# Dynamic Skipping and Blocking, Dead Path Elimination for Cyclic Workflows, and a Local Semantics for Inclusive Gateways

Dirk Fahland[a], Hagen Völzer[b]

[a] *Eindhoven University of Technology, The Netherlands*
[b] *IBM Research – Zurich, Switzerland*

**Abstract**

We propose and study dynamic versions of the classical flexibility constructs 'skip' and 'block' for workflows and motivate and define a formal semantics for them. We show that our semantics is a generalization of dead-path-elimination and solves the open problem to define dead-path-elimination for cyclic workflows. This in turn gives rise to a simple and fully local semantics for inclusive gateways. Finally, we show how our new constructs can be enacted on existing process engines by a workflow transformation that stores control-flow information in process variables.

*Keywords:* Process modeling, process modeling languages, BPMN, process flexibility by design, dead path elimination, inclusive gateway semantics, skipping and blocking

## 1. Introduction

One of the challenges in process management is striking a balance between the clarity of a process model on one hand and its ability to support a large variety of process flows on the other hand (also called process flexibility). A model can express flexibility in different ways: by design, by deviation, by underspecification, and by change [1, 2]. Flexibility *by design* faces the above challenge directly: including many different possible paths in a model tends to increase its complexity.

A public service process from a Dutch municipality (described in [3] in detail) illustrates the problem. The process model (Fig. 1) has 80 process steps (white)

*Email addresses:* d.fahland@tue.nl (Dirk Fahland), hvo@zurich.ibm.com (Hagen Völzer)

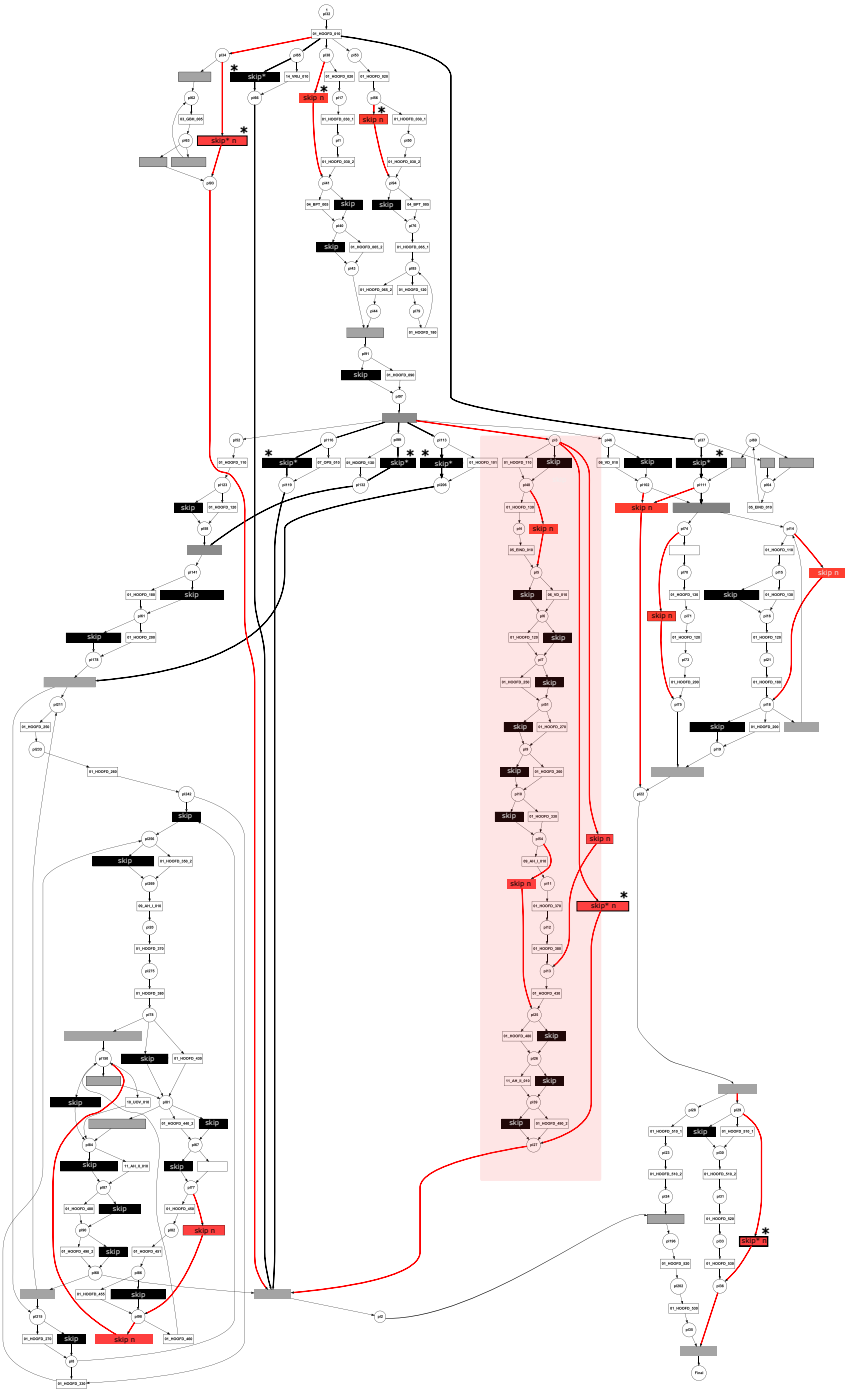

Figure 1: Model of a flexible process: 38 process steps can be skipped individually (*skip*, black), 13 segments of multiple process steps have to be skipped together (*skip n*, red).

and 21 routing constructs (grey); 65% of the process steps are optional under some more several conditions. To capture this behavior, the model contains 52 explicit paths for skipping process steps: 38 process steps can be skipped individually (in Fig. 1 by black transitions labeled *skip*), and 13 segments of multiple process steps that have to be skipped together (in Fig. 1 by red transitions labeled *skip n*). Paths for skipping are not mutually exclusive but overlap as indicated for example by the highlighted segment in the middle of Fig. 1. Further, 9 of the skip paths allow to *block* an entire branch of control between two parallel gateways (skip transitions tagged with $*$).

The complex routing logic is relevant: in 481 cases over a period of over 1 year, each case required steps to be skipped, amounting to 3087 skipped steps for 11846 executed steps (26%). Other municipalities running the same process face similar dynamics, their share of optional process steps ranges between 50% and 63% to allow for 516/5363 to 1574/7684 skips in 1 year [3]. Creating, understanding, and maintaining models for such flexible processes with explicit design constructs is tedious.

The classical concepts to *skip* tasks and to *block* a path can be used to express flexibility by design. They have been used predominantly for *static* flexibility, i.e., to remove tasks or paths from the model *before* deployment through process model configuration [4]. However, in many processes, skipping and blocking dynamically depend on user input or dynamically computed data, e.g., Fig. 1. Such *dynamic* skipping and blocking can be expressed to some extent in WS-BPEL by setting the status of a *link* through a combination of transition conditions, join conditions, suppressing join failure, and dead path elimination [5] - however, mapping classical skip and block to their implementation in BPEL is not straightforward. Furthermore, the link status can carry only the values 'true' or 'false', but this binary value can have many different causes, which merges the concepts of flexibility through data conditions, flexibility through alternate joining of paths, join failure, elimination of paths that were deliberately not taken in the process logic, and elimination of paths that are blocked through activity failure. This prevents the free combination of these concepts and can create unintended side effects [6, 7]. Moreover, BPEL restricts these constructs to acyclic control flow graphs.

In this paper, we study dynamic skipping and blocking in the context of BPMN with the following contributions:

1. We define dynamic skipping and blocking for BPMN-like languages, each with a dedicated local semantics, such that they can be used independently

from each other or freely combined. We define the semantics for general control-flow graphs, including cyclic graphs, and compare the semantics of static and dynamic skipping and blocking.

2. Our proposal for dynamic blocking includes a generalization of the Dead-Path-Elimination (DPE) concept [5, 7] to general control flow, which so far was limited to acyclic control flow.

3. We point out that dynamic blocking is closely related with the semantics of inclusive gateways (aka synchronizing merge pattern, OR-join semantics). Our generalization of DPE to cyclic flow graphs gives rise to a purely local semantics for inclusive join behavior. As a result, our semantics does not entail semantic anomalies such as 'vicious cycles' (see, e.g. [8]). In comparison with existing semantics, it can be enacted faster, i.e., in constant time, it is compositional for more models and therefore easier to understand and use, and it permits more refactoring operations for process models.

4. We show that dynamic skipping and blocking can be enacted on existing process execution engines without modifying them: We provide a local syntactic transformation of workflow graphs with skipping and blocking to regular workflow graphs *with process variables* but without inclusive gateways.

This article extends an earlier conference article [9] with a comparison between static and dynamic skipping and blocking, an extended discussion of the new semantics for inclusive gateways, and the translation of dynamic skipping and blocking to workflow graphs with variables. Further, we include here the complete formal semantics of the proposed concepts and the proof that dynamic blocking does not interfere with normal process execution.

We start by recalling some basic notations on workflow graphs in Sect. 2. We then discuss concepts for dynamic skipping and blocking based on literature for static skipping and blocking in Sect. 3. In Sect. 4, we generalize dead path elimination to all sound workflow graphs. The resulting local semantics for inclusive joins is discussed in Sect. 5. Our translation of dynamic skipping and blocking to regular workflow graphs with variables is given in Sect. 6. We conclude in Sect. 7 where we also compare conceptual and subtle differences of our approach to the literature.

## 2. Background

In this section, we briefly recall workflow graphs (with guard expressions). Further, the notions of dynamic skipping and blocking introduced in Sect. 3 are partially based on the view that a process is a synchronization of state machines which we explain here as well.

### *2.1. Workflow Graphs*

We work with *workflow graphs* [10] with guarding expressions over data as the core constructs of business process models. Other modeling constructs, e.g., BPMN events, can be added orthogonally and are out of scope of this paper. We use the following definitions.

A *two-terminal graph* is a directed graph (multiple edges between a pair of nodes are allowed) such that (i) there is a unique source and a unique sink and (ii) every node is on a path from the source to the sink. A *workflow graph* (WFG) is a two-terminal graph with four types of nodes: *task*, *exclusive gateway*, *parallel gateway*, and *dummy* such that (i) the source and the sink are exactly the dummy nodes, and the source has a unique outgoing edge, called the *source edge*, and the sink has a unique incoming edge, called the *sink edge*, and (ii) each task has at most one input and at most one output edge.

A gateway that has more than one outgoing edge is also called a *split* and a gateway that has more than one incoming edge is also called a *join*. Further, let $X$ be a set of *variables* for data objects and $\mathrm{Expr}(X)$ a set of Boolean expressions over $X$. Each outgoing edge $e$ of an exclusive split has a *guarding expression* $\gamma(e) \in \mathrm{Expr}(X)$. A *state*, also called a *marking* of a WFG assigns to each edge of the WFG, a number of indistinguishable black *tokens*. The marking with a single token on the source edge and no token elsewhere is called the *initial marking*. The marking with a single token on the sink edge and no token elsewhere is called the *final marking*.

We use BPMN semantics and visualization [11] for WFGs. Figure 2 shows an example. Starting with the initial state, i.e., a single token at the edge that leads to the task *Receive order*, the task is enabled and its execution moves the token to its output edge. The subsequent parallel split removes the token from its input edge and puts one token on *each* output edge. The exclusive split gateway moves the incoming token to the output edge where the guard expression *Credit Card* = *y* or *Credit Card* = *n* evaluates to true (depending on the current value of variable *Credit Card*). The exclusive join gateway forwards the incoming token to its output edge; the parallel join gateway waits until each input edge has a token, then

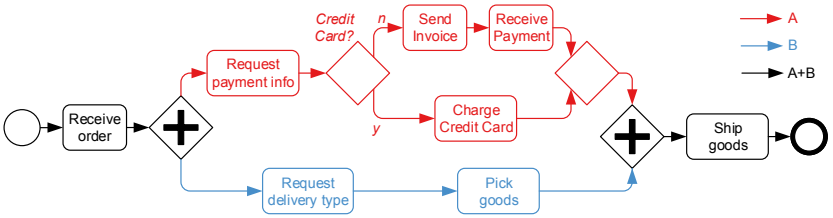


Figure 2: A workflow graph with two S-components A (red and black) and B (blue and black)

removes one token from each input edge and puts a token on its output edge. The subsequent discussion assumes a guarding expression has always a well-defined value, but makes no assumptions as to when and how values of variables and guarding expressions change. The reader may imagine that data is adjoined to process as in BPMN or BPEL, i.e., data variables are global to the process and changes during execution of any tasks.

Note that a workflow graph is equivalent to a corresponding isomorphic free-choice Petri net [10]. Therefore, we can directly apply the theory of free-choice Petri nets [12] to workflow graphs.

Recall that a workflow graph $G$ is *sound* iff starting with the initial marking, (i) no marking can be reached where an edge of $G$ is marked with two tokens (aka an *unsafe* marking or *lack of synchronization*), (ii) the final marking can be reached from each reachable marking. Soundness for workflow graphs has several equivalent characterizations, see e.g. [13].

### *2.2. State machine decomposition*

A natural way to understand a workflow graph is to view it as a synchronization of *state machines*, or *threads*, also called *S-components* or *P-components* in Petri net theory. An S-component represents purely sequential behavior, e.g., the lifecycle of a business object such as a purchase order or a payment document. Multiple objects may be completely synchronized, i.e., have exactly the same life cycle represented by the same S-component. Otherwise, different S-components are synchronized through parallel gateways. For example, Fig. 2 shows a decomposition of a simple workflow graph into two S-components $A$ and $B$, which are synchronized in the black part. A more complex example is shown in Fig. 8.

