Realising Dead Path Elimination in BPMN

Matthias Weidlich, Alexander Grosskopf
Hasso Plattner Institute
Potsdam, Germany
{*matthias.weidlich,alexander.grosskopf*}*@hpi.uni-potsdam.de*

Alistair Barros
SAP Research
Brisbane, Australia
alistair.barros@sap.com

***Abstract*—The Web Service Business Process Execution Language (BPEL) lacks any standard graphical notation. Various efforts have been undertaken to visualize BPEL using the Business Process Modelling Notation (BPMN). Although this is straightforward for the majority of concepts, it is tricky for the full BPEL standard, partly due to the insufficiently specified BPMN execution semantics. The upcoming BPMN 2.0 revision will provide this clear semantics. In this paper, we show how the dead path elimination (DPE) capabilities of BPEL can be expressed with this new semantics and discuss the limitations. We provide a generic formal definition of DPE and discuss resulting control flow requirements independent of specific process description languages.**

***Keywords*-BPEL, BPMN, dead path elimination, model transformation, round-tripping, process model alignment**

I. Introduction

The Business Process Modeling Notation (BPMN) [1] as standardised by the OMG has emerged as the de-facto standard for business process modelling. Its development was at least partly driven by the lack of any standard graphical notation for the Web Service Business Process Execution Language (BPEL) [2] which is the leading standard in the field of Web Service orchestrations. Between the poles of high-level business alignment and low-level process enactment, a lot of research has been conducted on transformations between both languages [3]–[7]. A bidirectional alignment, however, suffers from conceptual mismatches [8]. In an earlier work, we revealed BPEL's dead path elimination (DPE) capabilities as a major issue for BPEL-to-BPMN transformations, in particular [9].

In general, DPE refers to a control flow concept that supports explicit skipping of activities. Skipping an activity can then lead to the skipping of subsequent activities, to *eliminate* the *dead path*. By using conditions (conditional DPE), subsequent activities can be executed even though preceding activities were skipped.

This paper elaborates on the realisation of DPE in BPMN. It relies on the semantics of the inclusive OR gateway and the complex gateway. For both constructs, the current BPMN standard [1] does not provide an unambiguous definition [9], [10]. The upcoming revision 2.0 of BPMN [11] addresses the execution semantics of all constructs. In this paper, we evaluate BPMN's DPE capabilities based on the latest public draft of BPMN 2.0 [11] and the new semantics to be expected. We show how DPE can be realised in BPMN 2.0 and what limitations remain. Based thereon, we illustrate the application of our findings in the context of BPEL-to-BPMN transformations.

The remainder of this paper is structured as follows. Section II illustrates the concept of DPE by means of an example process, while we dedicate Section III to a formal definition of DPE and a discussion of its control flow requirements. Afterwards, Section IV introduces the realisation of DPE in BPM, discusses the mapping of DPE in BPEL to BPMN, and its limitations. We review related work in Section V and conclude in Section VI.

II. Example with Dead Path Elimination

In order to illustrate the essence of dead path elimination, Figure 1 depicts an exemplary BPEL process. This order-to-shipping scenario is visualised in a notation similar to the one used in the BPEL specification. The process is triggered by the reception of a purchase order. Afterwards, a *flow* activity is activated which, in turn, starts two single activities and two sequences of activities in parallel. Four of the activities have outgoing *control links* (*links* for short) named with small letters, while *transition conditions* are declared at the source of the *links*. Further on, a *join condition* is specified for activity *Schedule Production*. Semantics for this example are as follows: After an activity with an outgoing *link* (e.g. *Query Customer Status*) ends, the *transition condition* (e.g. *Master Agreement?*) is evaluated and depending on the result, the control link is set to either 'true' or 'false'. These Boolean *link* values are considered in *join conditions*. Thus, activity *Schedule Production* is executed only, if the ordered items are not on stock at the shipping hubs or if there exists a master agreement with the customer. Skipping of activity *Schedule Production* will set its outgoing *link* to 'false', which invariably leads to skipping of activity *Schedule Transport to Shipping Hub*. This behaviour is referred to as dead path elimination.

III. DPE in Business Processes

Dead path elimination (DPE) has been widely discussed in the context of executable workflows [12]–[14]. However, we are not aware of a concise model that captures the essence of DPE formally. Therefore, this section defines

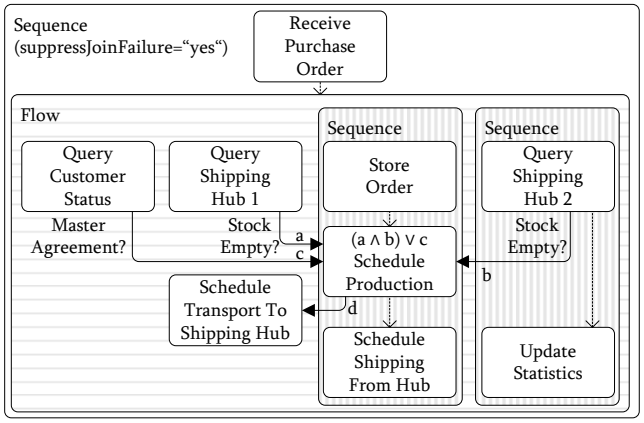


Figure 1. BPEL example scenario with DPE

DPE in the first part. Based thereon, we discuss the resulting requirements for token-flow based languages and analyse the well-known workflow patterns for coverage of these requirements. Afterwards, we present a short survey on the capabilities of common process languages to cope with DPE.

