BPEL to BPMN: The Myth of a Straight-Forward Mapping

Matthias Weidlich, Gero Decker, Alexander Großkopf, and Mathias Weske

Hasso Plattner Institute, Potsdam, Germany
{matthias.weidlich,gero.decker}@hpi.uni-potsdam.de,
{alexander.grosskopf,mathias.weske}@hpi.uni-potsdam.de

Abstract. An alignment of the Business Process Execution Language (BPEL) and the Business Process Modelling Notation (BPMN) promises seamless integration of process documentation and executable process definitions. Thus, a lot of research has been conducted on a mapping from BPMN to BPEL. The other perspective of this alignment, i.e. a BPEL-to-BPMN mapping, was largely neglected. This paper presents a condensed discussion of such a mapping and its pitfalls. We illustrate why such a mapping is not as straight-forward as commonly assumed and discuss the gaps to be bridged towards a better alignment of both languages.

1 Introduction

The domain of business process modelling has been shaped by the emergence of two languages in recent years. At first, the Web Service Business Process Execution Language 2.0 (BPEL) [1] has been standardised by the OASIS and is nowadays the leading standard for the description of executable business processes. On the other hand, the Business Process Modelling Notation 1.1 (BPMN) [2] was standardised by the OMG and established as a popular modelling language, which aims at documenting and communicating business processes between all business stakeholders.

The development of BPMN was also driven by the lack of any standard graphical notation for BPEL. Tool vendors introduced a wide variety of different notations hampering understanding and analysis. Addressing the demand for a standardised BPEL notation, the BPMN specification has included the definition of a BPMN-to-BPEL mapping since its first version. Limitations of this mapping have been discussed in academia in a comprehensive manner. That in turn led to more sophisticated transformations, cf. [3]. In particular, issues resulting from the incompatibility of block- and graph-structured languages have been in the centre of interest. While there is no doubt about this challenge, the one-sided focus repressed problems regarding the other direction of a BPEL and BPMN alignment.

An application of BPMN as a graphical notation for BPEL requires a bidirectional alignment of both languages. On the one hand, existing BPEL processes should be visualised using a standardised notation easing communication

R. Meersman and Z. Tari (Eds.): OTM 2008, Part I, LNCS 5331, pp. 265–282, 2008.

about these processes. On the other hand, changes in BPEL processes should be propagated to their BPMN representation in order to realise the vision of seamless round-trip engineering between the poles of high-level business alignment (BPMN) and low-level process enactment (BPEL). Thus, a bidirectional BPEL-to-BPMN mapping bridges the gap between process design and implementation. Despite the pressing demand for such a mapping, it has not been tackled by recent research. It is assumed that such a *'transformation is relatively straight-forward'*[4] and *'that arbitrary BPEL processes can be mapped to BPMN using the flattening or the hierarchy-maximisation strategy'*[5].

In fact, this assumption proves to be correct for numerous concepts. Essential BPEL activities can be mapped to BPMN directly and existing BPMN-to-BPEL mappings are bidirectional for many concepts. Nonetheless, there are a number of pitfalls in a BPEL-to-BPMN mapping. This paper presents a detailed analysis of these issues, which relate to a variety of different aspects, for instance event handling and the compensation mechanisms. We assume the reader to be familiar with BPEL and BPMN and refer to [6,7] for introductions to these languages.

The remainder of this paper is structured as follows. Section 2 gives an overview of the mapping by relating BPEL concepts to their counterparts in BPMN. We present a detailed analysis of mapping pitfalls in Section 3 and elaborate on related work in Section 4. Section 5 discusses our findings and outlines the steps to be taken in order to remove the presented pitfalls. We conclude with Section 6.

2 Sketch of a BPEL-to-BPMN Mapping

A BPEL-to-BPMN mapping is sketched in Table 1. It relates BPEL basic activities, structured activities, and generic concepts to the corresponding constructs in BPMN. The third column *Mapping* indicates, whether a concept is directly mappable (+), whether the mapping is partial (∘), or whether the concept is completely missing in BPMN (–). We note the section that explains issues of the respective mapping at the end of a row. A mapping is partial, if a concept can be mapped to a limited extent, due to one of the following reasons:

- BPMN semantics for the respective construct are ambiguous.
- A BPEL concept is not defined in BPMN and cannot be traced back to existing concepts.
- Similar constructs show different semantics in BPEL and BPMN.

The listed BPMN constructs solely sketch the mapping. Not all of them might be needed under all circumstances. Most of the mappings involve additional basic constructs such as sequence flows. According to Table 1, there are only few activities that cannot be mapped at all, namely *event handlers*, *termination handlers*, and the *validate* activity. Nonetheless, the majority of constructs can only be mapped partially.

Please note that there is no need to map the BPEL *empty* activity. It provides an explicit synchronisation point for *control links* or defines an empty *fault*

Table 1. Sketch of a BPEL-to-BPMN mapping

	BPEL	BPMN	Mapping
Basic Activities	*invoke*	sending/receiving task, message event	○ (3.1)
	receive (createInstance='no')	receiving task, message event	+
	reply	sending task, message event	○ (3.1)
	validate	—	– (3.10)
	assign	assignment	○ (3.10)
	wait	timer intermediate event	+
	exit	termination end event	+
	throw, rethrow	error end event	○ (3.4)
	compensate, compensateScope	compensation events	○ (3.5)
Structured Activities	*sequence*	sequence flow	+
	if-elseif-else	excl. data-based gateway, default flow	+
	while, *repeatUntil*	standard loop activity	+
	foreach	multiple-instance loop activity	+
	pick (createInstance='no')	event-based gateway, message/timer event	+
	flow, control links	parallel gateway, inclusive gateway, complex gateway	○ (3.7)
	scope	embedded subprocess	○ (3.6)
	fault handlers	exception flow	○ (3.4)
	event handlers	—	– (3.2)
	termination handler	—	– (3.3)
	compensation handler	compensation activity, compensation event, compensation association	+
Generic	*variables*	data object	○ (3.9)
	correlation mechanism	property	○ (3.1)
	process instantiation	message events, excl. event-based gateway	○ (3.8)
	communication abstractions	participant, web service, role	○ (3.1)