More formally, a subgraph $G'$ of a workflow graph $G$ is said to be *sequential* if for every parallel gateway, at most one incoming and at most one outgoing edge belongs to $G'$. $G'$ is an *$S$-component* of $G$ if (i) $G'$ contains the source and the sink

of $G$ and in $G'$, every node is on a path from the source to the sink, and (ii) every exclusive gateway belonging to $G'$ has all its incoming and all its outgoing edges in $G'$. A set of S-components of $G$ is called a *state machine decomposition* of $G$ if the component-wise union of all S-components yields $G$. Note that every sound workflow graph has a state machine decomposition, which can be computed in cubic time [14][1] An important property of S-components is that each S-component is always marked with exactly one token.

## 3. Dynamic skipping and blocking

In this section, we present dynamic versions of task skipping and path blocking as conservative extensions of workflow graphs together with modeling examples. These constructs are inspired by their well-known static counterparts. For example, the approach by Gottschalk et al. [4] allows to make a model configurable by adding visual annotations for 'execute', 'hide', and 'block' to tasks and to inputs and outputs of control-flow nodes. 'Execute' leaves the task as is, 'hide' removes the task, whereas 'block' removes the task and the entire flow after it until the next flow merge.

Next, we first introduce the syntax of our model which we named *multipolar workflow graph* based on the bipolar synchronization schemes of [15] which introduced true/false tokens for graphs without choices. Then we first discuss its semantics and a number of design decisions informally; their formal semantics are provided in Sect. 4.3.

### *3.1. Multipolar workflow graphs with dynamic skip and block guards*

Both constructs that we define, i.e., the *dynamic skip* and the *dynamic block*, allow the control flow of a process to skip one or more activities on its path depending on the evaluation of a dynamic data condition. More precisely, a *data expression* is a Boolean-valued expression that may contain variables that represent data objects of the business process, e.g., $amount > 1000$, $isGoldCustomer(client)$. A *guard* is a data expression associated with a point of the control flow of the workflow graph, which we model as a separate node with a single incoming and a single outgoing edge, depicted as a mini-diamond, cf. the grey and white

[1] Any workflow graph without inclusive gateways can be mapped in linear time to a free-choice workflow net [10], the S-components can be calculated in cubic time [14], and each S-component (which is a free-choice workflow net again), can be mapped to a workflow graph in linear time [10].

mini-diamonds in Fig. 3 and Fig. 4. This is similar to the data conditions (white mini-diamonds) in BPMN [11] and leads to the following syntax.

A *multipolar workflow graph* (multipolar WFG) $G$ is a workflow graph as in Sect. 2.1 but with two additional node types *skip guard* (visualized as grey mini-diamond) and *block guard* (visualized as white mini-diamond). Each guard $v$ has exactly one incoming and one outgoing edge and is annotated with a boolean expression $\gamma(v) \in \mathrm{Expr}(X)$ over some data variables $X$. Figures 4 and 6 show examples of multipolar WFGs.

Intuitively, the semantics of multipolar WFGs extends the semantics of regular WFGs as follows. Edges are marked with tokens, however, each token has a *color*: black, grey, or white. The enabling of a node depends on the presence of sufficiently many tokens, not on their color. A task enabled by a black token is *executed*, a task enabled by a grey or white token is *skipped*, and a token of the same color is produced on the output edge. An enabled skip or block guard can change the *color* of the token depending on the guarding expression. Gateways generally route tokens as in regular WFG but they can change token colors and routing may ignore guard conditions, as we discuss below. As the semantics of multipolar WFG follows that of regular WFG, *soundness* for multipolar WFG is defined in the same way as for regular WFG; detailed definitions are provided in Sect. 4.3.

Each sound multipolar WFG $G$ can be decomposed into S-components; for this decomposition the skip and block guards can be treated as task nodes. We will exploit this view on multipolar WFG in our exposition of the semantics of skip and block guards next.

### *3.2. Dynamic skip*

A token flowing through a guard triggers the evaluation of the guard. Only if the guard evaluates to true, the subsequent activities will be executed and they will be *skipped* otherwise. Seen from the perspective of an S-component, the scope of a *skip guard* $g$ is from $g$ until the next guard (of any type). Hence the guard can be considered as a precondition for the activities in its scope in the S-component.

A simple application of a skip guard (grey mini-diamond) in an S-component is shown in Fig. 3(a), where two activities are skipped when the data condition $amount > 1000$ evaluates to false. This is of course equivalent to the graph fragment shown in Fig. 3(b), however Fig. 3(a) represents the same behavior more compactly while still indicating the two cases of the flow graphically. This allows a modeler to represent more complex behavior more succinctly. The first guard can 'switch off' the corresponding S-component, the second guard switches it back on

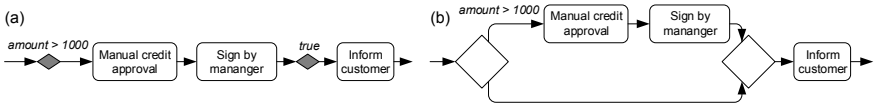


Figure 3: (a) A simple example for skip guards. (b) Its corresponding explicit representation.

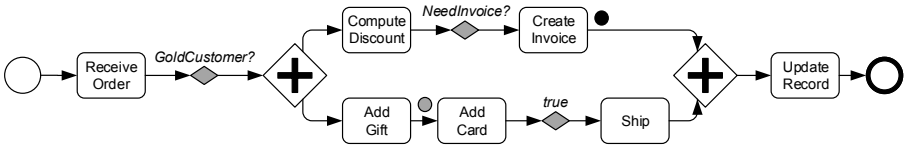


Figure 4: A multipolar workflow graph with three skip guards, a grey and a black token.

in order to make sure that the third activity 'Inform customer' is executed in any case.

The interplay of guards and multiple S-components is shown in Fig. 4, which models a part of an order fulfillment process. There are two S-components which split up in the center – the upper is concerned with the invoice whereas the lower is concerned with the physical items to be shipped. Both of these two components behave differently in case the item is shipped to a gold customer. Hence the guard is placed already before the parallel split and belongs to both S-components. If the customer is not a gold customer, then the three activities 'Compute discount', 'Add gift' and 'Add card' in its scope are skipped. The *NeedInvoice?* skip guard belongs to the upper S-component and its scope ranges only until the parallel join with the lower S-component. At this point the decisions whether to skip or execute activities of all joining S-components has to be reconciled: skipping continues only if all S-components have been "switched off", i.e., all respective prior skip guards evaluated to false.[2]

As we said above, we formalize the effect of guards using token colors. The normal token color is *black*. The workflow starts with a single black token on the source. A black token flowing through a skip guard remains black if the guard evaluates to true and turns into a *grey* token otherwise. Similarly, a grey token flowing through a skip guard turns black if the guard evaluates to true and remains grey otherwise. A black token flowing through an activity executes the

[2]Note that scopes of guards are not defined hierarchically (allowing for nesting), but as paths between nodes along the graph structure of a WFG.

activity, a grey token skips the activity. An activity does not change the color of a token flowing through it. Likewise, the token color does not change through split gateways and exclusive joins. A parallel join emits a black token iff at least one of its inputs is black, otherwise it emits a grey token. Hence, a skip guard evaluating to false switches off the S-component until it is switched on again by another skip guard or by synchronizing with another S-component that is switched on.[3] Fig. 4 shows a reachable marking of the corresponding graph with one black and one grey token.

### *3.3. Dynamic block*

A skip guard can switch on or off an S-component repeatedly. In contrast, a *block guard* blocks an S-component persistently if the guard evaluates to false, i.e., after a blocking, the S-component cannot be switched on again by another guard. Thus, the *scope* of a block guard $g$ to skip any activity in an S-component is always from $g$ until the S-component synchronizes at a parallel join with another S-component that is still active. This behavior is known from the synchronizing split/merge control-flow pattern, which is also known as the *inclusive split and join*, cf. Fig. 5(a). (We will introduce the inclusive split and join more formally below in Sect. 5.)

In Fig. 5(a), each of the branches, i.e., S-components, is either persistently switched on or off after execution of the inclusive split. The active and inactive branches are finally synchronized through the inclusive join gateway. Fig. 5(b) shows the same behavior in an alternative BPMN notation using white mini-diamonds to represent the data-based blocking of the corresponding branch. Hence we use the BPMN white mini-diamond to represent a block guard as shown in Fig. 5(c).

The block guard, however, generalizes this behavior of the inclusive join gateway as illustrated by Fig. 6. Similarly to Fig. 5, a subset of the branches can be activated. However, the upper branch is always taken, which does not need any guard. The lower branch is also always taken, and hence the activity 'Add standard travel insurance' is always executed - however this branch, i.e., S-component, can be switched off subsequently in various ways. Either by the block guard whenever an optional emergency insurance is not selected, which means that all remaining activities before the parallel join will be skipped. If an

[3] Note that this interpretation is agnostic to the particular S-component decomposition chosen for a WFG as at parallel splits token colors remain the same in all S-components, and at parallel joins, token colors are reconciled.

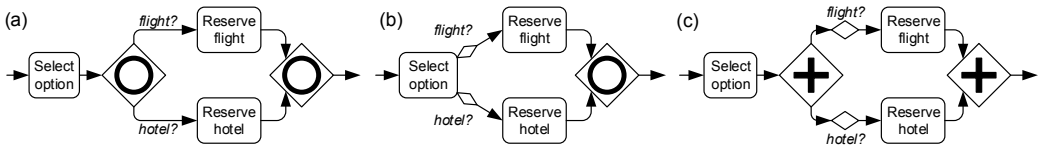


Figure 5: (a) Inclusive split and -join, (b) Inclusive split with BPMN mini-diamond, (c) Proposed notation with block guards

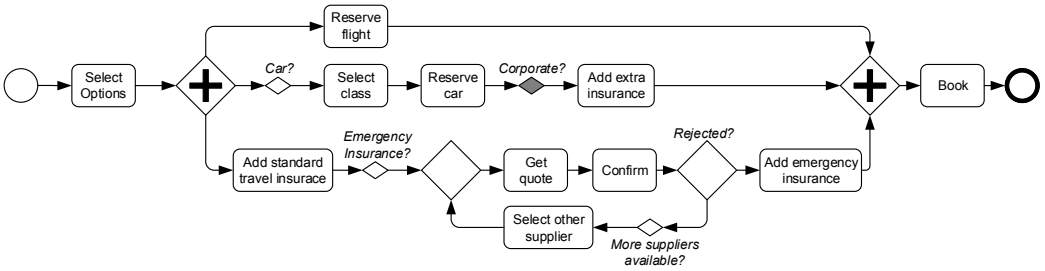


Figure 6: A more complex example for a block guard

optional emergency insurance is selected, the S-component may be switched off later by the second block guard in case no suitable supplier can be found.

The middle branch is persistently switched off if no car is selected and cannot be switched on by the skip guard for corporate customers. Since all those activities are skipped whenever a preceding block guard evaluates to false, the guard can again be viewed as a precondition to those activities; this precondition holds until the parallel join (as for skip guards).

To formalize block guards, we introduce *white* tokens. A black or grey token entering a block guard $g$ turns white when $g$ evaluates to false, otherwise it retains its color. A white token flowing through a block guard, skip guard, or an exclusive gateway always retains its color, hence the guard does not need to be evaluated in that case. If a white token flows through an activity, the activity is not executed and the color of the token does not change. Likewise, a white token entering a parallel split produces only white tokens on the outgoing edges of the parallel split. At the parallel join, decisions (colors) of several S-components are reconciled: A parallel join emits a black token iff at least one of its inputs is black, it emits a grey token iff none of its inputs is black but at least one is grey, and it emits a white token iff all its inputs are white.

Note that, so far, the difference between a skip and a block guard is merely that

blocking is more permanent than skipping, i.e., a grey token can easily be turned into a black one whereas a white token cannot. However, we will introduce another crucial difference in Sect. 4.2.

## 4. Dead path elimination for cyclic workflows

Section 3 introduced guards, and the behavior of grey and white tokens at guards and parallel gateways. In this section, we define the routing of grey and white tokens in exclusive splits, we present dead-path elimination for cyclic workflow graphs, and discuss differences between static and dynamic skipping and blocking.

### *4.1. Grey tokens in exclusive splits*

How should we route a grey or white token in an exclusive split? Both token colors represent inactive S-components – recall that all incoming and outgoing edges of an exclusive gateway belong to the same S-components. However, while a white token cannot execute any activity on any of the outgoing branches of the exclusive split since it cannot become black, a grey token can become black, and it will in general execute different activities on different outgoing branches just as a black token can. We therefore treat a grey token as a black token in this situation. That is, a grey token also triggers the evaluation of the boolean expressions on the outgoing edges of the split and the unique outgoing edge for which the expression evaluates to true gets the grey token.

In other words, the routing of grey tokens is firmly controlled by the modeler. This requires care since the data variables that the exclusive split refers to must be in a state the modeler expects them to be in at that point of control-flow, in particular the data variable must be at least initialized. This is not trivial since some tasks (this is where the data is set, e.g. in BPMN) have been skipped by the grey token. Consider for example the exclusive split in Fig. 6, labeled with 'Rejected?'. This decision refers to a Boolean variable *rejected*. Let's assume this variable is set in the preceding task 'Confirm'. However, this task is skipped if the preceding guard 'Emergency insurance selected' evaluates to false. Therefore the decision value in the exclusive split is not well defined if 'Confirm' is not executed (and that's why we cannot use a skip guard but must use a block guard–as we will see later–in the lower branch of Fig. 6). Also race conditions on the evaluation of a guard may arise, e.g., if a task can update a variable occurring in a guard on a

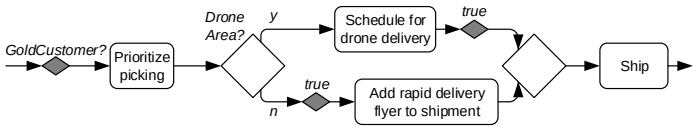


Figure 7: Flexible skipping in different branches of an S-component

parallel branch[4]. This explicit routing of grey tokens in an exclusive split must be carefully designed by the modeler if the split represents the exit condition of a loop to make sure that the process eventually exits the loop to be able to terminate.