A. Definition of Dead Path Elimination

First and foremost, we specify the notion of an acyclic process model in order to provide a concise definition of the very core of DPE. The definition of an acyclic process model has been inspired by the notion of *synchronizing workflow models* as presented in [15].

Definition 1 (Process Model (PM)): A process model is a tuple $P = (N, \mapsto)$, where

- N is a set of process nodes and
- $\mapsto \subseteq N \times N$ is a flow relation connecting process nodes.

P is called *acyclic*, if the flow relation $\mapsto$ is acyclic.

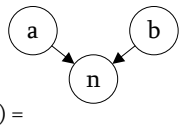
As an abbreviation we will use $\triangleright n$ as the set of all arcs directly preceding n, i.e. $\triangleright n := \{(x, y) \in \mapsto \mid y = n\}$. $n\triangleright$ is defined accordingly for succeeding arcs. Based thereon, we define process models, for which the execution of each node is captured by a relation that considers the state of its preceding arcs. The set of states is given by $S = \{\perp, executed, skipped\}$.

Definition 2 (PM with Explicit Enabling): A process model with explicit enabling is a tuple $P_e = (N, \mapsto, \mathit{signal}, \mathit{ex})$ where

- $(N, \mapsto)$ is an acyclic process model and
- for every node n, label $s \in S$, $\mathit{signal}(n, s)$ specifies states for arcs succeeding n:
  - $\mathit{signal}(n, executed) \subseteq (n\triangleright \rightarrow \{\perp, true, false\})$,
  - $\mathit{signal}(n, skipped) = (n \triangleright \times \{false\})$,
  - $\mathit{signal}(n, \perp) = (n \triangleright \times \{\perp\})$,
- for every node n, $\mathit{ex}(n)$ specifies the combinations of states of its preceding arcs for which n will execute, where $\mathit{ex}(n) \subseteq \wp(\triangleright n \times \{true, false\})$, such that $\forall\, e \in \mathit{ex}(n), a \in \triangleright n\ [\ \exists\ s \in S\ [\ (a, x) \in e\ ]]$.

From here on, we use the term process model to refer to an acyclic process model with explicit enabling. The definition of *ex* implies that arbitrary combinations of arc states can be defined to trigger execution. Figure 2 shows an example where node n has two predecessors a and b, i.e. $\triangleright n = \{(a, n), (b, n)\}$. As specified in the $signal$ relation, arc (a, n) is set to $false$ in all cases, i.e. a might have been executed or skipped. It is worth to mention that our definition allows arcs only to be set to $true$, if their source node has been executed. Further on, n will execute if arc (a, n) is in state $false$ and (b, n) in state $true$ or $false$. Further, we define $ex^{-1}(n)$ as the relation containing all tuples of arc / state combinations, not leading to an execution of n, i.e. $ex^{-1}(n) = \wp\,(\triangleright n \times \{true, false\}) \setminus ex(n)$.

signal(a,executed) = {((a,n),false)}
signal(a,skipped) = {((a,n),false)}



ex(n) =
{{((a,n),false),((b,n),true)} ,
{((a,n),false),((b,n),false)}}

Figure 2. Sample for conditional enabling

Next, we present restrictions on *ex* by introducing *process models with plain enabling* as special cases. While *process models with conditional enabling* can use *ex* to define arbitrary enabling conditions, for *process models with plain enabling* the *ex* relation contains all cases in which at least one incoming arc is $true$.

Definition 3 (PM with Plain Enabling): Let $P_e = (N, \mapsto, \mathit{signal}, \mathit{ex})$ be a process model. Then, P_e is a process model with plain enabling, if for all nodes $n \in N$ it holds $\mathit{ex}(n) \cap \{(\triangleright n \times \{false\})\} = \emptyset$.

The example from Figure 2 does not show plain enabling. Here, $\mathit{ex}(n)$ would look as follows in order to comply with plain enabling: $\mathit{ex}(n) = \{\{((a, n), true),\ ((b, n), true)\}, \{((a, n), true),\ ((b, n), false)\},\ \{((a, n), false), ((b, n), true)\}\}$. Further on, we characterise process instances by the notion of their state, while neglecting means to distinguish different instances.

Definition 4 (State of Process Instance): Let $P_e = (N, \mapsto, \mathit{signal}, \mathit{ex})$ be a process model. Then a tuple $I = (N, \mapsto, \mathit{signal}, \mathit{ex}, \mathit{st})$ is a process instance of P_e in a certain state, where $\mathit{st} : N \rightarrow \{\perp, executed, skipped\}$ is a function assigning status labels to nodes.

