Dynamic Skipping and Blocking and Dead Path Elimination for Cyclic Workflows

Dirk Fahland[1(✉)] and Hagen Völzer[2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
d.fahland@tue.nl
[2] IBM Research - Zurich, Zurich, Switzerland
hvo@ibm.zurich.com

Abstract. We propose and study dynamic versions of the classical flexibility constructs skip and block and motivate and define a formal semantics for them. We show that our semantics for dynamic blocking is a generalization of classical dead-path-elimination and solves the long-standing open problem to define dead-path elimination for cyclic workflows. This gives rise to a simple and fully local semantics for inclusive gateways.

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 [13,14]. 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 [3] illustrates the problem; the process model (Fig. 1) has 80 process steps (white) and 20 routing constructs (grey). As 65 % of the process steps are optional under some condition, the model also contains 52 explicit paths for skipping 38 single process steps (black) or 13 segments of multiple process steps (red). Paths for skipping are not mutually exclusive but overlap as indicated by the highlighted segment in the middle of Fig. 1. 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 [3], their share of optional process steps ranges between 50 % and 63 % to allow for 516/5363 to 1574/7684 skips in 1 year. Creating, understanding, and

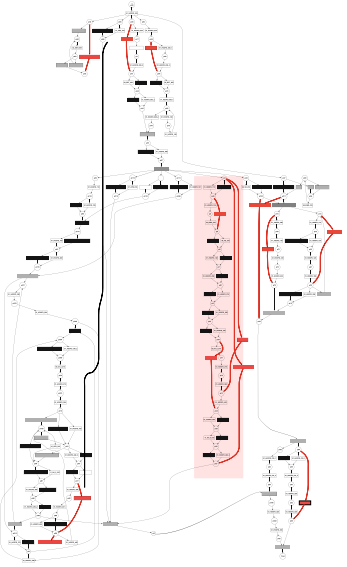

Fig. 1. Model of a flexible process (Color figure online)

M. La Rosa et al. (Eds.): BPM 2016, LNCS 9850, pp. 234–251, 2016.
DOI: 10.1007/978-3-319-45348-4_14

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 [7]. However, in many processes, skipping and blocking dynamically depend on user input or dynamically computed data, e.g., Fig. 1 [3]. 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 - however, mapping classical skip and block to their implementation in BPEL is not straight-forward. 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 [15,18]. 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.
2. We show that the proposed dynamic blocking generalizes the Dead-Path-Elimination (DPE) concept [18], which so far was limited to acyclic control flow.
3. We show that dynamic blocking is equivalent to having no control-flow on the edge. This allows a modeler to comprehend the semantics of 'block' flows as their intended concept: absence of flow. Therefore, 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 joins. As a result, our semantics does not entail semantic anomalies such as 'vicious cycles' (see, e.g. [9]). 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.

We start by discussing concepts for dynamic skipping and blocking based on literature for static skipping and blocking in Sect. 2. In Sect. 3, we generalize dead path elimination to all sound workflow graphs. The resulting local semantics for inclusive gateways is discussed in Sect. 4 in the context of the larger body of literature on inclusive join semantics. We conclude in Sect. 5 where we also compare conceptual and subtle differences of our approach to the literature.

2 Dynamic Skipping and Blocking

In this section, we present dynamic versions of task skipping and path blocking together with modeling examples. These constructs are inspired by their well-known static counterparts. For example, the approach by Gottschalk et al. [7] 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.

Our exposition is partially based on the view that a process is a synchronization of state machines. We start by explaining that view.

2.1 Workflow Graphs as Synchronized State Machines

We work with *workflow graphs* [5] as the model of the core constructs of business process models. Other modeling elements, 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* 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 such that 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. Further, each outgoing edge e of an exclusive gateway has a guarding expression $\gamma(e)$. We use the BPMN [11] semantics and visualization for workflow graphs. We will restrict to *sound* workflow graphs, which are defined in Sect. 3.3.

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-components 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.

