

Automatic Workflow Graph Refactoring and Completion

Jussi Vanhatalo^{1,2}, Hagen Völzer¹, Frank Leymann², and Simon Moser³

¹ IBM Zurich Research Laboratory, Switzerland, {juv,hvo}@zurich.ibm.com

² Institute of Architecture of Application Systems, University of Stuttgart, Germany, frank.leymann@iaas.uni-stuttgart.de

³ IBM Böblingen Software Laboratory, Germany, smoser@de.ibm.com

Abstract. Workflow graphs are used to model the control flow of business processes in various languages, e.g., BPMN, EPCs and UML activity diagrams. We present techniques for automatic workflow graph refactoring and completion. These techniques enable various use cases in modeling and runtime optimization. For example they allow us to complete a partial workflow graph, they provide local termination detection for workflow graphs with multiple ends, and they allow us to execute models containing OR-joins faster. Some of our techniques are based on workflow graph parsing and the Refined Process Structure Tree [10].

1 Introduction

A *workflow graph* shows the control flow of a business process similar to a flow chart as a directed graph. Figure 1(a) shows an example. Workflow graphs form the core of many specification languages, e.g., BPMN, EPCs and UML activity diagrams.

Different workflow graphs can model the same behavior, i.e., the same control flow. For example, the two workflow graphs in Figs. 1(a) and 1(b) model the same behavior. The workflow graph in Fig. 1(b) is *well-structured* in the sense that it consists of matching pairs of a node that splits the flow and a node that joins the flow. Well-structured workflow graphs are often preferred because they are easier to comprehend [9, 2] and analyze [11], and can be represented as a regular expression.

The workflow graph G can be transformed into the well-structured workflow graph G^* using local transformation rules that preserve the execution semantics. While those rules are known [7, 8], it is not clear how to apply them to obtain G^* from G automatically. This is because a transformation rule can be applied to different parts of the

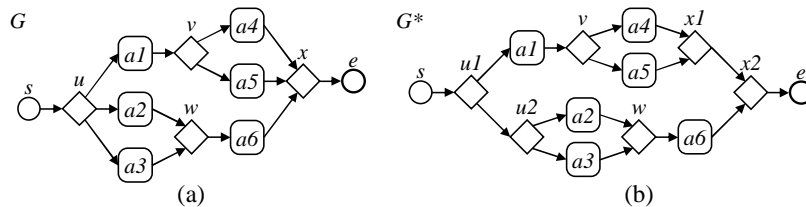


Fig. 1. (a) A workflow graph G , (b) A well-structured workflow graph G^* .

workflow graph, which can lead to different refactoring results. For example, applying the same rule that transforms G into G^* , we can transform G also into G' in Fig. 2. The transformation rule is given in Sect. 3.2. We are not aware of any work that specifies a desired refactoring result and proposes a concrete algorithm that computes it.

In this paper, we propose such a definition and such an algorithm. Given a workflow graph G , the algorithm computes a *normal form* G^* of G that makes the structure of G more explicit—and, as a side-product, making it more well-structured. Our approach is based on the *normal* [11] and the *refined process structure trees* [10] of a workflow graph.

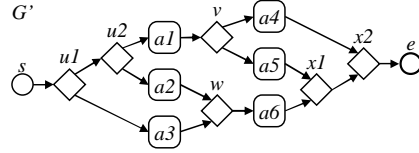


Fig. 2. Alternative transformation of G .

It is important that a workflow graph can be represented in an easily comprehensible form, as this makes it better accessible for users that may not be experienced in business process modeling. After all, workflow graphs are used for communicating business processes among different stakeholders, and not only among modeling experts. We hope that a workflow graph is, in general, easier to comprehend in our normal form than in its original form, as it is more well-structured.

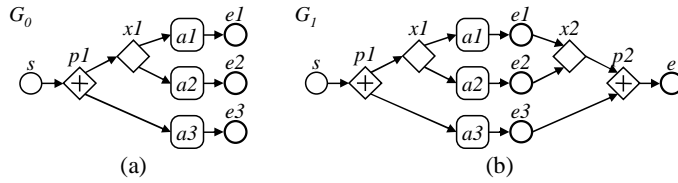


Fig. 3. (a) A workflow graph with multiple end nodes. (b) Its completion.

In the second part of the paper, we show how this refactoring technique can be used to compute a *completion* of a workflow graph. Figure 3 shows (a) a workflow graph with multiple end nodes and (b) its completion, which by definition has a unique end node. This has multiple use cases, which we discuss in Sect. 4. For example, it can be used to complete a partial workflow graph at modeling time [4], to provide local termination detection for workflow graphs with multiple ends, and to execute models containing OR-joins faster.

The refactoring technique efficiently computes a completion for many workflow graphs, but not for all. Sometimes a completion does not exist. We characterize these cases and also provide an algorithm that computes a completion in the general case when it exists. This algorithm is less efficient than the refactoring-based completion.

We review preliminary notions in Sect. 2. Then, we present our contributions: the refactoring technique in Sect. 3 and the completion technique in Sect. 4. Section 5 concludes this paper. We include short proofs of some theorems in this paper, whereas longer proofs of the other theorems can be found in a technical report [12].

2 Preliminaries

This section defines the basic preliminary notions of this paper, which include workflow graphs and their semantics, their extension by inclusive OR-gateways, and the soundness property for workflow graphs.

2.1 Workflow Graphs

A *workflow graph* G consists of a directed graph (V, E) , consisting of a set V of nodes, a set $E \subseteq V \times V$ of edges, and a partial mapping $\ell : V \rightarrow \{AND, XOR\}$ such that

1. $\ell(x)$ is defined if and only if x has more than one incoming edge or more than one outgoing edge,
2. there is exactly one source and at least one sink,
3. the source has exactly one outgoing edge and each sink has exactly one incoming edge, and
4. every node is on a path from the source to some sink.

