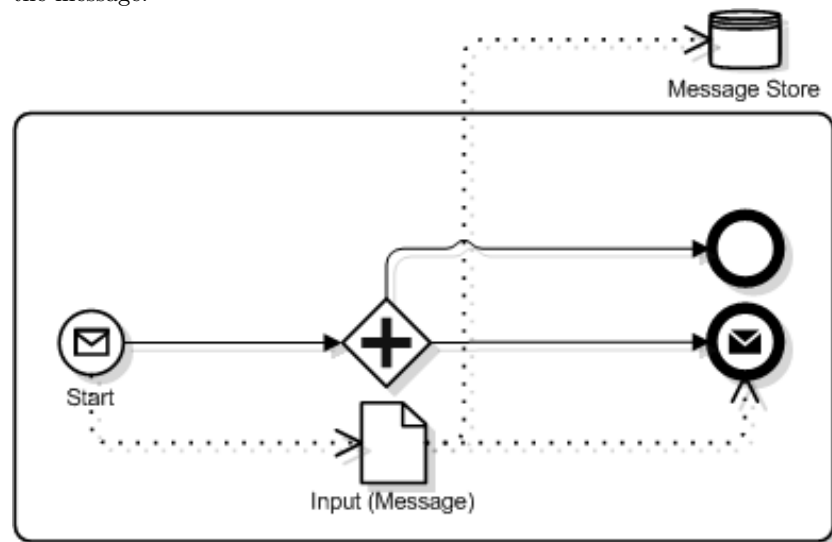
Table 21. Wire Tap Pattern.

Summary	1:1 message cardinality, 1:2 message channels, message generating (copy), stateless.

Table 22. Message Store (555)

Pattern Message Store (555)
Model	The *Message Store* allows operations on messages like persist (insert), update, delete and query. The message store can be any kind of (No)SQL, in-memory or file-based storage for arbitrary message storage formats (e. g., BLOB, XML, JSON). Many persistent pattern (e. g., *Aggregator*, idempotent receiver) use the store to guarantee the correct integration semantics. Other non-persistent patterns (e. g., wire tap) or integration semantics (e. g., *Guaranteed Delivery*, retry handling on channel level) can be assigned.
Syntax	Following the process model definition, associated *Data Objects* define the flow of data through a message channel. The same mechanism is used to assign the message store (i. e., represented as *Data Store*) to elements within a *Process* via *Data Association*. Since data stores are defined beyond the scope of a process in BPMN, the store can be associated by arbitrary many elements from different processes and can thus denote local or shared storages.

Additional Patterns Not all patterns in scope of this document have been covered by [3]. However, these patterns have been proposed and were partially taken up for implementation by [14,1]. The following patterns are subsequently discussed:

1. Join Router
2. External Service
3. Multicast

4 Concluding Remarks

The Enterprise Integration Patterns are a set of widely used patterns denoting the building blocks for a structured implementation of a messaging system. For these patterns we have continued the work of [15,16] by specifying a syntactic mapping to the Business Process Model and Notation (BPMN); thus each pattern is a set of elements in a BPMN Process, which can be composed to sets of message channels from the senders to their receivers. Together with the syntax, we defined a process model for its semantics (referred to as model). Although the syntax is compliant to BPMN, the BPMN execution semantics are replaced by a process model, we slightly adapted to the integration domain. The result is an integration domain specific language, with which integration aspects of messaging systems and their runtimes can be expressed.

Acknowledgements

<div align="center">**Table 23.** Join Router</div>

Pattern	Join Router
Model	The *Join Router* is a structuring element that combines several different incoming channels to one outgoing channe. Received messages from different channels are passed while not executing any operation on them. The join router is stateless. It is assumed that the formats of the incoming messages has to be the same (e. g., enforced by pre-processed *Message Translators*).
Syntax	The join route combines a several channels to one without specifying a merge condition. Figure 24 shows the usage of an *Inclusive Gateway* that is activated if at least one incoming flow passes the control to the gateway and emits a message if an incoming message via *Data Object* is received. In other words, all incoming messages are simply routed to one channel.
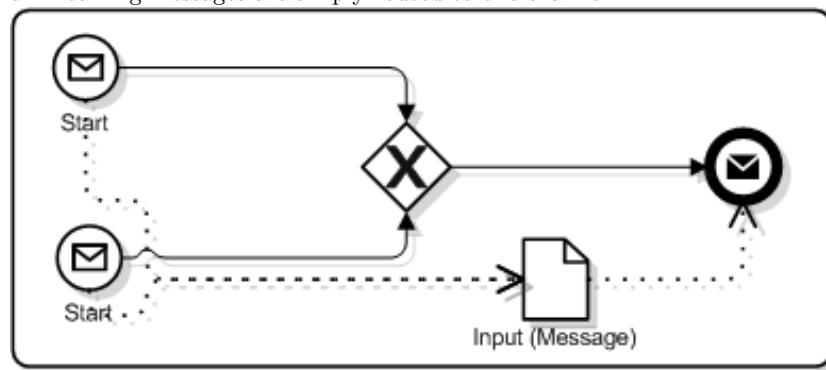
<div align="center">**Table 24.** Join Router Pattern.</div>

Summary 1:1 message cardinality, n:1 message channels, non-message generating, stateless.

