

The Refined Process Structure Tree

Jussi Vanhatalo^{1,2}, Hagen Völzer¹, and Jana Koehler¹

¹ IBM Zurich Research Laboratory, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland
{juv,hvo,koe}@zurich.ibm.com

² Institute of Architecture of Application Systems, University of Stuttgart, Germany

Abstract. We consider workflow graphs as a model for the control flow of a business process model and study the problem of *workflow graph parsing*, i.e., finding the structure of a workflow graph. More precisely, we want to find a decomposition of a workflow graph into a hierarchy of sub-workflows that are subgraphs with a single entry and a single exit of control. Such a decomposition is the crucial step, for example, to translate a process modeled in a graph-based language such as BPMN into a process modeled in a block-based language such as BPEL. For this and other applications, it is desirable that the decomposition be unique, *modular* and as fine as possible, where *modular* means that a local change of the workflow graph can only cause a local change of the decomposition. In this paper, we provide a decomposition that is unique, modular and finer than in previous work. It is based on and extends similar work for sequential programs by Tarjan and Valdes [11]. We show that our decomposition can be computed in linear time based on an algorithm by Hopcroft and Tarjan [3] that finds the triconnected components of a biconnected graph.

Keywords. Workflow graph parsing, Control flow, Model decomposition, BPMN to BPEL translation/ roundtripping, Subprocess detection, Graph theory

1 Introduction

The control flow of a business process can often be modeled as a *workflow graph* [10]. Workflow graphs capture the core of many business process languages such as UML activity diagrams, BPMN and EPCs. We study the problem of *parsing* a workflow graph, that is, decomposing the workflow graph into a hierarchy of sub-workflows that have a single entry and a single exit of control, often also called *blocks*, and labeling these blocks with a syntactical category they belong to. Such categories are *sequence*, *if*, *repeat-until*, etc., see Fig. 1(a). Such a decomposition is also called a *parse* of the workflow graph. It can also be shown as a *parse tree*, see Fig. 1(c).

The parsing problem occurs when we want to translate a graph-based process description (e.g. a BPMN diagram) into a block-based process description (e.g. BPEL process), but there are also other use cases for workflow graph parsing. For example, Vanhatalo, Völzer and Leymann [14] show how parsing speeds up control-flow analysis. Küster et al. [6] show how differences between two process models can be detected and resolved based on decompositions of these process models. We believe that parsing also helps in understanding large processes and in finding reusable subprocesses.

For a roundtripping between a BPMN diagram and a BPEL process, it is desirable that the decomposition be unique, i.e., the same BPMN diagram always translates to

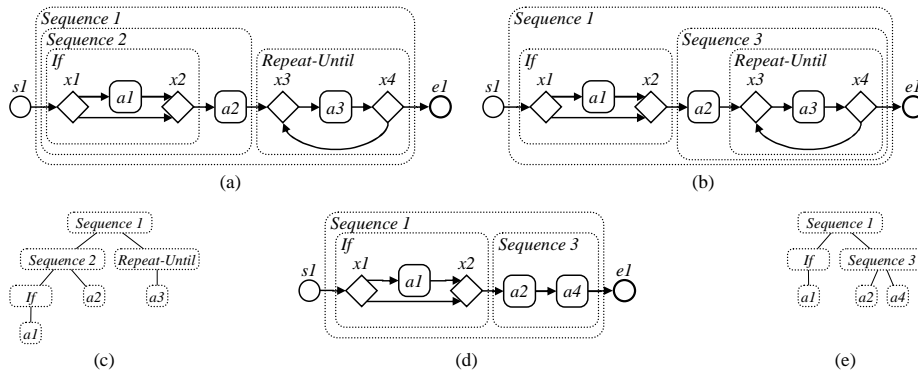


Fig. 1. (a), (b) Two parses of the same workflow graph. (c) Parse tree corresponding to (a). (d) Workflow graph obtained by a local change and its parse. (e) Parse tree corresponding to (d).

the same BPEL process. Consider, for example, the workflow graph in Fig. 1(a). The translation algorithm proposed by Ouyang et al. [9] is nondeterministic. It may produce one of the two parses shown in Fig. 1(a) and (b), depending on whether the if-block or the repeat-until-block is found first by the parsing algorithm.

One idea to resolve some of this nondeterminism is to define priorities on the syntactic categories to be found [9, 7, 8]. For example, if in each step the parsing algorithm tries to find sequences first, then if-blocks and then repeat-until-blocks, we can only obtain the parse in Fig. 1(a) in our example. However, this can introduce another problem. If we change a single block, say, the repeat-until block by replacing it, e.g. by a single task, we obtain the workflow graph shown in Fig. 1(d). Fig. 1(d) also shows the parse we obtain with the particular priorities mentioned above. The corresponding parse tree is shown in Fig. 1(e). It cannot be derived from the tree in Fig. 1(c) by just a local change, viz., by replacing the Repeat-Until subtree. For a roundtripping between a BPMN diagram and a BPEL process, it would be much more desirable that a local change in the BPMN diagram also result in only a local change in the BPEL process. Replacing a block in the BPMN diagram would therefore only require replacing the corresponding block in the BPEL process. We then call a such decomposition *modular*. The existing approach to the BPMN to BPEL translation problem [9] is not modular. Furthermore, it does not provide, because of the above problems, a specification of the translation that is independent of the actual translation algorithm.

A unique and modular decomposition is provided by the *program structure tree* defined by Johnson et al. [4, 5] for sequential programs. It was applied to workflow graphs by Vanhatalo et al. [14] to find control-flow errors. The corresponding decomposition for our first example is shown in Fig. 2. It uses the same notion of a block as Ouyang et al. [9] do, that is, a block is

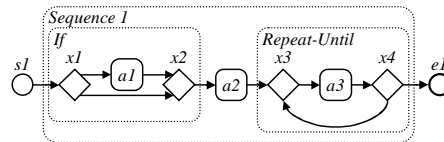


Fig. 2. Modular decomposition of the process from Fig. 1.

a connected subgraph with a single entry and a single exit edge. But in contrast to the approach of Ouyang et al. [9], non-maximal sequences are disregarded in the program structure tree. For example, Sequence 2 in Fig. 1(a) [likewise Sequence 3 in subfigure (b)] is non-maximal: it is in sequence with another block.