The state of a process instance is given by the function *st*. For obvious reasons, all nodes are neither executed nor skipped in the initial state, i.e. $\forall\, n \in N : \mathit{st}(n) = \perp$.

In order to capture the behaviour of a process instance, we specify its semantics using a state transition system. The transition relation $\rightarrow$ is defined by a set of transition rules, consisting of a *context*, a *source state* and a *target state*. The context is a set of guard conditions, which have to hold in order for the rule to apply. Semantics for the process instance are given by the following three transition rules (a context is noted above the line, while the transition from the

source state to the target state is noted below the line). As a shorthand notation, we use $\bullet n := \{x \in N |\ x \mapsto n\}$ for nodes preceding n.

$$\text{Init}\quad \frac{st(n) = \bot \ \wedge\ \triangleright n = \emptyset}{st\ \rightarrow\ (st \setminus (n, \bot)) \cup (n, executed)}$$

$$\text{Exec}\quad \frac{st(n) = \bot \wedge \bigcup_{p \in \bullet n} signal(p, st(p)) \cap (\triangleright n \times S) \in ex(n)}{st\ \rightarrow\ (st \setminus (n, \bot)) \cup (n, executed)}$$

$$\text{Skip}\quad \frac{st(n) = \bot \wedge \bigcup_{p \in \bullet n} signal(p, st(p)) \cap (\triangleright n \times S) \in ex^{-1}(n)}{st\ \rightarrow\ (st \setminus (n, \bot)) \cup (n, skipped)}$$

Rule $Init$ starts processing of the process instance, as it represents the execution of a node without predecessors, rule $Exec$ marks the execution of a node n, and rule $Skip$ realises skipping. The actual choice between these two rules is based on $ex(n)$. We test whether the combination of the states of preceding arcs is contained in $ex(n)$ or $ex^{-1}(n)$. These states, in turn, are determined by evaluating the signal function for all preceding nodes. Please note, that in both rules it is implicitly ensured that all predecessors of the respective node have been either skipped or executed.

After we introduced our formal foundation for process models and process instances, we are able to capture the characteristics of DPE.

Definition 5 (Dead Path Elimination): Let $I = (N, \mapsto, ex, st)$ be a process instance in a certain state. Then the term dead path elimination refers to a state transition described by transition rule $Skip$ during processing of I.

B. DPE Requirements on Token Flow

According to the formal model presented in the previous section, enabling of a node is based on the state of all preceding arcs. These arc states, in turn, depend on the state of their source nodes (however, they might not be equal, e.g., an activity might be skipped, but the outgoing links are set to $true$). In contrast to that, common process description languages activate nodes based on token passing. Thus, before we are able to summarise the requirements of DPE, we have to address how arc states are propagated using token flow. There are three possible realisations.

- *Typed Tokens*
  If the process description language supports at least two distinguishable types of tokens (*true* and *false* tokens), the state of an arc is indicated by sending a corresponding token.
- *One Arc and Non-Local Semantics*
  A token is sent on an arc to indicate the state *true*. Non-local semantics is applied to distinguish the states *false* and $\bot$ of the respective arc.
- *Dedicated Arcs*
  There are two arcs representing the states *true* and *false*, respectively. An untyped token is sent exclusively on one of them.

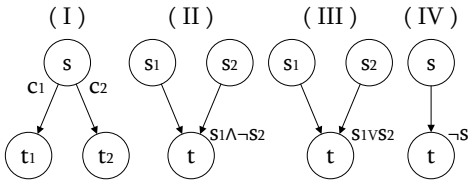


Figure 3. Control flow requirements

Based thereon, we summarise the requirements of DPE on token flow based languages as depicted in Figure 3. First, we need to be able to activate a branch under specified conditions, as node activation might not imply activation of the succeeding branches. Thus, means to send tokens based on conditions are essential (I). In order to realise DPE, we also have to merge control flow based on a *join condition* (II). This condition references the states of preceding arcs. Thus, it corresponds to the relation ex in Definition 2.

The concrete realisation of this flow pattern depends on the way the arc states are indicated using token flow. Requirement (II) can be further classified. In order to realise plain DPE, the join condition must require that one of the preceding arcs has been activated (III). Another important specialisation of requirement (II) is a join condition that can express that a preceding arc was not activated (IV).

C. Going Beyond Workflow Patterns

It is worth to notice that the aforementioned requirements are not covered by the well-known control-flow patterns [16] of the workflow patterns framework[1] to the full extent. While requirement (I) is captured by the *Multi-Choice* pattern, support for the requirements depends on how the arc state (*true* or *false*) is propagated.

In case of typed tokens, requirements (III) and (IV) are captured by the *Local Synchronizing Merge* pattern, whereas the generalisation of both patterns, namely requirement (II), cannot be traced back to any pattern. There is no means to specify an arbitrary join condition over the type of received tokens.

If one arc and non-local semantics are used to model the state of an arc, requirement (III) is covered by the *General Synchronizing Merge* pattern. However, requirement (IV) is not captured. The control flow pattern requires an activation of at least one of the incoming arcs, which is not the case for requirement (IV). Again, requirement (II) is also not support as join conditions based on the state of incoming arcs cannot be expressed.