The source is also called the *start node*, a sink is called an *end node*, $\ell(x)$ is called the *logic* of x . If the logic is AND or XOR, we call x a *gateway*; if x has no logic, then we call x a *task*. We use BPMN to depict workflow graphs, i.e., gateways are drawn as diamonds, where the symbol “+” inside stands for AND, whereas no decoration stands for XOR. Tasks are drawn as rectangles or circles. In particular, here start and end nodes are considered special tasks, which are always drawn as circles. A gateway that has more than one incoming edge and only one outgoing edge is also called a *join*, a gateway with more than one outgoing but only one incoming edge is also called a *split*. We say that an edge e is *incident* to a node n if e is incoming to n or outgoing from n .

The semantics of a workflow graph is, similarly to Petri nets, defined as a token game. A state of a workflow graph is represented by tokens on the edges of the graph. Let $G = (V, E, \ell)$ be a workflow graph. A *state* of G is a mapping $s : E \rightarrow \mathbb{N}$, which assigns a natural number to each edge. When $s(e) = k$, we say that edge e carries k *tokens* in state s . The semantics of the various nodes is defined as usual. An AND-gateway removes one token from each of its ingoing edges and adds one token to each of its outgoing edges. An XOR-gateway nondeterministically chooses one of its incoming edges on which there is at least one token, removes one token from that edge, then nondeterministically chooses one of its outgoing edges, and adds one token to that outgoing edge. As usual, we abstract from the data that controls the flow in XOR-gateways, hence the nondeterministic choice.

To be more precise, let s and s' be two states and x a node that is neither a start nor an end node. We write $s \xrightarrow{x} s'$ when s changes to s' by executing x . We have $s \xrightarrow{x} s'$ if

1. $\ell(x) = AND$ or the logic of x is undefined, and

$$s'(e) = \begin{cases} s(e) - 1 & e \text{ is an incoming edge of } x, \\ s(e) + 1 & e \text{ is an outgoing edge of } x, \\ s(e) & \text{otherwise.} \end{cases}$$

2. $\ell(x) = XOR$, and there exists an incoming edge e' and an outgoing edge e'' of x such that

$$s'(e) = \begin{cases} s(e) - 1 & e = e', \\ s(e) + 1 & e = e'', \\ s(e) & \text{otherwise.} \end{cases}$$

The *initial state* of G is the state where there is exactly one token on the unique outgoing edge of the start node and no token anywhere else. Node x is said to be *activated* in a state s if there exists a state s' such that $s \xrightarrow{x} s'$. A state s' is *reachable from* a state s , denoted $s \xrightarrow{*} s'$, if there exists a (possibly empty) finite sequence $s_0 \xrightarrow{x_1} s_1 \dots s_{k-1} \xrightarrow{x_k} s_k$ such that $s_0 = s$ and $s_k = s'$. A state is a *reachable state* of G if it is reachable from the initial state of G .

2.2 Inclusive OR-Gateways

A *generalized workflow graph* is a workflow graph in which a gateway x may also have OR-logic (*inclusive OR*), i.e., $\ell(x) = OR$. OR-gateways are drawn as diamonds with a circle inside. An OR-gateway has non-local join behavior, which is difficult to define if there is a cycle in the graph that contains an OR-join. As the semantics for the OR-join is not settled in that case, we do not consider that case. So, in the following we assume that a generalized workflow graph does not contain a cycle that contains an OR-gateway that has more than one incoming edge.

The OR-gateway behaves as follows. It is activated if for each incoming edge e' of the gateway that carries no token, and for each edge e'' of the graph that carries a token, there is no directed path from e'' to e' . When it executes, it consumes a token from each incoming edge that carries a token and produces a token for each edge of a nonempty subset of its outgoing edges. That subset is chosen nondeterministically. More precisely, we also have $s \xrightarrow{x} s'$ if

3. - $\ell(x) = OR$,
 - for each edge $e'' \in E$ and each incoming edge e' of x such that, $s(e'') \geq 1$ and $s(e') = 0$, there is no path from e'' to e' in the graph, and
 - there exists a nonempty set F of outgoing edges of x such that

$$s'(e) = \begin{cases} s(e) - 1 & e \text{ is an incoming edge of } x \text{ such that } s(e) \geq 1, \\ s(e) + 1 & e \in F, \\ s(e) & \text{otherwise.} \end{cases}$$

2.3 The Soundness Property

We now define what we understand by a “well-behaved” (generalized) workflow graph. A *final state* is a reachable state s of G that has no successor state, i.e., s activates no node. A final state is a *deadlock* if it contains a token on some edge that is not an incoming edge of an end node. (It follows that it must then be an incoming edge of a join.) A reachable state s contains a *lack of synchronization* if there is an edge e such

that $s(e) > 1$. A (generalized) workflow graph is *sound* if it contains neither a deadlock nor a lack of synchronization.

A deadlock is clearly undesired. Absence of lack of synchronization is a valuable design principle as it rules out that a task is executed concurrently to itself (“uncontrolled auto-concurrency”). Multiple concurrent instances of tasks can be modeled explicitly with dedicated constructs (e.g. “multiple instance activities” in BPEL and BPMN). Soundness is necessary for translating a (generalized) workflow graph to BPEL in a structured way, but has also an independent motivation: For workflow graphs with a unique end node and without OR-gateways, it is equivalent with the traditional definition of soundness (cf. [9, 11]).