Another general requirement for parsing is to find as much structure as possible, i.e., to decompose into blocks that are as fine as possible. As we will see (cf. Sect. 4), this allows us to map more BPMN diagrams to BPEL in a structured way. It has also been argued [9] that the BPEL process is more readable if it contains more blocks. Furthermore, debugging is easier when an error is local to a small block rather than to a large one.

In this paper, we provide a new decomposition that is finer than the program structure tree as defined by Johnson et al. [4, 5]. It is based on and extends similar work for sequential programs by Tarjan and Valdes [11]. The underlying notion of a block is a connected subgraph with unique entry and exit *nodes* (as opposed to *edges* in the previous approach). Accordingly, all blocks of the previous approach are found, but more may be found, resulting in a more refined parse tree. We prove that our decomposition is unique and modular. Moreover, we show that it can be computed in linear time, based on an algorithm by Hopcroft and Tarjan [3] that finds the triconnected components of a biconnected graph.

The paper is structured as follows. In Sect. 2, we define the refined process structure tree and discuss its main properties. In Sect. 3, we describe how to compute the process structure tree in linear time. Proofs of the main theorems can be found in a technical report [13].

2 The Refined Process Structure Tree

In this section, we define the refined process structure tree (PST, for short). First, we explain our notion of fragments in Subsection 2.1. Fragments have a strong relationship with the *triconnected components* of the workflow graph, which we explain in Subsection 2.2. Subsection 2.3 defines the process structure tree. Finally, we show that our decomposition is modular.

2.1 Fragments

We start by recalling some basic notions of graph theory. A *multi-graph* is a graph in which two nodes may be connected by more than one edge. This can be formalized as a triple $G = (V, E, M)$, where V is the set of nodes, E the set of edges and M a mapping that assigns each edge an ordered or unordered pair of nodes—for a directed or undirected multi-graph, respectively. We will use multi-graphs throughout the paper, directed and undirected, but will call them graphs for simplicity.

Let G be a graph. If e is an edge of G that connects two nodes u and v , we also say that u and v are *incident* to e , e is *incident* to u and v , and nodes u and v are *adjacent*.

Workflow graphs are based on *two-terminal graphs*³. A *two-terminal graph* (TTG for short) is a directed graph G without self-loops such that there is a unique source

³ A workflow graph is a two-terminal graph in which each node is labeled with some control flow logic such as AND, OR, etc.

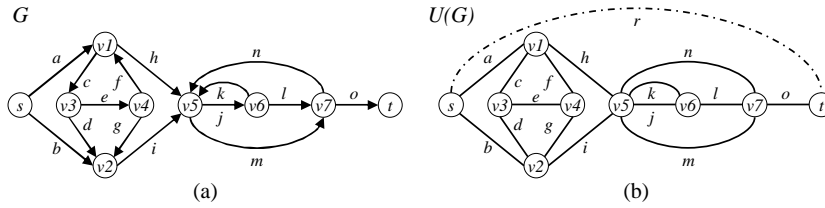


Fig. 3. (a) Two-terminal graph G , and (b) its undirected version $U(G)$, where r is the return edge.

node s and a unique sink node $t \neq s$ and each node v is on a directed path from s to t . The *undirected version* of G , denoted $U(G)$, is the undirected graph that results from ignoring the direction of all the edges of G and adding an additional edge between the source and the sink. The additional edge is called the *return edge* of $U(G)$. Figure 3 shows examples of (a) a two-terminal graph G , and (b) its undirected version $U(G)$, where r is the return edge.

For a subset F of edges, let V_F denote the set of nodes that are incident to some edge in F and let G_F denote the subgraph with nodes V_F and edges F . We say that G_F is *formed by* F .

Let G be a TTG and F a subset of its 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 G has edges $e \in F$ and $e' \notin F$ 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 a *fragment* of G if it has exactly two boundary nodes, an entry and an exit. Let $\mathcal{F}(u, v)$ denote the set of all fragments with entry u and exit v .

Figure 4 shows examples of fragments. A fragment is indicated as a dotted box. It contains all those edges that either are inside the box or cross the boundary of the box. Thus, the box in subfigure (a) denotes the fragment $F1 = \{a, b, c\}$. Node u is the entry and v is the exit of $F1$. In subfigure (b), $F2 = \{a, b, c, d\}$ is a fragment with entry u and exit v . In subfigure (c), $F3 = \{a, b, c, d\}$ has two boundary nodes, u and v , neither of them is an entry or an exit of $F3$. Therefore, $F3$ is not a fragment.

Note that it can be checked locally whether a boundary node is an entry or an exit. This notion of fragment was proposed by Tarjan and Valdes [11], where a TTG modeled the control flow of a sequential program. When control flows through any of the edges of a fragment, then it must have flown through the entry before and must flow through

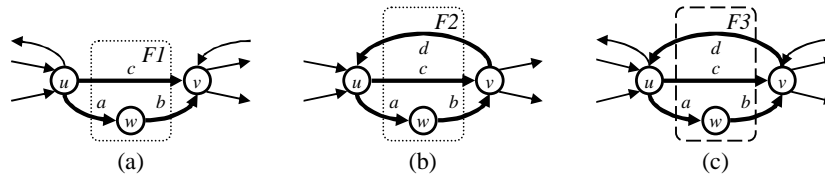


Fig. 4. (a), (b) Examples and (c) counterexamples of entry, exit and fragment.

the exit after. Their notion of fragment is, in a sense, the most general notion of fragment having this property that can still be verified locally [11].

2.2 Triconnected Components

Tarjan and Valdes [11] have shown that the fragments of a TTG are closely related to the *triconnected components* of its undirected version. We have to introduce a few more notions from graph theory.

