Experiences with Business Process Model and Notation for Modeling 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 best practices for integrating enterprise applications. However, a formal integration model is missing, such as *Business Process Model and Notation* (BPMN) from the workflow domain. There, BPMN is a "de-facto" standard for modeling business process semantics and their runtime behavior.

In this work we present the mapping of integration semantics represented by EIPs to the BPMN syntax and execution semantics. We show that the resulting runtime independent, BPMN-based integration model can be applied to a real-world integration scenario through compilation to an open source middleware system. Based on that system, we report on our practical experiences with BPMN applied to the integration domain.

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

1 Introduction

Integration middleware systems address the fundamental need for application integration by acting as the messaging hub between applications. As such, they have become ubiquitous in service-oriented enterprise computing environments in the last years. These systems control the message handling during service invocations and are at the core of each Service-Oriented Architecture (SOA) [8]. Since their implementation and operation remains challenging, best practices for building those systems, called *Enterprise Integration Patterns* (EIP), were collected by [7]. Later other practitioners (e. g., [18,1]) and researchers (e. g., [17]) added further patterns. Although these patterns describe typical concepts in designing a messaging system, they cannot be considered a modeling language. A modeling language would allow for the formal, runtime independent specification of integration scenarios and verification.

More precisely, the requirements that are important for developing integration systems, however, not covered by current approaches like the EIPs are collected subsequently. The EIPs propose a visual notation, which allows composition of patterns, while the notation does not specify a semantic model for integration (*REQ-1*: Define a semantic model for message-based integration as foundation

J. Cabot and J. Rubin (Eds.): ECMFA 2014, LNCS 8569, pp. 254–266, 2014.
© Springer International Publishing Switzerland 2014

of a Domain-specific Language (DSL) for integration). A semantic model for integration shall cover a human and computer readable, syntactical notation (*REQ-2*: Specify the syntax) as well as a behavioral runtime specification, which shall be independent of the specific runtime platform implementations (*REQ-3*: Define a platform independent behavioral semantics). The integration DSL shall consider the control flow (*REQ-4*: Support control flow modeling), similar to previous work on Coloured Petri Nets [4] that is used for verification of the EIPs' control flow [6], as well as the data flow for message exchange (*REQ-5*: Allow for data flow modeling for message exchange). The formal integration model shall allow for validation of integration programs and the verification of runtime systems (*REQ-6*: Validate programs and verify the runtime systems).

In this paper, these shortcomings (cf. *REQ*) are addressed by proposing a language for message-based integration grounded on a standard from the related workflow domain, called *Business Process Model and Notation* (BPMN) [15]. For instance, Figure 1 shows an asynchronous integration scenario of a corporate with its bank and business monitoring via SAP Cloud to Cash[1] (CTC), syntactically expressed in BPMN according to the definition proposed in this paper. The incoming message is of type "FSN" (short for Financial Services Network[2]), which has to be translated to its canonical data model incarnation "FSN:CDM" for further processing, 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 sent to the CTC application, which tracks the message exchange from a business perspective.

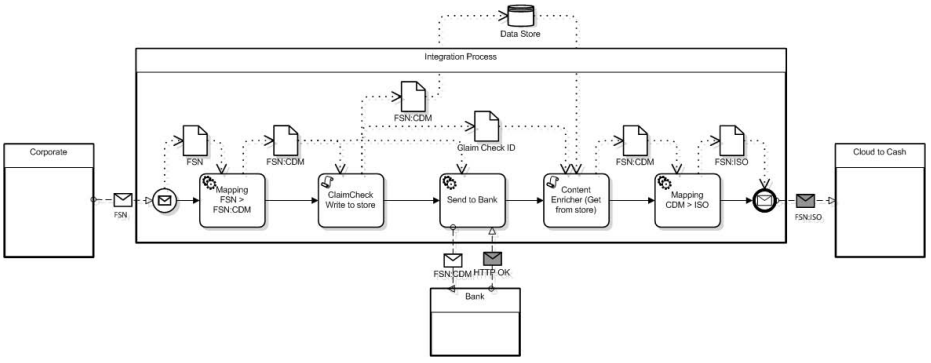


Fig. 1. Business Monitoring: Messages sent from Corporate to Bank are routed to SAP Cloud to Cash (CTC) for monitoring purpose (cf. [16])