This extra care will not be necessary when using white tokens as we show below. Yet, grey tokens and skip guards provide greater flexibility than block guards. In particular, the explicit control of routing grey tokens in exclusive splits can be leveraged to model different skipping behavior in different branches of the S-component. For example, Fig. 7 models that all *GoldCustomers* receive an immediate prioritization of picking their goods; the upper alternative branch is taken for members living in an area where delivery via drone is offered, but only *GoldCustomers* get their picked goods scheduled for delivery via drone (for non-*GoldCustomers* this activity is still skipped); for any customer (Gold or regular) living in a different area the lower branch is taken and a flyer about alternative rapid delivery options is added to their shipment.

### *4.2. White tokens in exclusive splits and dead path elimination*

In contrast to the explicit routing of grey tokens, we can route a white token implicitly at an exclusive split. Intuitively, the routing of a white token does not matter, because the S-component is dead anyway – neither of the branches can be executed because a white token remains white, hence all activities on each of the branches are skipped. In particular, we do not need to evaluate the data condition at an exclusive split if a white token arrives at it, which is important, because it may not be well defined – as we have discussed this above for Fig. 6.

Still we have to make sure that a white token arrives at the next synchronization point, i.e., 'eliminates the dead path', even or in particular in the presence of cycles as in Fig. 6, where we have to make sure that the white token is not following a cycle infinitely often and prevents termination of the process. We can do that by

[4]However, these race conditions are not specific to skip guards, but also arise for block guards and guards at exclusive splits in regular WFGs.

implicitly 'flushing out' the white tokens, i.e., route them automatically towards the sink, thereby providing a form of dead path elimination for cyclic workflow graphs. We operationalize such a behavior by help of an *exit allocation*.

**Definition 1.** Call any outgoing edge of an exclusive split a *choice edge* of the workflow graph. An *exit allocation* is a mapping $\phi$ that assigns to each exclusive split $v$ one of its choice edges $\phi(v)$, called the *exit edge*, such that, for each edge $e$ of the workflow graph there exists a path from $e$ to the sink such that each choice edge on the path is an exit edge.

The intuition behind Def. 1 is that white tokens will get flushed out of the graph by routing them via exit edges. The exit edges are statically fixed and can be considered as 'providing a compass' to the sink. To justify this definition, we first observe:

**Theorem 1.** *An exit allocation exists for each workflow graph and it can be computed in time* $O(|E| + |V| \cdot \log |V|)$.

*Proof.* We can use Dijkstras algorithm to compute, for each node the shortest path to the sink. We allocate a choice edge of an exclusive split $v$ as the exit edge if it starts the shortest path from $v$ to the sink. It follows that, for each edge $e$, every choice edge on the shortest path from $e$ to the sink is an exit edge. □

Note that an exit allocation is an *allocation pointing to* the sink in the sense of [12, Def.6.4] and its existence also follows from [12, Lemma 6.5 (1)].

Fig. 8 shows an example of an exit allocation where the exit edges are shown in bold. A white token produced by the block guard $g_1$ will be routed at the exclusive splits $d_1$ and $d_2$ towards the parallel join $j_2$, where it is joined with either a black token or a white token produced by block guard $g_2$. A white token arriving at $d_3$ is routed directly to the sink. Note that an exit allocation for the graph in Fig. 8 is

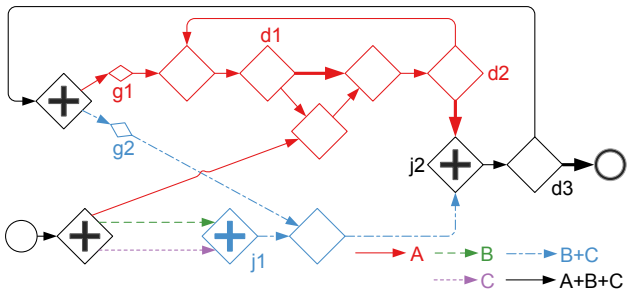


Figure 8: A workflow graph decomposed into three S-components $A$ (red and black), $B$ (orange, green, and black) and $C$ (blue, green, and black). An exit allocation is shown in bold.

not unique. We could have chosen also the other choice edge of $d_1$ (but not for $d_2$ or $d_3$).

An exit allocation defines the routing of white and only white tokens at an exclusive split and it does not need to be defined by the modeler – it is implicitly there, i.e., the compiler or execution engine provides the dead path elimination automatically. However, since an exit allocation is not unique and the modeler does not choose it, how does the modeler understand and control the behavior of the workflow graph? To this end, we prove that the particular choice of the exit allocation does not matter, i.e., all exit allocations and even more general, all *fair* routings of white tokens produce essentially the same behavior. Therefore any exit allocation operationalizes the same abstract behavior of dead path elimination.

Before we formally prove this, we formalize our extended model of workflow graphs and their semantics.

### *4.3. Semantics of multipolar workflow graphs*

In this section, we present the formal semantics for multipolar workflow graph (as defined in Sect. 3.1); see Appendix A for a rigorous formalization of our model.

Let $G$ be multipolar workflow graph. We write ${}^\circ v$ and $v^\circ$ for the set of input and output edges of node $v$ of $G$, respectively. A *marking* $m$ of $G$ assigns to each edge a nonnegative number of *tokens*, where each token has a *color* $c \in C = \{\text{black}, \text{grey}, \text{white}\}$. We write $m[e, c]$ for the number of tokens of color $c \in C$ on $e$ and $m[e] = \sum_{c \in C} m[e, c]$ for the number of all tokens of any color in $m$ on $e$; $m$ is *safe* iff $m[e] \leq 1$ for each edge $e$. The marking with exactly one black token on the source edge and no token elsewhere is called the *initial marking* of $G$. A marking that has a single token of any color on the sink edge and no token elsewhere is called a *final marking* of $G$.

Nodes are *enabled* as in regular workflow graphs: a task, exclusive gateway, or guard $v$ needs a token (of any color) on some edge $e^- \in {}^\circ v$; a parallel gateway $v$ needs a token on each edge $e^- \in {}^\circ v$. Each node $v$ defines several *transitions* $t$ that distinguish the possible colors of input tokens consumed and output tokens produced in a *step* $m \xrightarrow{t} m'$ as illustrated in Fig. 9; see Appendix A for the formalization.

A step $m \xrightarrow{t} m'$ of $G$ is called an *elimination* step, denoted $m \overset{t}{\dashrightarrow} m'$ if all tokens consumed (and produced) by $t$ are white, otherwise it is a *normal* step. If $m'$ can be reached from $m$ through zero or more elimination steps, we write $m \overset{*}{\dashrightarrow} m'$. We write $m \overset{\max}{\dashrightarrow} m^*$ if $m \overset{*}{\dashrightarrow} m^*$ and $m^*$ does not enable any further elimination step. Given an exit allocation $\phi$ for $G$, we say that an elimination step $m \overset{t}{\dashrightarrow} m'$ *complies*

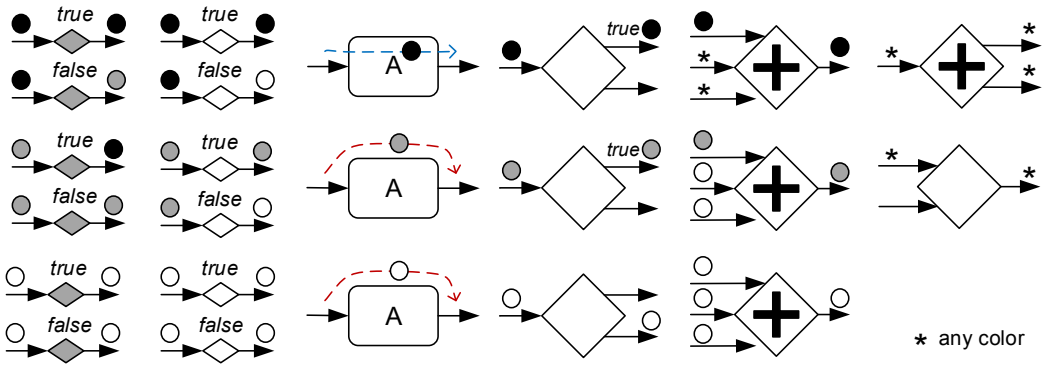


Figure 9: Possible transitions of nodes in a multipolar workflow graph.

*with* $\phi$ if the white token is produced on $e^+ = \phi(v) \in v^\circ$ whenever $t$ is a step of an exclusive split $v$.

A *trace* of $G$ is a sequence $\sigma = m_0, t_1, m_1, \ldots$ of markings and transitions s.t. $m_0$ is the initial marking of $G$ and $m_i \xrightarrow{t_{i+1}} m_{i+1}$ for each $i \geq 0$; $\sigma$ is *maximal* if it is either infinite or ends in a marking $m_n$ such that no transition is enabled in $m_n$. A trace $\sigma$ is *fair* with respect to a choice edge $e$ of an exclusive split $v$ of $G$ if the following holds: If $v$ is executed infinitely often in $\sigma$, then $e$ is marked infinitely often in $\sigma$. A trace of $G$ is *fair* if it is fair with respect to all choice edges of $G$.

A node $v$ (edge $e$) is *dead* in $m$ if no marking reachable from $m$ enables $v$ (marks $e$). A node or edge $x$ is *live* in $m$ if $x$ is not dead in each marking reachable from $m$. A *local deadlock* is a marking in which a node $v$ other than the sink is dead and an edge $e \in {}^\circ v$ is marked. $G$ is *live* if no marking reachable from the initial marking $m_0$ is a local deadlock. $G$ is *safe* if each marking reachable from $m_0$ is safe. $G$ is *sound* if $G$ is safe and live. Equivalently, $G$ is sound iff $G$ is safe, the sink edge is live in $m_0$ and only a final marking is a reachable marking that marks the sink edge. Soundness guarantees that each maximal and fair trace of $G$ terminates in a final marking of $G$.

### *4.4. Justification of exit allocations*

In this section, we justify the use of exit allocations as implementation of dead path elimination. Let $G$ be henceforth a sound multipolar WFG. We first observe that an exit allocation implements fair behavior:

**Proposition 1.** *Every sequence of elimination steps that complies with an exit allocation* $\phi$ *is finite.*

*Proof.* The proof is indirect. Consider an infinite elimination sequence $\sigma$ that complies with $\phi$, i.e., an infinite trace that involves only white tokens. It follows that some edge $e$ is marked infinitely often in $\sigma$. The edge $e$ has a path $\pi$ to the sink where each choice edge is an exit edge of $\phi$. We can consider an S-component $S$ that covers $\pi$, i.e., every edge of $\pi$ is an edge of $S$. It follows that transitions of $S$ occur infinitely often in $\sigma$. However, the token on $e$ must travel along $\pi$ in $\sigma$ because $\sigma$ complies with $\phi$; $\pi$ leads to the sink, which implies that at some point of $\sigma$, the sink is marked. However, soundness implies that when the sink is marked, no other transition of $G$ is enabled, which contradicts $\sigma$ being infinite. □

Note that Prop. 1 is also a consequence of [12, Lemma 6.5, (2)] after translating all concepts back to the corresponding free-choice workflow net.

In the following, we will prove that all exit allocations generate essentially the same behavior. We prove first that two finite maximal elimination sequences that comply with the same exit allocation end in the same marking. We have shown that compliance with an exit allocation guarantees the finiteness of a maximal elimination sequence. Note that this finiteness is also guaranteed by fairness of the maximal elimination sequence, as any infinite elimination sequence in a finite graph violates fairness.

**Lemma 1.** *Let $m_0$ be a reachable marking, $\phi$ be an exit allocation and $m_0 \overset{max}{\dashrightarrow} m_1$ and $m_0 \overset{max}{\dashrightarrow} m_2$ be two finite maximal elimination sequences that comply with $\phi$. Then we have $m_1 = m_2$.*

*Proof.* Every elimination step that is enabled in $m_0$ will be executed in a maximal elimination sequence because it cannot be disabled by another elimination step. Using this, it can then be shown by induction on the size of elimination sequences that each maximal elimination sequence contains the same steps, i.e., one maximal sequence is a permutation of the other. The marking equation for Petri nets (cf. [12]) implies that both sequences end in the same marking. □

As a result, we can consider two elimination sequences both starting in $m_0$ and ending in $m_1 = m_2$ to be equivalent, as neither executes any activities. An even stronger result holds: any two finite maximal elimination sequences starting in $m_0$ necessarily reach the same marking even if they do not comply with a particular exit allocation.

**Lemma 2.** *Let $m_0$ be a reachable marking, $m_0 \overset{max}{\dashrightarrow} m_1$ and $m_0 \overset{max}{\dashrightarrow} m_2$ be two finite maximal elimination sequences. Then we have $m_1 = m_2$.*

*Sketch.* In the state-machine decomposition of the sound WFG, any S-component contains exactly one token. During elimination steps, the single white token only travels edges of 'its' S-component until reaching a parallel join. Two different sequences reaching two different markings $m_1 \neq m_2$ would reach different parallel joins. But then one would cause either a local deadlock or an *improper termination*, i.e., a reachable marking that has a token on the sink edge and another token elsewhere. Both cases contradict soundness of the WFG. Appendix B contains the detailed proof. □

We can now prove that the behavior of the WFG does not depend on the choice of a particular exit allocation, and hence *the behavior of the WFG does not depend on the routing of white tokens*. In other words, routing white tokens in exclusive splits in a fair but otherwise nondeterministic way suffices to reach a unique marking after finite elimination steps.