Let G be an undirected graph. The following notions are also used for directed graphs by ignoring the direction of their edges. Let W be a subset of nodes of G . Two nodes $u, v \notin W$ are *connected without W* if there is a path that contains u and v but no node $w \in W$. For instance, in Fig. 3(a) nodes s and t are connected without $v6$, but not connected without $v5$. Two edges e, f are *connected without W* if there exists a path containing e and f in which a node $w \in W$ may only occur as the first or last element. A graph without self-loops is *k -connected*, $k > 0$, if it has at least $k + 1$ nodes and for every set W of $k - 1$ nodes, any two nodes $u, v \notin W$ are connected without W . We say *connected* for 1-connected, *biconnected* for 2-connected and *triconnected* for 3-connected. A *separation point* (*separation pair*) of G is a node u (pair $\{u, v\}$ of nodes) such that there exists two nodes that are not connected without $\{u\}$ (without $\{u, v\}$). Therefore a graph is biconnected (triconnected) if and only if it has no separation point (separation pair). For instance in Fig. 3, G is not biconnected, because $v5$ is a separation point, whereas $U(G)$ is biconnected, because it has no separation points. $U(G)$ is not triconnected, because $\{v5, v7\}$ is a separation pair. In Fig. 5(a), $T2$ is an example of a triconnected graph if the dashed edge x is considered as an ordinary edge.

We say that a graph is *weakly biconnected* if it is biconnected or if it contains exactly two nodes and at least two edges between these two nodes. For instance, in Fig. 5(a), $B1$ is weakly biconnected, but not biconnected.

Throughout the paper, we assume that $U(G)$ is weakly biconnected. This can easily be achieved by splitting each separation point into two nodes, where the only outgoing edge of the first node is the only incoming edge of the second node.⁴

Let $\{u, v\}$ be a pair of nodes. A *separation class* with respect to (w.r.t.) $\{u, v\}$ is a maximal set S of edges such that any pair of edges in S is connected without $\{u, v\}$; S is a *proper separation class* or *branch* if it does not contain the return edge; $\{u, v\}$ is called a *boundary pair* if there are at least two separation classes w.r.t. $\{u, v\}$. Note that a pair $\{u, v\}$ of nodes is a boundary pair if and only if it is a separation pair or u and v are adjacent in G . For instance in Fig. 3(b), $\{v5, v7\}$ and $\{v6, v7\}$ are boundary pairs. The pair $\{v5, v7\}$ is also a separation pair, but $\{v6, v7\}$ is not.

Each weakly biconnected graph can be uniquely decomposed into a set of graphs, called its *triconnected components* [3], where each triconnected component is either a *bond*, a *polygon* or a triconnected graph. A *bond* is a graph that contains exactly two nodes and at least two edges between them. A *polygon* is a graph that contains at least three nodes, exactly as many edges as nodes such that there is a cycle that contains all its nodes and all its edges.

⁴ It is often assumed for workflow graphs that no node has both multiple incoming and multiple outgoing edges. In that case it follows that $U(G)$ is biconnected. See also Sect. 4.

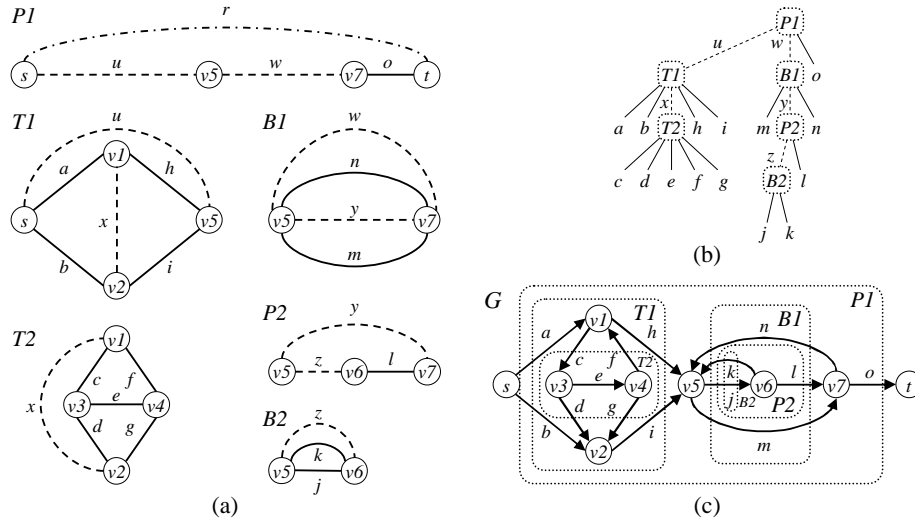


Fig. 5. (a) The triconnected components of $U(G)$ in Fig. 3, (b) the tree of the triconnected components of $U(G)$, and (c) the corresponding component subgraphs of G .

Part (a) in Fig. 5 shows the six triconnected components of graph $U(G)$ from Fig. 3. $P1$ and $P2$ are polygons, $B1$ and $B2$ are bonds, and $T1$ and $T2$ are triconnected graphs. Each component contains *virtual edges* (shown as dashed lines), which are not contained in the original graph. They contain the information on how the components are related to each other: Each virtual edge occurs in exactly two components, whereas each original edge occurs in exactly one component. For example, the virtual edge x occurs in components $T1$ and $T2$. In component $T1$, x represents the component $T2$, whereas x represents $T1$ in $T2$. Therefore, we obtain the original graph by merging the triconnected components at the virtual edges (which removes them).

The triconnected components can be arranged in a tree, cf. Fig. 5(b), where two components are connected if they share a virtual edge. The root of the tree is the unique component that contains the return edge. Each original edge is also shown in the tree under the unique component that contains that edge. Therefore, each component C determines a set F of edges of the original graph, namely all the leaves of the subtree that C corresponds to. For example, component $T1$ determines the set $F = \{a, b, c, d, e, f, g, h, i\}$ of edges. We call the subgraph formed by such a set F of edges the *component subgraph* of C . Figure 5(c) shows the component subgraphs of G . Note that the component subgraphs $B1$, $P1$ and $T1$ are fragments, whereas $B2$, $P2$ and $T2$ are not. There are also fragments that are not component subgraphs, for instance, $\{j, k, l, m\}$.