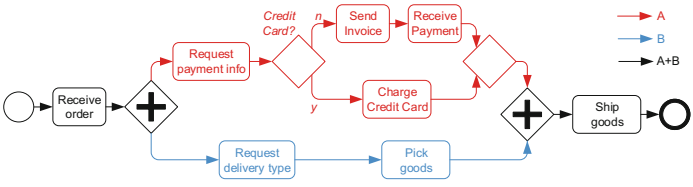

Fig. 2. A workflow graph with two S-components A (red and black) and B (blue and black) (Color figure online)

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 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 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 [8]. An important property of S-components is that each S-component is always marked with exactly one token.

2.2 Dynamic Skip

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

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

A simple application of a skip guard (grey mini-diamond) 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 in order to make sure that the third activity 'Inform customer' is executed in any case.

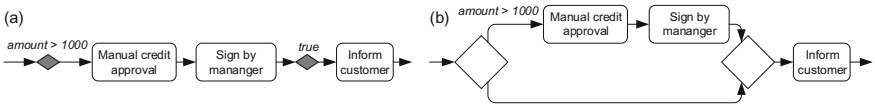

Fig. 3. (a) A simple example for skip guards. (b) Its corresponding explicit representation.

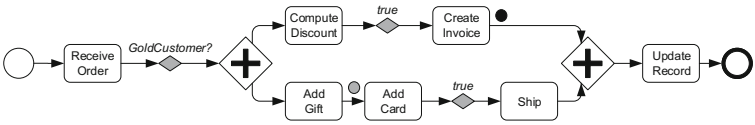

Fig. 4. A workflow graph with three skip guards, a grey and a black token.

A more complex example for skip guards 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. If the customer is not a gold customer, then the activities 'Compute discount', 'Add gift' and 'Add card' are skipped.

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 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. Figure 4 shows a reachable marking of the corresponding graph with one black and one grey token.

2.3 Dynamic Block

A skip guard can switch on or off an S-component repeatedly. In contrast, a *block guard* blocks an S-component persistently, i.e., after a blocking, the S-component cannot be switched on again by another guard. Thus any activity on the S-component is skipped until the S-component synchronizes 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). 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. Figure 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).

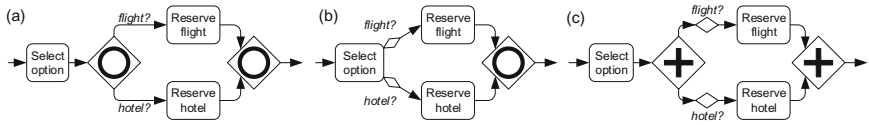

Fig. 5. (a) Inclusive split and -join, (b) Inclusive split with BPMN mini-diamond, (c) Proposed notation with block guards

Figure 6 shows a more elaborate example for the use of block guards. 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, is switched off subsequently 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. 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.

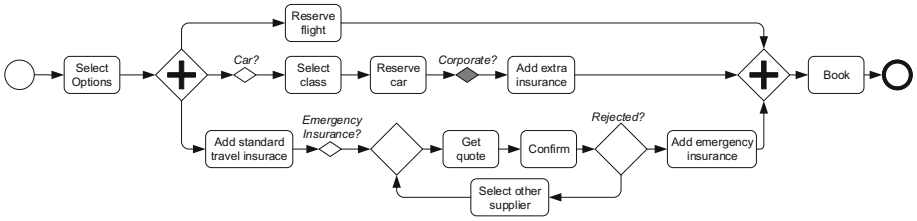

Fig. 6. A more complex example for a block guard

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. A parallel split 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. 3.2.

3 Dead Path Elimination for Cyclic Workflows

In this section, we define the routing of grey and white tokens in exclusive splits, and we present dead-path elimination for cyclic workflow graphs.

3.1 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. Therefore, it matters how we route a grey token and we route it as a black token, i.e., according to the evaluation of the data expression in the exclusive split – firmly controlled by the modeler.

This requires care since the data variables that the exclusive split refers to must be in an expected state, in particular must be defined at all. This is not trivial since some activities (this is where data is set) 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* which 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). Likewise, 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. On the other hand, grey tokens, i.e., 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.