We do not prove the result in full generality, i.e., for the most general behavioral equivalence possible, as the necessary technical overhead would not justify the additional insight within the scope of this paper. In a technically simplified form, we assume that the WFG is executed with *eager* elimination, i.e., after each normal step, we execute a sequence of maximal elimination steps before we execute the next normal step.

**Theorem 2.** *Let $\sigma$ and $\sigma'$ be two* eager *traces of a sound multipolar WFG, i.e., they are of the form $m_0 \xrightarrow{t_1} m_1 \overset{max}{\dashrightarrow} m_2 \xrightarrow{t_2} \ldots$ where $m_0$ is the initial marking of the WFG, and*

1. *$t_i, i > 0$ are normal steps such that for any two markings $m_i, m_j$ of $\sigma$ and $\sigma'$, we have $m_i = m_j$ implies $t_{i+1} = t_{j+1}$, i.e. the program behaves deterministically from a given marking, and*
2. *$m_i \overset{max}{\dashrightarrow} m_{i+1}$ are finite maximal elimination sequences.*

*Then, $\sigma$ and $\sigma'$ have the same sequence of normal steps and in particular the same sequence of executed activities.*

*Proof.* The theorem follows directly from Lemma 2. □

Note that the proof of Theorem 2 rests on the essential property of white tokens, i.e., that a white token always remains white. This property allows us to route them automatically. As any fair routing of white tokens is permissible, the compiler or execution engine can choose an exit allocation that optimizes some cost measure, e.g., the average number of elimination steps, to implement the fair routing.

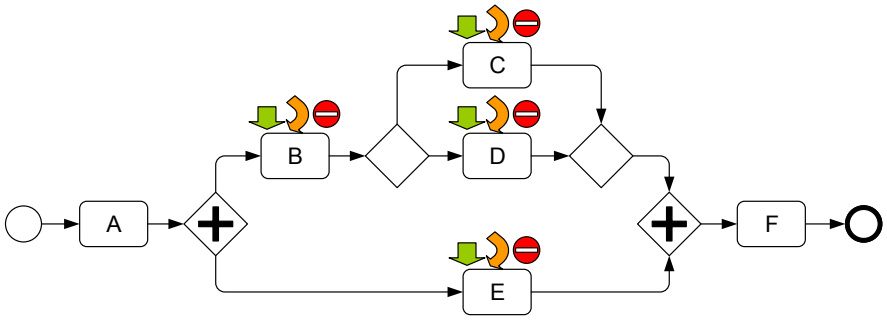


Figure 10: A workflow graph with possible configuration options for statically 'allowing', 'hiding', and 'blocking' tasks.

### *4.5. Static skipping and blocking vs. dynamic skipping and blocking*

We conclude this section with a comparison between the proposed dynamic skipping and blocking and the operators for statically 'executing', 'hiding', and 'blocking' tasks when configuring process models [4]. Figure 10 shows a workflow graph where each task *B*, *C*, *D*, and *E* can be configured by selecting one of the three operators. Selecting 'execute' (green straight arrow) keeps the task in the model; selecting 'hide' (yellow curved arrow) replaces the task with a sequence flow; selecting 'block' (red circular sign) removes the task from the model.

Here, 'executing' a task at configuration time (i.e., keeping it in the model) corresponds to executing the task when a black token arrives at runtime. 'Hiding' a task at configuration time allows the flow to continue but omits the task from execution. Both dynamic skipping of tasks (with grey tokens) and dynamic blocking of paths (with white tokens) achieve this behavior, but differ in their *scope*: static task hiding is local to the task, dynamic skipping ranges from one skip guard to the next skip guard or synchronization, whereas dynamic blocking ranges from one block guard until the next synchronization with a black token, which cannot be expressed through static hiding. Thus, dynamic *skipping* most closely corresponds to static hiding. For instance, in Fig. 10 we may choose to statically hide *B* and *C* and to execute *D*; dynamically, this can be realized with skip guards.

Statically 'blocking' a task at configuration time *removes* a path from the model. For example, blocking *C* would remove *C* and its two adjacent edges [4] which eliminates the choice between *C* and *D* and makes *D* mandatory. However, blocking *C* and *D* together removes all flows between the exclusive split and join, and the model has a deadlock at the parallel join which cannot receive a token from *B* anymore. Thus blocking requires global considerations to obtain sound

configurations [16]. Our proposed dynamic blocking cannot mimic removal of paths as by design we preserve all paths in the model to preserve soundness. Rather, entire paths can be dynamically excluded from execution by appropriate guard conditions at exclusive splits.

Altogether, the proposed dynamic skipping corresponds to the static task 'hiding' operator, while the static path 'blocking' operator corresponds to guards at exclusive splits. In contrast, the proposed dynamic blocking introduces a different form of variability: We have shown in Theorem 2 that a modeler can abstract from the behavior of white tokens and consider block guards as a means to disable control-flow along the subsequent path as if no token is present. Control-flow will consistently and predictably re-emerge at parallel joins with black tokens. Next, we show that this property allows us to give inclusive gateways a simple local semantics.

## 5. Dynamic blocking as inclusive gateway semantics

We have pointed out informally above that block guards with their semantics of white tokens and dead path elimination are closely related to *inclusive gateways*. In this section, we formalize this relationship and indeed we argue that block guards give rise to a new semantics of inclusive gateways. First we recall existing semantics of inclusive gateways and define ours. Then we argue, by leveraging prior analysis in the literature, that our semantics agrees with existing semantics, and, by virtue of being local, the new semantics has several advantages over existing semantics for inclusive gateways.

### *5.1. Inclusive gateways and their semantics*

*Syntax and BPMN semantics.* A *workflow graph with inclusive gateways* is a workflow graph as defined in Sect. 2.1 with an additional node type *inclusive gateway*, which is depicted as in Fig. 5(a). Each inclusive gateway has at least one incoming and at least one outgoing edge, and each outgoing edge $e$ has a guarding expression $\gamma(e) \in \mathrm{Expr}(X)$.

The semantics of the inclusive gateway has been subject of an extensive debate; cf. [17] for entry points into the discussion. The behavior of an inclusive gateway once it has started executing is mostly agreed upon: a token is consumed from each incoming edge that has a token and then *all* expressions on the outgoing edges are evaluated. On exactly those edges where the associated expression has evaluated to *true*, a token is produced.

The debated part of the inclusive gateway semantics is when exactly it is enabled. The following semantics is given by the BPMN 2.0 specification [11]: An inclusive gateway $j$ is *Q-enabled* in marking $m$ if

1. there is an incoming edge $e$ of $j$ such that $m$ marks $e$ and
2. for each edge $e'$ of the graph that is marked in $m$, we have: If there is a path $\pi$ (called *inhibiting path*) from $e'$ to some incoming edge $e$ of $j$ such that $e$ is unmarked in $m$ and $\pi$ does not visit $j$, then there is a path $\pi'$ (called *anti-inhibiting path*) from $e'$ to some incoming edge $e$ of $j$ such that $e$ is marked in $m$ and $\pi'$ does not visit $j$.

This semantics is motivated in detail in [17]. Note that this semantics is, in contrast to the semantics of the exclusive and parallel gateways, *non-local*, i.e., whether an inclusive gateway is Q-enabled depends in general not only on the tokens of its incoming edges but also on other tokens in the graph.

*A new semantics.* Our new proposal for the semantics of an inclusive gateway $j$ is to rewrite $j$ as a parallel gateway with block guards, as depicted in Fig. 5(c) with the associated semantics introduced in Sect. 4.3. Hence the following definition applies to *multipolar* workflow graphs. We say that $j$ is *L-enabled* in a (multipolar) marking $s$ of a multipolar workflow graph whenever its rewritten version, i.e., the parallel gateway is enabled in s. Note that a regular workflow graph with inclusive gateways becomes a multipolar workflow graph through the rewriting.

In the following, we show in what sense our semantics captures of what we believe is the essence of the inclusive gateway. Note that there is no universally agreed-upon reference semantics to which we can compare, nor are there any formally documented requirements for the semantics that we are aware of. Hence, the essence can only be captured through the examples that have been given in the literature for the behavior of the inclusive gateways in specific workflow graph examples. Such a discussion was given in detail in [17] for the definition of Q-enabledness. We follow a similar discussion here. In particular, we show in what sense the new semantics induced by L-enabledness agrees with the BPMN semantics induced by Q-enabledness for a large subset of workflow graphs, but not for all.

### *5.2. Comparison of the new semantics with the BPMN 2.0 semantics*

For comparison, we recall prior definitions and analyses of inclusive gateway semantics for workflow patterns, and acyclic and cyclic WFGs.

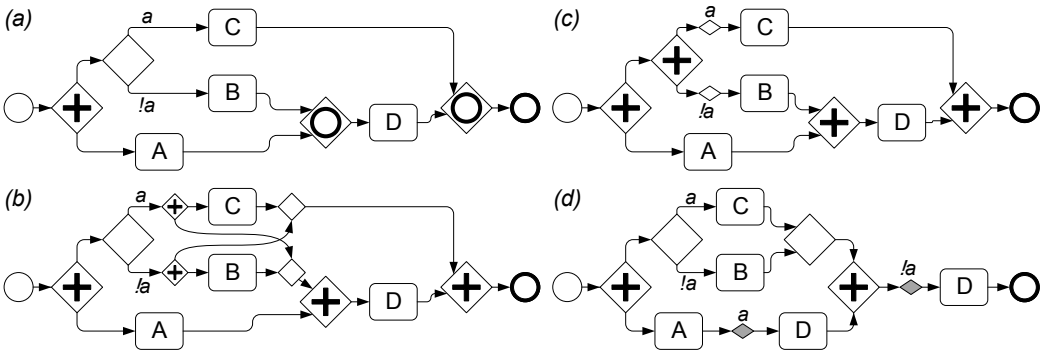


Figure 11: An acyclic workflow graph with inclusive joins (a) and its equivalent models (b-d)

*The synchronizing merge pattern.* The best documented modeling use case for inclusive gateways is the structured synchronizing merge pattern [18], cf. Fig. 5(a). If we understand Fig. 5(a) as a 'normal' workflow graph with inclusive gateways that behave according to BPMN semantics induced by Q-enabledness, then we can understand Fig. 5(c) as the multipolar workflow graph that corresponds to the normal workflow graph in Fig. 5(a). The formal correspondence of their behavior can be established by defining a correspondence between the multipolar markings (Fig. 5(c)) and 'normal' markings that have only black tokens (Fig. 5(a)). Note that we do not have to deal with grey tokens as our rewriting does not create any skip guards. Call a multipolar marking *bipolar* if it has no grey tokens.

The following correspondence was introduced in [17]: We say that a safe marking $m$ and a safe bipolar marking $x$ *correspond*, denoted $m \sim x$, if for all edges $e$, $m$ has a token on $e$ if and only if $x$ has a black token on $e$. Thus a marking may correspond to more than one bipolar marking. A normal transition sequence between normal markings corresponds with a bipolar transition sequence between bipolar markings if they are the same after deleting the *elimination steps* from the bipolar sequence, i.e., the steps that consume and produce only white tokens. It is now easy to prove that Fig. 5(c) simulates the BPMN semantics for Fig. 5(a) in a formal sense given by [17, Thm. 2].

Indeed [17, Thm. 2] proves the correspondence between the semantics induced by Dead Path Elimination (DPE) and Q-semantics for any sound acyclic workflow graphs[5].

[5] [17, Thm. 2] actually shows the correspondence between DPE-semantics and P-semantics. But the paper also shows that P-semantics and Q-semantics agree on sound acyclic graphs.

*Acyclic graphs.* The inclusive join semantics induced by the original definition of DPE on acyclic graphs, cf. [17], has a notable technical difference to our version here: In its original version, also exclusive splits send white tokens on the outgoing edges not taken. This allows a modeler to pair an exclusive split with an inclusive join without creating a deadlock. While this is possible in acyclic graphs, this creates problems when an exclusive split is part of a loop, e.g. in a structured loop, where we would get unsafe markings, viz. multiple white tokens on the exit edge of the loop.

However, as argued in Sect. 4, the role of a split gateway that emits white tokens on outgoing edges not taken is played in multipolar WFG by the inclusive split, i.e., a parallel split with block guards on the outgoing edges. Fig. 11(a) shows an example of an acyclic workflow graph, where a task $D$ is executed after concurrent tasks $A$ and $B$, but $B$ is executed only when the condition $a$ is false. Fig. 11(c) shows the the multipolar workflow graph that is obtained from Fig. 11(a) by rewriting the exclusive split by an inclusive split and then rewriting the inclusive gateways as suggested above. The graphs in Figs. 11(a) and (c) intuitively behave the same.

Indeed, if we apply this transformation to any sound acyclic workflow graph with inclusive gateways, i.e. first rewrite each exclusive split with an inclusive split and then rewrite each inclusive gateway with a parallel gateway with block guards on the outgoing edges, then the obtained graph simulates the original one in the sense of [17, Thm. 2] (proven there). In this sense, the semantics induced by Q-enabledness and L-enabledness agree on sound acyclic graphs.

Note that the behavior of the graph in Fig. 11(a) can be also generated with exclusive and parallel gateways only by help of additional auxiliary paths called *bridges*, cf. Fig. 11(b). The extent to which inclusive gateways can be replaced in this way has been studied in depth in [10]. Yet another way to model the behavior of the graph in Fig. 11(a) is by help of skip guards, cf. Fig. 11(d). Note that the resulting graph is even well-structured, i.e. the gateways are matching pairs of split and join. However, the conditions $a$ and $!a$ and the task $D$ are duplicated.