The contribution of this paper is the syntactic and semantic formalization of common integration patterns using BPMN (cf. *REQs-1–3*). Due to the manifold collection of patterns, we focus on some hand-picked, core patterns. The complete list of pattern to BPMN mapping can be found in supplementary material [16]

[1] `http://www.sap.com/pc/tech/cloud/software/cloud-applications/index.html`
[2] `http://scn.sap.com/docs/DOC-40696`

(not mandatory). With the EIPs as business process building blocks, integration semantics can be expressed as implementation independent BPMN syntax. From this formal model, we show the realization of the sample integration scenario in Figure 1 to Apache Camel [1], which is a widely used, open-source system for message-based integration and event processing [5] (cf. *REQs-4–5*). Based on the interaction with customers and integration domain experts, we report on experiences with our modeling approach and discuss advanced integration modeling techniques.

Section 2 discusses the contribution of the paper in the context of related work. Section 3 introduces general integration semantics and defines the syntactic and semantic mapping from selected EIPs to BPMN. In Section 4 we apply our approach to an open-source ESB for the "business monitoring" scenario. In Section 5 we share our experiences, before concluding in Section 6.

2 Related Work

The patterns described by [7,17,18,1] 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) [15], the *Workflow Patterns* defined by [20] or *Service Integration Patterns* [2]. Enterprise Integration Patterns (EIPs) complement these notations by a set of typical designs found in a messaging infrastructure.

Processes and Data. The 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" [10] shows that besides *Configuration-based Release Processes* (COREPRO) [14,12,13], 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* [19] nor Petri nets do support data modeling at all [10]. For example, the work on the EIPs' control flow uses *Coloured Petri Nets* [4], which are used for verification of composed EIPs [6]. Based on the work on control and data flow, BPMN was further evaluated by [9,11] with respect to data dependencies within BPMN processes, however, not towards a combined control and data flow as in our approach.

Process Languages for Integration. The work builds on this foundation and combines it with executable integration patterns, their configuration and mapping to the *Web Services Business Process Execution Language* (WSBPEL) proposed by [17] and leverages the work of [18] that started to map the EIPs to the BPMN syntax and some semantics by example. In this document, we provide a systematic continuation of this work by defining a comprehensive syntax and model for widely-used patterns.

3 The BPMN Integration Pattern Language

Before defining the mapping of some selected EIPs to BPMN, we discuss the relevant BPMN syntax (mainly taken from the *BPMN Collaboration Diagram* [15]) directly in the context of core integration concepts.

3.1 Core Integration Concepts and BPMN

The main syntactical artifacts in BPMN denote process steps, sequences and the representation of messages that are exchanged between processes during runtime. The core concepts of message-based integration are *Message*, *Message Channel* and *Integration Adapters* (cf. *Message Endpoint*) [7].

A message is informally defined as a piece of information to be exchanged between sender and receiver. This information can be a piece of data (i. e., EIP *Document Message*), a command for execution (i. e., EIP *Command Message*), or an event for logging (i. e., EIP *Event Message*). This notion is shared by BPMN, in which the sender and receiver applications are *Participant* elements. In BPMN a participant may have internal details, in the form of an executable process.

The connection between sender and receiver participants is called message channel, which is the fundamental infrastructure of a messaging system. For example, there are EIP *Point-to-Point Channels*, connecting exactly one sender with one receiver, and one-to-many channels like EIP *Publish-Subscribe* or *Broadcast/Multicast*. A message sent to such a channel can be received by multiple receivers, while for n receivers, n copies of the original message have to be provided. In general, channels specify non-functional qualities like the *Quality of Service* (QoS; best-effort, exactly once), *Message Exchange Pattern* (MEP: InOnly or InOut), and *Capacity* (e. g., maximum message size). For instance, a file poller acts one-way (InOnly), since it cannot handle response messages and can be configured to ensure the delivery of messages (exactly once).

