# Compilation of BPMN-based Integration Flows

Daniel Ritter

SAP SE, Dietmar-Hopp-Allee 16, 69190 Walldorf
daniel.ritter@sap.com

**Abstract** Enterprise Application Integration plays an integral role for the communication between applications not only in service-oriented architectures. However, their modeling and configuration remain underrepresented. In previous work, the integration control and data flow syntax and semantics have been expressed in the Business Process Model and Notation (BPMN) as a semantic model for message-based integration, while the concrete compilation to several runtime systems was left open. In this work we share our ideas for a general compilation approach along the *Message Redelivery on Exception* (MRoE) integration capability and basic message processing strategies, from which we derive compilation patterns. These patterns are used to translate BPMN models via an intermediate property graph model to a runtime system graph representation that allows the generation of executable runtime code.

**Keywords:** Business Process Model and Notation, Graph Model, Integration Flow, Message-based Integration, Runtime Systems

## 1 Introduction

Although *Enterprise Application Integration* (EAI) continues to receive widespread focus by organizations, e. g., for integrating existing business with cloud applications, the modeling of integration scenarios remains vendor-specific and covers mostly their control flow aspects [10]. Besides other requirements, suitable modeling approaches should offer (P1) well-defined, standard modeling capabilities for interoperability and ease of use, (P2) cover the integration semantics (e. g., message creation, routing), and (P3) executable runtime semantics. Most prominent, non-commercial examples are the *Enterprise Integration Pattern* (EIP) icon notation [7], the text-based *Apache Camel* [2] or the UML-based *Guaraná* DSLs, however, none of them supports all of the requirements (P1–3).

Our *Integration Flow* (IFlow) modeling approach [10], which is productively used in SAP's *Integration as a Service* product, maps the common EIPs and integration semantics (P2) to the *Business Process Model and Notation* (BPMN) [8] (P1), which is a "de-facto" standard for modeling business process semantics and their runtime behavior [10] and specifies their composition to integration and adapter processes [10,12] as well as their behavior in exceptional cases [13]. Despite some deviations (e. g., BPMN *Message Flow* as integration adapter, process instantiation / termination [10]), the IFlow execution semantics (P3) can

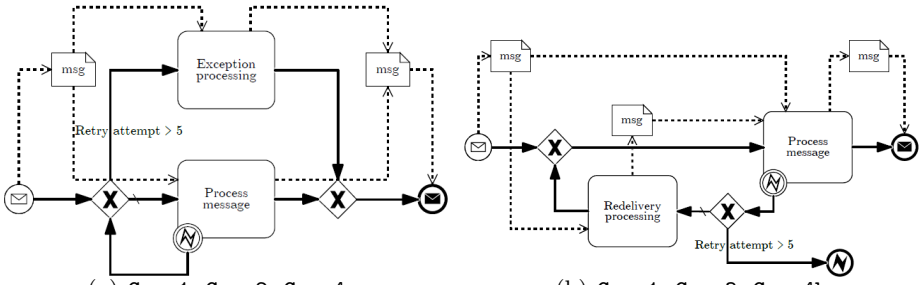
T. S. Heinze, T. M. Prinz (Eds.): Services and their Composition, 7th Central European Workshop, ZEUS 2015, Jena, Germany, 19-20 February 2015, Proceedings – published at http://ceur-ws.org

be represented close to the BPMN specification, which makes IFlows partially executable on standard BPMN engines. Open remaining questions are the compilation to standard integration systems and a general compilation model for different kinds of integration runtime systems (e. g., databases [11]), which we address in this paper. Similar work can be found, e. g., in the related business process domain [9].

The contributions of this work are the collection of general compilation (model) requirements and the concrete list of capabilities for one integration processing aspect, i. e., message redelivery, in Sect. 2, the mapping of the most relevant BPMN concepts for IFlows to Apache Camel constructs representing a standard (open-source) integration system in Sect. 3, the definition of basic compilation patterns for common message processing strategies in Sect. 4, and a graph-based model and compilation approach explained by sample graph re-writings in Sect. 5.