*Beyond acyclic graphs.* Acyclic graphs can be seen as the core subset of workflow graphs where most inclusive gateway semantics agree up to some minor technical differences and this was indeed proven for some further selected semantics in [17]. It was then proposed in [17] to extend this agreed-upon semantics from acyclic graphs to workflow graphs that can be composed from sound acyclic graphs and structured loops, i.e., while and repeat-until loops, and further beyond in what is called *separable* graphs in [17]. There, a workflow graph is called *separable* if it

can be composed by single-entry-single-exit nesting from sound acyclic graphs and arbitrary sequential workflow graphs. Such graphs are sound by construction. For separable graphs, it was shown in [17] that the semantics induced by Q-enabledness agrees with a semantics that uses DPE for the acyclic subgraphs and structured loop semantics for the structured loops (called $\mathcal{D}$*-enabledness* in [17]).

In a separable graph, first rewrite its acyclic fragments as describe above, i.e., rewrite each exclusive gateway in an acyclic single-entry single-exit fragment [19] as inclusive. In the resulting graph, an inclusive gateway is L-enabled whenever it is $\mathcal{D}$-enabled in the original graph, which follows from [17, Thm. 2] and soundness. Therefore, by [17, Thm. 3], the semantics induced by L-enabledness and by Q-enabledness also agree on separable graphs in some formal sense.

Outside this class of separable workflow graphs, almost no two semantics agree on what the behavior of an inclusive gateways should be nor is there any documented requirement for that behavior we are aware of. A few such graphs with sound behavior have been documented for technical discussion [17, 20]; the model of Fig. 13(a) is an example. We discuss this example in more depth below.

### *5.3. Advantages of the new semantics*

In this section, we show three advantages of the new semantics with respect to earlier proposed semantics: faster enactment, compositionality, and more refactoring operations.

*Enactment.* Existing semantics that do not restrict to a subset of workflow graphs are *non-local*, i.e., the enablement of an inclusive join there depends not only on the tokens of the incoming edges of the join but also on the presence of tokens on other edges of the graph. Two kinds of non-local semantics have been proposed: In the first, the enablement can depend on the entire state space of the workflow graph, e.g., [8]. Therefore, *enactment*, i.e., deciding whether a given inclusive join is enabled in a given marking, takes exponential time. In the second kind of non-local semantics, which includes the current BPMN semantics [11], the enablement depends on the existence of paths from other tokens in the graph to the inclusive join [17, 20]. It can be determined in linear [17] or quadratic time [20] respectively whether a particular inclusive join is enabled. The run time can be reduced in both cases by trading time for space, i.e., by creating data structures of quadratic size.

The local semantics presented in this paper has a small constant overhead for storing the additional token colors. It can be determined in constant time whether

a particular inclusive join is enabled under the assumption that the in-degree of nodes is bounded by a constant.

*Compositionality.* A process model can be better understood if it is composed out of simpler patterns or modules. However, this is only the case when the simple module can be understood in isolation, i.e., independent from the context it will be embedded in. Note that many textbooks explain the semantics of BPMN gateways, in particular the inclusive gateway by help of simple patterns.

One of the simplest and most popular notions of module is a single-entry-single-exit fragment, cf. e.g.[19]. Fig. 12 shows such a fragment (shaded) nested in another fragment. Considered in isolation, each fragment has the expected intuitive and sound behavior in the BPMN semantics, cf. [11, 17]. However, if we compose them in the way shown, the composed workflow graph has a deadlock (the marking shown in Fig. 12) in the BPMN semantics: $j_1$ is not Q-enabled because there is an inhibiting path from the token at $j_2$ (via the return edge of the loop) to the unmarked incoming edge of $j_1$ without a corresponding anti-inhibiting path; $j_2$ is not Q-enabled because there is an inhibiting path from the token at $j_1$ to the unmarked incoming edge of $j_2$ without a corresponding anti-inhibiting path. Thus the deadlock arises due to the non-local semantics of the inclusive join in BPMN where the synchronization behavior can depend on tokens outside the containing fragment.

In our local semantics, the behavior of any subgraph $G$ depends only on the tokens it exchanges at its border, i.e., the behavior of a composed graph is the composition of the behaviors of its constituent graphs. Even the non-local property of soundness is compositional for single-entry-single-exit-fragments in the local semantics, i.e., a composition is sound if and only if all its constituent fragments are sound [19]. The example in Fig. 12 shows that the non-local BPMN semantics [11, 17] is not compositional in general in this sense.

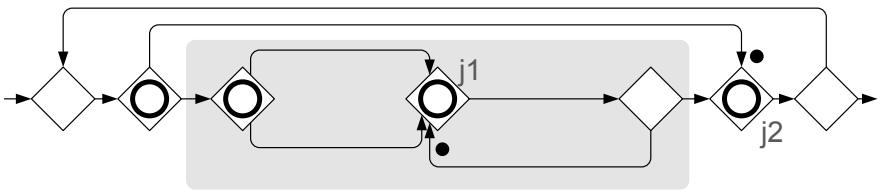


Figure 12: The current BPMN semantics for inclusive joins is not fully compositional w.r.t. single-entry-single-exit fragments.

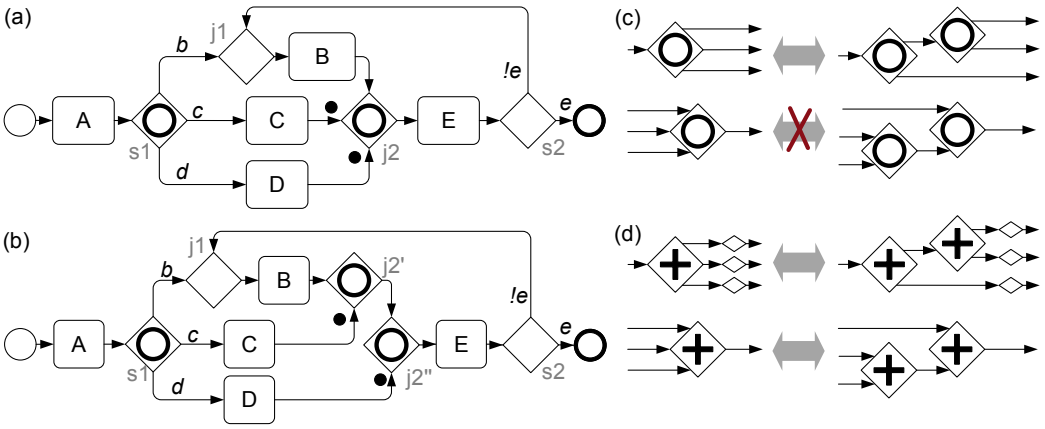


Figure 13: The BPMN inclusive join semantics is not robust under node splitting (a) and (b). The corresponding refactoring rule (c) is valid in our semantics (d).

*Refactoring.* Compositionality supports refactoring, i.e., maintenance of a process model. A fragment can be extracted from a process model, outsourced into a separate process and then called from different places without changing the behavior. There are other well-known refactoring operations that preserve the local semantics, viz. various structural transformation rules originally stated for Petri nets [21], but which apply to workflow graphs as well. For example, the two patterns shown in Fig. 13(d) are equivalent in any context in our local semantics as this is a well-known rule for parallel gateways. Note that the combination of colors in a parallel join as defined in Sect. 4.3 can be seen as a form of disjunction or maximum operation. In particular, it is commutative and associative.

However, the corresponding rule for the non-local semantics for inclusive gateways in BPMN is not valid (Fig. 13(c)). As a counterexample, we consider the model Fig. 13(b) that is obtained by applying the rule for inclusive joins from Fig. 13(c) on join $j_2$ of the model of Fig. 13(a). The workflow graph in Fig. 13(b) is not sound – the marking shown is a deadlock in the non-local semantics of BPMN, i.e., both $j_2'$ and $j_2''$ are not Q-enabled in the marking shown — whereas the workflow graph Fig. 13(a) is sound. Therefore, for that semantics, the rule in Fig. 13(c) is not a valid refactoring rule. Whether a deadlock occurs or not depends in our local semantics only on whether all incoming edges are live (a black, grey, or white token will eventually arrive) or whether one edge is dead. Transformation rules such as the one of Fig. 13(d) preserve whether an edge is live or dead, and hence can be applied in any situation.

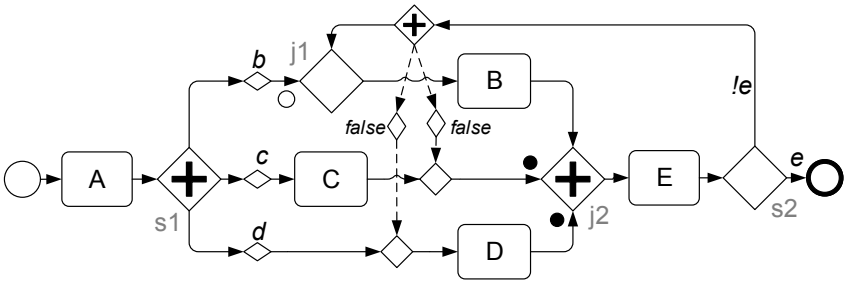


Figure 14: Representation of the unstructured loop with inclusive gateways of Fig. 13(a) in our semantics.

Figure 14 shows how the behavior of the model of Fig. 13(a) can be expressed in our semantics using bridges. The bridges producing white tokens (by the 'false' block guards) can be merged anywhere along the *c* and *d* branches.

*Conclusion.* We have shown that using our local semantics for inclusive gateways has several advantages over existing non-local semantics. Recall that we have argued in Sect. 5.2 that the new semantics agrees with the BPMN semantics for a large class of workflow graphs, viz. separable graphs. We are not aware of any documented real business process model outside that class. Nevertheless, we have verified that the technical examples outside that class given in [17] can be easily equivalently modeled with our local semantics by adding bridges, similar to the example of Figs. 13(a) and 14 given above. We do not know whether all non-separable graphs can be equivalently expressed with bridges and block guards. However, since there are no known requirements for the behavior of inclusive gateways in non-separable graphs and existing non-local semantics disagree outside separable graphs, we do not consider that to be a problem.

Note that in sound graphs, a token on the sink signals termination. In BPMN, the default edge of an inclusive split makes sure that at least one token remains in the graph if all other expressions on the outgoing edges of the split evaluate to false. In the local semantics, this is not necessary, because on each branch not taken, a white token is produced. If only white tokens remain in the graph, termination will be signaled through a white token on the sink.

## 6. Realizing dynamic skipping and blocking on existing engines

Processes with rich variability in flows already exist in practice, but standardized modeling languages such as BPMN and corresponding execution engines

require modeling all paths explicitly, as illustrated by the model in Fig. 1. We proposed two new syntactic constructs, skip guards (grey mini-diamonds) and block guards (white mini-diamonds) to overcome this problem. We provided executable semantics for skip and block guards by introducing token colors (black, grey, white) and generalizing the semantics for tasks and gateways. Further, we showed that extending workflow graph semantics with white tokens allows us to replace inclusive gateways with parallel gateways providing simpler local semantics that can be enacted in constant time. However, supporting the proposed semantics would require extending the BPMN standard [11] as well as existing software such as process execution engines.

In this section, we show that the semantics of skip and block guards can be realized on existing process execution engines without changing them — including all the mentioned advantages. More specifically, we provide a behavior-preserving local translation of multipolar WFGs as defined in Sect. 4.3 to regular WFGs *without* inclusive gateways where some tasks read from and write to a fixed number of *process variables*. In the following, we first outline the general translation approach and then provide the local patterns for our translation. Finally, we discuss implications of applying this translation in practice, and show that our translation also allows us to correctly execute processes with inclusive gateways on process engines that do not fully support the inclusive gateway.

### *6.1. Local skipping by data precondition*

The central idea is to translate the flow of colored tokens in a multipolar WFG to a flow of black tokens in a regular WFG where the color of a token is stored in a dedicated (global) process variable. Then skipping or executing a task can be made dependent on the value of the dedicated variable, which corresponds to the workflow data pattern 35 [22, 23]. A straight-forward realization of this pattern in BPMN[6] is shown in Fig. 15 where task $A$ gets executed if and only if guard $g$ holds true.

Our central idea for preserving token colors in variables comes from seeing each WFG as a set of synchronized state machines or S-components (see Sect. 2.2). As each S-component always holds exactly one token, it is sufficient to introduce for each S-component $S$ a variable $v_S$ that holds the color of the token in $S$. The

[6]Strictly speaking, variables and expressions over variables are outside the scope of the BPMN standard [11] and included in models via BPMN's extension mechanism. However, all existing execution engines that support BPMN also provide an extension for variables and expressions.

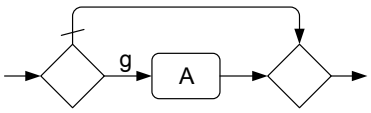


Figure 15: BPMN realization of data-precondition pattern (WDP35) [22, 23]

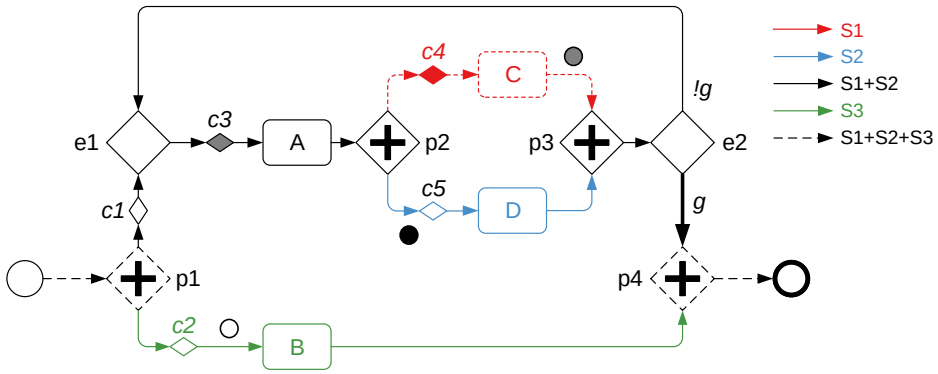


Figure 16: Multipolar WFG with decomposition into S-components.

obligation of our translation then is to ensure consistent updating of the value of $v_S$, in particular when two S-components synchronize.