On the other hand, most document message exchange works according to the *Request-reply Pattern*, which specifies a two way communication (InOut). This corresponds to the *Process* flow in BPMN, which is controlled by a combination of flow objects (e. g., events, activities, gateways) and connections. We consider BPMN *Start Event* and *End Event* that initiate the process flow with a "message-receive" semantic or terminate the flow, thus terminates a process or part of a process, after having sent the message. The BPMN throwing/catching *Intermediate Event* is used to express message events, errors, and timed message processing. The BPMN *Activity/Task* represents process steps that allow to manipulate a message within the channel and has to be executed, before the flow can proceed. BPMN *Gateway* elements are able to handle multiple process flows, where they route or fork the flow. For that, BPMN *Sequence Flow* definitions connect flow elements within a channel. Besides the control flow, the message channel requires a data flow, which is expressed as a sequence of BPMN *Data Object* and BPMN *Data Store* definitions that are associated to the flow elements. More formally, these BPMN execution semantics for messaging are

defined as process model in Definition 1. In a nutshell, 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 in form of a *Token* to the next flow element in the process (e. g., activity, gateway, event) and puts it into ready state. The data flow is handled by associated data objects from one element to the next one. More precisely, a flow element in the ready state gets activated, if all associated data is supplied, and executes its inherent behavioral semantics on the data (e. g., script, service call). The process ends through the invocation of a message end event firing the outgoing message before the process context stops.

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 ExclusiveGateway 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`*.*

Finally, a message endpoint connects an application to a messaging system. In BPMN, the *Message Flow* specifies message exchange (e. g., process status information, error messages, data) between participants or participants and process elements. When mapped to messaging systems, the message flow 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. The model that describes the complete system for an integration flow is specified in Definition 2.

Definition 2 (Integration Flow). *An* integration flow $IFlw = (CO, PO, MF)$ *consists of a* Collaboration CO *containing a finite non-empty set of* Pool $PO \subseteq P \cup M$ *of* Participant P *and process model* M *(cf. Definition 1), where* $M \subseteq P_{int}$ *and* P_{int} *is the participant referencing the (integration) process steps, and the* Message Flow *relation for the sender* $MF_S \subseteq P_s \times (P_{int} \cup E_s)$ *and the receiver* $MF_r \subseteq P_s \times (P_{int} \cup E_e)$*, where* P_s *denotes an arbitrary amount of sender participants,* E_s *and* E_e *represent sets of start and end events of an integration process* P_{int} *and* P_r *denotes the set of receiving participants. The message* MS *is part of the message exchange from sender to receiver via the* MF *relation.*

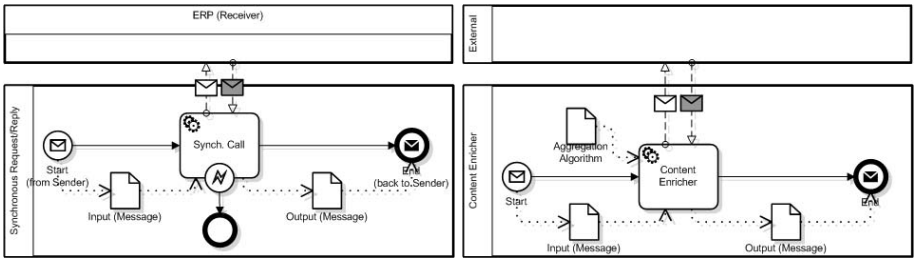
3.2 Mapping Enterprise Integration Patterns to BPMN

The definitions of a *Process Model* and an *Integration Flow* (IFlow) are used to map some selected EIPs to BPMN. We have selected basic integration patterns

like *Request-Reply*, *Content Enricher*, the two antipodes *Splitter* and *Aggregator*, and the *Message Translator*. The other patterns from the literature we covered in [16] as non-mandatory supplementary reading.

The basic message exchange patterns are one-way or two-way communication on an end-to-end IFlow level. These patterns are fundamental, since they let the sender participant communicate synchronously (InOut) or asynchronously (InOnly). Figure 2 (a) shows a two-way integration flow in BPMN using a synchronously "waiting" *Service Task*. Definition 3 specifies the pattern's runtime behaviour. The definitions of the subsequently discussed patterns and the "business monitoring" integration scenario can be expressed in the same way, however, is informally described due to brevity.