handler. In BPMN, these synchronisation points are implicitly given by gateways as a result of the mapping of *control links*. Additionally, exception flow does not enforce the specification of an activity. In Table 1, we also neglect *opaqueness*, a concept that allows for hiding of syntactic elements in BPEL abstract processes. Although BPMN lacks a corresponding concept, opaque activities or opaque expressions can easily be marked as unspecified using dedicated keywords.

3 Pitfalls in a BPEL-to-BPMN Mapping

This section discusses pitfalls in a BPEL-to-BPMN mapping and addresses all concepts that are marked as being not or only partially mappable according to Table 1. We explain what prevents us from mapping certain constructs and why various of the apparent mappings have to be considered as being incomplete or imprecise.

3.1 Message-Based Interactions

BPEL introduces the concept of *partner links* to model relationships between interacting business processes. Via *partner link types*, such a conversational relationship can be characterised by means of the *roles* each participant plays in the conversation and the *port types* used to receive messages from the partner. Therefore, all message-exchanging activities of BPEL reference a certain *partner link* and additionally specify the applied *port type* and *operation*.

A service call is realised via an *invoke* activity supporting two different interaction scenarios — asynchronous (one way) and synchronous (request-response). Inbound message activities, i.e. *receive*, *pick*, and *event handler* activities, are applied to handle incoming service calls. Corresponding *reply* activities might be used to react on received messages. In case the relation between an inbound message activity and a *reply* activity cannot be derived implicitly, BPEL introduces the concept of *message exchanges* to correlate these activities. BPEL therefore directly supports the single-transmission bilateral interaction patterns (send, receive, send/receive, and receive/send) as introduced in [8]. In order to correlate multiple interactions between process instances, BPEL introduces *correlation sets*. They allow for referencing correlation tokens by means of WSDL properties. Messages with equal values for these properties belong to the same conversation.

The BPMN meta-model introduces various concepts to deal with BPEL's communication abstractions. Message-based interactions are defined between participants, which are characterised by a certain role or entity. However, the relation between participants is captured solely by the concept of a message. Thus, an abstraction of that relation in the sense of the BPEL *partner links* and *partner link types* is not included in the BPMN meta-model. Moreover, the technical endpoints for a certain communication (*port types* and *operations* in BPEL) can be specified using the concept of a web service in BPMN.

The actual message-based interaction is realised via sending and receiving tasks, and message intermediate events. With version 1.1 of BPMN, the latter can have catching as well as throwing semantics. An asynchronous *invoke* activity corresponds to a sending task or message intermediate event, whereas the synchronous scenario cannot be represented by a single construct, as illustrated in Figure 1 (the mapping of *variables* is omitted as it is discussed separately). An additional subprocess spanning both events has to be inserted in case the decoupling of sending and receiving is of relevance for exception or compensation

```
<invoke
  name='m1'
  inputVariable='x'
  outputVariable='y'
  ... />
```

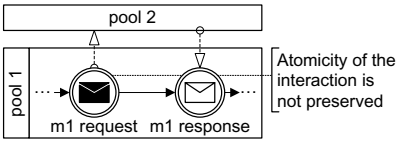

Fig. 1. Proposal for a mapping of a synchronous *invoke* activity

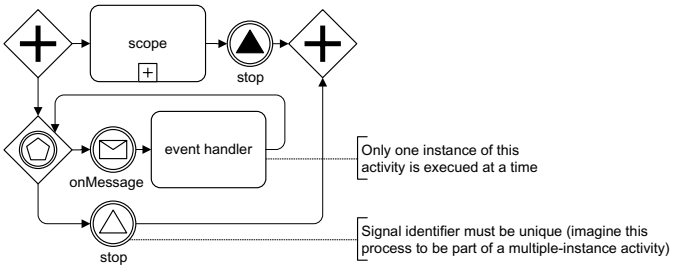

Fig. 2. Structure to partially simulate non-interruptive event-handling in BPMN

handling. A service task cannot be applied to model the synchronous *invoke*, as it receives a message *before* a message is sent.

While the reception of messages can be realised using receiving tasks, service tasks, and message intermediate events in BPMN, a mapping of *reply* activities is not straight-forward. In particular, the question of a relation between message receiving and corresponding message sending constructs is not addressed in BPMN. One might argue that, similar to BPEL, this relation might be derived from the configuration of the message handling constructs (the referenced messages, participants, and web services). However, there is no means to explicitly define the relation in case the aforementioned mechanism is ambivalent. The absence of a counterpart for BPEL *message exchanges* therefore represents a pitfall in the mapping of message handling activities.

The BPMN meta-model introduces the concept of properties, which might be specified for messages. A dedicated attribute allows for the marking of these properties for correlation purposes. Actual semantics of a correlation mechanism — when is a correlation initialised and how long is it kept? — is not addressed in the BPMN specification. Thus, it is left open, whether semantics of the intended correlation mechanism comply with the one defined for BPEL. Please note that BPMN correlation properties are not visualised.

3.2 Non-interruptive Event-Handling

BPEL supports non-interruptive event-handling through the specification of *event handlers*. They are part of the normal processing and are introduced at the level of a *scope*. An event handler instance is spawned either by a message or timer-based trigger. Further on, multiple triggers will create concurrent instances of the according handler. These instances can only be spawned while the *scope* is active.