The precise definition of the triconnected components is rather lengthy and has therefore been omitted (see [12, 2, 3]). Instead we present here the exact relationship between the triconnected components and fragments we are going to exploit. The proofs

of the following two theorems can be found in [12]. First, we observe that triconnected components are closely related to boundary pairs.

Theorem 1. *A set $\{u, v\}$ of two nodes is a boundary pair of $U(G)$ if and only if*

1. *nodes u and v are adjacent in $U(G)$,*
2. *a triconnected component of $U(G)$ contains a virtual edge between u and v , or*
3. *a triconnected component of $U(G)$ is a polygon and contains u and v .*

We show examples based on $U(G)$ in Fig. 3(b) and its triconnected components in Fig. 5(a). For instance, the boundary pair $\{v6, v7\}$ contains two adjacent nodes of $U(G)$, the boundary pair $\{v1, v2\}$ corresponds to a virtual edge x of $T2$, and the boundary pair $\{s, v7\}$ contains two nodes of the polygon $P1$. Boundary pairs are closely related to fragments as follows.

Theorem 2. *1. If $F \in \mathcal{F}(u, v)$, then $\{u, v\}$ is a boundary pair of $U(G)$ and F is the union of one or more proper separation classes w.r.t. $\{u, v\}$.*
2. Let $\{u, v\}$ be a boundary pair of $U(G)$ and F the union of one or more proper separation classes w.r.t. $\{u, v\}$. If u is an entry of F and v is an exit of F , then $F \in \mathcal{F}(u, v)$.

For instance, the boundary pair $\{v5, v7\}$ has three proper separation classes $\{m\}$, $P2 = \{j, k, l\}$, and $\{n\}$. $P2$ is not a fragment, because $v5$ is neither its entry nor its exit, whereas $\{m\} \in \mathcal{F}(v5, v7)$ and $\{n\} \in \mathcal{F}(v7, v5)$ are fragments. The union of $P2$ and $\{m\}$ is a fragment, whereas $P2 \cup \{n\}$ and $\{m\} \cup \{n\}$ are not. $P2 \cup \{m\} \cup \{n\}$ is a fragment.

Theorem 1 says that the boundary pairs can be obtained from the triconnected components while Thm. 2 says that the fragments can be obtained from the boundary pairs.

2.3 Canonical Fragments and the Process Structure Tree

Two fragments F_1 and F_2 may *overlap*, that is, we have $F_1 \cap F_2 \neq \emptyset$, $F_1 \setminus F_2 \neq \emptyset$ and $F_2 \setminus F_1 \neq \emptyset$. Examples of overlapping fragments are shown in Fig. 6. Overlapping fragments give rise to nondeterministic parsing as explained in Sect. 1. We are therefore interested in a subset of fragments that do not overlap with each other. These will be called *canonical*. We comment on our particular definition of canonical fragments in Sect. 4. We start by defining various types of *bond fragments*.

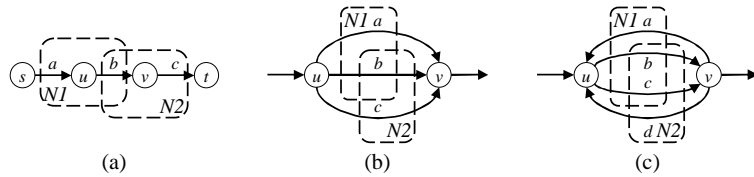


Fig. 6. Examples of overlapping fragments.

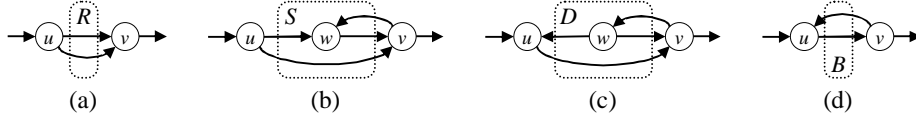


Fig. 7. Examples of (a) a pure bond fragment, (b) a semi-pure bond fragment, (c) a directed bond fragment, and (d) a bond fragment.

Definition 1 (Bond fragments). Let S be a proper separation class (i.e., a branch) w.r.t. $\{u, v\}$. S is directed from u to v if it contains neither an incoming edge of u nor an outgoing edge of v . $\mathcal{D}(u, v)$ denotes the set of directed branches from u to v . S is undirected if it is neither in $\mathcal{D}(u, v)$ nor in $\mathcal{D}(v, u)$. The set of undirected branches between u and v is denoted by $\mathcal{U}(u, v)$. A fragment $X \in \mathcal{F}(u, v)$ is

1. a bond fragment if it is the union of at least two branches from $\mathcal{D}(u, v) \cup \mathcal{U}(u, v) \cup \mathcal{D}(v, u)$.
2. a directed bond fragment if it is the union of at least two branches from $\mathcal{D}(u, v) \cup \mathcal{U}(u, v)$.
3. a semi-pure bond fragment if it is the union of at least two branches from $\mathcal{D}(u, v) \cup \mathcal{U}(u, v)$, and
 - (a) there exists no $Y \in \mathcal{U}(u, v)$ such that $Y \subseteq X$, Y has an edge incoming to u , or
 - (b) there exists no $Y \in \mathcal{U}(u, v)$ such that $Y \subseteq X$, Y has an edge outgoing from v .
4. a pure bond fragment if it is the union of at least two branches from $\mathcal{D}(u, v)$.

Note that the various bond-fragment types form a hierarchy, i.e., each pure bond fragment is a semi-pure bond fragment, each semi-pure bond fragment is a directed bond fragment etc. Fig. 7 shows examples of various classes of bond fragments that do not belong to a lower class. Bond fragments are closed under composition, i.e., we have:

Proposition 1. If $X, Y \in \mathcal{F}(u, v)$ and $F = X \cup Y$, then $F \in \mathcal{F}(u, v)$. If X and Y are bond fragments, so is F . If X and Y are directed (semi-pure) [pure] bond fragments, so is F .