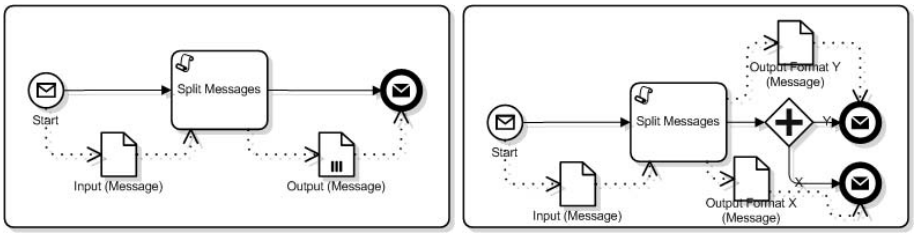
Definition 3 (Request-Reply (synchronous)). *The control flow* CF *is defined as* $CF = E_s \times ServiceTask \times MF_{req} \times P_r \times MsgFlow_{res} \times ServiceTask \times E_e$, *where the set of ServiceTask of type A, while the process instantiation* `constructor`, E_s *and termination* `destructor` $= E_e \cup E_{err}$, *where* E_{err} *denote error events. The data flow* DF *is* $E_s \times DO_{in} \times ServiceTask \times MF_{req} \times P_r \times MF_{resp} \times ServiceTask \times DO_{out} \times E_e$, *where* MF_{req} *and* MF_{resp} *denote message flows for request, response, respectively. If an exceptional situation occurs during the execution of the ServiceTask, a separate channel* Exc *is instantiated to handle the error:* $Exc = ServiceTask \times E_e$.



(a) Request-Reply (b) Content Enricher

Fig. 2. Request-Reply Pattern synchronous (a), Content Enricher Pattern (b) (cf. [16])

A pattern that adheres to Definition 3 is the content enricher. The content enricher consumes messages from the channel and merges additional information into the header or body of the original message according to an *Aggreg. Algorithm*, shown in Figure 2 (b). The data can come from local tables or remote services (not shown). The content enricher is non-persistent by default: if any operation during or after the enrichment operation fails, intermediate results of the operation are lost and the operation has to be re-processed from the latest persistence state onwards. The enricher uses a *Service Task* to map the incoming message (Input) to a request understandable to the external participant, waits for the response and aggregates it to the original message resulting to the output message (Output) according to an aggregation algorithm denoted by a *Data Object*.

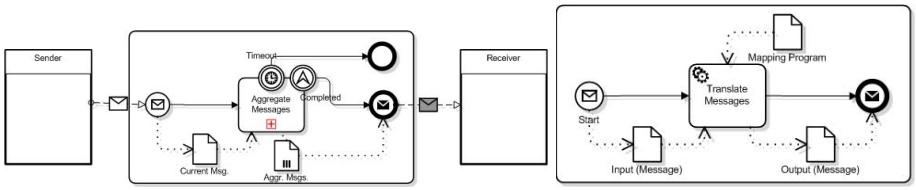


(a) Message Splitter (b) Multiple Formats

Fig. 3. Message Splitter Pattern (a), with differing output messages (b) (cf. [16])

An interesting pair of patterns are the antagonists *Splitter* and *Aggregator*. Both patterns have a channel cardinality of 1:1, however, the splitter is message creating (1:n message cardinality) and the aggregator is also message creating, while the new message is an aggregate of multiple incoming messages (message cardinality n:1) The splitter breaks one original message into multiple (smaller) messages. For that, the splitter creates as many new messages as the split function (*Script Task*) results to. Figure 3 (a) denotes a splitter, whose split results are of the same format. The split function could result to multiple messages of different format. However, in BPMN a *Message End Event* can only handle a single message definition. Hence, for each message with differing format a new control and data flow with dedicated end events is required. Figure 3 (b) shows the usage of a *Parallel Gateway* for that purpose. In contrast, an aggregator receives a stream of messages and correlates them according to a correlation condition *Message Receive Task* (Figure 6). When a complete set of correlated messages has been received, the aggregator applies an aggregation function *Service Task* and publishes a single, new message containing the aggregated result correlated message identifiers for lineage. The aggregator is persistent, because it stores list of aggregates. Completion conditions like *Timer Event* and *Escalation Event* are used to end the aggregation, e. g., with the following strategies: wait for all, wait for first best, timeout. The outer workings of the aggregator are shown in Figure 4 (a). The first message instantiates a stateful aggregator that can only be ended through its completion conditions or an exceptional situation. The inner workings and the instantiation mechanics are shown in Figure 6 and discussed in detail later.