In case dedicated arcs are used to indicate arc states, requirement (IV) is covered by the *Simple Merge* pattern, whereas requirements (II) and (III) are not captured directly. A workaround would be to encode a join condition using *Synchronisation* and *Simple Merge* patterns. Due to the implied

[1] http://www.workflowpatterns.com/

Table I
SUPPORT FOR DPE

Language	Support for plain DPE	Support for cond. DPE
WS-BPEL 2.0	control links	control links, join conditions
UML 2.1.1 AD	–	–
EPC	OR-split, OR-join[2], conditional events	–
YAWL	OR-split, OR-join	–

verbosity, it seems questionable, whether this workaround is feasible for complex scenarios. All possible combinations of activity states that satisfy the join condition would have to be modelled explicitly.

We observe that the limited coverage of the requirements for DPE mainly results from the neglect of arc identities in all control flow patterns. In contrast to diverging patterns (e.g. a *Multi-Choice*), converging patterns always assume indistinguishable arcs. This precludes any possibility to specify merging behaviour based on a condition over the type of incoming tokens or the state of incoming arcs. Even if two dedicated arcs are used to propagate a node state, these two arcs cannot be correlated in the set of incoming arcs.

D. DPE in Process Description Languages

In the following, we present a short survey on the capabilities of common process description languages to realise DPE scenarios. An overview is given in Table I.

As discussed for the example in Section II, BPEL [2] supports DPE through *control links*, *transition conditions*, and *join conditions* in a *flow* construct. BPEL prohibits cyclic link dependencies and control links must not enter or leave repeatable constructs. The BPEL semantics might lead to different behaviour of a process with and without DPE. That results from the potential usage of a negation operator in join conditions and the requirement to wait for synchronisation of all branches before the join condition is evaluated. This phenomenon, referred to as a *side effect*, has been studied by van Breugel and Koshkina in [17]. They also propose different DPE semantics for BPEL to avoid side effects, as they assume these effects to be unintended in any case. With our definition of DPE in Section III-A, we do not follow this opinion and comply with the existing BPEL semantics that has explicitly been affirmed by the OASIS WS-BPEL Technical Committee. The committee rejected a proposal to restrict join conditions in order to avoid side effects (cf. issue 14 in the WS-BPEL Issues List[3]).

UML [18] introduces Activity Diagrams (AD) in order to describe business processes. Conditional divergence of branches can be expressed by *fork nodes* with *guard conditions*. In order to join concurrent branches, UML AD introduce *join nodes*. A plain *join node* synchronises, if all incoming arcs have been activated. Further on, a *join specification* allows for further specification of the merging behaviour. According to [18], a *join specification* is a conjunction of identifiers of a subset of incoming arcs as illustrated in Figure 4. Thus, there are no means to synchronise multiple branches, of which only a subset has been activated before (i.e. there is no *OR-join* in UML AD). Even in case the disjunction operator would be allowed in the *join specification*, this issue would not be resolved as discussed in [19]. Work on formalisation of semantics of UML AD [20], [21] typically neglects the *join specification*. Owing to the restricted expressiveness of the *join specification*, UML AD support neither plain DPE nor conditional DPE.

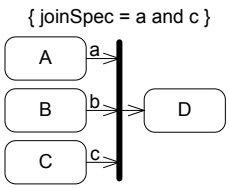


Figure 4. JoinSpec in UML AD

While we can model conditional divergence of branches in EPCs [22] directly, a lot of research has been conducted on the corresponding convergence of branches through an OR-join [23], [24]. Grounded on an OR-join formalisation, Mendling shows the realisation of plain DPE in [25] when mapping BPEL to EPCs. However, conditional DPE is not supported due to the absence of a construct that allows for merging of multiple branches based on a condition. This also prohibits any realisation of DPE with dedicated arcs to distinguish the *true* and *false* state.

Aiming at increased support for workflow pattern and well-defined semantics for the OR-join construct, van der Aalst and ter Hofstede introduced YAWL in [26]. YAWL semantics are grounded on untyped token flow. Plain DPE is supported via OR-splits and OR-joins. As flow relation entities are indistinguishable in YAWL, we cannot specify conditional convergence behaviour. Thus, conditional DPE is not supported.

IV. DEAD PATH ELIMINATION IN BPMN

This section shows how dead path elimination can be realised in BPMN. We first show the realisation of plain DPE and then discuss the challenges associated with conditional DPE. Afterwards, DPE is reviewed discussed in the context of BPEL-BPMN-transformations.

In general, semantics for BPMN are based on untyped token flow, which excludes the possibility to apply *true* and *false* tokens in order to propagate a state of an activity (cf. Section III-B). As mentioned before, two constructs that are important for DPE have underspecified semantics in the current BPMN standard [9], [10]. Therefore, we base our discussion on the semantics proposed in the course of the BPMN 2.0 standardisation [11].

[2]The OR-join has underspecified execution semantics in EPCs.