Proposition 1 assures that a maximal bond fragment, maximal directed, maximal semi-pure, or maximal pure bond fragment is unique if it exists. We are now ready to define canonical fragments.

Definition 2 (Canonical fragment).

1. If $F_0 \in \mathcal{F}(v_0, v_1)$ and $F_1 \in \mathcal{F}(v_1, v_2)$ such that $F_0 \cup F_1 = F \in \mathcal{F}(v_0, v_2)$, we say that F_0 and F_1 are in sequence (likewise: F_1 and F_0 are in sequence) and that F is a sequence. F is a maximal sequence if there is no fragment F_2 such that F and F_2 are in sequence.
2. A bond fragment (directed bond fragment etc.) $F \in \mathcal{F}(u, v)$ is maximal if there is no bond fragment (directed bond fragment etc.) $F' \in \mathcal{F}(u, v)$ that properly contains F . A bond fragment $F \in \mathcal{F}(u, v)$ is canonical if it is a maximal bond fragment, a maximal directed, maximal semi-pure, or maximal pure bond fragment such that F is not properly contained in any bond fragment $F' \in \mathcal{F}(v, u)$.

3. A fragment is canonical if it is a maximal sequence, a canonical bond fragment, or neither a sequence nor a bond fragment.

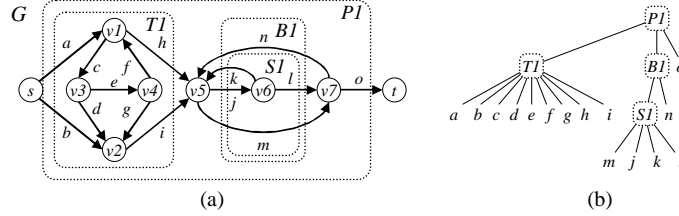


Fig. 8. (a) The non-trivial canonical fragments of G , and (b) the process structure tree of G .

Note that each edge is a canonical fragment, which we call a *trivial fragment*. Part (a) of Fig. 8 shows the non-trivial canonical fragments of graph G . $S1 \in \mathcal{F}(v_5, v_7)$ is a maximal semi-pure bond fragment, and $B1 \in \mathcal{F}(v_5, v_7)$ is a maximal bond fragment. $P1$ is a maximal sequence. $T1$ is neither a sequence nor a bond fragment.

To prove that canonical fragments do not overlap, i.e., two canonical fragments are either nested or disjoint, this claim is proven first for bond fragments that have the same entry-exit pair.

Lemma 1. Let $X, Y \in \mathcal{F}(u, v)$ be canonical bond fragments. Then $X \subseteq Y$, $Y \subseteq X$ or $X \cap Y = \emptyset$.

We continue by showing that two canonical bond fragments that share the same boundary pair do not overlap. In general, we can encounter two situations depending on whether the union of all branches with respect to a boundary pair is a fragment. These two cases are shown in Fig. 9.

In Fig. 9(a), the union of all branches with respect to the boundary pair $\{u, v\}$ is the maximal bond fragment from u to v called B . Fragments D, S and R are the maximal directed bond, semi-pure bond, and pure bond fragment from u to v respectively. Compared with part (a) of Fig. 9, part (b) has an additional edge outgoing from u that is outside of the union of all branches with respect to the boundary pair $\{u, v\}$. Because of this added edge, neither u or v is an entry of this subgraph. Thus, this set of edges is not a fragment. Fragment $R1$ is the maximal pure bond fragment from u to v . Fragment S is the maximal semi-pure bond fragment from u to v . As there is no larger bond fragment from u to v , S is also the maximal directed bond fragment and the maximal bond fragment from u to v . $R2$ is the maximal pure bond

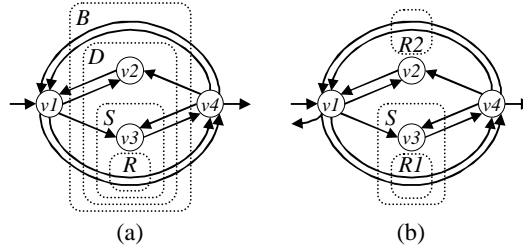


Fig. 9. Examples of canonical bond fragments.

fragment from v to u . As there is no larger bond fragment from v to u , $R2$ is also the maximal semi-pure bond, the maximal directed bond and the maximal bond fragment from v to u . Through an analysis of these two cases, we can prove the following:

Lemma 2. *Let $X \in \mathcal{F}(u, v)$ and $Y \in \mathcal{F}(v, u)$ be canonical bond fragments. Then $X \subseteq Y$, $Y \subseteq X$ or $X \cap Y = \emptyset$.*

We are now ready to present the main theorem.

Theorem 3. *Let X, Y be two canonical fragments. Then $X \subseteq Y$, $Y \subseteq X$ or $X \cap Y = \emptyset$.*

Theorem 3 allows us to define the unique process structure tree of a workflow graph.

Definition 3 (Process structure tree). *Let G be a TTG. The process structure tree (PST, for short) is the tree of canonical fragments of G such that the parent of a canonical fragment F is the smallest canonical fragment of G that properly contains F .*

Thus, the largest fragment that contains a whole workflow graph G is the root fragment of the PST. Part (b) of Fig. 8 shows the PST of graph G in part (a). The child fragments of a sequence $P1$ are ordered left to right from the entry to the exit of $P1$. For example, the order of child fragments of maximal sequence $P1$ is $T1$, $B1$ and o . Moreover, as $T1$ has the same entry as $P1$, the exit of $T1$ ($B1$) is the entry of $B1$ (o), and o has the same exit as $P1$. We use this ordering in Sect. 2.4 to derive all fragments from the canonical fragments. For this, it is not necessary to order the child fragments of a bond or a triconnected graph.

2.4 Computing all Fragments from the Canonical Fragments

The following proposition indicates how to derive all fragments from the canonical fragments. This is useful for example if one wants to find the smallest fragment that contains some given set of graph elements.