The message translator, shown in Figure 4 (b), converts an incoming message (format) into a data format expected by its corresponding receiver. Therefore it does not create new messages, but changes the original message. The translator is stateless and has a channel cardinality of 1:1. The message translator is structurally similar to the enricher, while the translator calls an internal mapping program (not shown), instead of an external one.



(a) Aggregator (b) Message Translator

Fig. 4. Aggregator Pattern (collapsed) (a), Message Translator Pattern (b) (cf. [16])

4 Case Study: BPMN Integration Patterns in Action

We demonstrate the practicability of our integration modeling approach through the application to the "business monitoring" scenario from the *Financial Service Network* domain (cf. Figure 1). The scenario features messages sent from a corporate to one or multiple banks, while all messages are passed to the "Cloud to Cash" application, which correlates the technical messages to their business contexts and provides an overview.

The technical implementation uses a well-known open source integration middleware system called *Apache Camel* [1], to which we "compile" our BPMN-based IFlow definitions. In Apache Camel the basic concepts are implemented in a proprietary way. A message consists of a set of name-value pair headers, a variable body or payload and a set of attachments. The addressable message endpoints are defined within *Component* runtime artifacts, which represent a factory for endpoint objects. The inbound and outbound message adapters are part of the component as *Consumer* (`camel:from`) and *Producer* (`camel:to`) objects. A Camel *Route* realizes a concrete implementation of a channel. The *Camel Context* is the container for runtime services (e. g., mapping program, aggregation algorithm) and (multiple) routes.

Figure 5 illustrates the compilation from an IFlow model to Apache Camel artifacts. For better understanding, we used BPMN *Group* elements, which we annotated with the Camel syntax, to overlay the BPMN integration scenario. Hereby, the integration flow is represented by a `camel:context` with exactly one assigned `camel:route`. Between the `camel:from` inbound the two outbound calls `camel:to uri="cfx:"`, several Camel components `camel:to` are executed. The compilation of this definition results in an executable runtime, if all runtime services are attached to the Camel context.

The execution semantics of Camel differ from Definition 2, since Camel has no separation of data and control flow during the execution of a route and behaves rather like a call stack, i. e., the Camel components remain active during the complete route processing. The BPMN workings involve a token-based state model including associated data that lets activities finish after processing. In this case study we mapped the BPMN to the Camel execution semantics by synthesising the control and data semantics to Camel route processing.

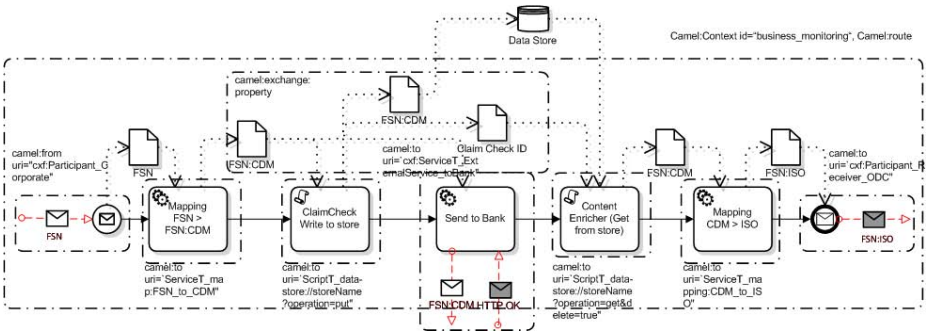


Fig. 5. Translating Business Monitoring: Messages IFlow to Apache Camel. The Camel representation is shown as *Group* overlay for better understanding only. *Message Flow* elements denote Message Endpoint definitions.

Listing 1.1. Message Processing Log for "Business Monitoring (anonymized)"

```
1 Entering CXF Inbound Request{
2     contextName = business_monitoring, MessageGuid =
          iXGKWlUtoQPl0xg..., OverallStatus = COMPLETED,
          ReceiverId = ctc, SenderId = corporate, ...
3         Entering Camel route route27 {
4         Processing exchange ID-0001 in
              To[map:FSN_to_CDM] {...}, To[data-store
              ?op=put] {...}, To[cxf:toBank] {...},
              To[data-store ?op=get&delete=true] {...},
              To[map:CDM_to_ISO] {...}, To[cxf:bean:ODC]
              {...}
5         }
6     }
```

When instrumenting the camel runtime with a technical "Message Processing Log" (MPL) monitoring capabilities, we can follow the message during the route processing. Listing 1.1 shows a shortened and anonymized MPL for our scenario, marking the most relevant steps during the processing of the Camel route. As discussed, the log illustrates the slightly different execution semantics of Camel.