3 Automatic Refactoring of Workflow Graphs

In this section, we present our technique that automatically refactors a workflow graph into a normal form that makes the structure of the workflow graph explicit. By structure we mean the decomposition of the workflow graph into logically atomic parts, which we call *fragments*. We use two alternative ways to do this decomposition, the *normal* [11] and the *refined process structure tree* [10]. We review the definitions of these process structure trees in Sect. 3.1. In Sect. 3.2, we present our novel refactoring technique.

3.1 The Refined and the Normal Process Structure Trees

Let $G = (V, E, \ell)$ be a (generalized) workflow graph. We assume that G has a unique end node. A detailed discussion on how to extend a workflow graph with multiple end nodes to a workflow graph with a unique end node is given in Sect. 4.

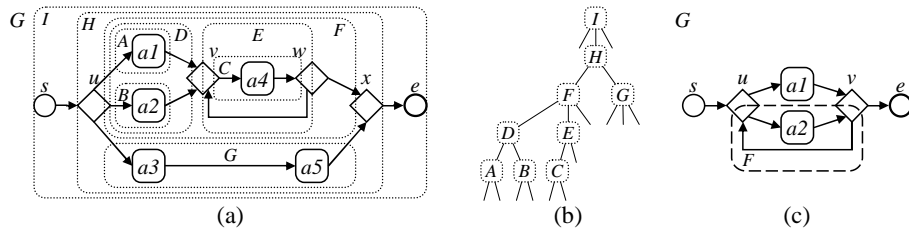


Fig. 4. (a) The canonical R-fragments of a workflow graph G . (b) The refined process structure tree of G . (c) F is not an R-fragment of G .

Figure 4(a) shows a workflow graph G and its decomposition into *canonical R-fragments*. An R-fragment is a connected subgraph with a unique entry and a unique exit node. The canonical R-fragments form a hierarchy, which can be represented as a tree. This tree, called the *refined process structure tree* (RPST), is shown in Fig. 4(b). These notions can be formally be defined as follows.

For a set $F \subseteq E$ of edges, let V_F denote the set of nodes that are incident to some edge in F and let $G_F = (V_F, F)$. We say G_F is *formed by* F .

Let $F \subseteq E$ be a set of edges such that G_F is a connected subgraph of G . A node $v \in V_F$ is a *boundary node* of F if it is the source or sink node of G , or if there exist edges $e \in F$ and $e' \in E \setminus F$ such that v is incident to e and e' . A boundary node v is an *entry* of F if no incoming edge of v is in F or if all outgoing edges of v are in F . A boundary node v is an *exit* of F if all incoming edges of v are in F or if no outgoing edge of v is in F . F is called an *R-fragment* of G if it has exactly two boundary nodes, an entry and an exit.

Figure 4(a) shows examples of R-fragments, which are indicated dotted boxes. They contain all those edges that are either inside the box or cross the boundary of the box. Thus, the box D denotes the R-fragment $D = \{(u, a1), (a1, v), (u, a2), (a2, v)\}$. Node u is the entry and v is the exit of D . $E = \{(v, a4), (a4, w), (w, v)\}$ is an R-fragment with entry v and exit w . In Fig. 4(c), $F = \{(u, a2), (a2, u), (v, u)\}$ has two boundary nodes, u and v , but neither of them is an entry or an exit of F . Thus, F is not an R-fragment.

An R-fragment F is *canonical* if it does not overlap with any other R-fragment, that is, for each R-fragment F' of G , we have $F \subseteq F'$ or $F' \subseteq F$ or $F \cap F' = \emptyset$. It follows that canonical R-fragments do not overlap with each other and hence form a hierarchy, which is represented as the *refined process structure tree* (RPST) of G . Note that each leaf node of the RPST represents an edge of the workflow graph, as each edge forms an R-fragment. The boundary nodes of this R-fragment are the two nodes that this edge connects. Vanhatalo, Völzer and Koehler [10] show how the RPST can be computed in linear time.

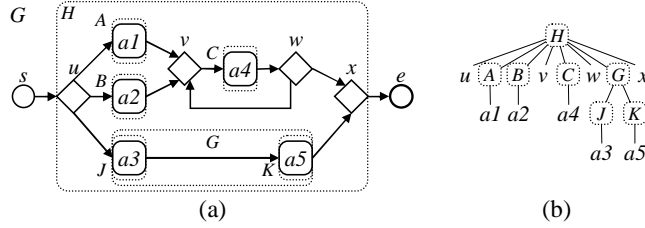


Fig. 5. (a) The canonical N-fragments of a workflow graph G . (b) The NPST of G .

Figure 5(a) shows an alternative decomposition of the same workflow graph into *N-fragments*. An N-fragment has unique entry and exit edges (as opposed to nodes). The corresponding tree, shown in Fig. 5(b), is called the *normal process structure tree* (NPST).

Let $G = (V, E, \ell)$ be a workflow graph. An *N-fragment* $G_F = (V', E')$ is a nonempty connected subgraph of G such that there exist edges $e, e' \in E$ with $E \cap ((V \setminus V') \times V') = \{e\}$ and $E \cap (V' \times (V \setminus V')) = \{e'\}$; e and e' are called the *entry* and the *exit* edge of G_F , respectively. An N-fragment is *canonical* if it does not overlap with any other N-fragment. The tree representing the hierarchy of canonical N-fragments is called the *normal process structure tree* (NPST). It can also be computed in linear time [5, 1, 11].