Figure 2 shows a possibility to capture non-interruptive event-handling in BPMN. While the scope is running, new messages can trigger an event handler activity. When the scope completes a signal event is thrown and caught to prevent the creation of further event handler instances and route the control flow to the merging parallel gateway. This does not fully capture the BPEL semantics because it does not allow multiple *event handler* activities to run in parallel.

Moreover, this workaround implies a distinction between the signal events of different processes or branch instances, which, in turn, requires dynamic definition of the signal identifier.

At a glance, the workaround introduced in Figure 2 realizes non-interuptive event-handling. But the requirement to dynamically create concurrent instances prohibits the visualisation of BPEL *event handlers* in BPMN. Any solution incorporating boundary events is not feasible, due to the termination semantics of these events.

3.3 Termination Handling

Termination handlers can be specified in BPEL in order to control termination of *scopes*. They are invoked after execution of the activity enclosed by the *scope* has been terminated. Forced termination can be triggered either by a fault or by an enclosing *for each* activity with a *completion condition*. BPEL defines *termination handlers* to be rather encapsulated. That is, any internal faults of *termination handlers* are not propagated.

BPMN does not define a construct similar to the BPEL *termination handler*. It is important to notice that the behaviour of a *termination handler* cannot be traced back to existing constructs. In case of termination that is triggered by a fault, it is feasible to model the termination handling activities as part of all potentially invoked *fault handlers*. This workaround is illustrated in Figure 3. However, any scenario with termination triggered in the context of processing of multiple branches (i.e. a *for each* activity with a *completion condition*) cannot be captured. In these cases, termination of certain activities is part of the intended behaviour and the absence of a fault prohibits the execution of *fault handlers*. On the other hand, *compensation handlers* cannot be applied to tackle these scenarios, as successful completion of an activity is the prerequisite for its compensation.

Thus, the concept of termination handling is neither supported in a native way in BPMN, nor can it be traced back to existing BPMN constructs.

3.4 Exception Handling

The exception handling framework of BPEL comprises *throw* and *rethrow* activities, as well as *fault handlers* defined for *scopes*. *Throw* and *rethrow* activities enable explicit throwing of exceptions, while *fault handlers* can catch explicitly or

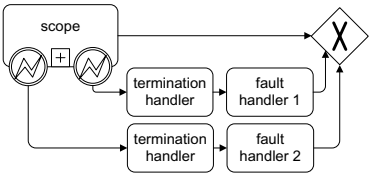

Fig. 3. Termination handling as part of the exception flow in BPMN

```
<sequence>
  <invoke name='1' ... />
  <throw ... />
  <invoke name='2' ... />
</sequence>
```

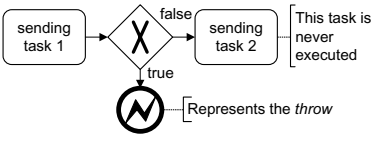

Fig. 4. Proposal for a mapping of exception-throwing activities

implicitly thrown exceptions. The matching between an exception and a *fault handler* is based on a combination of fault identifiers or fault data types, respectively.

In general, the exception handling framework of BPEL can be mapped to BPMN. Exception throwing activities correspond to error end events. A mapping to intermediate error events is no longer possible, as these events have been removed from the BPMN specification with version 1.1 (please note that the specification is still inconsistent regarding the definition of these events), due to their ambiguous semantics. Exception-throwing activities might be followed by further (unreachable) activities in BPEL. In these cases, a mapping might involve an exclusive data-based gateway preceding the error end event in order to preserve the BPEL process structure, as illustrated in Figure 4.

With respect to the matching of exceptions and exception handlers, BPMN relies solely on exception identifiers. Consequently, any mapping of the exception handling framework of BPEL requires a careful definition of the exception identifiers for error events in BPMN (that is, values for the attribute *error code*). It might be necessary to introduce additional exception identifiers, which represent exception data types or even tuples of exception names and data types, in order to achieve the same matching behaviour in both languages.

Another pitfall is the lack of well-defined semantics for the propagation of exceptions in BPMN. In particular, it remains unclear, how exceptions in concurrent instances of an activity are treated. Please refer to [9] for a discussion of this issue.

3.5 Compensation Handling

In order to treat the reversible part of the behaviour of a *scope*, BPEL introduces *compensation handlers*. Compensation can be triggered explicitly using a *compensate scope* or *compensate* activity inside a *fault*, a *compensation*, or a *termination handler* (FCT-handler). Both activities can solely trigger compensation for *scopes* enclosed by the *scope* to which the calling FCT-handler is attached to. While activity *compensate scope* targets one out of these *scopes*, activity *compensate* considers all of them, which is referred to as *compensation broadcasting*. Although compensation can be triggered without knowledge about the state of the target activity instance, the actual compensation depends on the availability of compensation snapshots, which are created on successful completion of an activity instance. The order in which these snapshots are processed equals the reversed order of their creation.

Compensation activities are the counterparts for *compensation handlers* in BPMN. Moreover, the compensation order equals the one defined for BPEL. Nevertheless, semantics in case of invalid compensation triggers (no compensation snapshot was created before) and the execution context of compensation activities are underspecified in BPMN. The latter is of particular importance for the propagation of exceptions arising during execution of compensation activities. Against the background of BPMN as a modelling language for BPEL processes, one might assume BPEL semantics with respect to these open issues. However, this does not resolve all pitfalls related to compensation handling.