For example, the multipolar WFG of Fig. 16 can be decomposed into the three S-components $S_1$, $S_2$, and $S_3$. The translation introduces variables $v_1$, $v_2$, and $v_3$ which are initialized to the value 'black'. When translating block guard $c_2$ and task $B$ in S-component $S_3$, we only read and write variable $v_3$ representing the color of the token flowing through $S_3$. Correspondingly, when translating $c_4$ and $C$ in $S_1$, we only read and write variable $v_1$, etc. For the colored marking depicted in Fig. 16 the translated WFG has to assign values $v_1$ = grey, $v_2$ = black, $v_3$ = white.

Other model elements are part of several S-components, for instance, task $A$, parallel join $p3$, and exclusive split $e2$ are in $S_1$ and in $S_2$. Our translation has to ensure that a token flowing through these parts is represented by a single color only, i.e., variables $v_1$ and $v_2$ always carry the same value in these situations. At parallel joins, such as $p3$ and $p4$ tokens of different colors are synchronized into a single color (see Fig. 9); our translation has to ensure that variables of all involved S-components are updated accordingly. Finally, guards at exclusive splits such as $e2$ in S-components $S_1$ and $S_2$ have to be extended to route white tokens according to the exit allocation, e.g., in case variables $v_1$ and $v_2$ hold value white.

The next section gives a precise definition of the translation and the corresponding patterns.

*6.2. Translation procedure*

Let $G$ be a sound multipolar WFG as defined in Sect. 4.3. We translate the multipolar WFG $G$ into a regular WFG $G'$ by adding variables and locally transforming $G$ in several steps.

*Step 1: Variables for token colors.* First we obtain a state machine decomposition $SMD(G) = \{S_1, \ldots, S_n\}$ of $G$ [14]. The translation does not depend on which specific decomposition is used. By the decomposition, each node or edge $x$ of $G$ is in one or more S-components, written $SMD(x) = \{S_{i_1}, \ldots, S_{i_k}\}$ in the following. Extend $G$ with fresh global process variables $v_1, \ldots, v_n$ (one for each S-component) each of which can take the values black, grey, and white. In the following, we add tasks to $G$ that update the values of $v_1, \ldots, v_n$ to reflect the color of the token in each S-component. Insert a fresh (automated) task *init* right after the source node of $G$ which initializes $v_1, \ldots, v_n :=$ black (the initial token is black). For example, the state machine decomposition $S_1, S_2, S_3$ of the multipolar WFG shown in Fig. 16 leads to variables $v_1, v_2, v_3$.

*Step 2: Implement task skipping.* For each original task $v$ of $G$, insert an exclusive split $s_v$ immediately before $v$ and an exclusive join $j_v$ immediately after $v$, and add an edge from $s_v$ to $j_v$ to bypass $v$. Further, let $SMD(v) = \{S_{i_1}, \ldots, S_{i_k}\}$. Add the guard expression $v_{i_1} =$ black to the edge $e$ from $s_v$ to $v$. Figure 17 illustrates this translation for task $A$ from the WFG in Fig. 16. This step implements the data-precondition pattern [22, 23] for $v$ and ensures that $v$ is executed only when the control-flow arriving at $v$ corresponds to a black token in $G$. For example, the white token in front of task $B$ in $S_3$ of Fig. 16 translates to a 'regular' token in front of the inserted exclusive split and variable $v_3 =$ white, i.e., in both cases execution $B$ is not executed.

Note that this step assumes that variable $v_{i_1}$ holds the color of the token on the incoming edge of $v$, i.e., that all variables $v_{i_1}, \ldots, v_{i_k}$ agree on the their values if the incoming edge of $v$ holds a token. We will show later that this *consistency assumption* indeed holds.

*Step 3: Translating skip and block guards.* We translate the semantics of skip and block guards for setting token colors given in Fig. 9 into two functions $\text{grey}(\gamma, c)$ and $\text{white}(\gamma, c)$ that take the guard expression $\gamma$ and the color of the incoming token as argument:

- $\text{grey}(\gamma, c) =$ black, if $\gamma =$ true $\wedge\ c =$ black; and $\text{grey}(\gamma, c) =$ white if $c =$ white; and $\text{grey}(\gamma, c) =$ grey, otherwise.

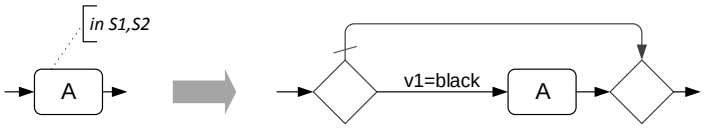


Figure 17: Task skipping is translated to explicit skipping with exclusive gateways that read variables representing token colors.

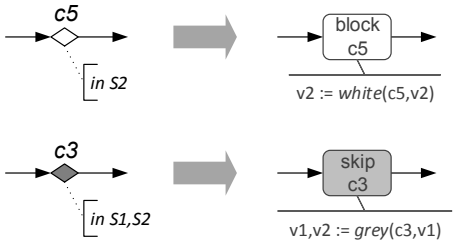


Figure 18: Skip and block guards translate to tasks setting values for variable that represent token colors.

- white$(\gamma, c)$ = black, if $\gamma$ = true $\wedge$ $c$ = black; and white$(\gamma, c)$ = grey, if $\gamma$ = true $\wedge$ $c$ = grey; and white$(\gamma, c)$ = white, otherwise.

We translate skip and block guards to fresh tasks. For each skip guard $v$ (grey mini-diamond) with guard expression $\gamma(v)$ in $G$, where $SMD(v) = \{S_{i_1}, \ldots, S_{i_k}\}$, replace $v$ with a fresh automated task *skip-v* which updates $v_{i_1}, \ldots, v_{i_k} :=$ grey$(\gamma(v), v_{i_1})$. For each block guard $v$ (white mini-diamond) with guard expression $\gamma(v)$ in $G$, where $SMD(v) = \{S_{i_1}, \ldots, S_{i_k}\}$ replace $v$ with a fresh automated task *block-v* which updates $v_{i_1}, \ldots, v_{i_k} :=$ white$(\gamma(v), v_{i_1})$. For example, Fig. 18 illustrates this translation for skip guard $c3$ and block guard $c5$ from the WFG in Fig. 16. The black token in front of block guard $c5$ of Fig. 16 then translates to a 'regular' token in front of *block-c*5 and variable $v_2$ = black; if $\gamma(c5)$ evaluates to false, executing *block-c*5 sets $v_2$ = white(false, black) = white.

Again, we assume that all variables $v_{i_1}, \ldots, v_{i_k}$ are consistent when there is a token on $v$'s input edge; by definition $v_{i_1}, \ldots, v_{i_k}$ will be consistent when there is a token on $v$'s output edge.

*Step 4: Translating parallel gateways.* At a parallel split, a single colored token is split into several tokens of the same color, each following their own S-component, e.g., a token reaching parallel split $p2$ of Fig. 16 is split into two tokens following S-components $S_1$ and $S_2$. At a parallel join, several tokens of different colors coming from different S-components synchronize into a single token with a specific color. For example, at join $p3$ of Fig. 16, the grey token of S-component $S_1$ and the black

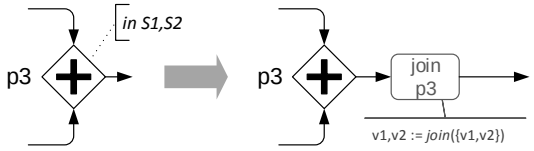


Figure 19: Merging of token colors at AND-joins translates to a task setting variable values.

or white token of S-component $S_2$ are synchronized into a single token on the outgoing edge which is in both S-components $S_1$ and $S_2$. Our translation has to ensure, that the variables $v_1$ = grey and, say, $v_2$ = white are updated consistently, e.g., to $v_1 = v_2$ = grey, as the tokens pass $p3$.

As for the skip and block guards, we translate the synchronization of token colors at a parallel join (see Fig. 9) into a function $join(\{c_1, \ldots, c_k\})$ that takes the set of colors of the incoming tokens as argument:

- $join(\{c_1, \ldots, c_k\})$ = black, if $\exists 1 \leq i \leq k : c_i$ = black; and $join(\{c_1, \ldots, c_k\})$ = grey, if $\forall 1 \leq i \leq k : c_i \neq$ black $\wedge \exists 1 \leq i \leq k : c_i =$ grey; and $join(\{c_1, \ldots, c_k\})$ = white, otherwise.

To translate a parallel join from a multipolar WFG to a regular WFG, we first synchronize the incoming flows with a parallel join and then update the token color in a task. For every parallel join $v$, where $SMD(v) = \{S_{i_1}, \ldots, S_{i_k}\}$, insert a fresh automated task *join-v* on the outgoing edge of $v$ and let *join-v* update all variables $v_{i_1}, \ldots, v_{i_k} := join(\{v_{i_1}, \ldots, v_{i_k}\})$. For example, Fig. 19 illustrates this translation for the parallel join $p3$ from the WFG in Fig. 16.

The parallel split in a multipolar WFG simply preserves the color of the incoming token for all outgoing tokens. If for the incoming edge $e$ of the parallel split $v$ with $SMD(e) = \{S_{i_1}, \ldots, S_{i_k}\}$ all variables $v_{i_1}, \ldots, v_{i_k}$ are consistent, then the parallel split $v$ can remain unchanged. For each token on an outgoing edge of $v$, there is an S-component $S_{i_j}$ for which the corresponding variable $v_{i_j}$ already holds the correct value. We will show later that the assumption holds.

*Step 5: Translating exclusive gateways.* An exclusive gateway forwards an incoming token to an outgoing edge without changing its color. Here, the gateway and all surrounding edges are in the same set of S-components. Thus, our translation of exclusive gateways does not have to update the variables that represent token colors in an S-component. However, we have to translate that white tokens are routed to the exit edge at an exclusive split.

For every exclusive split $v$, where $SMD(v) = \{S_{i_1}, \ldots, S_{i_k}\}$, and for each outgoing edge $e$ of $v$ replace the guard condition $\gamma(e)$ with the following condition:

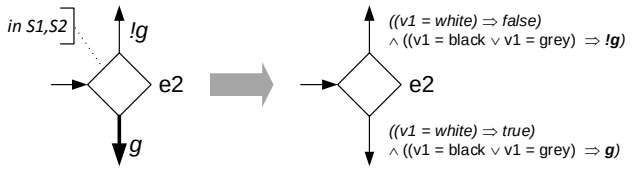


Figure 20: Guards at XOR-gateways are expanded with the token-color to take the exit allocation into account.

- $(v_{i_1} = \text{white} \Rightarrow \text{true}) \wedge ((v_{i_1} = \text{black} \vee v_{i_1} = \text{grey}) \Rightarrow \gamma(e))$, if $e = \phi(v)$ is the exit edge of $v$ (see Def. 1);
- $(v_{i_1} = \text{white} \Rightarrow \text{false}) \wedge ((v_{i_1} = \text{black} \vee v_{i_1} = \text{grey}) \Rightarrow \gamma(e))$, otherwise.

By this condition, black and grey tokens are routed according to the guard expressions, whereas white tokens are always routed to the exit edge. Also here, we assume that all variables $v_{i_1}, \ldots, v_{i_k}$ hold the same value. Figure 20 illustrates this translation for $e2$ from Fig. 16. The situation where the input edge of $s2$ of Fig. 16 holds a white token is translated to a 'regular' token on the input edge of $s2$ and $v_1 = v_2 =$ white, and the token is routed towards the exit ignoring $g$'s value.

An exclusive join only forwards the token to the single outgoing edge; the token neither changes its color nor leaves/enters a new S-component. Thus, our translation leaves exclusive join gateways unchanged. This is the last step in the translation.

*Resulting workflow graph with variables.* Figure 21 shows the result of translating the multipolar WFG of Fig. 16 with the procedure given above (and a simplification of guard expressions at exclusive split $e2$). Altogether, our translation introduced one variable $v_1, v_2, v_3$ for each S-component $SMD(G) = \{S_1, S_2, S_3\}$ of $G$. Task skipping is realized through exclusive splits with guard expressions reading the $v_i$. The semantics of skip guards, block guards, and parallel joins is realized by locally replacing/introducing additional tasks that read and update the $v_i$ according to the semantics of Sect. 4.3. The routing of white tokens towards the exit is realized by updating guards at exclusive splits. Parallel splits and exclusive joins remain as before.

### *6.3. Properties of the translation*

The translation procedure in Sect. 6.2 assumes at several steps that when a token reaches an edge $e$ which is in $SMD(e) = \{S_{i_1}, \ldots, S_{i_k}\}$, then the variables

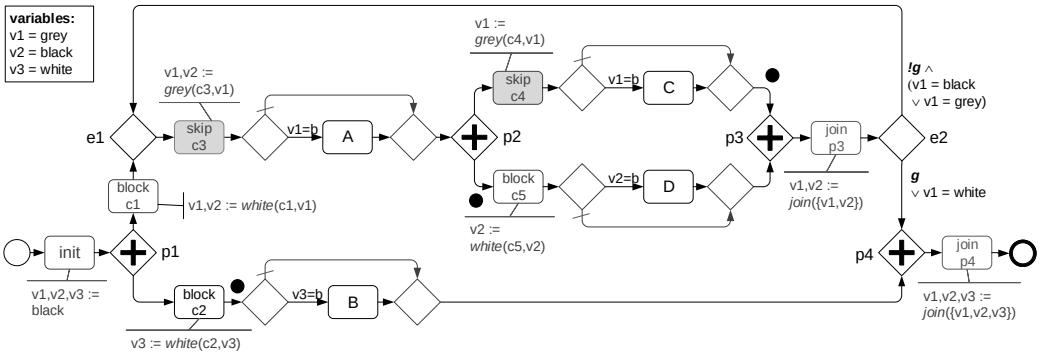


Figure 21: Result of translating the multipolar WFG of Fig. 16 to a regular WFG with variables ($vi = b$ in guards is shorthand for $vi$ = black).