## 2 General Requirements

The integration flows are a *Domain-specific Language* (DSL) in the sense of Fowler [6], from which we derived the following, general requirements. The XML representation of the IFlow–BPMN file shall be parsed (*REQ-1: Parser*) and populated to a *Semantic Model* (*REQ-2: Runtime independent Semantic Model*), i. e., an in-memory object model similar to the domain model. The semantic model shall capture the control- and data flow to represent the integration process and exception flow (*REQ-3: Capture flow semantics*). For instance, one important




(a) `Cap-1`, `Cap-2`, `Cap-4a` (b) `Cap-1`, `Cap-3`, `Cap-4b`

**Figure 1.** Message redelivery patterns in BPMN expressing capabilities `Cap-1`–`Cap-4`.

integration processing type is the *Message Redelivery on Exception* (MRoE; *REQ-4: Support MRoE*), which is ubiquitous in integration scenarios [13]. MRoE requires at least the following capabilities (`Cap-1-4`): `Cap-1` redeliver message $n \in N$ times, `Cap-2` process the exception, when retry limit is reached, `Cap-3` handle processing on redelivery attempts, `Cap-4a` continue after exhaustion or `Cap-4b` end the process after exhaustion of redeliveries. Figure 1(a) shows the

message redelivery in BPMN fulfilling the capabilities `Cap-1`, `Cap-2` and `Cap-4a`, and Fig. 1(b) for `Cap-1`, `Cap-3` and `Cap-4b`. Eventually, the semantic model shall allow to generate code for multiple runtime systems (*REQ-5: Code generation*), which is packaged and deployed to the runtime system. We use *compilation* (i. e., code generation) over *interpretation* due to *REQ-2*.

## 3 From BPMN to Apache Camel in a Nutshell

In this section we sketch the idea of mapping from BPMN to runtime constructs. The lightweight integration system Apache Camel [2] represents the runtime system, which executes so called Camel routes (i. e., *Message Channel* [7] that can be combined to integration scenarios), described either in form of a Java DSL (used here) or several XML formats. Table 1 shows the mapping of BPMN elements used in IFlows to the corresponding Camel DSL statements. Notably,

**Table 1.** Camel Java DSL equivalents for BPMN elements

| BPMN element | Camel DSL statement |
|---|---|
| Message Start Event / Message Receiving Activity | `from` |
| Plain Activity | `process` |
| Message End Event / Message Sending Activity | `to` |
| Sub-Process | `to` |
| Activity with Message Flows | `to` |
| Event Sub-Process | `errorHandler` |
| Transaction | `transacted` |
| Boundary Error Event | `onException` |
| Boundary Timer Event | `?timeout=value` |

the message receiving / sending elements in BPMN represent *Message Endpoints* like HTTP, SOAP, FILE in the sense of [7], which are mapped to configurable `from` / `to` statements in Camel. BPMN activities are implementations of the `processor` interface in Camel, while BPMN sub-processes can be represented as separate Camel routes, which are addressable within the same Camel VM instance by `to` with additions like `to:direct` or `to:direct-vm`. The BPMN elements like event sub-process and boundary error event, used for the exception handling, find their counterparts only partially in the Camel `errorHandler` and `onException` statements, which are applicable on route or Camel context[1] instance level. The major issue is that the BPMN boundary element semantics cannot be adequately matched by route / context-level statements, since the matching Camel `try-catch` statement is incompatible with `onException` and `errorHandler` in version 2.x. Other BPMN boundary events like timer can be mapped to the Camel `timeout` on statement or route level. The specific MRoE capabilities of *REQ-4* can be mapped to Camel as shown in Table 2. While the MRoE has to be modeled in BPMN,

[1] A Camel context is a collection of routes, separated from other contexts at runtime.

**Table 2.** Camel Java DSL equivalents for capabilities.

| BPMN pattern | Camel DSL statement |
|---|---|
| Retry pattern `Cap-1` | `maximumRedeliveries` |
| Retry pattern `Cap-2` | `process` / `to` (after onException definition) |
| Retry pattern `Cap-3` | `onRedelivery` |
| Retry pattern `Cap-4a` | `continued(true)` |
| Retry pattern `Cap-4b` | `continued(false)` (default, can be omitted) |

we use the `onException` statement with additions like `maximumRedeliveries` for specifying the redelivery limit or `continued` to specify the behavior after the limit has been reached. The Camel `useOriginalMessage` statement (not shown) indicates whether the original message or the potentially modified one will be forwarded after an exception.