3.2 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

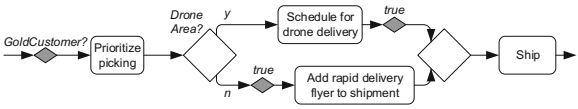

Fig. 7. Flexible skipping in different branches of an S-component

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 seen in 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 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 ϕ 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 Definition 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. Note that a workflow graph is equivalent to a corresponding isomorphic free-choice Petri net [5]. Therefore, we can directly apply the theory of free-choice Petri nets to workflow graphs. An exit allocation is an *allocation pointing to* the sink in the sense of [1, Definition 6.4]. Existence follows from [1, Lemma 6.5 (1)]. 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 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.

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

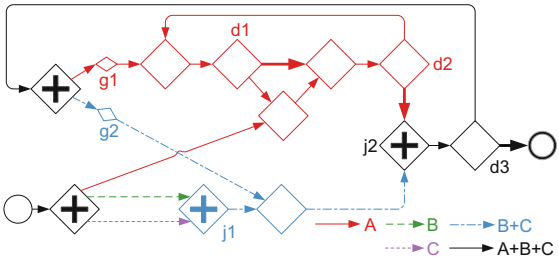

Fig. 8. A workflow graph decomposed into three S-components A (red and black), B (orange, green, and black) and C (blue, green, and black). An exit allocation is shown in bold. (Color figure online)

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 have to formalize our extended model of workflow graphs and their semantics.

3.3 Multipolar Workflow Graphs

In this section, we present the formal concepts for our model, which we named *multipolar workflow graph* based on the bipolar synchronization schemes of [6] which introduced true/false tokens for graphs without choices; see [4, Appendix A] for a rigorous formalization of our model.

A *multipolar workflow graph* G consists of a workflow graph with two additional nodes types *skip guard* (small grey diamond) and *block guard* (small white diamond); $^\circ v$ and v° denote the input and output edges of node v, respectively. Each guard v has one incoming and one outgoing edge and is annotated with an expression $\gamma(v)$ over some data variables, where the data is accessed during process and updated during task execution. A *marking* m 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*

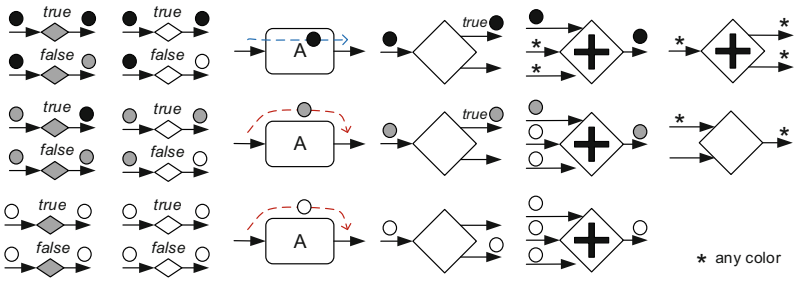

Fig. 9. Possible transitions of nodes in a multipolar workflow graph.

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 classical 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 [4, 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 ϕ for G, we say that an elimination step $m \overset{t}{\dashrightarrow} m'$ *complies with* ϕ if the white token is produced on $e^+ = \phi(v)$ 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 $i \geq 0$; σ 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 σ 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 σ, then e is marked infinitely often in σ.

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.

3.4 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 graph. We first observe that an exit allocation implements fair behavior:

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

Proof. The claim follows from [1, Lemma 6.5, (2)]: The free-choice workflow net that corresponds to the workflow graph, cf. [5], is slightly modified by adding a transition, called the *return transition*, that consumes a token from the sink and produces a token on the source. This version is called the *connected version*. It is strongly connected and the exit allocation points to the return transition in the sense of [1]. Furthermore, each reachable marking of a sound workflow graph is bounded. [1, Lemma 6.5, (2)] now implies that an infinite elimination sequence implies infinitely many firings of the return transition, which implies the claim.