$v_{i_1}, \ldots, v_{i_k}$ hold the same value. This property follows from the even stronger claim that the multipolar WFG and the translated WFG $G'$ correspond in their behaviors.

Let $G$ be a sound multipolar workflow graph with $SMD(G) = \{S_1, \ldots, S_n\}$ and let $G'$ be the WFG obtained by applying the translation procedure of Sect. 6.2. Then for each trace $\sigma$ of $G$ reaching marking $m$ there exists a trace $\sigma'$ of $G'$ reaching marking $m'$ and an assignment $\beta'_m : \{v_1, \ldots, v_n\} \to \{\text{black, grey, white}\}$ such that:

1. For any edge $e$ with $SMD(e) = \{S_{i_1}, \ldots, S_{i_k}\}$ and $m[e] > 0$ holds: $m[e, c] > 0$ iff $m'[e] > 0$ and $\beta'_m(v_{i_1}) = \ldots = \beta'_m(v_{i_k}) = c$ for all $c \in \{\text{black, grey, white}\}$.

2. The projection of $\sigma$ and $\sigma'$ to steps involving visible executions of tasks from the original WFG $G$ are identical (i.e., ignore task skips and ignore automated tasks introduced by the translation).

This claim can be proven by induction on the length of traces. In the following, we sketch the key arguments.

- The token on the initial edge $e$ is black in all S-components $SMD(e) = \{v_1, \ldots, v_n\}$. The new initialization task setting $v_1, \ldots, v_n :=$ black ensures the correspondence initially.

- For the induction step, it is sufficient to consider situations where a node $v$ with $SMD(v) = \{S_{i_1}, \ldots, S_{i_k}\}$ is enabled in a marking $m$ where the above claims hold. As $G$ is sound, and the translation does not change the structure, also $G'$ is sound and on any edge there is at most one token of some color $c$.

- For an enabled task $v$, the incoming edge $e$ is in the same S-components as $v$. By assumption, all $v_{i_1}, \ldots, v_{i_k}$ hold the same value $c = \beta'_m(v_{i_1})$ corresponding to the token color on $e$. So, $G'$ skips $v$ iff $c$ is not black iff $G$ skips $v$. The outgoing edge $e'$ of $v$ is in the same S-components as $v$, $G$ does not change the color of the token, and $G'$ does not change the values of any $v_{i_1}, \ldots, v_{i_k}$.

- For an enabled skip or block guard $v$, the same arguments apply as for enabled tasks; the incoming and outgoing edges are in the same S-components as $v$; the change in token color by $v$ in $G$ is mimicked in $G'$ by updating all variables $v_{i_1}, \ldots, v_{i_k}$ according to the grey(.) and white(.) functions.

- In an enabled parallel split and in an enabled exclusive join, token colors and variable values do not change and $G$ and $G'$ route tokens correspondingly.

- For an enabled parallel join $v$ with outgoing edge $e$ in $SMD(v) = SMD(e) = \{S_{i_1}, \ldots, S_{i_k}\}$, all variables $v_{i_1}, \ldots, v_{i_k}$ are set together by the *join*(.) function in the new task of $G'$ according to the parallel join of $G$.

- For an exclusive split $v$, the incoming and outgoing edges are all in $SMD(v) = \{S_{i_1}, \ldots, S_{i_k}\}$. By assumption all $v_{i_1}, \ldots, v_{i_k}$ hold the same value $c = \beta'_m(v_{i_1})$ corresponding to the token color on the incoming edge of $v$. Hence, the updated guards in $G'$ ensure that the token is routed to the same outgoing edges in $G$ and in $G'$ according to color $c$.

### *6.4. Translation in practice*

Next, we discuss a few aspects regarding applying the translation of multipolar WFG to process models in practice.

To support skip and block guards for process modeling *and* enactment, it is sufficient to extend only the process modeling tool with the syntactic constructs for skip and block guards. Due to the translation, also a model with skip and block guards can be deployed on an existing process execution engine as long as the engine supports global process variables. As the translation requires no user input, it can in principle be applied at any point, i.e., in the modeling tool, during validation, or directly on the execution engine. However, a modeler should generally not be allowed to modify the translated model to ensure that the variables introduced by the translation remain consistent with the introduced tasks and modified guard expressions. The translation itself has polynomial complexity: a state machine decomposition can be computed in cubic time [14], the translation of the WFG is linear in the number of nodes and edges. This makes it possible to

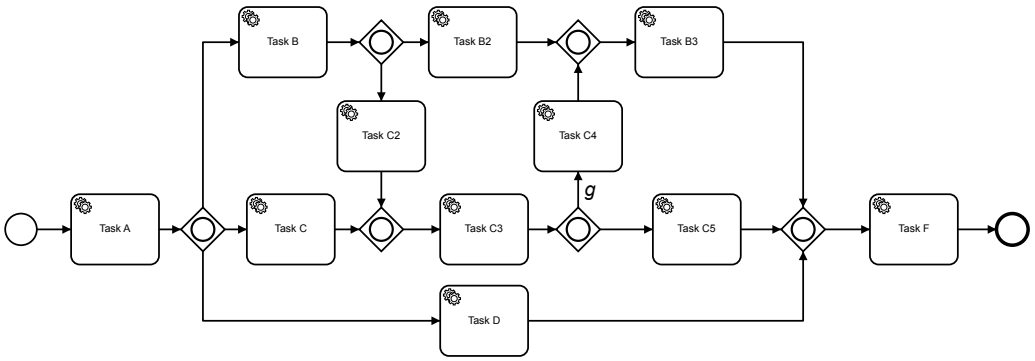


Figure 22: Acyclic, but unstructured process with inclusive gateways.

automatically translate the WFG, for instance, during deployment of a multipolar WFG on a current execution engine.

The translation can also be applied for converting a regular WFG with inclusive gateways into a regular WFG without inclusive gateways. First, manually translate the WFG with inclusive gateways into a multipolar WFG with block guards as discussed in Sect. 5, then translate the multipolar WFG into a regular WFG with variables (and without inclusive gateways) as discussed in Sect. 6.2. This way, processes with inclusive gateways can be deployed on existing process execution engines with no or incomplete support for inclusive gateways. Such a use case is relevant in the light that the non-local inclusive gateway semantics is non-trivial to implement in practice even if it is part of a standard. For instance, the process model shown in Fig. 22 is acyclic and contains inclusive gateways in an unstructured fragment. The model cannot be enacted correctly, for instance, on the Camunda open-source BPM engine[7] which has otherwise strong and efficient support for BPMN 2.0 processes [24]. Executing the model in a situation where guard $g$ on the edge to task $C4$ evaluates to false leads to a trace $A, B, B2, C, (C2), C3, C5, D$ where the process deadlocks as the inclusive gateway in front of $B3$ expects another incoming token from $C4$; the model is executed correctly if $g$ evaluates to true. Translating the model via a multipolar WFG to a WFG without inclusive gateways leads to the correct behavior in all situations. Note that our translation is not limited to acyclic process models, and hence can also enable execution of sound cyclic process models with inclusive gateways on existing engines. Further, the cost of repeatedly deciding at run-time whether an inclusive gateway is enabled can,

[7]https://camunda.org/, version 7.6.0, released Nov 2016

depending on the existing engine, be reduced from exponential [8], quadratic [20], or linear [17] time to constant time, at the cost of a single cubic time translation during deployment.

## 7. Conclusion

We have defined dynamic versions of task skipping and path blocking together with a local semantics which supports efficient execution. Dynamic path blocking comes with dead path elimination, which we have generalized to work for all sound workflow graphs. We argued that when inclusive gateways in BPMN are semantically replaced with parallel gateways with block guards, the advantages outweigh the disadvantages. Note also that workflow graphs with a fully local semantics can be easier and more naturally mapped to Petri nets where a wealth of tools and algorithms is available for their analysis. In particular, verifying soundness can be done in polynomial time for the local semantics (cf. [13]), whereas no polynomial-time algorithm is known to verify soundness under the non-local BPMN semantics for inclusive joins. Finally, the proposed translation to regular workflow graphs makes dynamic skipping and blocking readily available on existing process execution engines. As a consequence, execution engines with no or incomplete support for inclusive joins can support processes that require this behavior.

A future implementation of the proposed constructs and techniques has to take the following considerations into account. The proposed skip and block guards cannot be considered as extensions of the BPMN as allowed by the current standard [11, Sect. 2.1.3]: guards are neither tasks, nor events, nor gateways, and new node types may only be 'Artifacts' not related via sequence flows, i.e., control-flow. Thus, a direct implementation of guards requires to extend both the meta-model of the modeling language, the corresponding graphical editors, as well as the implementation of the semantics. Translating models with skip and block guards to standard-compliant models with data requires the extension of a graphical editor (such as bpmn.io[8] which allows for arbitrary model extensions) and the implementation of the model transformation. However, as the data perspective of BPMN has not been standardized, each execution engines provides its own solution. Thus, the model transformation has to be specific for the target execution engine. We envision an open-source BPMN engine such as Camunda to be suitable for a proof-of-concept implementation and validation in a future study.

[8] http://bpmn.io

*Further related work.* We already discussed in Sect. 3 how our semantics for dynamic skipping and blocking originates from ideas of configurable process models, e.g. [4]. In Sect. 5, we extensively discussed our semantics wrt. works on the inclusive join semantics. Another way to support dynamic skipping and blocking is to give each task an explicit *activation condition* (over process data and control-flow) that, when evaluated to false, leads to *skipping* of the activity [25]; the modeler can choose to further propagate 'true' or 'false' control-flow values by a corresponding start condition.[26, pp.55]. This model was restricted to acylic processes, but can be generalized to models where 'false' flows are contained in a block-structured loop [27]. In these models, an exclusive choice evaluates one outgoing arc to true and all other outgoing arcs to false. However, the modeler has to ensure consistent propagation of 'false' to skip downstream activities of paths that should not be taken. Automated propagation of 'false' edges, i.e., dead path elimination (DPE) as provided by BPEL was discussed in Sect. 1. Where existing DPE proposals suffer from only distinguishing 'false' and 'true' as analyzed in [6, 7], our semantics provides different token colors to distinguish skipping and blocking. Moreover, as we showed in Sect. 4, the ability to also route skipping and blocking flows across exclusive choices (rather than propagating 'false'), makes our approach applicable to any model, including cyclic ones. Yet, our translation of dynamic skipping and blocking to classical models can be simplified if the engine directly supports local task skipping [25, 27], allowing for stronger coherence between modeled and deployed process.

*Additional remarks.* We have assumed a unique sink only for simplicity of the presentation. Multiple sinks can be easily admitted as they are equivalent to an implicit inclusive join that merges all sinks into one. The use of blocking for paths that lead to a sink is compatible with that view and our definition of blocking.

## References


[1] M. Reichert, B. Weber, Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies, Springer, ISBN 978-3-642-30408-8, doi:\bibinfo{doi}{10.1007/978-3-642-30409-5}, URL `http://dx.doi.org/10.1007/978-3-642-30409-5`, 2012.

[2] M. L. Rosa, W. M. P. van der Aalst, M. Dumas, F. Milani, Business Process Variability Modeling: A Survey, ACM Comput. Surv. 50 (1) (2017) 2:1–2:45, doi:\bibinfo{doi}{10.1145/3041957}, URL `http://doi.acm.org/10.1145/3041957`.

[3] D. Fahland, W. M. P. van der Aalst, Model repair - aligning process models to reality, Inf. Syst. 47 (2015) 220–243, doi:\bibinfo{doi}{10.1016/j.is.2013.12.007}, URL http://dx.doi.org/10.1016/j.is.2013.12.007.

[4] F. Gottschalk, W. M. P. van der Aalst, M. H. Jansen-Vullers, M. L. Rosa, Configurable Workflow Models, Int. J. Cooperative Inf. Syst. 17 (2) (2008) 177–221, doi:\bibinfo{doi}{10.1142/S0218843008001798}, URL http://dx.doi.org/10.1142/S0218843008001798.

[5] F. Leymann, W. Altenhuber, Managing Business Processes as an Information Resource, IBM Systems Journal 33 (2) (1994) 326–348, doi:\bibinfo{doi}{10.1147/sj.332.0326}, URL https://doi.org/10.1147/sj.332.0326.

[6] F. van Breugel, M. Koshkina, Dead-Path-Elimination in BPEL4WS, in: Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St. Malo, France, IEEE Computer Society, 192–201, doi:\bibinfo{doi}{10.1109/ACSD.2005.11}, URL http://dx.doi.org/10.1109/ACSD.2005.11, 2005.

[7] M. Weidlich, A. Großkopf, A. P. Barros, Realising Dead Path Elimination in BPMN, in: 2009 IEEE Conference on Commerce and Enterprise Computing, CEC 2009, Vienna, Austria, July 20-23, 2009, IEEE Computer Society, ISBN 978-0-7695-3755-9, 345–352, doi:\bibinfo{doi}{10.1109/CEC.2009.32}, URL http://dx.doi.org/10.1109/CEC.2009.32, 2009.

[8] E. Kindler, On the semantics of EPCs: Resolving the vicious circle, Data Knowl. Eng. 56 (1) (2006) 23–40.

[9] D. Fahland, H. Völzer, Dynamic Skipping and Blocking and Dead Path Elimination for Cyclic Workflows, in: Business Process Management - 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18-22, 2016. Proceedings, vol. 9850, Springer, 234–251, doi:\bibinfo{doi}{10.1007/978-3-319-45348-4_14}, URL http://dx.doi.org/10.1007/978-3-319-45348-4_14, 2016.

[10] C. Favre, D. Fahland, H. Völzer, The relationship between workflow graphs and free-choice workflow nets, Inf. Syst. 47 (2015) 197–219, doi:\bibinfo{doi}{10.1016/j.is.2013.12.004}, URL http://dx.doi.org/10.1016/j.is.2013.12.004.