Proposition 2. *Let F be a set of edges in a TTG. F is fragment if and only if F is a canonical fragment or F is*

1. *a union of consecutive child fragments of a maximal sequence,*
2. *a union of child fragments of a maximal pure bond fragment, or*
3. *a union of child fragments of a maximal bond fragment B such that B is not a maximal directed bond fragment.*

For example, the maximal sequence $P1$ in Fig. 8 has $T1$, $B1$ and o as ordered child fragments. Besides these canonical fragments and the maximal sequence, also the union of $T1$ and $B1$ ($B1$ and o) is a fragment. However, the union of $T1$ and o is not a fragment, as these are not consecutive child fragments, i.e., they do not share a boundary node.

Part (a) of Fig. 10 shows a maximal pure bond fragment $R = \{a, b, c\}$. Its child fragments are $\{a\}$, $\{b\}$, and $\{c\}$. It follows from Prop. 2 that $\{a, b\}$, $\{b, c\}$, and $\{a, c\}$ are the non-canonical fragments in R . Part (b) of Fig. 10 shows a maximal bond fragment $B = \{a, b, c, d\} \in \mathcal{F}(u, v)$ and a maximal directed bond fragment $R = \{a, b\} \in \mathcal{F}(u, v)$ such that $B \neq R$. It follows from Prop. 2 that $\{c, d\}$, $\{a, b, c\}$ and $\{a, b, d\}$ are the non-canonical fragments in B . Note that $\{a, c, d\}$ and $\{b, c, d\}$ are not fragments, because their boundary nodes are neither entries nor exits.

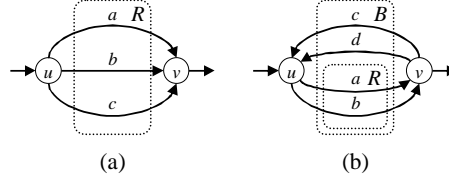


Fig. 10. Deriving non-canonical fragments from child fragments of (a) a maximal pure bond fragment and (b) a maximal bond fragment.

2.5 Modularity

Finally, we state what we mean by saying that our decomposition is modular.

Theorem 4. *Let G be a TTG and $X \in \mathcal{F}(u, v)$ be a canonical fragment of G . Let G' be the TTG obtained by replacing the subgraph that is formed by X by some other subgraph formed by a set of (fresh) edges X' such that $X' \in \mathcal{F}(u, v)$ is again a fragment of G' (but not necessarily canonical) with the same entry-exit pair as X . Assume that A is the parent fragment of X in G and $F \neq X$ is a child fragment of A in G . Let $A' = (A \setminus X) \cup X'$ and $F' = F$. Then A' and F' are canonical fragments of G' where F' is a child fragment of A' in G' .*

Theorem 4 means that a local change to the TTG, i.e., changing a canonical fragment X , only affects the PST locally. The parent and siblings of a changed canonical fragment remain in the PST in the same place and it follows inductively that this is also true for all canonical fragments above the parent and all canonical fragments below the siblings of X .

3 Computing the PST

In this section, we describe an algorithm that computes the PST in linear time. We have extended the algorithm by Valdes [1] to find all the canonical fragments (his algorithm produces a coarser decomposition, cf. Sect. 4). The algorithm has three high-level steps that are illustrated in Fig. 11 and described in Alg. 1. In Step 1, the tree of the triconnected components is computed, using e.g. the linear-time algorithm by Hopcroft and Tarjan [3]. Gutwenger and Mutzel [2] present some corrections to this algorithm. We illustrate the computed triconnected components through the respective component subgraphs in Fig. 11.

In Step 2, we analyze each triconnected component to determine whether the respective component subgraph is a fragment. This can be done in linear time with the following approach that takes advantage of the hierarchy of fragments. We analyze the tree of the triconnected components bottom-up—all children before a parent. For each

Algorithm 1 Computes the PST for a two-terminal graph.

buildPST(Graph G)

Require: G is a weakly biconnected TTG.

// Step 1. Compute the tree of the triconnected components of the TTG.

Tree := Compute the tree of the triconnected components of G .

// Step 2. Analyze each component to determine whether it is a fragment.

for each Component c in Tree in a post-order of a depth-first traversal **do**

 Count the number of edges (that are children of c) incoming to/outgoing from each boundary node. (4 counts)

 Add these edge counts to the respective edge counts of the parent component for each shared boundary node.

 Compare these edge counts to the total number of incoming/outgoing edges to determine whether each boundary node is entry, exit, or neither.

 Based on these boundary node types, determine whether c is a fragment.

if c is a polygon **then**

 Count the number of entry and exit nodes of the child components.

 If a child component is a fragment, order the child components from entry to exit.

// Step 3. Restructure the tree of the triconnected components into the tree of the canonical fragments (the PST).

for each Component c in Tree in a post-order of a depth-first traversal **do**

if c is a polygon **then**

 Merge consecutive child components (that are not fragments if any exist) if those form a minimal child fragment.

if c is not a fragment and c has at least two child fragments **then**

 Create a maximal sequence (that contains a proper subset of children of c).

if c is a bond **then**

 Classify each branch of c based on the edge counts of the boundary nodes of the respective child components of c .

if c is a fragment **then**

 Based on the classifications of the branches, create the maximal pure, the maximal semi-pure, and the maximal directed bond fragment, if any exists.

else

 Based on the classifications of the branches, create the maximal pure bond fragments, the maximal semi-pure bond fragment, if any exists.

for each Component d that has been created in this iteration **do**

 Merge the child fragments of each component in the to-be-merged-list of d to d .

if c is not a fragment **then**

 Add c to the to-be-merged list (a linked list) of its parent component.