In the following, we will prove that all exit allocations generate essentially the same behavior. We prove first that two maximal elimination sequences that comply with the same exit allocation end in the same marking:

Lemma 1. *Let m_0 be a reachable marking, ϕ be an exit allocation and $m_0 \overset{max}{\dashrightarrow} m_1$ and $m_0 \overset{max}{\dashrightarrow} m_2$ be two maximal elimination sequences that comply with ϕ. 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. Therefore, it can be shown by induction 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. [1]) 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 fair, 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 fair maximal elimination sequences. Then we have $m_1 = m_2$.*

Proof (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. The detailed proof is given in [4, Appendix B]

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 σ and σ' 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 σ and σ', 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 fair and maximal elimination sequences.*

Then, σ and σ' 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 tokes, i.e., that a white token always remains white. This property allows us to route them automatically.

We have shown 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. 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. 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 and fully local semantics.

4 Dynamic Blocking as Inclusive Gateway Semantics

We have argued already in Sect. 2.3 that parallel gateways in combination with block guards with their semantics of white tokens and dead path elimination provide a generalization of inclusive gateways. This allows us to propose to use the semantics proposed above as an inclusive join semantics. Since we only use the existing constructs of parallel gateways and white mini-diamonds, the new semantics would allow us to use inclusive gateways merely as syntactic sugar for parallel gateways with block guards or to abolish the inclusive gateways altogether. Next, we discuss such a proposal in more detail.

Many papers on the inclusive join semantics (aka Or-join semantics) problem have been published, for a survey see [17]. In this section, we compare our proposal with existing semantics from the literature with respect to various properties.

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., [9]. Therefore, enactment 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 [2,17]. It can be determined in linear [17] or quadratic time [2] 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. [16]. Figure 10 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. 10) in the BPMN semantics. This is 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

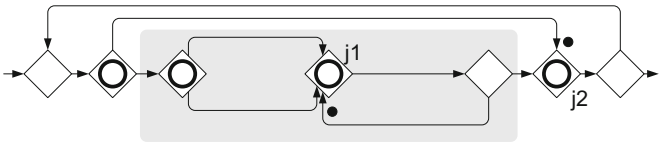

Fig. 10. The current BPMN semantics for inclusive joins is not fully compositional w.r.t. single-entry-single-exit fragments.

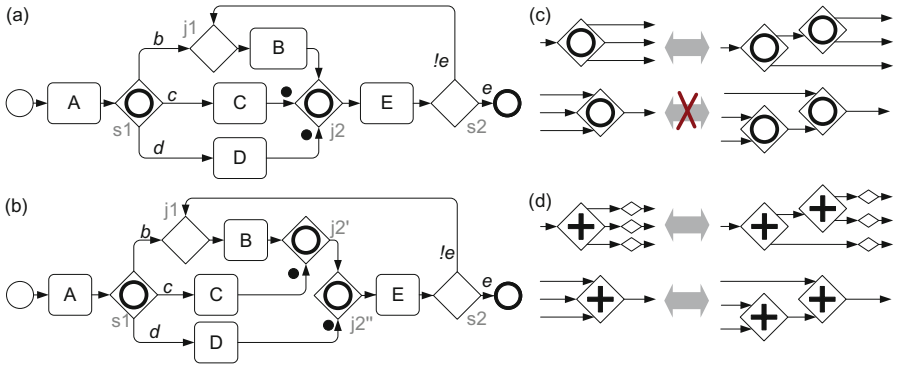

Fig. 11. The BPMN inclusive join semantics is not robust under node splitting (a) and (b). The corresponding refactoring rule (c) is valid in our semantics (d).

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 [16]. The example in Fig. 10 shows that the non-local BPMN semantics [11,17] is not compositional in general in this sense.

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 [10], but which apply to workflow graphs as well. For example, the two patterns shown in Fig. 11(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 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. 11(c)). As a counterexample, we consider the model Fig. 11(b) that is obtained by applying the rule for inclusive joins from Fig. 11(c) on join j_2 of the model of Fig. 11(a). The workflow graph in Fig. 11(b) is not sound – the marking shown is a deadlock in the non-local semantics of BPMN, whereas the workflow graph of Fig. 11(a) is sound. Therefore, for that semantics, the rule in Fig. 11(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 (there may no token arrive). Transformation rules such as the one of Fig. 11(d) preserve whether an edge is live or dead, and hence can be applied in any situation. How to express the model of Fig. 11(a) in our semantics is discussed next; Fig. 13 shows the result.