We define special kinds of R-fragments and N-fragments as follows. An R-fragment F is *trivial* if F has exactly one edge. The union of two R-fragments F, F' is an *R-*

sequence if the exit node u of F is the entry node of F' , and each edge incident to u is in $F \cup F'$. An N-fragment F is *trivial* if the entry and the exit edge of F are incident to the same node of F . An N-fragment F is an *N-sequence* if F is the union of two N-fragments F', F'' such that the exit edge of F' is the entry edge of F'' . An R-fragment (N-fragment) is *proper* if it is neither trivial nor an R-sequence (N-sequence). A node u is a *child* of a canonical N-fragment F if F is the smallest canonical N-fragment that contains u .

While the RPST exhibits more structure than the NPST, the NPST shows the structure more explicitly, in a less dense, more readable, form. The NPST produces a decomposition of the edges and nodes, defining a home fragment for each node and each edge—whereas the RPST produces a decomposition of edges only, while nodes may be shared between adjacent fragments. In the following, we show how to compute the normal form of a workflow graph G , which has the best of both worlds: It has all the structure of the RPST of G , but shows it explicitly in the more readable form of the NPST. The normal form is also more well-structured than the original workflow graph.

3.2 Refactoring Based on the RPST

Let G be a (generalized) workflow graph with a unique end node. We want to transform G , maintaining its structure given by its RPST but showing it more explicitly in form of N-fragments. Some R-fragments can be considered as N-fragments. We call them *normal*:

Definition 1 (Normal R-fragment, normal (generalized) workflow graph). A proper canonical R-fragment F is normal if exactly one edge outside F is incident to the entry of F and exactly one edge outside F is incident to the exit of F . A (generalized) workflow graph G is normal if every proper canonical R-fragment F is normal.

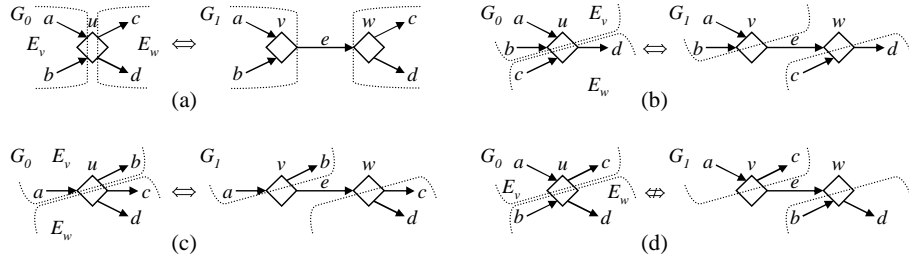


Fig. 6. (a), (b), (c) Valid expansions that split a node u into nodes v, w . (d) An invalid expansion.

Normal R-fragments can be obtained by splitting nodes. Figures 6(a)-(c) show three examples of a valid expansion (G_0, G_1) that splits a node u into two nodes v and w in a way that it preserves the execution semantics of the workflow graph. Subfigure (a) shows the splitting of a node into a join and a split. Subfigures (b) and (c) show a splitting that separates different inputs and outputs of the node respectively. The splitting

shown in subfigure (d) is invalid as the simultaneous separation of inputs and outputs does not preserve execution semantics. (The original path from b to c gets lost.) The three valid cases are instances of a single rule that splits a node into two nodes v, w :

Definition 2 (Valid expansion). Let $G_i = (V_i, E_i, \ell_i), i = 0, 1$ be two (generalized) workflow graphs. The pair (G_0, G_1) is a valid expansion if there exists two nodes $v, w \in V_1$ and a surjective mapping $\phi : V_1 \rightarrow V_0$ such that

- $(v, w) \in E_1$ and $(w, v) \notin E_1$,
- $(v, x) \in E_1$ and $(y, w) \in E_1 \Rightarrow x = w$ or $y = v$,
- $\phi(x) = \phi(y) \Leftrightarrow x = y$ or $\{x, y\} = \{v, w\}$,
- $(\phi(x), \phi(y)) \in E_0 \Leftrightarrow (x, y) \in E_1$ and $(x, y) \neq (v, w)$,
- $\ell_0(\phi(x)) = \ell_1(x)$, or $\ell_1(x)$ is undefined.

We define two parameters for a valid expansion: 1. the node $u = \phi(v) = \phi(w)$ and 2. a partition of the edges E_u that are incident to u into two sets $E_u = E_v \cup E_w$. We define

$$E_v = \{(u, \phi(y)) \mid (v, y) \in E_1, y \neq w\} \cup \{(\phi(x), u) \mid (x, v) \in E_1, x \neq w\} \quad (1)$$

E_w can be defined similarly or as $E_w = E_u \setminus E_v$.

Sadiq and Orlowska [8] have shown for workflow graphs that if (G_0, G_1) is a valid expansion that splits a node u into nodes v and w , and the logic of u is AND or XOR, then G_0 and G_1 have the same behavior. An analogous result is also known from Petri nets [7]. The behavior is also the same if the logic of u is OR and u is not in a cycle.

Some valid expansions create redundant nodes that we want to exclude from our refactoring technique. We call these *undesired expansions*. A *desired expansion* splits a gateway into two gateways, whereas an undesired expansion creates at least one task. Figure 7 shows two examples of undesired expansions. Each of these valid expansions is undesired, because node w has only one incoming and only one outgoing edge.

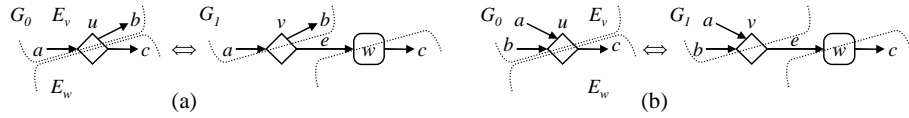


Fig. 7. Two examples of undesired expansions that split node u into nodes v and w .