In BPMN, compensation is triggered by compensation events. It might be reasonable to represent BPEL *throw* or *rethrow* activities by error events in BPMN, due to the negligible lifecycle of these activities (termination is triggered immediately). A similar approach is not feasible for compensation activities. In BPMN, control flow is directly passed to downstream activities after compensation has been triggered by a compensation intermediate event, while BPEL's *compensate* and *compensate scope* activities block the control flow until compensation has finished. Figure 5 illustrates this issue by means of exemplary process parts. The *invoke* activity inside the *fault handler* (*invoke 6*) is executed *after* compensation finished, while the corresponding task (*sending task 6*) runs *concurrently* to the compensation activities in BPMN.

Assuming that BPMN compensation events are used to model BPEL compensation calling activities, there is another mismatch with respect to the broadcasting of compensation triggers. Compensation broadcasting in BPEL considers all activities in the *scope*, to which the compensation calling FCT-handler is assigned to. On the contrary, compensation is broadcasted throughout the process instance in BPMN, if no target activity has been specified. Again, Figure 5 illustrates the mismatch. In the BPEL process part, compensation is broadcasted solely to the *compensation handler* inside the scope (*invoke 4*), whereas BPMN semantics lead to execution of all compensation activities (*sending task 2* and

```
...<invoke name='1' ... >
 <compensationHandler>
  <invoke name='2' ... />
 </compensationHandler>
</invoke>
<scope ... > <faultHandlers>
 <catchAll> <sequence>
  <compensate />
  <invoke name='6' ... />
 </sequence> </catchAll>
 </faultHandlers>
 <sequence>
  <invoke name='3' ... >
   <compensationHandler>
   <invoke name='4' ... />
   </compensationHandler>
  </invoke>
  <invoke name='5' ... />
 <sequence> </scope>...
```

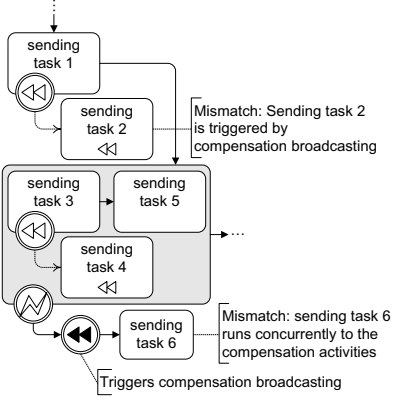

Fig. 5. Exemplary process parts illustrating the compensation handling mismatches

sending task 4). Thus, emulation of compensation broadcasting as it is defined for BPEL, requires the application of compensation events with specific targets in BPMN. They reference all subprocesses that correspond to BPEL *scopes* directly enclosed by the *scope* to which the respective FCT-handler (containing the corresponding *compensate* activity) is attached to. This workaround might require the usage of multiple compensation calling events per BPEL *compensate* activity as these events can reference solely one activity. Therefore, the workaround does not seem to be feasible with respect to BPEL-BPMN-round-tripping.

3.6 Default Handlers

According to the BPEL specification, the visibility of all FCT-handlers is restricted to the immediately enclosing *scope*. In order to reuse handlers in enclosed *scopes* and to enable propagation of faults and compensation triggers, BPEL defines a set of default handlers for fault, compensation, and termination handling. These default handlers are installed, whenever a user defined handler is missing. All default handlers trigger compensation broadcasting, whereas the default *fault handler* throws any caught exception again.

Besides the issues mentioned in Section 3.5, the implicit behaviour of BPEL processes realised by default handlers imposes serious challenges for a BPEL-to-BPMN transformation. Propagation of exceptions as implemented in the default *fault handler* complies with BPMN semantics. Nevertheless, the mechanisms to trigger compensation broadcasting, as the result of an uncaught exception (default *fault handler*) or termination of an activity (default *termination handler*), have to be modelled explicitly in BPMN. We consider that as a serious pitfall. A mapping of all these default handlers would increase complexity of the mapped process a lot, owing to their definition for every *scope*. Apparently, default handlers must only be mapped, if the respective *scope* contains activities with user-defined *compensation handlers*.

```
...
<scope name='1' ... >
 <faultHandlers>
  <catchAll>
   <compensate />
  </catchAll>
 </faultHandlers>
 <scope name='2' ... >
  <scope name='3' ... >
   <compensationHandler>
     <invoke name='2' ... />
   </compensationHandler>
   <invoke name='1' ... />
  </scope>
 </scope>
</scope>
...
```

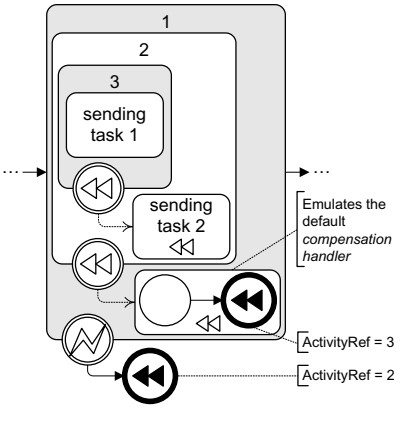

Fig. 6. Exemplary process parts illustrating propagation of compensation triggers

However, it is questionable, whether a direct emulation of the BPEL mechanism to propagate compensation triggers is feasible for BPMN. The workaround aligning the targets for compensation broadcasting (please refer to Section 3.5) must also be applied for any mapping of default *compensation handlers*. Modelling compensation targets explicitly in BPMN creates an overhead as illustrated in Figure 6. Here, the compensation handler of subprocess *2* emulates the default *compensation handler* of the BPEL *scope*. An alternative approach would be the propagation of compensation triggers to indirectly contained subprocesses (i.e. subprocess *3* in our example). However, this would complicate BPEL-BPMN-round-tripping as the complete containment hierarchy needs to be analysed in order to identify BPEL default handlers in BPMN.