5 Experiences and Limitations

During customer user studies and extensive, hands-on sessions with integration domain experts, we gained practical insight in the usage of our modelling approach. Subsequently, we discuss some technical aspects of the usage of the proposed syntax and semantics and list practical and conceptual limitations, for which we give solutions in the context of BPMN.

Business vs. Technical View. The Business Process Model and Notation was originally defined for business users (e. g., business analysts, business experts): the technical developers implement and use the processes and the business experts monitor and manage the processes [15]. However, the more complex examples within this document (e. g., Fig. 6) show that bridging the gap between the business process design and process implementation in the domain of EIPs with BPMN can become difficult, if not impossible. For complex integration problems, the composition of EIPs quickly leads to technical BPMN syntax, which becomes intractable for business users.

(Sub-) Process Instantiation, Instance Handling. The instantiation of processes and sub-processes in BPMN is statically defined. This is sufficient for stateless, short-running processes. However, there are cases of stateful patterns (e. g., resequencer (not shown), aggregator), for which a more dynamic, conditional instance handling would be required.

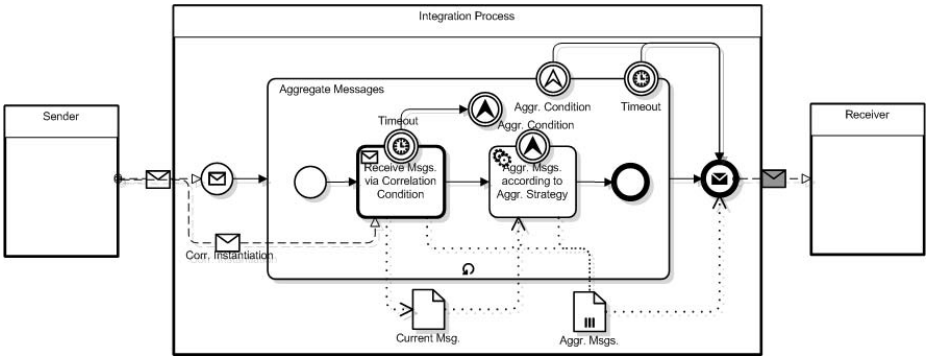


Fig. 6. Aggregator with timeout on sequence sub-process instance and conditional start mechanism for instance correlation per sequence

For instance, if for different messages sequences (i. e., consisting of correlation, sequence identifiers and a sequence termination information) separate aggregation sub-processes shall be started and independently interrupted by a *Timer* during an aggregation, the resulting BPMN syntax becomes tricky and the instantiation semantic seems violated. Figure 6 shows one possible syntactic approach to sequence-based timeouts in BPMN. An aggregator is a complex pattern and requires an embedded sub-process to define its tasks. Using a mechanism known as "Conditional Start" [3] combined with the BPMN correlation makes the instantiation tractable. When a message arrives, to which no aggregate is assigned, a new instance of an aggregator sub-process is created and the message is dispatched to this new instance. The subsequent messages for an existing, active aggregate are dispatched to the respective sub-process instance. For each correlated message sequence, an active sub-process instance exists and can be terminated through time out or if the aggregate is complete. The aggregated

message is sent and the sub-process is closed for further messages. In case new messages arrive for a closed sequence, a new sub-process is created.

Although the combination of BPMN correlation and conditional start makes the dynamic instantiation of sub-processes tractable, the mechanism comes with the implication of "redundant" syntax. Let's assume before the messages are aggregated (in Figure 6), a message translator has to transform them to a specific format (preprocessing) and the aggregated message has to be mapped to the target format (post-processing). The latter can be clearly added between the aggregator sub-process and the message end event. However, the preprocessing would be either executed in the parent process before the sub-process execution for the first message and copied to the aggregator sub-process for subsequent messages, or directly "in-lined" into the sub-process.

The topic of (sub-) process instantiation applies to all other complex patterns with sub-processes (e. g., inline synch/asynch bridge, resequencer).