Table 25. External Service

Pattern	External Service
Model	The *External Service* is a building block for all patterns that require external data (e.g., *Synch-Asynch Bridge*, *Asynch-Synch Bridge*, *Content Enricher*). The pattern is stateless and has an internal and external message and channel cardinality of 1:1, respectively.
Syntax	Figure 26 shows an external service that receives request messages defined through a *Data Object* and emits the response message in the same way. The request is passed via *Message Flow* to the external endpoint and the response from the endpoint is set on the channel for subsequent processing.
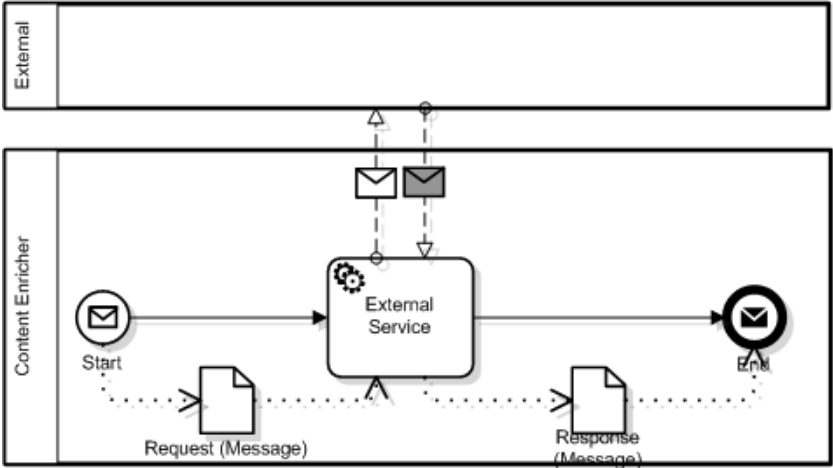
Table 26. External Service Pattern.

Summary	1:1 message cardinality (format might change) and message channels within the process and 1:1 message flow channel to the external service, message generating (passes response), stateless.

Table 27. Multicast

Pattern	Multicast
Model	The *Mutlicast* routes one incoming message to several receivers in parallel. For that, the incoming message is copied to multiple output channels. In contrast to the *Recipient List* the receiver determination has to be (statically) configured and is not determined based on the incoming messages. The multicast is stateless and will continue processing messages to the separate output channels even if one of them fails. To reach the same robust messaging capabilities as the recipient list, the same configurations can be done (see Section 3.1 (a)–(c)).
Syntax	The multi-cast takes one incoming messages as *Data Object* and emits several copies of the original message to the recipients via data objects. Figure 28 shows the syntax of the multi-cast using a *Parallel Gateway*.
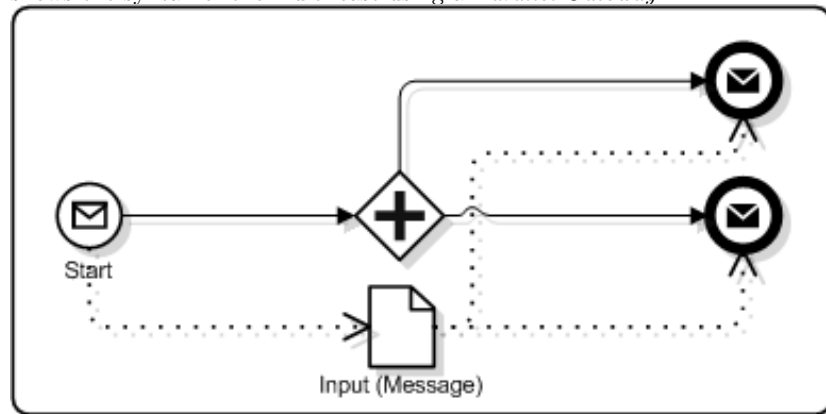
Table 28. Multicast Pattern.

Summary 1:n message cardinality, 1:n message channels, message generating (copy message with new message identifier), stateless (similar to *Recipient List*), behavior in exceptional cases (e. g., stop processing).

References

1. J. Anstey and H. Zbarcea. *Camel in Action*. Manning, 2011.
2. D. Fahland and C. Gierds. Analyzing and completing middleware designs for enterprise integration using coloured petri nets. In *CAiSE*, pages 400–416, 2013.
3. G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
4. G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.
5. A. Meyer, L. Pufahl, D. Fahland, and M. Weske. Modeling and enacting complex data dependencies in business processes. In *BPM*, pages 171–186, 2013.
6. A. Meyer, S. Smirnov, and M. Weske. Data in business processes. *EMISA Forum*, 31(3):5–31, 2011.
7. A. Meyer and M. Weske. Data support in process model abstraction. In *ER*, pages 292–306, 2012.
8. D. Müller. *Management datengetriebener Prozessstrukturen*. PhD thesis, 2009.
9. D. Müller, M. Reichert, and J. Herbst. Flexibility of data-driven process structures. In *Business Process Management Workshops*, pages 181–192, 2006.
10. D. Müller, M. Reichert, and J. Herbst. Data-driven modeling and coordination of large process structures. In *OTM Conferences (1)*, pages 131–149, 2007.
11. O. M. G. (OMG). Business process model and notation (bpmn) version 2.0. Technical report, jan 2011.
12. C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut fr instrumentelle Mathematik der Universitt Bonn, 1962.
13. I. Raedts, M. Petkovic, Y. S. Usenko, J. M. E. M. van der Werf, J. F. Groote, and L. J. Somers. Transformation of bpmn models for behaviour analysis. In *MSVVEIS*, pages 126–137, 2007.
14. T. Scheibler. *Ausführbare Integrationsmuster*. PhD thesis, 2010.
15. V. Stiehl. *Composite Application Systems - Systematisches Konstruieren von Verbundanwendungen unter Verwendung von BPMN*. PhD thesis, TU Darmstadt, September 2011.
16. V. Stiehl. *Prozessgesteuerte Anwendungen entwickeln und ausfhren mit BPMN: Wie flexible Anwendungsarchitekturen wirklich erreicht werden knnen*. dpunkt.verlag GmbH, 2012.
17. W. M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.
18. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

List of Patterns