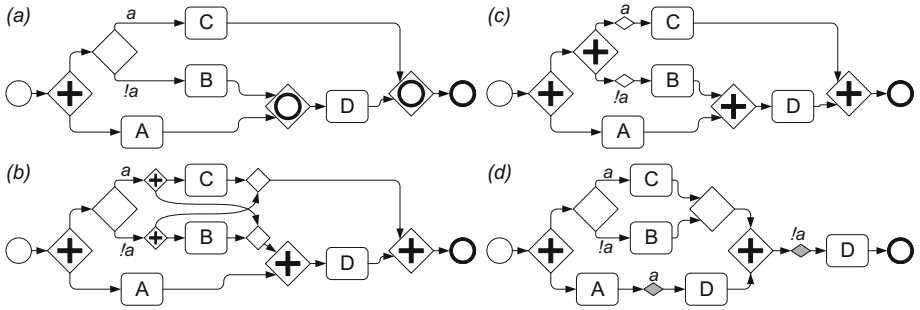

Fig. 12. An acyclic workflow graph with inclusive joins (a) and its equivalent models (b–d)

Expressiveness. A semantics is preferable to another if it can model more (realistic) process behaviors than the other in a concise way. The best documented modeling use case for inclusive gateways is the structured synchronizing merge pattern, cf. Fig. 5(a). Figure 5(c) already showed that this case can be modeled isomorphically with parallel gateways and block guards.

Another well-understood context for the use of inclusive joins are acyclic workflow graphs. There is little debate what semantics inclusive joins should have in acyclic graphs, only how to express that semantics such that it generalizes to a semantics for general workflow graphs. Figure 12(a) shows a paradigmatic example, where a task D is executed after concurrent tasks A and B, but B is executed only when the condition a is false. Often, but not always, all inclusive joins can be replaced by parallel joins [5], at the price of introducing additional auxiliary paths called *bridges*, cf. Fig. 12(b). However, each acyclic graph in BPMN semantics is equivalent when each inclusive and exclusive split is replaced with a parallel split with block guards and each exclusive and inclusive join is replaced with a parallel join, which follows from [17, Theorem 2]. The resulting workflow graph is a BPEL flow with DPE semantics. For the graph in Fig. 12(a), the resulting graph is shown Fig. 12(c). Note that we can also remodel the graph in Fig. 12(a) into a well-structured graph, i.e., a graph where the gateways are matching pairs of split and join, cf. Fig. 12(d). We do this by using skip guards at the expense of duplicating the conditions a and $!a$ and the task D.

Acyclic workflow graphs can be composed by single-entry-single-exit nesting with arbitrary sequential workflow graphs. For the resulting workflow graphs, the BPMN semantics and our local semantics here agree, which follows from the theorems in [17]. Only for workflow graphs that cannot be obtained in this way, the two semantics disagree. A few such graphs with sound behavior have been documented for technical discussion [2,17]; the model of Fig. 11(a) is an example. We have verified that these workflow graphs documented there can be easily equivalently modeled with our local semantics, again by adding bridges. Figure 13 shows the model of Fig. 11(a) in our semantics. The bridges producing white

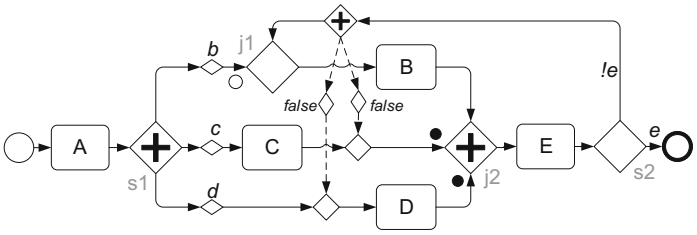

Fig. 13. Representation of the unstructured loop with inclusive gateways of Fig. 11(a) in our semantics.

tokens (by the 'false' block guards) can be merged anywhere along the c and d branches. In summary, the rare modeling cases where concurrency and loops cannot be separated by single-entry-single-exit decomposition can still easily be supported by the local semantics at the expense of extra bridges. The extra bridges for rare models are a small price to pay for faster execution across all models and the other advantages of the local semantics discussed above.