Definition 3 (Desired expansion, undesired expansion). Let $G_i = (V_i, E_i, \ell_i), i = 0, 1$ be two (generalized) workflow graphs and $v, w \in V_1$ be distinct nodes such that (G_0, G_1) is a valid expansion and $\phi(v) = \phi(w)$. The pair (G_0, G_1) is a desired expansion if

- v has at least two incoming edges or at least two outgoing edges, and
- w has at least two incoming edges or at least two outgoing edges.

Otherwise (G_0, G_1) is an undesired expansion.

The valid expansions in Fig. 6 are also desired expansions. Desired expansions restrict the application of node splitting. However, application of the desired expansion can still lead to different results, as shown by the example in Sect. 1. As we want to maintain the structure of the graph, we apply the expansion only based on the R-fragments. This will lead to a unique result.

Definition 4 (*F*-based expansion of node u). Let (G_0, G_1) be a desired expansion with parameters u, E_v and E_w as in Def. 2 and let F be an R-fragment of G_0 . Furthermore, let E_u denote the set of edges that are incident to u . We say that the expansion is *F*-based if either u is the entry of F and $E_w = F \cap E_u$, or u is the exit of F and $E_v = F \cap E_u$.

The fragment-based expansions can be applied repeatedly until we obtain a normal generalized workflow graph.

Definition 5 (Expansion). Let G_0 and G_n be (generalized) workflow graphs. G_n is an expansion of G_0 if there is a sequence G_0, \dots, G_n of (generalized) workflow graphs such that (G_i, G_{i+1}) is a fragment-based expansion, when $0 \leq i < n$.

Although a given workflow graph G allows different sequences of fragment-based expansions, we nevertheless obtain a unique result, which we call the *expanded normal form* of G , denoted as G^* .

Theorem 1. For every (generalized) workflow graph G , there exists a unique (generalized) workflow graph G^* such that G^* is an expansion of G and G^* is normal.

Proof. Each *F*-based expansion replaces a fragment that is not normal by a sequence of a trivial fragment and F . It is a local change to the original generalized workflow graph G that preserves the structure of the RPST. This feature, which was named *modularity* of the RPST, was proved earlier [10]. In fact, if we do not consider trivial fragments and sequences, the RPST stays the same after the expansion. Because expansions can be considered as modular replacements that do not change the RPST up to trivial fragments and sequences, and because fragments are either nested or disjoint, then all fragment-based expansions that can be applied to the original graph G are mutually independent, i.e. they can be applied in any order to obtain the same result. (Note however, that a particular expansion can be based on different fragments.)

Furthermore, an *F*-based expansion can only be applied if either the entry or the exit of F violates the normality constraint and if F is neither trivial nor a sequence. An *F*-based expansion removes such a violation of a normality constraint and cannot introduce a new one. It follows that we arrive at a unique normal generalized workflow graph after any sequence of all the expansions that are possible in the original graph G . \square

It is clear from the proof of Thm. 1 that the normal form G^* arises as the maximal expansion of G . Therefore, Alg. 1 computes G^* . It can be implemented in linear time to the number of edges of G .

Theorem 2. Let G be a (generalized) workflow graph. Algorithm 1 computes the *expanded normal form* of G .

Algorithm 1 Computes the expanded normal form of a generalized workflow graph G .

computeExpandedNormalForm(G)

 Compute the RPST T of G , and a list L of proper canonical R-fragments that are not normal.

while L has a proper canonical R-fragment F such that F is not normal **do**

 Transform G into G' based on an F -based expansion (G, G').

 Locally update T to correspond to the RPST of this updated G .

if F is normal, **then** remove F from L .

return G

Next we show that the NPST and the RPST coincide if G is normal. As R-fragments and N-fragments are different objects, there are subtle differences in the trivial fragments. However, the main structure that is represented by the proper fragments coincides:

Theorem 3. *Let G be a normal (generalized) workflow graph, F a set of edges and G_F the subgraph formed by F . F is a proper canonical R-fragment of G if and only if G_F is a proper canonical N-fragment of G .*

Proof. Presented in a technical report [12].

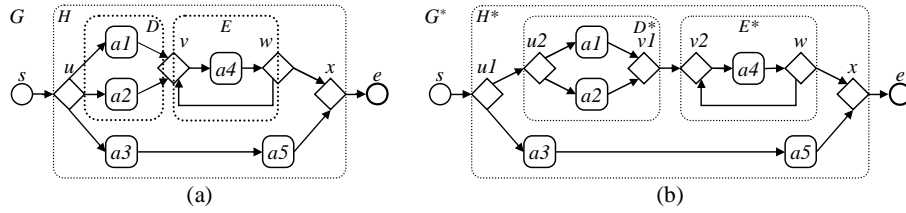


Fig. 8. (a) A workflow graph G . (b) The expanded normal form G^* of G .

Figure 8(a) shows a workflow graph G and its proper canonical R-fragments. The R-fragment H is normal, whereas D and E are not. Thus, we transform G into its extended normal form G^* shown in Fig. 8(b) through D -based and E -based expansions. It is possible to transform G with D -based expansion of u and v , and an E -based expansion of v . The D -based and E -based expansions of v split v into nodes $v1$ and $v2$ the same way. Thus, we can obtain G^* from G through two D -based expansions, or through one E -based and one D -based expansion. The proper canonical R-fragments of G^* are also the proper canonical N-fragments of G^* . G^* has two more proper canonical N-fragments (D^* and E^*) than G .