[3]http://www.oasis-open.org/apps/group_public/download.php/11285/wsbpel_issues34.html#Issue14

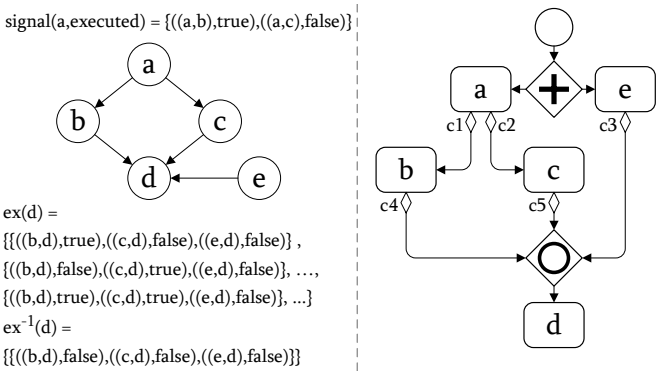


Figure 5. Realisation of plain DPE in BPMN

A. Realisation of Plain DPE

According to our formal model defined in Section III-A, plain dead path elimination requires a certain activity to be executed, if *one* of the preceding arcs has been set to $true$. Therefore, the enabling condition can be seen as an inclusive disjunction over the states of preceding arcs. Consequently, we can base our realisation on one arc and non-local semantics (cf. Section III-B) in order to distinguish the states (*true* and *false*) of an arc. We illustrate the realisation by means of the exemplary process model that is depicted on the left side of Figure 5. It has two nodes without predecessors, a and e, that are executed initially. Afterwards, the state for arcs starting at these nodes are set based on the relation $signal$ which is illustrated for node a. Further on, we see that the ex relation requires at least one of the preceding arcs of node d to be in state $true$ which corresponds to plain DPE configuration.

The realisation of this scenario in BPMN is grounded on conditional flow and the merging inclusive OR-gateway as illustrated on the right side of Figure 5. The joint execution of all nodes without predecessors is realised using a parallel split gateway in front of the respective BPMN activities. In general, the state $true$ of an arc in our formal model is represented by sending a token on the respective sequence flow in BPMN. In our formal model, a state of an arc is derived via the relation $signal$. This condition, in turn, is reflected by the conditional flow in the corresponding BPMN model. The relation $signal$ does only consider the node states, however, the corresponding conditions ($c1, c2, c3, c4$, and $c5$) in the BPMN model might also be based on process data, as the distinct relation between the node states and the arc states is not required to observe DPE.

Further on, the states $false$ and $\perp$ of an arc in our formal model have to be distinguished in the respective BPMN model as well. This is achieved by the non-local semantics of the OR-join. That is, the join waits for all tokens *'anywhere upstream of this sequence flow, i.e. there is no path from a token to this sequence flow unless the path visits a dominator or postdominator [..] A node dominates a sequence flow if each path from a source of the graph to the sequence flow visits that node.'* [11]. Due to the acyclic nature of our DPE model, postdominators will never impact on the semantics of the OR-join and at most one token will be send on each arc. Therefore, the arc state $false$ of our formal model can be represented by the absence of a sent token which is detected by the OR-join semantics.

Note that stalling of the process is possible even in plain DPE configuration. Regarding the example in Figure 5, we could define *signal*$(a, executed) = \{((a, b), false), ((a, c), false)\}$. Thus, the arcs succeeding node a are always set to $false$. That would result in skipping of both nodes b and c. If also *signal*$(e, executed) = \{((e, d), false)\}$, node d will be skipped. In plain DPE configuration all potential nodes after d will be skipped either, as the relation $signal$ can set an arc state to $true$ solely if the source node has been executed (cf. Definition 2). This complies with our BPMN model as no tokens are sent after activities a and e have been executed. Consequently, another activity might never be executed afterwards.

B. Realisation of Conditional DPE

In contrast to plain DPE, conditional DPE allows for the specification of an arbitrary combination states of preceding arcs in order to execute a certain activity. Therefore, the OR-join cannot be applied as in the case of plain DPE. *Nomen est omen*, however, the complex gateway is intended *'to model complex synchronization behaviour'* [11]. In general, firing behaviour of the complex gateway is specified using an *activation expression* which is evaluated with the reception of every token. It might reference separate token counters for each of the directly preceding flow arcs (but not for flow arcs elsewhere in the model). Once the activation expression evaluates to true, a token is created on the *normal output* flow. Afterwards, *'the gateway waits for a token on each of those incoming gates that it has not yet received a token in the first phase unless such a token does not come (cf. inclusive join behaviour)'* [11]. Thus, the non-local semantics known from the OR-join is reused. Finally, the gateway resets by creating a token on the optional *reset output* flow.

Conditional DPE without negation. First, we consider conditional DPE without negation. That is, a join condition requires a certain flow arc to be either in state $true$ or both states ($true$,$false$) are allowed in the join condition (see arc (c, d) in Figure 6).