3.7 Dead Path Elimination

Dead path elimination (DPE) can be realised in BPEL using a *flow* activity, enabling parallel execution, and named *control links* (*links*, for short), which are applied to specify execution dependencies. *Links* always connect two activities and might assume three different states: 'unset' (the initial state), 'true', or 'false'. In the case an activity with incoming *links* is enabled, the execution is delayed until all *links* are set to either 'true' or 'false'. Afterwards the *join condition* is evaluated and according to the result, the activity is executed or skipped. Subsequently, all outgoing links are set, either according to the *transition condition* (if the activity has been executed) or to 'false' (if the activity has been skipped). *Links* have to create an acyclic graph of dependencies and must not cross the boundary of repeatable constructs.

In order to realise DPE in BPMN the different states of a *link* have to be expressed. Two approaches seem to be reasonable. *Two* separate sequence flows represent both *link* states, 'true' and 'false' and a token is sent exclusively on one of these flows. Alternatively, *one* sequence flow represents the *link*. This approach would require non-local semantics to distinguish the states 'false' and 'unset' of the according *link*.

In any case, parallel gateways realise the parallel execution and the activation of sequence flows representing *links* in scenarios without *transition conditions* and *join conditions*. The activation of sequence flows representing *links* with *transition conditions* is modelled using inclusive gateways. BPMN assumes a structured setting for the converging inclusive gateway. This assumption does not hold for DPE scenarios, which raises various questions concerning the synchronisation of the gateway. Recent research led to various proposals to overcome these issues [10,11], however, they have not yet been adopted by BPMN.

Moreover, a mapping of the *join condition* is not straight-forward. In order to activate outgoing sequence flows based on the activation of incoming sequence flows, we have to apply a complex gateway. All other gateway types require a clear separation of splitting and merging behaviour. The BPMN specification allows for referencing the names of incoming branches in the *incoming condition* of the complex gateway. Although it remains unclear in the BPMN specification, we assume that this condition is evaluated, whenever a token arrives at the

```
... <invoke name='1' ...
    suppressJoinFailure='yes|no'>
 <targets>
   <target linkName='a' />
   <target linkName='b' />
   <target linkName='c' />
   <joinCondition ... >
   (a AND b AND c) OR NOT c
   </joinCondition>
  </targets> </invoke>...
```

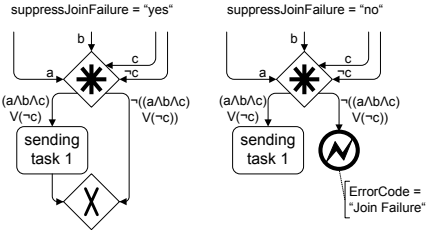

Fig. 7. Activity with incoming *control links* and a *join condition*

gateway. In addition, BPMN does not specify, whether tokens not satisfying the *incoming condition* are collected (they may satisfy the condition together with tokens arriving later) or discarded. A reasonable interpretation of semantics of the complex gateway would be to wait for *synchronisation* of all incoming sequence flows and then activate the outgoing sequence flows according to the assigned conditions. Besides the fact that multiple outgoing sequence flows with different conditions cannot be modelled according to the BPMN specification, the notion of *synchronisation* of the complex gateway is an open issue. Figure 7 sketches the mapping of the join condition. It also illustrates the relevance of this pitfall even if DPE is disabled in BPEL (attribute *suppressJoinFailure*).

We summarise that BPEL processes involving *transition conditions* or *join conditions* cannot be mapped to BPMN, due to the ambiguous definitions of both, the inclusive gateway (merging semantics) and the complex gateway.

3.8 Process Instantiation

A classification of process instantiation mechanisms and an evaluation of BPEL and BPMN has been presented in [12]. In general, BPEL processes are always instantiated through a single message, received by a *receive* activity or by a *pick* activity. We refer to these activities as start activities, if they have been configured for instantiation purposes via the Boolean attribute *create instance*. With respect to multiple start activities, BPEL allows for a variety of different scenarios. According to [12], subscriptions are established either for all of the remaining start activities (multiple *receive* activities), or only for reachable start activities (after one *on message* branch has been triggered, subscriptions for the remaining *on message* branches are discarded). Further on, BPEL enables precise definition of expiration for these subscriptions. Timer-based (*event handlers*) or message-based (*pick* activity) triggers might remove subscriptions, while they might also reside until message consumption or process termination.

In line with BPEL, BPMN processes are also instantiated by the reception of messages. In fact, simple BPEL processes can directly be mapped to BPMN, as *receive* activities and *pick* activities applied to instantiate the process have their counterparts in message start events and the event-based gateway (configured via its attribute *instantiate*).

However, process instantiation mechanisms of BPEL and BPMN are compatible only at a first glance. The analysis in [12] reveals fundamental differences with respect to the conjunction of multiple start events. In particular, there is no means to issue subscriptions for message events in the course of process instantiation. Therefore, there is only one way to model scenarios that require multiple start events to occur. That is, multiple start events are directly connected to an activity, which furthermore is configured via a dedicated attribute. In this case, it is waited for the occurrence of all events before the process instance is created. Please note that the actual configuration of the activity remains unclear, as there is no such attribute in the BPMN meta-model. The attribute *start quantity* is inapplicable in this context, as it does not relate to process instantiation.

Owing to the lack of a subscription mechanism, BPEL scenarios involving multiple start activities followed by different activities cannot be expressed in BPMN. Obviously, scenarios with complex dependencies between these subscriptions cannot be modelled in BPMN either. The Listing 1.1 shows an exemplary BPEL scenario that cannot be mapped to BPMN.