An N-fragment F is *well-structured* (c.f. [6, 11]) if F is trivial, an N-sequence, or if F has exactly two gateways as children, a split and a join j , that have the same logic and if the entry edge of F is incoming to j , then j is an XOR-join. Note that a well-structured fragment cannot be further refactored. However, if a fragment is not well-structured it may become well-structured as shown in the examples of Figs. 1 and 8. In

this sense, the normal form of G is “more well-structured” than G . Note that we only use local refactorings. It is known [6] that some workflow graphs can only be transformed into well-structured graphs through non-local transformations, whereas there are also workflow graphs that have no well-structured equivalent.

4 Automatic Completion of Workflow Graphs

In this section, we show how the refactoring technique of Sect. 3 can be used to compute a *completion* of a workflow graph.

Definition 6 (Completion). Let $G = (V, E, \ell)$ be a sound workflow graph (sound generalized workflow graph) and $G' = (V', E', \ell')$ a workflow graph (generalized workflow graph). G' is called a completion of G if

1. G is a subgraph of G' such that if an edge $e \in E' \setminus E$ is incident to some node in $x \in V$, then x is an end node of G and e is outgoing from x ,
2. G' has a unique end node,
3. G' is sound.

A completion of a generalized workflow graph is easy to construct. We just add an OR-join j and an end node e , and connect each original end node to j and j to e . Figure 9 illustrates this.

Proposition 1. The construction shown in Fig. 9 defines a completion of a generalized workflow graph.

Proof. We have to prove soundness of G' . We claim that the final OR-join can fire only if G has no more tokens. Suppose the contrary. Then there is a state s that activates the OR-join and there is some token inside G , say on edge e . As G has no deadlock, we can move the token from e to some end node of G . As G has no lack of synchronization, this end node was unmarked in s . It follows that in state s , there is a path from some token to an unmarked incoming edge of the OR-join. It follows that the OR-join is not activated in s , contradicting our supposition. The claim of the proposition follows now directly. \square

This simple completion can be used to compute a translation from generalized workflow graphs (e.g. BPMN diagrams) with multiple end nodes to BPEL using the refined process structure tree [10], which needs a generalized workflow graph with a unique start¹ and a unique end node as input. However, there are various use cases where we want a completion without using OR-gateways:

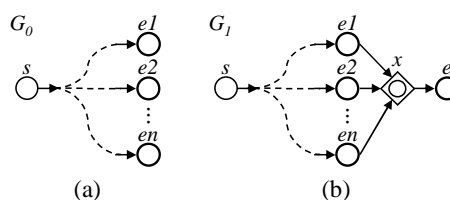


Fig. 9. G_1 is the simple completion of G_0 .

¹ In most languages, multiple start nodes stand for either an implicit AND-split or an implicit XOR-split. Therefore, we restrict to workflow graphs having exactly one start node.

- A user modeling a process has drawn a part of a diagram where she opened a number of parallel and alternative paths, i.e., she used a combination of AND-splits and XOR-splits, possibly also some AND-joins or XOR-joins. Now the user does not know what logic, i.e., combination of AND-joins and XOR-joins to use to close a given set of paths correctly. In that situation, an OR-join could also be used to close these paths. However, the OR-join may not be available in the language chosen or may be considered too expensive to execute (see next use case). Gschwind et al. [4] present this use case described above, but do not provide a technical solution.
- An OR-join was used to close a particular set of paths. In our restricted setting, the evaluation of the OR-join at runtime requires that the whole graph preceding the OR-join be checked for tokens. If we compute a completion of the graph preceding the OR-join, we can replace the OR-join with a combination of merges and joins, all of which can be executed locally. Removing OR-joins also allows us to translate the generalized workflow graph to a Petri net in a simple way [9]. (Petri net transitions have a local semantics.) A translation to Petri nets may not only be useful as an intermediate language between two different business process specification languages, but also to apply some of the various analysis techniques and tools available for Petri nets.
- Dead Path Elimination of BPEL [3] is a way to simulate the OR-join by gateways that can be executed locally. There, *dead tokens* are sent along those paths that are not taken. An OR-join can then wait for a token, dead or alive, on each incoming edge before it executes. If we can replace OR-joins by merges and joins, dead path elimination can be switched off, saving the overhead of sending, propagating and synchronizing dead tokens along potentially long paths.
- Checking whether a process has terminated also requires checking the entire generalized workflow graph for tokens. A sound generalized workflow graph with a unique end node will, however, signal termination through a single token on the unique end node. Thus, a completion at compile time provides a local termination detection at runtime.

In the general case, a completion may be difficult to compute or not even exist. In the following section, we show how to efficiently compute a completion based on the refactoring technique from Sect. 3 for many cases. In Sect. 4.2, we characterize for which workflow graphs a completion exists. Finally, in Sect. 4.3, we discuss how to compute a completion in the general case where it exists.

4.1 Completion by Refactoring

Let G be a sound generalized workflow graph with multiple end nodes. We first complete G using the simple OR-join completion described above. This adds a unique end node, which allows us to compute the RPST for the completed graph. We obtain one or more proper canonical R-fragments that contain the added OR-join of G as exit. Such an R-fragment is called an *end fragment* of G . Figures 10(a) & (b) show an example of a workflow graph G_0 and its simple completion G_1 that has two end fragments A and B .