Note that the default outgoing edge of inclusive splits as defined in BPMN [11] to ensure a token in the sink upon termination is not necessary in the local semantics, because the white tokens that are sent take care of these issues.

5 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, whereas no polynomial-time algorithm is known to verify soundness under the non-local BPMN semantics for inclusive joins.

Further related work. We already discussed in Sect. 2 how our semantics for dynamic skipping and blocking originates from ideas of configurable process models, e.g. [7]. In Sect. 4 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 [20]; the modeler can choose to further propagate 'true' or 'false' control-flow values by a corresponding start condition [19, p. 55]. This model was restricted to acylic processes, but can be generalized to models where 'false' flows are

contained in a block-structured loop [12]. 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 [15,18], our semantics provides different token colors to distinguish skipping and blocking. Moreover, as we showed in Sect. 3, the ability to also route skipping and blocking flow across exclusive choices (rather than propagating 'false'), makes our approach applicable to any model, including cyclic ones.

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. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge University Press, New York (1995)
2. Dumas, M., Grosskopf, A., Hettel, T., Wynn, M.T.: Semantics of standard process models with OR-joins. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 41–58. Springer, Heidelberg (2007)
3. Fahland, D., van der Aalst, W.M.P.: Model repair - aligning process models to reality. Inf. Syst. **47**, 220–243 (2015)
4. Fahland, D., Völzer, H.: Dynamic Skipping and Blocking and Dead Path Elimination for Cyclic Workflows (Ext. Version). BPM Center Report BPM-16-05 (2016). http://bpmcenter.org
5. Favre, C., Fahland, D., Völzer, H.: The relationship between workflow graphs and free-choice workflow nets. Inf. Syst. **47**, 197–219 (2015)
6. Genrich, H.J., Thiagarajan, P.S.: A theory of bipolar synchronization schemes. Theor. Comput. Sci. **30**, 241–318 (1984)
7. Gottschalk, F., van der Aalst, W.M.P., Jansen-Vullers, M.H., La Rosa, M.: Configurable workflow models. Int. J. Coop. Inf. Syst. **17**(2), 177–221 (2008)
8. Kemper, P., Bause, F.: An efficient polynomial-time algorithm to decide liveness and boundedness of free-choice nets. In: Jensen, K. (ed.) ICATPN 1992. LNCS, vol. 616, pp. 263–278. Springer, Heidelberg (1992)
9. Kindler, E.: On the semantics of EPCs: resolving the vicious circle. Data Knowl. Eng. **56**(1), 23–40 (2006)
10. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
11. OMG: Business process model and notation (BPMN) version 2.0, OMG document number dtc/2010-05-03. Technical report (2010)
12. Reichert, M., Dadam, P.: ADEPT$_{\text{flex}}$-supporting dynamic changes of workflows without losing control. J. Intell. Inf. Syst. **10**(2), 93–129 (1998)

13. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer, Heidelberg (2012)
14. La Rosa, M., van der Aalst, W.M.P., Dumas, M., Milani, F.P.: Business process variability modeling: a survey. QUT e-Print 61842, QUT, Australia (2013)
15. van Breugel, F., Koshkina, M.: 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, pp. 192–201. IEEE Computer Society (2005)
16. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models through SESE decomposition. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 43–55. Springer, Heidelberg (2007)
17. Völzer, H.: A new semantics for the inclusive converging gateway in safe processes. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 294–309. Springer, Heidelberg (2010)
18. Weidlich, M., Großkopf, A., Barros, A.P.: Realising dead path elimination in BPMN. In: 2009 IEEE Conference on Commerce and Enterprise Computing, CEC 2009, Vienna, Austria, 20–23 July 2009, pp. 345–352. IEEE Computer Society (2009)
19. Weske, M.: Workflow management systems: formal foundation, conceptual design, implementation aspects. Habilitationsschrift Fachbereich Mathematik und Informatik, Universität Münster (2000)
20. Weske, M.: Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In: 34th Annual Hawaii International Conference on System Sciences (HICSS-34), 3–6 January 2001, Maui, Hawaii, USA. IEEE Computer Society (2001)