That is, whenever an execution configuration (i.e. a set in $ex(n)$) specifies state *false* for an arc, $ex(n)$ has to contain another execution configuration, in which this arc is required to be *true*, whereas the states for all other arcs remain unchanged. Formally, for all nodes n it holds $\forall e \in ex(n), t \in \mapsto [(t, false) \in e \Rightarrow \exists\ e_2 \in ex(n)[e_2 \setminus e = \{(t, true)\}]]$. In this case, semantics of the complex gateway allow for the realisation of conditional DPE as illustrated in Figure 6. This

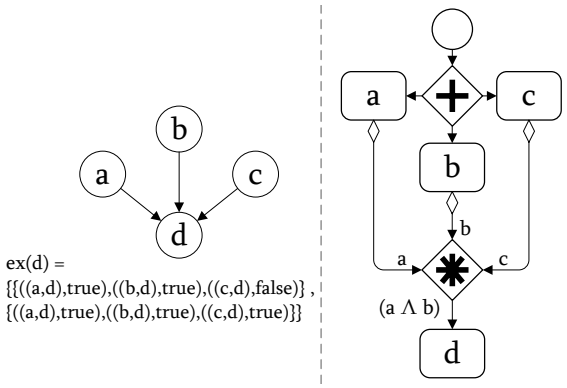


Figure 6. Realisation of conditional DPE without negation in BPMN

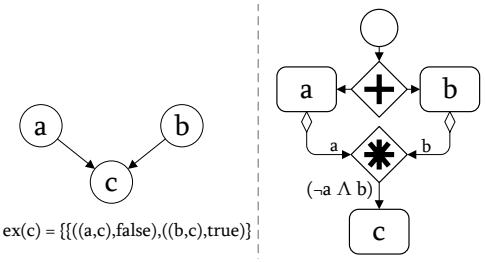


Figure 7. Conditional DPE with negation cannot be realised in BPMN

realisation resembles the one introduced for plain DPE except for the mapping of the join condition. In the process model on the left side node d is executed solely if the preceding arcs from node a and b are set to $true$. This condition has to be encoded in the activation expression of a complex gateway. It requires the token counters of two gates to be bigger than one, i.e. $tc_a \geq 1 \ \wedge \ tc_b \geq 1$. With the arrival of every token (indicating the state $true$ for the respective arc) this condition is evaluated. Please note that for all activities potentially succeeding activity d the following property holds due to our assumptions. For such an activity, the decision of execution is either based on the state $true$ propagated on the path coming from d, or this path does not impact on this decision.

Conditional DPE with negation. In case a join condition contains negation, conditional DPE cannot be realised in BPMN directly. That is due to the missing possibility to test for token absence on one of the incoming arcs of the complex gateway. Even if the activation expression requires a certain token counter to be zero, it might be the case that a token arrives at a later stage.

Consider, for instance, Figure 7. Setting the activation expression to $tc_a = 0 \ \wedge \ tc_b \geq 1$ would lead to unintended firing of the complex gateway in case execution of activity b finishes before execution of a finished (assuming that both activities send a token on their outgoing conditional flow).

Therefore, this kind of conditional DPE can only be realised if a different approach is chosen in order to distinguish the arc states of our formal model. As mentioned in Section III-B, two dedicated arcs might be applied in order to represent the *true* and *false* states of such an arc. However, this approach does not seem feasible due to the enormous overhead which is illustrated in Figure 8 for the aforementioned join condition.

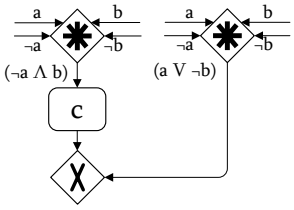


Figure 8. Realisation of conditional DPE with negation is not feasible

C. BPEL Control Links in BPMN

In contrast to BPMN, BPEL supports the full range of DPE variants. BPEL scenarios showing plain DPE or conditional DPE without negation can be mapped to BPMN. For these scenarios the mapping directly follows from the realisation as described above. However, block-structured control flow elements that are part of a process with DPE have to be considered accordingly. This is shown in Figure 9 which illustrates the BPEL process from Section II (which shows conditional DPE without negation) in BPMN. We see that the block-structured BPEL activities, e.g. the BPEL *sequences*, result in additional sequence flows. These flows have to be synchronised using OR-join constructs in order to preserve the execution semantics of the block-structured activities. Consider, for instance, activity *Schedule Shipping from Hub*. It has to be executed after activity *Store Order*, whereas activity *Schedule Production* might be executed in between, depending on the evaluation of its join condition. To cope with this potential execution, the inclusive OR gateway succeeding *Schedule Production* is activated once the first activity *Store Order* finished execution. Due to the non-local semantics, it is ensured that *Schedule Shipping from Hub* will only be executed if *Schedule Production* has finished or cannot be executed at all.

Please note that there is still a slight difference in execution semantics between the BPEL and the BPMN process. As a complex gateway representing a join condition fires immediately after the condition is fulfilled, a subsequent activity (e.g. *Schedule Production*) might run concurrently to activities preceding the complex gateway (e.g. *Query Hub 1*). This potential concurrency cannot occur in the BPEL process because BPEL join conditions are evaluated only after all links are set.