We then split the final OR-join into several OR-joins, one per end fragment according to the refactoring technique presented in Sect. 3.2. This preserves behavior as stated

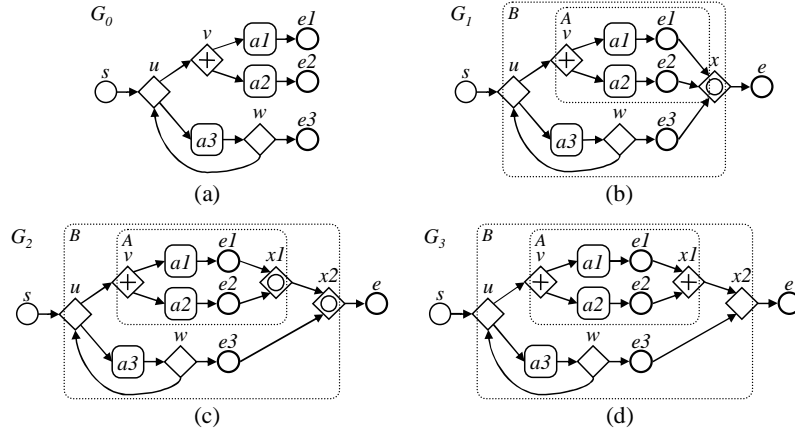


Fig. 10. Example of a completion by refactoring. The refactoring step produces G_2 from G_1 .

in Sect. 3.2 and therefore also soundness. We refactor the end fragments until they are normal, and thus also N-fragments. Figure 10(c) shows an example of the resulting expansion G_2 of G_1 . Now we can replace an OR-join j by an XOR-join if the fragment F of j is *sequential*, that is, if F has no AND-split and no OR-split as a child. On the other hand, an OR-join j can be replaced by an AND-join if F of j is *deterministic*, that is, F has no XOR-split and no OR-split as a child. This replacement preserves soundness.

Theorem 4. *Let F be an N-fragment of a sound generalized workflow graph G , and an OR-join j be a child of F .*

1. *If F has neither an XOR-split nor an OR-split as children and j is replaced with an AND-join, then the resulting generalized workflow graph G' is sound.*
2. *If F has neither an AND-split nor an OR-split as children and j is replaced with an XOR-join then, the resulting generalized workflow graph G' is sound.*

Proof. Presented in a technical report [12].

Applying this theorem repeatedly allows us to replace all OR-joins in such a fragment F with either 1) AND-joins or 2) XOR-joins. Figure 10(d) shows an example of a completion G_3 of G_0 obtained with our technique. Note that this completion technique requires only linear time. After a review of more than 150 sound workflow graphs created from realistic business processes, we believe that in practice most workflow graphs can be completed using this fast technique. The power of this technique stems from the fact that it abstracts from the interior of subfragments. For example, fragment B in Fig. 10(c) is sequential, although it contains a subfragment with concurrency.

Figure 11(a) shows a workflow graph that can be completed, but not with the fast technique presented here. We discuss these cases in the next section. It is clear that this technique can also replace OR-joins in the middle of a generalized workflow graph, provided that we restrict its application to settings where an OR-join is not in a cycle.

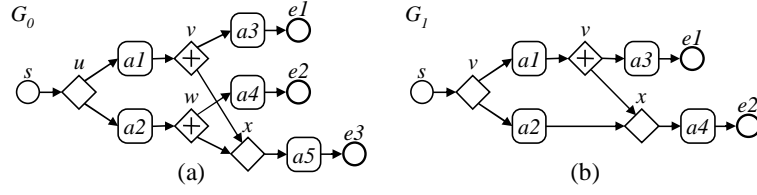


Fig. 11. Two workflow graphs. (a) G_0 has a completion and (b) G_1 has no completion.

4.2 Existence of Completions

In this section, we present a technique that computes a completion for each workflow graph that has a completion, and we characterize for which workflow graphs a completion exists. Let G be a sound workflow graph. As G is sound, we can represent a final state as a set of end nodes. Let \mathcal{F} be the set of final states of G , presented in this way.

Definition 7. Two distinct end nodes x, y are mutually exclusive if there is no final state $M \in \mathcal{F}$ such that $x \in M$ and $y \in M$. A test is a set T of pairwise mutually exclusive end nodes such that for each final state M , $M \cap T \neq \emptyset$.

Figure 11(a) shows a workflow graph with final states $\mathcal{F} = \{\{e1, e3\}, \{e2, e3\}\}$. There are two tests, $T_1 = \{e1, e2\}$ and $T_2 = \{e3\}$. Figure 11(b) shows a workflow graph with final states $\mathcal{F} = \{\{e1, e2\}, \{e2\}\}$. There is only one test $T_3 = \{e2\}$.

Theorem 5. A sound workflow graph has a completion if and only if each end node belongs to a test.

Proof. Presented in a technical report [12].

This completion can be constructed as shown in Fig. 12. We extend a workflow graph G by creating an XOR-join for each test of G and one final AND-join that is connected to the unique end node. Each end node x is connected to those XOR-joins that correspond to a test T such that $x \in T$. If an end node has to be connected to more than one XOR-join, an AND-split is inserted after the end node. All XOR-joins are connected to the final AND-join.

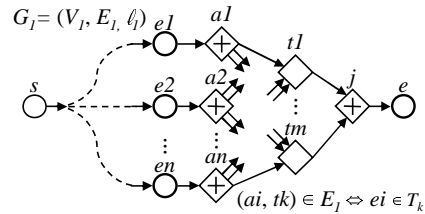


Fig. 12. General completion.

Figure 13(a) shows the completion of G_0 from Fig. 11. Note that gateways of completion that would have only a single incoming and a single outgoing edge can be omitted. Node y corresponds to the test $\{e1, e2\}$ whereas the node created by the test $\{e3\}$ was omitted. Also no AND-split was necessary.

G_1 in Fig. 11(b) does not have a completion because $e1$ is not in any test. Note that it is impossible to obtain a completion if there exist two final states M and M' such that $M \subseteq M'$. Figure 13(b) shows that it is nevertheless possible to “complete” G_1 in a more liberal sense; AND-split w added in the middle of G_1 allows us to complete G_1 .