## 4 Basic Compiler patterns

Starting with the normal processing, in this section we describe basic compiler patterns for the different MRoE processing types (cf., *REQ-4*). These patterns serve as building blocks for our compilation approach, since they enumerate the basic processing types within an integration process (cf., [13]) and their flow semantics (cf., *REQ-3*). The normal message processing (*Pattern 1*), depicted in Fig. 2(a), represents the integration process with control- and data flow (cf., *REQ-1*) for an integration process. The integration operations are indicated by a BPMN sub-process, which will become a separate Camel route with the Camel instance internal addressing `to:direct:log`. If an exception occurs during the processing of the sub-process, the process can be either stopped (cf., `Cap-4b`), as shown in Fig. 2(b) (*Pattern 2*), a message redelivery can be started (cf., `Cap-1`, `Cap-2`), sketched in Fig. 2(d) (*Pattern 4*), or the processing can be continued (`Cap-4a`), denoted in Fig. 2(c) (*Pattern 3*).

## 5 Compilation Models and Runtime Synthesis

In this section, we define a graph-based, (semantic) compilation model that fulfills *REQ-1–4*, explain the graph re-writing to a runtime graph representation and the generation of executable code (cf., *REQ-5*) by example of compiler *Pattern 4* and Apache Camel.

### 5.1 Compilation Models

The key for the IFlow compilation to different runtime systems is a runtime-independent compilation model and approach (cf. *REQ-2*). Hence, we have decided for a compilation approach, where the semantic model is split into a logical (close to DSL) and a physical (runtime-near) representation. The *Logical*

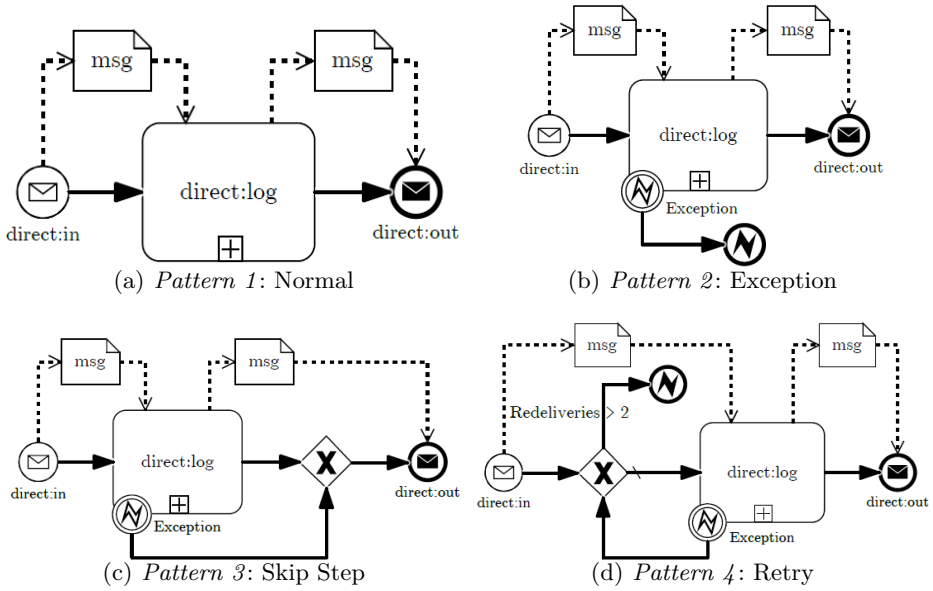



**Figure 2.** Compiler Patterns 1–4: From normal processing (with BPMN *Sub-Process*) over exception handling to skip step and message redelivery.

*Model* (LM) is defined as directed property graph (e. g., refer to [14]), where the BPMN flow elements, data store and data object are vertices, and BPMN sequence flow, message flow and data associations are the edges. In this LM property graph, vertices and edges define semantic information about the represented elements, which are populated during the parsing (cf., *REQ-1*). Figure 3(a) shows the LM of the *Pattern 4* BPMN model from the `direct:in` node of type *TStartEvent* (white) to the `direct:out` *TEndEvent* (black), with the intermediate sub-process (purple) and the attached boundary error event (red) that uses an exclusive gateway for the message redelivery attempts that lead to another end event (e. g., of type *TErrorEventDefinition*) to stop the process after exhausted delivery. The data flow is denoted as *TDataObjectReference* nodes (blue). The LM is used to optimize the actual process model independent of the runtime along the given integration semantics (not further discussed). Then a runtime-specific re-writing logic is applied to translate the LM to its respective *Physical Runtime Model* (PRM). For instance, the PRM of an Apache Camel route is again a directed graph, where message endpoints (`from` (white) / `to` (black)), processors (`process`) and other route level statements (e. g., `onException` (red)) are nodes linked by edges, the control flow. Fig. 3(b) depicts the corresponding PRM for the discussed example. Notably, the resulting PRM for Camel does not make the data flow explicit, but assumes an implicit data flow handling in the Camel runtime. However, this observation helps to understand why the Camel DSL

is not a (business) user facing model or LM for integration (P1), but rather a physical runtime model.