Listing 1.1. Exemplary BPEL process instantiation scenario

```
<sequence> <flow> <sequence>
  <receive ... createInstance='yes'>
    <correlations> <correlation set='c' initiate='join' /> </correlations>
  </receive>
  ... </sequence>
  <sequence>
  <receive ... createInstance='yes'>
    <correlations> <correlation set='c' initiate='join' /> </correlations>
  </receive>
  ... </sequence> ... </flow> ... </sequence>
```

3.9 Data Variables

BPEL defines *variables*, which belong to a certain *scope* (or the whole *process*, respectively). Further on, a *scope* enforces constraints with respect to data visibility — a variable is visible inside the *scope* containing its definition and all of its enclosed *scopes*. In addition, BPEL enforces lexical scoping of variables. In other words, *variables* of outer *scopes* can be hidden by introducing *variables* of the same name in an inner *scope*. Further on, BPEL supports *variables* based on a variety of type definitions, comprising WSDL message types and properties, XML Schema types, BPEL *partner link types*, *expressions*, and *literals*.

Variables are used to store content of incoming messages and to provide content for outgoing messages. If the input (output) message contains only one part, the respective message activity can specify an input (output) *variable*. In order to cope with more complex scenarios, BPEL provides *from* and *to* constructs to assemble or disassemble the message using several variables.

Furthermore, *variables* play an important role in the context of exception handling. A *throw* activity might reference a certain *variable* that contains the data needed for treatment of the exception. In case an exception is thrown using a *rethrow* activity, the fault data is also propagated. In addition, BPEL defines how WSDL faults are mapped into the process.

In general, BPMN introduces the notion of data objects to model the data flow. These data objects are state-full and might be structured using properties. Properties are the basic building blocks of the data layer in BPMN as they are the grounding not only of data objects, but also of messages. In addition, they are part of an activity definition and might be applied for correlation purposes. Properties and data objects can be specified as inputs or outputs of a certain activity. That might be visualised using data associations, however, visualisation of data objects and associations is optional.

Data scoping is not explicitly addressed in BPMN, neither on the level of properties, nor for data objects. While the notion of input and output sets might be applied to clarify data visibility for activities, there is no way to restrict the set of data objects (or properties) accessed by expressions of sequence flows or gates. Therefore, especially the mapping of multiple BPEL *variables* with the same name is cumbersome. All constraints resulting from the lexical scoping of these *variables* have to be implemented manually in BPMN at the expense of additional data objects.

In BPMN, the set of type definitions is more restricted than in BPEL. A fundamental limitation in BPMN is the absence of an alignment of the data perspective on the one hand, and the service addressing information on the other hand. As mentioned before, BPEL *variables* might be typed to contain *partner links*, which, in turn, might be modified and applied in message activities. Consequently, dynamic binding of services can be realised in BPEL. On the contrary, service addressing information (participants and web services) are not defined using properties in BPMN. Therefore, they cannot be modified.

The mechanism to provide exception handlers with fault data is another pitfall in a BPEL-to-BPMN mapping. Neither data objects, nor properties can be referenced by an error end event to specify data for propagation. Owing to the absence of explicit data scoping rules, the question which data might be accessed by activities in exception flow remains an open issue.

3.10 Data Mediation and Validation

In order to realise data mediation, BPEL introduces the *assign* activity. It can be applied to copy data from *variables* to other *variables* or to generate new data by means of *expressions*. Besides the basic copy functionality, the *assign* activity allows for the usage of arbitrary data manipulation operations, which must be defined as extension elements. Addressing potential data inconsistencies, a *validate* activity enables verification of data values against the type definition of the *variable*, e.g. an XML Schema definition.

Data manipulation is realised via assignments in BPMN. They can be defined for all types of activities and flow objects (e.g. events and gateways). Whether they are executed before or after the actual construct is handled depends on the configuration of the assignment. Advanced transformation concepts are assumed to be part of the applied expression language. This aspect has to be considered in a BPEL-to-BPMN mapping. Imagine a scenario, in which a BPEL process uses XPath as its expression language. Regarding a BPEL-to-BPMN, the usage

of XPath as the expression language in BPMN might not be sufficient. In case the BPEL process uses an advanced data transformation operation as part of an *assign* activity, it has to be ensured that this transformation can also be expressed using XPath in BPMN. Further on, validation of data cannot be considered in a BPEL-to-BPMN mapping, owing to the absence of a corresponding mechanism in BPMN.

4 Related Work

A detailed discussion of the conceptual discrepancies between BPEL and BPMN has been published by Recker and Mendling [5]. Their analysis reveals mismatches with respect to domain representation capabilities, control flow support and process representation paradigms. It is based on assessments of BPMN and BPEL in the field of Representation Theory [13,14] and the workflow patterns framework [15,16]. The latter implies that not all eventualities might be captured, as for instance BPEL's dead path elimination capability cannot be traced back to control flow patterns in its entirety. Further on, the analysis is based on an older version of BPEL (WS-BPEL 1.1) and does not consider semantic details (e.g. of the compensation handling framework). As a result, the pitfalls identified in this paper have not been addressed.

Pattern-based assessments of BPMN and BPEL have also been presented in the field of data flow and exception handling. BPEL in its version 1.1 has been evaluated against the data and exception patterns by Russel et al. [17,18]. An assessment of BPMN 1.0 against the data flow patterns was presented by Wohed et. al. [15], while exception handling has been discussed in [17]. These assessments show that there are serious deviations between both languages in terms of their expressiveness.

As mentioned in Section 3.8, incompatibilities between BPMN and BPEL regarding process instantiation have been discussed in [12]. The authors introduce a framework to categorise process instantiation mechanisms. As part of that, an evaluation of BPMN and BPEL reveals different expressiveness for certain instantiation patterns.

Various work on BPMN-to-BPEL transformations — the complement to the mapping discussed in this paper — has been published in recent years. White [19] presented initial ideas on such a transformation without providing a concrete mapping definition. As mentioned above, the BPMN specification [2] also outlines a BPMN-to-BPEL mapping. However, these approaches focus on high-level alignment and do not dive into semantic details.