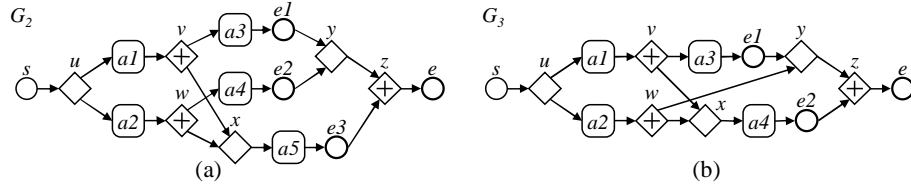


Fig. 13. (a) Completion of G_0 from Fig. 11(a). (b) Completion of G_1 from Fig. 11(b) after adding AND-split w in the middle of the graph.

4.3 Computation of the Completion in the General Case

The completion technique above requires computing the final states of the workflow graph, of which there can be exponentially many. The computation can be done by a simple state-space exploration. However, using the refactoring in Sect. 3.2 first, allows us to restrict the completion to single end fragments, that is, we also have to compute the state space only for individual end fragments, which are typically much smaller. We transformed our library of more than 150 industrial process models into workflow graphs having, on average, 57 edges. The largest has 203 edges and more than 100,000 states. However, those fragments that are neither sequential nor deterministic have at most 38 states.

Once the final states have been computed, a suitable set of tests can be computed. Sometimes a simple set of tests exists that can be computed in a simple way: An end node x is called an *identifier* of a final state M if x is contained in M but not in any other final state. Assume every state M has an identifier x_M . Let y be an end node. Then $T_y = \{x_M \mid M \in \mathcal{F}, y \notin M\} \cup \{y\}$ is a test containing y . So, in that special case, which is easy to check, we get a simple set of tests that suffices.

5 Conclusion

This paper presents two new techniques for generalized workflow graphs—an RPST-based refactoring technique that renders their structure more explicit, and a completion technique that builds on this refactoring technique. We also provide a characterization of workflow graphs that have a completion.

Only few related papers on workflow graph refactoring and completion exist and these papers, having different focus, are only loosely related. Sadiq and Orlowska [8] transform a workflow graph preserving its behavior to analyze its soundness. Eder et al. [2] define equivalence of workflow graphs through rewriting rules. Zhang and D'Hollander [13] use hammocks to structure flow graphs of sequential programs, which can all be made well-structured in the absence of concurrency. A hammock is a special case of an R-fragment, but their technique to restructure hammocks differs from ours. Well-structuredness makes workflow graphs more readable [9, 2]. Gschwind et al. [4] identify a use case for a completion technique, but do not provide a solution to solve it.

We believe that our refactoring-based completion technique is also useful for removing OR-joins that occur in cycles. Figure 14 shows an example. Note that the

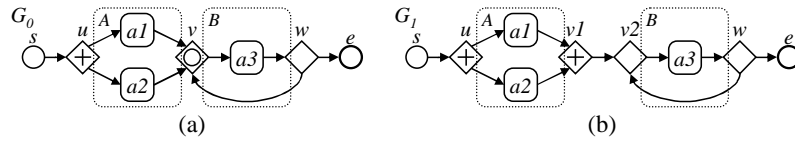


Fig. 14. Separating sequential cyclic fragments from concurrent fragments.

refactoring produces a well-structured workflow graph. A formal treatment of these examples would, however, exceed the scope of this paper as it requires a semantics for OR-joins in cycles. In EPCs, our refactoring-based technique can also provide a completion in the beginning of a graph, which describes the start node combinations that lead into a sound execution.

Acknowledgments The work published in this article was partially supported by the SUPER project (<http://www.ip-super.org/>) under the EU 6th Framework Programme Information Society Technologies Objective (contract no. FP6-026850).

References

1. C. Scott Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, September 1999.
2. Johann Eder, Wolfgang Gruber, and Horst Pichler. Transforming workflow graphs. In *INTEROP-ESA 2005*, pages 203–214, 2005.
3. Alexandre Alves et al. *Web Services Business Process Execution Language Version 2.0*. OASIS Org., 2007.
4. Thomas Gschwind, Jana Koehler, and Janette Wong. Applying patterns during business process modeling. In *BPM 2008*, volume 5240 of *LNCS*, pages 4–19. Springer, Heidelberg, 2008.
5. Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *PLDI 1994*, pages 171–185. ACM, New York, 1994.
6. Bartek Kiepuszewski, Arthur H. M. ter Hofstede, and Christoph Bussler. On structured workflow modelling. In *CAiSE*, pages 431–445. Springer, Heidelberg, 2000.
7. Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
8. Wasim Sadiq and Maria E. Orlowska. Analyzing process models using graph reduction techniques. *Inf. Syst.*, 25(2):117–134, 2000.
9. Wil M. P. van der Aalst, Alexander Hirnschall, and H. M. W. Verbeek. An alternative way to analyze workflow graphs. In *CAiSE*, volume 2348 of *LNCS*, pages 535–552. Springer, 2002.
10. Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The refined process structure tree. In *BPM 2008*, volume 5240 of *LNCS*, pages 100–115. Springer, Heidelberg, 2008.
11. Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and more focused control-flow analysis for business process models through SESE decomposition. In *ICSOC 2007*, volume 4749 of *LNCS*, pages 43–55. Springer, Heidelberg, 2007.
12. Jussi Vanhatalo, Hagen Völzer, Frank Leymann, and Simon Moser. Automatic workflow graph refactoring and completion. IBM Research Report RZ 3715, 2008.
13. Fubo Zhang and Erik H. D'Hollander. Using hammock graphs to structure programs. *IEEE Trans. Softw. Eng.*, 30(4):231–245, 2004.