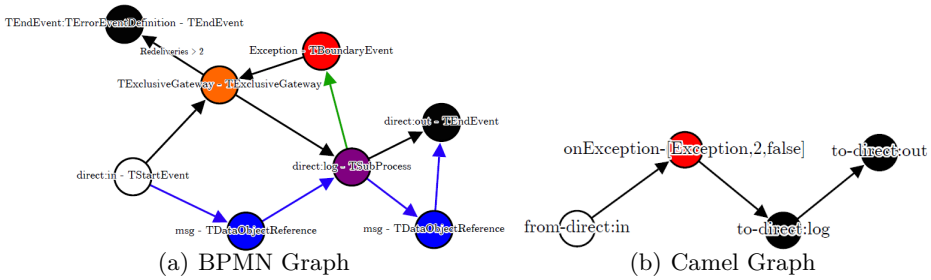



**Figure 3.** Compiler *Pattern 4* as Apache Camel model graph (right) from its BPMN graph (left).

### 5.2 Rule-based Logic to Physical Model Re-Writing

The translation from the runtime-independent LM to the specific physical runtime model has to be done for each runtime implementation by a system engineer. Hence, a pluggable, rule-based graph re-writing approach is preferred. Hereby, graph re-writing is conducted by graph node and edge traversers, which execute all applicable, registered rules (cf., Listing 1.1). Each rule specifies a `match` and an `execute` function. Listing 1.2 shows the implementation of the `match` function for detecting a MRoE (cf. *Pattern 4*). If the specified condition is evaluated to `true`, then the `execute` function for the current node is evaluated. Listing 1.3 shows the pseudo code for detecting the retry pattern: if a boundary error event is found during the `match`, the node is investigated further. The exception type becomes the name of the error event. Then, in case the leaving sequence flow leads to a gateway, preceding the activity, the redelivery count is extracted. If the outgoing flow leads to a succeeding gateway instead, no redeliveries are attempted and the route continues after the exception. Finally, the gather information is stored to the node and obsolete nodes are removed from the graph. Since the data flow of a Camel route cannot be configured explicitly, the BPMN message flow and data object associations are used to verify the correct behavior and the optimization of the LM. However, let us recall the `useOriginalMessage` statement from Sect. 3. With data flow information present, the case of "which message to use" in case of `continue==true` can be answered and applied if the runtime supports it.

**Listing 1.1.** Rewrite graph.

```
1 void rewrite (graph)
2     rule1 =
3         new detectFromEndpoints
4     //...
5     ruleX =
6         new detectRetryPatterns
7     runner =
8         new ruleRunner(graph)
9     runner.run(
10        rule1, ..., ruleX)
```

**Listing 1.2.** `match` function.

```
1 match(node)
2     return (node.getType()
3     equals
4     "BoundaryErrorEvent")
```

**Listing 1.3.** Detect retry patterns.

```
1 void execute(node)
2     exception = node.getName()
3     redeliveries = 0
4     continued = false
5
6     if node.leadsToPrecGtw()
7         redeliveries =
8             node.getPrecGtw()
9             .getRedeliveries()
10    if node.leadsToSuccGtw() OR
11        node.getPrecGtw()
12            .leadsToSuccGtw()
13        continued = true
14
15    node.add([exception,
16        redeliveries,
17        continued])
18    node.getGraph()
19        .rmObsolteNodes()
```

### 5.3 Code Generation

The resulting PRM contains all necessary information for the code generation. Listings 1.4 and 1.5 show the generated code for the compiler patterns *Pattern 3* and *Pattern 4* for comparison. The main difference between the two patterns lies in the behavior after an exception occured: *Pattern 3* continues with the processing (Listing 1.4, line 3), while *Pattern 4* starts with two message redelivery attempts (Listing 1.5, line 2, 3) and stops the execution through `handled(false)`, which ends the processing and throws the previously caught exception.