 Concatenate the to-be-merged list of c to the to-be-merged list of its parent.

else

 Merge the child fragments of each component in the to-be-merged list of c to c .

return Tree

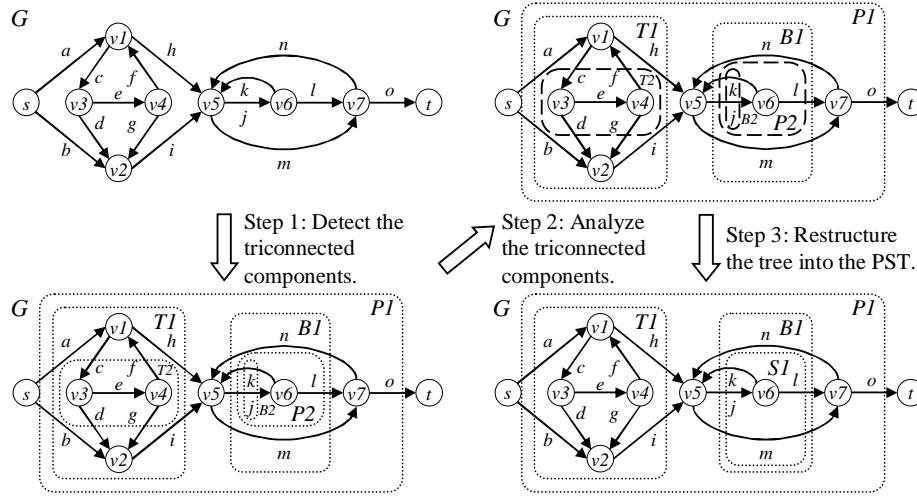


Fig. 11. The high-level steps of Alg. 1. Step 1: Detect the triconnected components. Step 2: Analyze each triconnected component to determine whether the respective component subgraph is a fragment. Step 3: Create the missing canonical fragments and merge the triconnected components that are not fragments.

child edge of a triconnected component c , we check whether it is incoming to or outgoing from one of the two boundary nodes of c . We count these edges to determine whether a boundary node is an entry, an exit, or neither. Based on this information, we can determine whether the respective component subgraph is a fragment. Note that when a triconnected component shares a boundary node with its parent, the same edges do not have to be counted twice, because an edge inside a child is also inside its parent.

In Step 3, we create the missing canonical fragments, and merge each component subgraph that is not a fragment to the smallest canonical fragment that contains it. This restructuring is based on the information computed in Step 2. New fragments are created only in those cases where a bond or a polygon contains canonical fragments that are not component subgraphs. Such a fragment is created as a union of at least two (but not all) children of this bond or polygon. We show examples in the following.

We process the tree of the triconnected components bottom-up as in Step 2. Thus, in Fig. 11, we can begin with $T2$. It contains no new canonical fragments, because it is neither a sequence nor a bond. $T2$ is not a fragment, because v_1 is neither its entry nor its exit. Thus, it will be merged into its parent fragment $T1$, that is, the children of $T2$ become children of $T1$.

The bond $B2$ is not a fragment, so it will be merged. $B2$ contains no new canonical fragments, because it has only two children. The same applies to $P2$. More interestingly, $B1$ is a fragment and has three children. Each child of a bond is a branch, and we classify them to find out whether they form new canonical bond fragments. $\{m\}$ is a directed branch from v_5 to v_7 , $P2$ is an undirected branch that has no outgoing edges from v_7 , and $\{n\}$ is a directed branch from v_7 to v_5 . Note that the branches can be classified based

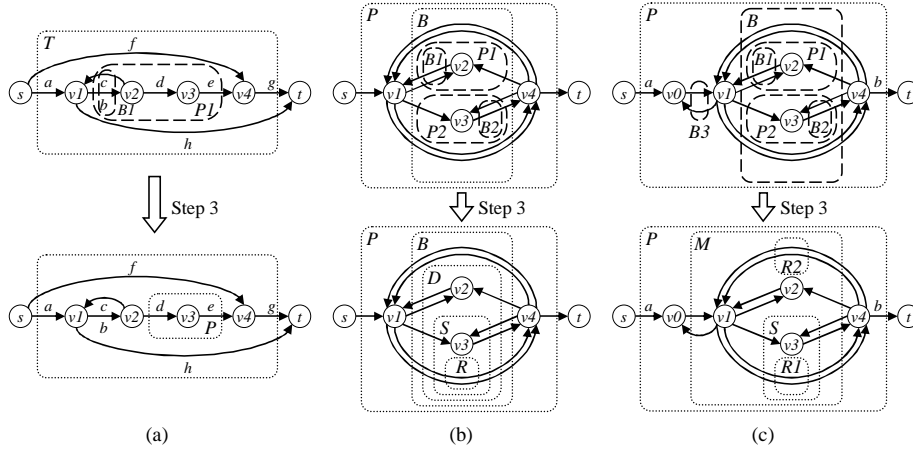


Fig. 12. Step 3: From the tree of the component subgraphs to the tree of the canonical fragments.

on the counts of edges incident to each boundary node of a branch computed in Step 2. There is a new semi-pure bond fragment $S1 = \{m\} \cup P2$. $B2$ and $P2$ are merged to $S1$. $S1$ and $\{n\}$ become the children of the restructured $B1$. Finally, $P1$ and all its children are fragments, thus there is no need to restructure $P1$.

In the following examples we show polygons and bonds in which more restructuring is required. In Fig. 12(a), $B1$ and $P1$ are not fragments. However, polygon $P1$ has two consecutive child fragments $\{d\}$ and $\{e\}$ that form a maximal sequence $P = \{d\} \cup \{e\}$. To determine whether a polygon contains such a new maximal sequence, we compute the number of entries and exits of its children already at the end of Step 2. A polygon that is not a fragment contains a maximal sequence as a union of its children if and only if its children have at least three entries or at least three exits in total.

In Fig. 12(b), $B1$, $P1$, $B2$, and $P2$ are not fragments and will be merged. Bond B is a fragment from $v1$ to $v4$ and has six branches: two edges as directed branches from $v1$ to $v4$, and one undirected branch, $P2$, that has no edge incoming to the entry of B , one undirected branch, $P1$, that has both an edge incoming to the entry of B and an edge outgoing from the exit of B , and another two edges as directed branches from $v4$ to $v1$. The directed branches from the entry to the exit of B form a new maximal pure bond fragment R . The union of $P2$ and R is a new maximal semi-pure bond fragment S . The union of $P1$ and S is a new maximal directed bond fragment. D and the remaining two directed branches are the children of B . $B1$ and $P1$ are merged to D , and $B2$ and $P2$ to S . P is a maximal sequence.