Message Flow as Integration Adapter. The BPMN *Message Flow* is used to model message-consuming and producing adapters that are capable of handling various technical protocols (e. g., HTTP, SOAP, FTP). From an integration semantic point-of-view, adapters define the message interface/service and behaviour of the message channels, e. g., with respect to its quality of service or the message exchange pattern. That leads to advanced concepts like the retry handling for failed messages from a persistence. The standard message flow, however, does only allow to reference a *Message* specification, which does not even cover the required interface/service definition. The mentioned behavioural concepts cannot be covered and require an extension to the message flow beyond its specification.

6 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. In this work we proposed a syntactic mapping to the Business Process Model and Notation (BPMN) (cf. *REQ-2*); 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 the receivers of a message. In contrast to [18] we showed that an extension of BPMN for the integration domain with specific EIP constructs is not necessary.

Together with the syntax we defined corresponding execution semantics (cf. *REQ-3*). We developed the concept of composed patterns further to a complete definition of an *Integration Flow* (*REQ-1*). Although the syntax is compliant to BPMN, the BPMN execution semantics had to be slightly changed for *Message Endpoint* represented as *Message Flow*. The result is an integration domain specific language, with which integration aspects of messaging systems and their execution semantics can be expressed independent of the runtime implementation (cf. *REQs-5–6*). The approach allows for validation of integration programs and runtime systems (cf. *REQ-6*).

The "Business Monitoring" case study shows that our runtime independent modelling approach can be successfully compiled to the well-known, open source integration middleware *Apache Camel* and lets us assume that the application to other runtime systems is possible.

Acknowledgments. We thank Volker Stiehl and Ivana Trickovic for their support on BPMN and the formalization of integration semantics.

References

1. Anstey, J., Zbarcea, H.: Camel in Action. Manning (2011)
2. Barros, A., Dumas, M., ter Hofstede, A.H.M.: Service interaction patterns. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 302–318. Springer, Heidelberg (2005)
3. Bihari, S., Fischer, R., Loos, C., Reddy, P., Stiehl, V.: Sap netweaver process orchestration–build a complete integration scenario (sap teched 2013). Technical report. SAP AG (2013)
4. Gierds, C., Fahland, D.: Using petri nets for modeling enterprise integration patterns. Technical Report BPM Center Report BPM-12-18. BPMcenter.org. (2012)
5. Emmersberger, C., Springer, F.: Tutorial: Open source enterprise application integration - introducing the event processing capabilities of apache camel. In: DEBS, pp. 259–268 (2013)
6. Fahland, D., Gierds, C.: Analyzing and completing middleware designs for enterprise integration using coloured petri nets. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) CAiSE 2013. LNCS, vol. 7908, pp. 400–416. Springer, Heidelberg (2013)
7. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
8. Josuttis, N.M.: SOA in Practice. O'Reilly Media (2007)
9. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 171–186. Springer, Heidelberg (2013)
10. Meyer, A., Smirnov, S., Weske, M.: Data in business processes. EMISA Forum 31(3), 5–31 (2011)
11. Meyer, A., Weske, M.: Data support in process model abstraction. In: Atzeni, P., Cheung, D., Ram, S. (eds.) ER 2012. LNCS, vol. 7532, pp. 292–306. Springer, Heidelberg (2012)
12. Müller, D.: Management datengetriebener Prozessstrukturen. PhD thesis (2009)
13. Müller, D., Reichert, M., Herbst, J.: Flexibility of data-driven process structures. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 181–192. Springer, Heidelberg (2006)
14. Müller, D., Reichert, M., Herbst, J.: Data-driven modeling and coordination of large process structures. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 131–149. Springer, Heidelberg (2007)
15. O.M.G. (OMG). Business process model and notation (bpmn) version 2.0. Technical report (January 2011)

16. Ritter, D.: Using the business process model and notation for modeling enterprise integration patterns. CoRR, abs/1403.4053 (2014)
17. Scheibler, T.: Ausführbare Integrationsmuster. PhD thesis (2010)
18. Stiehl, V.: Prozessgesteuerte Anwendungen entwickeln und ausführen mit BPMN: Wie flexible Anwendungsarchitekturen wirklich erreicht werden können. dpunkt.verlag GmbH (2012)
19. van der Aalst, W.M.P.: The application of petri nets to workflow management. Journal of Circuits, Systems, and Computers 8(1), 21–66 (1998)
20. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)