**Listing 1.4.** Camel DSL for *Pattern 3*

```
1 from("direct:in")
2   .onException(Exception.class)
3     .continued(true)
4   .end()
5   .to("direct:log")
6   .onException(Exception.class)
7     .handled(true)
8   .end()
9   .to("direct:out");
```

**Listing 1.5.** Camel DSL for *Pattern 4*

```
1  from("direct:in")
2  .onException(Exception.class)
3    .maximumRedeliveries(2)
4  .end()
5  .to("direct:log")
6  .onException(Exception.class)
7    .handled(false)
8  .end()
9  .to("direct:out");
```

## 6 Conclusion and Outlook

In this work we have collected basic requirements (cf., *REQ-1–5*) for the compilation of BPMN-based integration flows to integration runtime systems by example of Apache Camel. We sketched our idea for a runtime-independent, logical compilation model, identified compilation patterns for the case of message redelivery on exception and discussed the runtime-dependent re-writing to the physical runtime model with code generation. Our approach is comparable to the transformation from process model (i. e., IFlow), over executable workflow (i. e., Camel DSL) to IT infrastructure in Appel et al. [3] and the re-writes for security policies and compliance rules on $\mu$BPMN in Accorsi [1]. We decided for a "two-step" approach due to required compilation to different runtime systems (from one logical model) and separate optimizations on logical / physical level.

Future work will consider the optimization of the logical model along the integration semantics and the application of the compilation approach to other runtime systems. We will check the transformation of our logical model to colored petri nets for checking middleware designs for enterprise integration, e. g., Fahland et al. [5], and are highly interested in collaboration partners for the formalization of integration runtime systems, e. g., as in Dijkman et al. [4].

**Acknowledgments** Thanks go to Jan Sosulski for his contribution to the compiler implementation.

## References

1. Accorsi, R.: On process rewriting for business process security. In: Proceedings of the 3rd International Symposium on Data-driven Process Discovery and Analysis, Riva del Garda, Italy, August 30, 2013. pp. 111–126 (2013)
2. Anstey, J., Zbarcea, H.: Camel in Action. Manning (2011)
3. Appel, S., Frischbier, S., Freudenreich, T., Buchmann, A.P.: Event stream processing units in business processes. In: Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings. pp. 187–202 (2013)
4. Dijkman, R.M., Gorp, P.V.: BPMN 2.0 execution semantics formalized as graph rewrite rules. In: Business Process Modeling Notation - Second International Workshop, BPMN 2010, Potsdam, Germany, October 13-14, 2010. Proceedings. pp. 16–30 (2010)
5. Fahland, D., Gierds, C.: Analyzing and completing middleware designs for enterprise integration using coloured petri nets. In: CAiSE. pp. 400–416 (2013)
6. Fowler, M.: Domain Specific Languages. Addison-Wesley Professional, 1st edn. (2010)
7. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
8. (OMG), O.M.G.: Business process model and notation (bpmn) version 2.0. Tech. rep. (jan 2011)

9. Prinz, T.M., Spieß, N., Amme, W.: A first step towards a compiler for business processes. In: Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. pp. 238–243 (2014)
10. Ritter, D.: Experiences with business process model and notation for modeling integration patterns. In: Modelling Foundations and Applications - 10th European Conference, ECMFA 2014, York, UK, July 21-25, 2014. Proceedings. pp. 254–266 (2014)
11. Ritter, D.: What about database-centric enterprise application integration? In: Proceedings of the 6th Central-European Workshop on Services and their Composition, ZEUS 2014, Potsdam, Germany, February 20-21, 2014. pp. 73–76 (2014)
12. Ritter, D., Holzleitner, M.: Integration adapter modeling. In: Advanced Information Systems Engineering - 27th International Conference, CAiSE 2015 (accepted), Stockholm, Sweden, June 8-12, 2015. Proceedings (2015)
13. Ritter, D., Sosulski, J.: Modeling exception flows in integration systems. In: 18th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2014, Ulm, Germany, September 1-5, 2014. pp. 12–21 (2014)
14. Rodriguez, M.A., Neubauer, P.: Constructions from dots and lines. Bulletin of the American Society for Information Science and Technology 36(6), 35–41 (2010)