From an academic perspective, a notable mapping has been published by Ouyang et al. [3]. Albeit restricted to a subset, they specify a formal mapping of BPMN models into BPEL processes and implemented it in a tool[1]. Their approach realises discovery of certain process patterns, which, in turn, are mapped on structured activities in BPEL.

[1] http://www.bpm.fit.qut.edu.au/projects/babel/tools/

Notwithstanding the issues discussed in academia, first products implementing BPMN-to-BPEL transformations become available in industry. Telelogic's System Architect[2] supports the generation of BPEL processes from BPMN diagrams for a limited subset. Going one step further, eClarus' Business Process Modeler[3] promises enabling of round-trip engineering for BPMN and BPEL. The implemented mapping algorithm has been sketched by Gao [4]. It potentially involves restructuring of the BPMN process flow in order to create BPEL-isomorphic processes.

As some of the presented pitfalls result from ambiguous semantics, formalisations of both languages are another field of related work. Numerous approaches have been presented in order to formalise BPEL semantics, please refer to van Breugel and Koshkina [20] for an overview. BPEL formalisations are based on abstract state machines [21,22], Petri nets [23,24], and process algebras [25] to name just a few of the applied formalisms. Most of these approaches have been defined for a certain field of application. Therefore, they consider a subset of BPEL constructs or focus on the control flow only.

For BPMN, two formalisations have recently been presented. Dijkman et al. [9] introduced a formalisation for a subset of BPMN using Petri nets. Wong and Gibbons [26] specified semantics for a BPMN subset using CSP process algebra. Targeting clarification of semantics instead of a complete formal mapping, a formalisation of an extended version of BPMN has been presented in [27,28]. Naturally, such an extended version of BPMN avoids most of the presented pitfalls, as it is more expressive and comes with well-defined semantics.

5 Discussion

In order to approach a feature-complete mapping, the presented issues have to be tackled. That involves clarification of BPMN semantics (for instance for the complex gateway) and the definition of new concepts in BPMN (for instance non-interruptive event-handling). Besides, we see the handling of discrepancies in the semantics as the main challenge. Assuming that design decisions that have been taken in BPMN (for instance to trigger compensation by events and not by activities) should be kept to ensure backwards-compatibility, these mismatches have to be handled in the transformation. It is foreseeable that these issues complicate BPEL-BPMN-round-tripping significantly.

Another important aspect of an alignment of BPEL and BPMN is the question, which concepts are considered to be first-class-citizens in both languages. That, in turn, requires to decide, which concepts have to be seen as execution relevant process configuration and which of them are essential for understanding of the process. Despite the distinct goals of both languages, the vision of BPEL-BPMN-round-tripping requires all information to be mapped from BPEL to BPMN, even the low-level process configuration. If any information is lost during transformation, it would have to be added again during the generation of

[2] http://www.telelogic.com/products/systemarchitect/systemarchitect/
[3] http://www.eclarus.com/products_soa.html

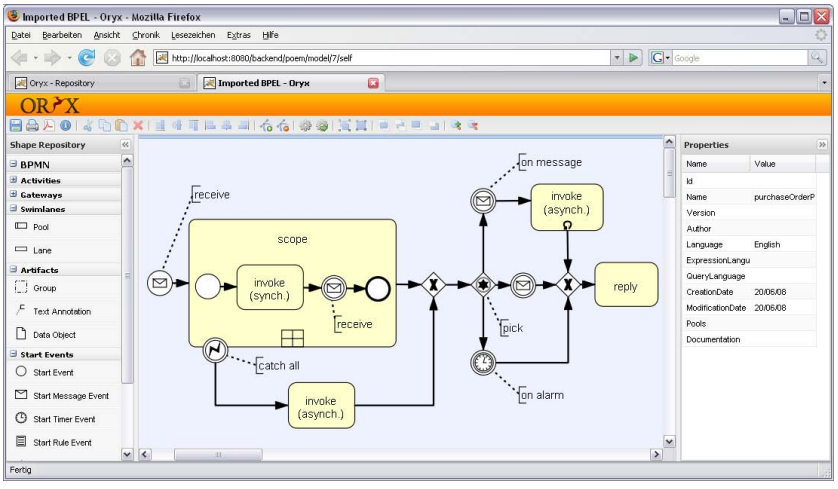

Fig. 8. A BPEL process imported in the web-based process editor Oryx

BPEL processes out of the BPMN model. Nevertheless, it is reasonable to hide most of the process configuration in attributes of the BPMN model, as it is done with information about message correlation. However, we would assume each BPEL activity (except the *empty* activity) to be represented by one or more constructs in BPMN. This is not always the case. For instance, the BPEL *assign* activity is reflected by assignment attributes in the respective BPMN model. Further on, data associations are applied to visualise input and output parameters and are not intended to visualise data transformations. Therefore, BPMN lacks any visualisation of data mediation, which hampers understanding of the data flow. Despite missing graphical representation, it might be possible to map a BPEL construct to BPMN in its entirety. Nonetheless, consideration of this aspect is important with respect to applicability of a BPEL-to-BPMN mapping.

6 Conclusion

Recent research dedicated to an alignment of BPEL and BPMN largely neglects the challenges of a mapping from BPEL to BPMN. Up to now, only a few aspects (e.g. process instantiation) were examined in this context. In this paper we presented a detailed analysis of a BPEL-to-BPMN mapping covering all BPEL constructs and their explicit and implicit semantics. In particular, we revealed various issues with respect to concepts, for which a mapping seems to be apparent. Therefore, we raise the awareness of the pitfalls of such a mapping.