V. Related Work

Besides the numerous publications on BPEL-BPMN-transformations [3]–[9], two fields of research are further related to our work. On the one hand, there are multiple pattern-based assessments of BPMN and BPEL revealing differences in their expressiveness with respect to control

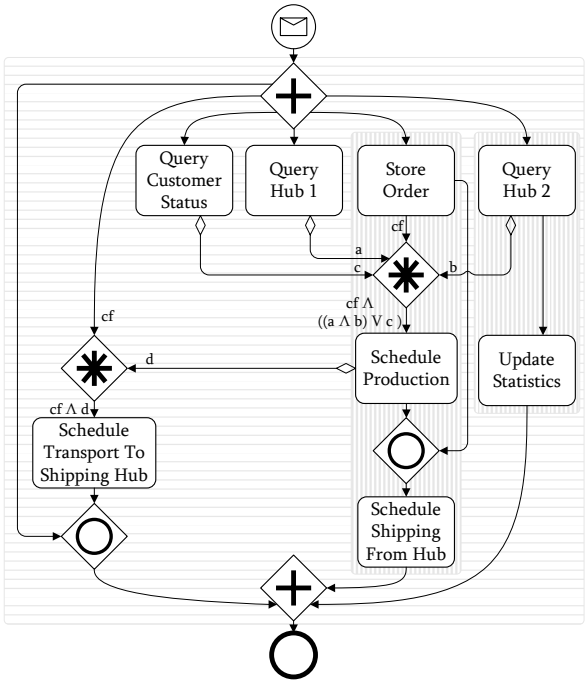


Figure 9. The example scenario in BPMN

flow, data flow, and process instantiation mechanisms [27]–[30]. Albeit based on older versions of both standards, these assessments reveal serious deviations between both languages.

On the other hand, work on formalisations of both languages is related. Numerous approaches have been presented in order to formalise BPEL semantics, refer to van Breugel and Koshkina [31] for an overview. Most of these approaches capture also BPEL's DPE capabilities. For BPMN, existing formalisations [32]–[35] are based on former versions of BPMN. They either omit the inclusive OR and complex gateway or make assumptions that do no longer hold true due to BPMN 2.0.

VI. Conclusion

In the first part, we elaborated on the concept of dead path elimination (DPE) in general. We introduced a formal model capturing the essence of DPE, derived requirements for token flow based languages, and gave an overview of support for DPE in common process description languages. Based thereon, our second contribution is a description of how DPE can be realised in BPMN assuming semantics proposed for BPMN 2.0. As DPE in BPMN is motivated by an alignment with BPEL, we also discussed to which extent BPMN 2.0 can be used to represent BPEL regarding DPE scenarios.

A major result of our work is the precise description of different kinds of DPE along with a discussion, which of these DPE variants can be realised in BPMN. Plain DPE and conditional DPE without negation can be realised directly (although the latter implies a slight deviation in semantics), whereas conditional DPE with negation cannot be expressed in a feasible manner. That, in turn, implies limitations for any BPEL-BPMN-transformation. We solely sketched such a transformation for DPE, so that a concrete transformation algorithm is still to be presented.

Moreover, the rather complex realisation of conditional DPE without negation in BPMN raises questions concerning a negative impact on the understandability of the model. Although semantics might be precise, the intrinsic complexity of such scenarios might be seen as a reason to avoid conditional DPE in BPMN models.

References

[1] OMG, *Business Process Modeling Notation (BPMN) 1.2*, January 2009.

[2] *Web Services Business Process Execution Language Version 2.0*, OASIS, January 2007.

[3] S. White, "Using BPMN to Model a BPEL Process," *BPTrends*, vol. 3, no. 3, pp. 1–18, 2005.

[4] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, and W. M. P. van der Aalst, "From BPMN Process Models to BPEL Web Services," in *ICWS*. IEEE Computer Society, 2006, pp. 285–292.

[5] O. Kopp, D. Martin, D. Wutke, and F. Leymann, "On the Choice Between Graph-Based and Block-Structured Business Process Modeling Languages," in *MobIS*, ser. LNI, vol. 141. GI, 2008, pp. 59–72.

[6] J. Mendling, K. Lassen, and U. Zdun, "On the Transformation of Control Flow between Block-Oriented and Graph-Oriented Process Modeling Languages," *IJBPIM*, vol. 3, pp. 96–108, October 2008.

[7] D. Schumm, D. Karastoyanova, F. Leymann, and J. Nitzsche, "On Visualizing and Modelling BPEL with BPMN," in *Proceedings of the 4th International Workshop on Workflow Management (ICWM2009)*. IEEE Computer Society, May 2009, Workshop-Beitrag.

[8] J. Recker and J. Mendling:, "On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages," in *CAiSE Workshop Proceedings - EMMSAD*, June 2006, pp. 521–532.

[9] M. Weidlich, G. Decker, A. Großkopf, and M. Weske, "BPEL to BPMN: The Myth of a Straight-Forward Mapping," in *OTM Conferences (1)*, ser. LNCS, vol. 5331. Springer, 2008, pp. 265–282.

[10] M. Dumas, A. Großkopf, T. Hettel, and M. T. Wynn, "Semantics of Standard Process Models with OR-Joins," in *OTM Conferences (1)*, ser. LNCS, vol. 4803. Springer, 2007, pp. 41–58.