Figure 12(c) shows an example of a bond B that is not a fragment, but its children form new canonical fragments. As there are at least two directed branches to each direction, these branches form two new pure bond fragments, $R1$ and $R2$. The union of $R1$ and branch $P2$ is a semi-pure bond fragment S . Thus, $B2$ and $P2$ are merged to S . The polygon P has four children $\{a\}$, $B3$, B , and $\{b\}$. $B3$ and B not fragments, but the union of these consecutive siblings is a fragment. Thus, B is merged to $B3$ to form a new fragment M . $B1$ and $P1$ are also merged to M . The fragment P has only three children.

Each step of the algorithm can be performed in linear time. Thus, also the entire algorithm has linear time complexity.

Theorem 5. *The PST of a TTG G can be computed in time linear in the number of edges of G .*

4 Conclusion

We have presented a modular technique of workflow graph parsing to obtain fine-grained fragments with a single entry and single exit node. The result of the parsing process, the process structure tree, is obtained in linear time. We have mentioned a couple of use cases in Sect. 1. Coarser-grained decompositions may be desirable for some use cases. Those can easily be derived from the refined process structure tree by flattening. One such coarser decomposition, which can be derived and which is also modular, is the decomposition into fragments with a single entry edge and a single exit edge presented by Vanhatalo, Völzer and Leymann [14]. The new, refined decomposition presented here allows us to translate more BPMN diagrams to BPEL in a structured way. As an example, consider the workflow graph in Fig. 13 and (a) its decomposition with the existing techniques [9, 14] and (b) with our new technique. In Fig. 13(a), X cannot be represented as a single BPEL block, whereas in Fig. 13(b) each fragment can be represented as a single BPEL block.

The main idea of the technique presented is taken from Tarjan and Valdes [11, 1]. They describe an algorithm that produces a unique parse tree. However, they do not provide a specification of the parse tree, i.e., a definition of canonical fragments or claim or prove modularity. Moreover, our PST is more refined than their parse tree. Figure 12 shows examples of workflow graphs where this is the case. The fragments that are not identified by them are P in (a), D , S and R in (b), and S , $R1$ and $R2$ in (c).

We have made some simplifying assumptions about workflow graphs. The assumption that we have unique source and sink nodes can be lifted. Also the assumptions that the undirected version of the workflow graph is weakly biconnected and does not contain self-loops can be lifted. The necessary constructions to deal with these cases will be presented in an extended version of this paper. Thus the remaining assumption on workflow graphs will be that each node is on a path from some source to some sink.

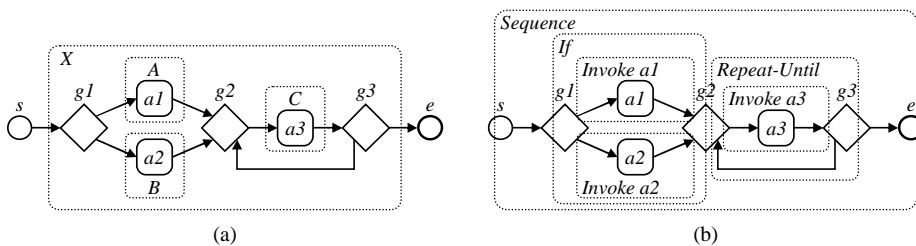


Fig. 13. A workflow graph and (a) decomposition presented in [9, 14] and (b) our decomposition.

The reader might wonder what justifies our particular definition of canonical fragments. It can be shown that the canonical fragments are exactly those fragments that do not overlap with any (canonical or non-canonical) fragment. This means, they are exactly the ‘objective’ fragments in the sense that they are compatible with any parse and hence appear in every maximal parse. Any finer decomposition into fragments can only be obtained by arbitrating between overlapping fragments. Our definition is further justified by Prop. 2, i.e., by the fact that all fragments and hence all parses can be derived from the PST in a simple way.

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. Jacobo Valdes Ayesta. *Parsing flowcharts and series-parallel graphs*. PhD thesis, Stanford University, CA, USA, 1978.
2. Carsten Gutwenger and Petra Mutzel. A linear time implementation of SPQR-trees. In Joe Marks, editor, *Graph Drawing*, volume 1984 of *LNCS*, pages 77–90. Springer, 2000.
3. J. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2:135–158, 1973.
4. Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI)*, pages 171–185, 1994.
5. Richard Craig Johnson. *Efficient program analysis using dependence flow graphs*. PhD thesis, Cornell University, Ithaca, NY, USA, 1995.
6. Jochen Küster, Christian Gerth, Alexander Förster, and Gregor Engels. Detecting and resolving process model differences in the absence of a change log. In *BPM 2008*, *LNCS*.
7. Kristian Bisgaard Lassen and Wil M. P. van der Aalst. WorkflowNet2BPEL4WS: A tool for translating unstructured workflow processes to readable BPEL. In *OTM Conferences (1)*, volume 4275 of *LNCS*, pages 127–144. Springer, 2006.
8. Niels Lohmann and Jens Kleine. Fully-automatic translation of open workflow net models into human-readable abstract BPEL processes. In *Modellierung 2008*, volume P-127 of *Lecture Notes in Informatics (LNI)*, pages 57–72. GI, March 2008.
9. Chun Ouyang, Marlon Dumas, Arthur H. M. ter Hofstede, and Wil M. P. van der Aalst. From BPMN process models to BPEL web services. In *ICWS*, pages 285–292. IEEE Computer Society, 2006.
10. Wasim Sadiq and Maria E. Orłowska. Analyzing process models using graph reduction techniques. *Inf. Syst.*, 25(2):117–134, 2000.
11. Robert E. Tarjan and Jacobo Valdes. Prime subprogram parsing of a program. In *POPL ’80: Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 95–105, New York, NY, USA, 1980. ACM.
12. Jussi Vanhatalo. *Structural analysis of business process models using the process structure trees*. (To appear).
13. Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The refined process structure tree. IBM Research Report RZ 3712, 2008.
14. 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, 2007.