[11] OMG, Business process model and notation (BPMN) version 2.0, OMG document number dtc/2010-05-03, Tech. Rep., 2010.

[12] J. Desel, J. Esparza, Free choice Petri nets, Cambridge University Press, New York, NY, USA, ISBN 0-521-46519-2, 1995.

[13] C. Favre, H. Völzer, P. Müller, Diagnostic Information for Control-Flow Analysis of Workflow Graphs (a.k.a. Free-Choice Workflow Nets), in: M. Chechik, J. Raskin (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, vol. 9636 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-662-49673-2, 463–479, doi:\bibinfo{doi}{10.1007/978-3-662-49674-9_27}, URL https://doi.org/10.1007/978-3-662-49674-9_27, 2016.

[14] P. Kemper, F. Bause, An Efficient Polynomial-Time Algorithm to Decide Liveness and Boundedness of Free-Choice Nets, in: ICATPN '92, Proceedings, vol. 616 of *LNCS*, Springer, ISBN 3-540-55676-1, 263–278, doi:\bibinfo{doi}{10.1007/3-540-55676-1_15}, URL http://dx.doi.org/10.1007/3-540-55676-1_15, 1992.

[15] H. J. Genrich, P. S. Thiagarajan, A Theory of Bipolar Synchronization Schemes, Theor. Comput. Sci. 30 (1984) 241–318, doi:\bibinfo{doi}{10.1016/0304-3975(84)90137-3}, URL http://dx.doi.org/10.1016/0304-3975(84)90137-3.

[16] W. M. P. van der Aalst, N. Lohmann, M. L. Rosa, J. Xu, Correctness Ensuring Process Configuration: An Approach Based on Partner Synthesis, in: BPM 2010, vol. 6336 of *Lecture Notes in Computer Science*, 95–111, doi:\bibinfo{doi}{10.1007/978-3-642-15618-2_9}, URL http://dx.doi.org/10.1007/978-3-642-15618-2_9, 2010.

[17] H. Völzer, A new semantics for the inclusive converging gateway in safe processes, in: Proceedings of the 8th international conference on Business process management, BPM'10, Springer-Verlag, Berlin, Heidelberg, ISBN 3-642-15617-7, 978-3-642-15617-5, 294–309, URL http://portal.acm.org/citation.cfm?id=1882061.1882089, 2010.

[18] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, A. Barros, Workflow patterns, Distributed and parallel databases 14 (1) (2003) 5–51.

[19] J. Vanhatalo, H. Völzer, F. Leymann, Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition, in: ICSOC, vol. 4749 of *LNCS*, Springer, ISBN 978-3-540-74973-8, 43–55, 2007.

[20] M. Dumas, A. Großkopf, T. Hettel, M. T. Wynn, Semantics of Standard Process Models with OR-Joins, in: OTM Conferences (1), vol. 4803 of *LNCS*, Springer, ISBN 978-3-540-76846-3, 41–58, 2007.

[21] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580.

[22] N. Russell, A. H. M. ter Hofstede, D. Edmond, W. M. P. van der Aalst, Workflow Data Patterns: Identification, Representation and Tool Support, in: Conceptual Modeling - ER 2005, 24th International Conference on Conceptual Modeling, Klagenfurt, Austria, October 24-28, 2005, Proceedings, vol. 3716 of *Lecture Notes in Computer Science*, 353–368, doi:\bibinfo{doi}{10.1007/11568322_23}, URL http://dx.doi.org/10.1007/11568322_23, 2005.

[23] N. Russell, A. H. M. ter Hofstede, D. Edmond, W. M. P. van der Aalst, Workflow Data Patterns, QUT Technical report FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[24] M. Skouradaki, V. Ferme, C. Pautasso, F. Leymann, A. van Hoorn, Micro-Benchmarking BPMN 2.0 Workflow Management Systems with Workflow Patterns, in: CAiSE 2016, vol. 9694 of *Lecture Notes in Computer Science*, Springer, 67–82, doi:\bibinfo{doi}{10.1007/978-3-319-39696-5_5}, URL http://dx.doi.org/10.1007/978-3-319-39696-5_5, 2016.

[25] M. Weske, Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System, in: 34th Annual Hawaii International Conference on System Sciences (HICSS-34), January 3-6, 2001, Maui, Hawaii, USA, IEEE Computer Society, doi:\bibinfo{doi}{10.1109/HICSS.2001.927082}, URL http://dx.doi.org/10.1109/HICSS.2001.927082, 2001.

[26] M. Weske, Workflow Management Systems: Formal Foundation, Conceptual Design, Implementation Aspects., Habilitationsschrift Fachbereich Mathematik und Informatik, Universität Münster, 2000.

[27] M. Reichert, P. Dadam, ADEPT$_{\text{flex}}$-Supporting Dynamic Changes of Workflows Without Losing Control, J. Intell. Inf. Syst. 10 (2) (1998) 93–129, doi:\bibinfo{doi}{10.1023/A:1008604709862}, URL http://dx.doi.org/10.1023/A:1008604709862.

## Appendix A. Formal Semantics of Multipolar Workflow Graphs

We define two versions of our model, one where the data decisions are explicit and another where the data decisions are implicit modeled through nondeterminism.

**Definition 2.** Let $X$ be a set of *variables* for data objects and Expr($X$) a set of Boolean expressions over $X$. A (high-level ) *multipolar workflow graph* $G$ consists of (i) a workflow graph with two additional node types *skip guard* and *block guard* where each guard has a unique incoming and a unique outgoing edge and (ii) a mapping $\gamma : V \to \text{Expr}(X)$ that assigns an expression to each guard (and to each outgoing edge of an exclusive split).

A *marking* of $G$ is a mapping $m : E \times \{\text{black}, \text{grey}, \text{white}\} \to \mathbb{N}$ that assigns each edge a nonnegative number of black, grey and white tokens. The marking that has a single black token on the source and no token of any color elsewhere is called the *initial marking* of $G$. We write $m[e, c]$ instead of $m(e, c)$. Markings can be added and compared pointwise: $(m_1 + m_2)[e, c] = m_1[e, c] + m_2[e, c]$ and $m_1 \leq m_2$ iff there exists a marking $m$ such that $m_1 + m = m_2$. We write $m[e]$ for $\sum_c m[e, c]$. A marking $m$ is *safe* if each edge has at most one token of any color, i.e., $m[e] \leq 1$ for each edge $e$.

A *state* $s = (m, \text{eval}_s)$ of $G$ consists of a marking $m$ of $G$ and a mapping $\text{eval}_s : \text{Expr}(X) \to \{\text{true}, \text{false}\}$ that evaluates any expression $\alpha \in \text{Expr}(X)$.

A *transition* $t = (t^-, v, t^+)$ of $G$ consists of a node $v \in V$ and two safe markings $t^-, t^+$; which represent the set of tokens consumed by $t$ and produced by $t$, respectively. Let, for a subset $C \subseteq \{\text{black}, \text{grey}, \text{white}\}$, $\max C$ denote the maximum of the set $C$ w.r.t. the order black $>$ grey $>$ white. For a transition $t = (t^-, v, t^+)$ and two states $s = (m, \text{eval}_s)$ and $s' = (m', \text{eval}_{s'})$, we define the relation $s \xrightarrow{t} s'$, pronounced $t$ is *enabled* in $s$ and *firing* of $t$ in $s$ results in $s'$ as follows: $s \xrightarrow{t} s'$ iff $t^- \leq m$ and $m + t^+ = m' + t^-$ such that either

- $v$ is a parallel gateway such that (i) $t^-[e] = 1$ iff $e \in {}^\circ v$ (which implies $t^-[e] = 0$ iff $e \notin {}^\circ v$ by safeness of $t^-$) and (ii) $t^+[e, c] = 1$ iff $e \in v^\circ$ and $c = \max t^-$.

- $v$ is an exclusive gateway, a task, or a guard and there exist $e^- \in {}^\circ v$ and $e^+ \in v^\circ$ such that (i) $t^-[e] = 1$ iff $e = e^-$ and (ii) $t^+[e] = 1$ iff $e = e^+$ and furthermore (iii) where $c^-$ and $c^+$ denote the colors such that $t^-[e^-, c^-] = 1$ and $t^+[e^+, c^+] = 1$ such that
  - if $v$ is an exclusive gateway or a task, then $c^- = c^+$,
  - if $v$ is a skip guard, then $c^+ = \text{white}$ iff $c^- = \text{white}$ and $c^+ = \text{black}$ iff $\text{eval}_s(\gamma(v)) = \text{true}$
  - if $v$ is a block guard, then (i) $c^+ = \text{white}$ or $c^- = c^+$ and (ii) $c^+ = \text{white}$ iff $c^- = \text{white}$ or $\text{eval}_s(\gamma(v)) = \text{false}$
  - if $v$ is an exclusive split and $c^- \neq \text{white}$, then $\text{eval}_s(\gamma(e^+)) = \text{true}$ and $\text{eval}_s(\gamma(e)) = \text{false}$ for $e \neq e^+$
  - $\text{eval}_s = \text{eval}_{s'}$ or $v$ is a task and $c^- = \text{black}$.

To define *low-level multipolar workflow graph*, we use the above definition of a high-level multipolar workflow graph without any reference to variables, expressions, and the mappings $\gamma$ and $\text{eval}_s$. A state consist only of a marking and we simply write $m \xrightarrow{t} m'$. Since we also remove any reference to $\text{eval}_s$ in the transition rules, the data-based decisions are replaced by non-determinism.

Given an exit allocation $\phi$ for $G$, we say that an elimination step $m \overset{t}{\dashrightarrow} m'$ *complies with* $\phi$ where $t = (t^-, v, t^+)$ if the following holds: If $v$ is an exclusive split where $t^+[e^+] = 1$, then $e^+ = \phi(v)$.

## Appendix B. Proof of Lemma 2 (Elimination Steps)

In Sect. 4.3 we use the formal model of multipolar WFG to show that the routing of white tokens across tasks and gateways does not influence the behavior of the grey and black tokens. Next we give the proof that any two maximal fair elimination sequences starting in $m_0$ reach the same marking $m_1$.

*Lemma 2.* The proof is indirect. Suppose $m_1 \neq m_2$. Since $m_1$ and $m_2$ can only differ in the location of white tokens, it follows that there is an edge $e_1$ such that $m_1[e_1, \text{white}] = 1$ and $m_2[e_1] = 0$.

Consider a state machine decomposition of the WFG and an S-component $S$ such that $e_1 \in S$. Since $S$ is always marked with exactly one token, call $e_0$ and $e_2$ the edges in $S$ such that $m_0[e_0] = 1$ and $m_2[e_2] = 1$. It follows that these two tokens

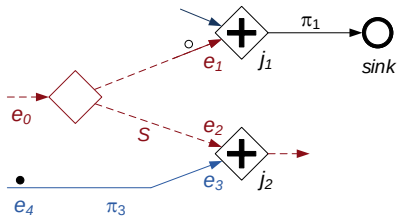


Figure B.23: Illustration for the proof of Lemma 2

are also white. Because $m_2[e_1] = 0$ and $m_2[e_2] = 1$, we have $e_1 \neq e_2$. Because the eliminations are assumed to be maximal, the white token on $e_1$ and $e_2$ are in front of an parallel join or the sink.

We consider here the case that the target nodes of both, $e_1$ and $e_2$ are parallel joins, denoted $j_1$ and $j_2$ respectively. The other case that the target node of one of these edges is the sink is similar.

Because $e_1$ and $e_2$ belong to the same S-component, we have $j_1 \neq j_2$. Since component $S$ is strongly connected, let $\pi_i$, $i = 1, 2$ be a simple path from $e_i$ to the sink. It is clear that the paths $\pi_i$ can be chosen such that $j_2$ is not on $\pi_1$ or that $j_1$ is not on $\pi_2$. W.l.o.g., let's assume that $j_2$ is not on $\pi_1$.

We consider now marking $m_2$ and recall that there is a white token on $e_2$ in front of the parallel join $j_2$. Let $e_3 \neq e_2$ be another incoming edge of $j_2$. Because our graph is sound and hence $m_2$ is not a local deadlock, there must be a token on some edge that has a path to $e_3$. Because $m_2$ is maximally eliminated, there must be a such a token that is black. Hence, let $e_4$ be an edge and $\pi_3$ be a simple path from $e_4$ to $e_3$ such that $m_2[e_4, \text{black}] = 1$. Therefore, we also have $m_0[e_4, \text{black}] = 1$ and $m_1[e_4, \text{black}] = 1$.

Fig. B.23 shows the overall situation with the marking $m_1$, which has a white token on $e_1$ and a black token on $e_4$. We consider now a marking $m$ that is reachable from $m_1$ such that there is a token (of any color) on $\pi_1$ and a token (of any color) on $\pi_3$. Clearly $m_1$ itself is such a marking. Hence, there is a marking $m^*$ that is a maximal such marking in the sense that tokens on $\pi_1$ and $\pi_3$ have progressed maximally, i.e., have a minimal distance to the sink and to $j_2$ respectively, and we furthermore assume that no node other than $j_2$ is enabled in $m^*$.

If the token on $\pi_1$ has reached the sink in $m^*$, then $m^*$ manifests an improper termination (i.e. a marking with a token on the sink edge and a token elsewhere) because of the other token on $\pi_3$, which contradicts soundness. Therefore and because $m^*$ was chosen maximally in the sense above, both tokens are in front of

an parallel join in $m^*$. Since $m^*$ cannot be a deadlock, some node must be enabled, which can be only $j_2$ due to our assumption. However, this is impossible because a token on $\pi_1$ implies that there is no token on $e_2$ (recall that $e_2$ and $\pi_1$ belong to the same S-component and $e_2$ cannot be on $\pi_1$ because we assumed $j_2$ is not on $\pi_1$.) □