[11] *Proposal for: Business Process Model and Notation (BPMN) Specification 2.0, V0.9.6*, Submitting Organisations, February 2008. [Online]. Available: http://www.omg.org/cgi-bin/doc?bmi/09-02-01

[12] F. Leymann and W. Altenhuber, "Managing Business Processes as an Information Resource," *IBM Systems Journal*, vol. 33, no. 2, pp. 326–348, 1994.

[13] F. Leymann and D. Roller, *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 1999.

[14] F. Curbera, R. Khalaf, F. Leymann, and S. Weerawarana, "Exception Handling in the BPEL4WS Language," in *Business Process Management*, ser. LNCS, vol. 2678. Springer, 2003, pp. 276–290.

[15] B. Kiepuszewski, A. H. M. ter Hofstede, and W. M. P. van der Aalst, "Fundamentals of Control Flow in Workflows," *Acta Informatica*, vol. 39, no. 3, pp. 143–209, 2003.

[16] N. Russell, A. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar, "Workflow Control-Flow Patterns: A Revised View," BPMcenter.org, BPM Center Report BPM-06-22, 2006.

[17] F. van Breugel and M. Koshkina, "Dead-Path-Elimination in BPEL4WS," in *ACSD*. IEEE Computer Society, 2005, pp. 192–201.

[18] OMG, *Unified Modeling Language: Superstructure, version 2.1.1*, February 2007.

[19] P. Wohed, W. M. van der Aalst, M. Dumas, A. H. ter Hofstede, and N. Russell, *Pattern-based Analysis of UML Activity Diagrams*. Eindhoven, The Netherlands: Working Paper Series, 2004, vol. WP129.

[20] S. Sarstedt and W. Guttmann, "An ASM Semantics of Token Flow in UML 2 Activity Diagrams," in *Ershov Memorial Conference*, ser. LNCS, vol. 4378. Springer, 2006, pp. 349–362.

[21] E. Börger, A. Cavarra, and E. Riccobene, "An ASM Semantics for UML Activity Diagrams," in *AMAST*, ser. LNCS, vol. 1816. Springer, 2000, pp. 293–308.

[22] G. Keller, M. Nüttgens, and A.-W. Scheer, "Semantische Prozeßmodellierung auf der Grundlage 'Ereignisgesteuerter Prozeßketten (EPK)'," Universität des Saarlandes, Tech. Rep., January 1992.

[23] J. Mendling and W. M. P. van der Aalst, "Formalization and Verification of EPCs with OR-Joins Based on State and Context," in *CAiSE*, ser. LNCS, vol. 4495. Springer, 2007, pp. 439–453.

[24] E. Kindler, "On the Semantics of EPCs: A Framework for Resolving the Vicious Circle," in *Business Process Management*, ser. LNCS, vol. 3080. Springer, 2004, pp. 82–97.

[25] J. Mendling and J. Ziemann, "Transformation of BPEL Processes to EPCs," in *EPK*, ser. CEUR Workshop Proceedings, vol. 167, December 2005, pp. 41–53.

[26] W. van der Aalst and A. ter Hofstede, "YAWL: Yet Another Workflow Language (Revised version)," QUT Technical report, Tech. Rep. FIT-TR-2003-04, 2003.

[27] P. Wohed, W. M. van der Aalst, M. Dumas, A. H. ter Hofstede, and N. Russell, "Pattern-based Analysis of BPMN-An extensive evaluation of the Control-flow, the Data and the Resource Perspectives," *BPM Center Report BPM-05-26, BPMcenter. org*, 2005.

[28] P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede, "Analysis of Web Services Composition Languages: The Case of BPEL4WS," in *ER*, ser. LNCS, vol. 2813. Springer, 2003, pp. 200–215.

[29] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst, "Workflow Data Patterns (Revised Version)," Queensland University of Technology, Brisbane, Australia, Tech. Rep. FIT-TR-2004-01, April 2004.

[30] G. Decker and J. Mendling, "Instantiation Semantics for Process Models," in *BPM*, ser. LNCS, vol. 5240. Springer, 2008, pp. 164–179.

[31] F. van Breugel and M. Koshkina, "Models and Verification of BPEL," York University, Tech. Rep., September 2006, to appear.

[32] R. Dijkman, M. Dumas, and C. Ouyang, "Semantics and Analysis of Business Process Models in BPMN," *Information and Software Technology (IST)*, 2008.

[33] P. Y. H. Wong and J. Gibbons, "A Process Semantics for BPMN," in *ICFEM*, ser. LNCS, vol. 5256. Springer, 2008, pp. 355–374.

[34] E. Börger and B. Thalheim, "Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach," in *ABZ*, ser. Lecture Notes in Computer Science, E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, Eds., vol. 5238. Springer, 2008, pp. 24–38.

[35] A. Großkopf, "xBPMN - formal control flow specification of a BPMN based process execution language," Master's thesis, Hasso-Plattner-Institute, Potdam, Germany, 2007.