We expect our findings to be of high value for implementations targeting BPEL-BPMN-round-trip engineering. The aforementioned products implementing such a mapping tend to assume BPEL semantics for all BPMN constructs. This is not in line with the semantics defined by the BPMN specification. As

part of our analysis, we implemented a BPEL-to-BPMN mapping for Oryx[4], an open-source modelling framework. This mapping implementation takes the presented issues into account. A BPMN process that was created from an exemplary BPEL file is shown in Figure 8.

Further on, our findings relate to the upcoming standardisation of BPMN 2.0. The respective Request For Proposal (RFP)[5] requests not only precise definition of execution semantics, but also the specification of a BPMN subset that can be mapped to BPEL. In this context, our work could guide the discussion on a mappable subset. Moreover, an analysis of the relevance of certain BPEL constructs (e.g. *termination handlers*) for real-world scenarios would be usefull to judge about the impact of our identified pitfalls. That, in turn, would hint at the necessity of introducing new contructs into BPMN.

References

1. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. Technical report, OASIS (January 2007)
2. OMG: Business Process Modeling Notation (BPMN) 1.1 (January 2008)
3. Ouyang, C., Dumas, M., ter Hofstede, A.H.M., van der Aalst, W.M.P.: From BPMN Process Models to BPEL Web Services. In: ICWS, pp. 285–292. IEEE Computer Society, Los Alamitos (2006)
4. Gao, Y.: BPMN-BPEL Transformation and Round Trip Engineering. Technical report, eClarus Software (2006)
5. Recker, J., Mendling, J.: On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In: Proceedings of the 11th EMMSAD (June 2006)
6. Barreto, C., et al.: Web Services Business Process Execution Language Version 2.0 Primer. Technical report, OASIS (May 2007)
7. White, S.A.: Introduction to BPMN. Technical report, IBM (2004)
8. Barros, A.P., Dumas, M., ter Hofstede, A.: Service Interaction Patterns. In: Proceedings of the 3rd BPM, Nancy, France. Springer, Heidelberg (2005)
9. Dijkman, R., Dumas, M., Ouyang, C.: Semantics and Analysis of Business Process Models in BPMN. In: IST (2008) (accepted for publication)
10. Wynn, M., Edmond, D., van der Aalst, W., ter Hofstede, A.: Achieving a General, Formal and Decidable Approach to the OR-join in Workflow using Reset nets. In: Applications and Theory of Petri Nets, vol. 3536, pp. 423–443 (2005)
11. Dumas, M., Grosskopf, A., Hettel, T., Wynn, M.: Semantics of BPMN Process Models with OR-joins. In: Proceedings of the 15th CoopIS, Vilamoura, Portugal (November 2007)
12. Decker, G., Mendling, J.: Instantiation Semantics for Process Models. In: Proceedings of the 6th BPM, Milan, Italy (September 2008)
13. Recker, J., Indulska, M., Rosemann, M., Green, P.: Do Process Modelling Techniques Get Better? A Comparative Ontological Analysis of BPMN. In: Proceedings of the 16th ACIS, Sydney (2005)
14. Green, P., Rosemann, M., Indulska, M., Manning, C.: Candidate interoperability standards: An ontological overlap analysis. Data Knowl. Eng. 62, 274–291 (2007)

[4] http://www.oryx-editor.org/
[5] http://www.bpmn.org/Documents/BPMN%202-0%20RFP%2007-06-05.pdf

15. Wohed, P., van der Aalst, W.M., Dumas, M., ter Hofstede, A.H., Russell, N.: Pattern-based Analysis of BPMN-An extensive evaluation of the Control-flow, the Data and the Resource Perspectives. BPM Center Report BPM-05-26 (2005)
16. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Analysis of Web Services Composition Languages: The Case of BPEL4WS. In: Proceedings of the 22nd ER, Chicago, USA (October 2003)
17. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Workflow exception patterns. In: Proceedings of the 18th CAiSE, Luxembourg (June 2006)
18. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow Data Patterns (Revised Version). Technical Report FIT-TR-2004-01, Queensland University of Technology, Brisbane, Australia (April 2004)
19. White, S.: Using BPMN to Model a BPEL Process. BPTrends 3(3), 1–18 (2005)
20. van Breugel, F., Koshkina, M.: Models and Verification of BPEL. Technical report, York University (September 2006) (to appear)
21. Fahland, D.: Complete Abstract Operational Semantics for the Web Service Business Process Execution Language. Informatik-Berichte 190, Humboldt-Universität zu Berlin (September 2005)
22. Farahbod, R., Glässer, U., Vajihollahi, M.: A Formal Semantics for the Business Process Execution Language for Web Services. In: Proceedings of the WSMDEIS, Miami, USA (May 2005)
23. Ouyang, C., van der Aalst, W., Breutel, S., Dumas, M., ter Hofstede, A., Verbeek, H.: Formal Semantics and Analysis of Control Flow in WS-BPEL. BPM Center Report BPM-05-13, BPMcenter. org (2005)
24. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: Proceedings of the 4th WS-FM, Brisbane, Australia (September 2007)
25. Ferrara, A.: Web Services: A Process Algebra Approach. In: Proceedings of 2nd ICSOC, New York City, USA (November 2004)
26. Wong, P.Y., Gibbons, J.: A process semantics for BPMN (submitted for publication) (2007)
27. Grosskopf, A.: xBPMN - Formal Control Flow Specification of a BPMN based Process Execution Language. Master's thesis, Hasso Plattner Institute for IT Systems Engineering, Potsdam, Germany (July 2007)
28. Schreiter, T.: xBPMN++ - Towards Executability of BPMN: Data Perspective and Process Instantiation. Master's thesis, Hasso Plattner Institute for IT Systems Engineering, Potsdam, Germany (February 2008)